Enabling Direct Bluetooth-WiFi Communications

by

Hsun-Wei Cho

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2024

Hsun-Wei Cho

hsunweic@umich.edu

ORCID iD: 0000-0002-6372-5928

DEDICATION

To my family.

ACKNOWLEDGEMENTS

I sincerely thank everyone for helping me over the last 6 years. For me, Ph.D. is a very personal journey, but I have met so many wonderful people along the way. I am grateful to Prof. Kang Shin for giving me the freedom to work on technologies that I am really passionate about and believe in strongly. I would like to thank the thesis committee members to guide me through the writing process. I am grateful for my fellow lab mates, graduated and current, for helping me travel through tough times. I would also like to thank many friends I met through trivia, honor societies, and the summer at Analog Devices. I am not good with words so excuse me for brevity, but I truly cherish the time and friendship I have with each and every one of you.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

ABSTRACT

Wireless communication has become an essential and ubiquitous technology in modern society. However, wireless technologies are *heterogeneous* where different wireless protocols were developed for various types of devices and applications. Since different wireless technologies use different waveforms, modulations, and protocol designs, direct communication between devices of different technologies is not possible. WiFi and Bluetooth are two most prominent and ubiquitous wireless technologies, but they are incompatible with each other, and thus cannot exchange information directly between them. As a result, additional hardware, such as dedicated radio chips or gateways, is required to connect devices of different technologies.

This thesis investigates this long-standing incompatibility in general and explores the direct communication between WiFi and Bluetooth in particular. It presents key concepts, theories, system designs and implementations that realize direct communication between WiFi and Bluetooth devices. The novel solutions proposed in this thesis enable mismatched wireless devices compatible down to the physical-layer and waveform levels. This intercommunication opens new opportunities for information to go beyond the boundary of one wireless technology, thus enabling highly connected networks with heterogeneous communication links. Furthermore, by allowing transmitters and receivers to use different wireless technologies, heterogeneous communication links combine the strength of both technologies. For example, Internet-of-Things (IoT) networks are enabled to use simple, energy-efficient, low-cost sensor nodes with direct Internet access by leveraging existing WiFi infrastructures.

This thesis presents pioneering research of Bluetooth–WiFi communication, which consists of several innovations. The first innovation, **BlueFi**, enables direct communication from widely-used WiFi chips to unmodified Bluetooth chips. BlueFi carefully crafts 802.11n packets on the transmitter side so that conventional Bluetooth receivers can accurately decode the waveform as legitimate Bluetooth packets. It enables practical use cases such as sending Bluetooth localization beacons with WiFi access points and streaming Bluetooth audio from WiFi cards to Bluetooth headphones. The thesis then presents **FLEW** that enables bidirectional communication between WiFi and FSK chips. FLEW identifies key signal processing properties and enables FSK chips to both send and receive standard WiFi packets with innovative system designs and implementations. By enabling bidirectional heterogeneous wireless links, FLEW is especially useful for IoT networks to combine the advantages of both Bluetooth (FSK) and WiFi. The third innovation is **Unify**, which is the first *single-chip* solution for bidirectional Bluetooth–WiFi communication. Unify works on widely popular FSK and BLE SoCs (System-on-a-Chip) and enables direct WiFi connectivity. Because of the high system integration of Unify, future IoT devices can be inexpensive, energy-efficient, and compact while also being capable of directly connecting to WiFi access points. To further improve the throughputs while also reducing power consumption, this thesis then proposes **DREW**, which enables WiFi-to-Bluetooth communication with twice the throughputs and Bluetooth-to-WiFi communication without relying on power-consuming mixers. With DREW, this thesis extends Bluetooth–WiFi communication to benefit more use cases, particularly for wearable devices and multimedia applications. The higher throughputs allow DREW to stream uncompressed, wired-equivalent stereo audio from WiFi to Bluetooth chips. Finally, this thesis applies the principle of Bluetooth–WiFi communication for other heterogeneous devices. We present **BBC**, which enables direct communication between Bluetooth Classic and BLE devices. BBC allows simple, energy-efficient BLE chips to communicate with Bluetooth Classic devices, such as Bluetooth headphones. Furthermore, future devices can use simpler radio hardware to support both Bluetooth Classic and BLE connections.

CHAPTER 1

Introduction

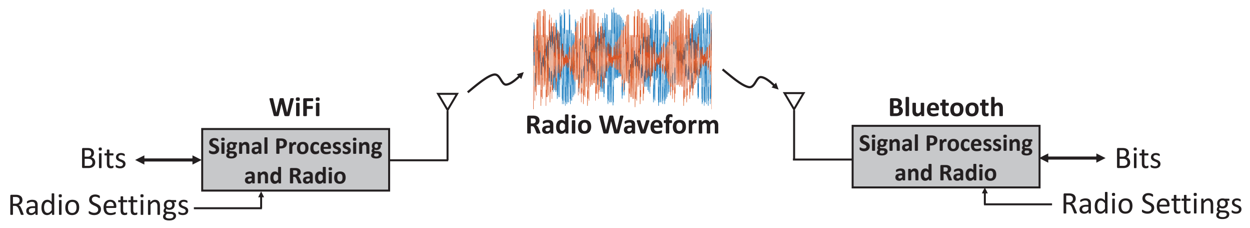


Figure 1.1: The key goal of this thesis is enabling off-the-shelf WiFi and Bluetooth chips to directly communicate with each other. This thesis applies signal processing techniques and advanced system engineering so that bits sent at one end can be fully recovered on the other end. The key variables within the communication system include the bits produced or consumed by the WiFi or Bluetooth chip, and the radio configuration of each device. Before the bits are transmitted or after the bits are received, pre-processing or post-processing is applied to enable Bluetooth–WiFi communication.

The communication technology is the foundation for exchanging information between people around the globe. In the information and data-driven era, the Internet plays a key role in the modern and connected society. While both wired and wireless connections can be used to connect devices to the Internet, wireless communication has become ever more important with the proliferation of smart, mobile and wearable devices. Using the electromagnetic waves to carry the information and with the signal processing algorithms at the transmitters and receivers, wireless communication eliminates the need for physical wires and allows users to enjoy greater mobility and convenience. These benefits of wireless communication are especially desirable for connecting end users, wearable devices, and sensors. For example, smart devices, such as laptops, smartphones or tablets, commonly use WiFi for Internet access. In another example, wireless (such as Bluetooth) headphones allow users to enjoy music with greater convenience.

Wireless devices come in different sizes, power budget and throughput requirements. Because of these different design goals, multiple wireless communication standards have been developed and adopted over the years. As a result, wireless devices in our daily life are *heterogeneous* where multiple wireless standards are used and coexist with each other. Within a group of heterogeneous wireless devices, devices with the same wireless protocol can communicate with each other. However, heterogeneous devices cannot intercommunicate *across* protocols. Consequently, supporting multiple wireless technologies requires separate radio components, radio chips or additional devices, such as wireless gateways.

Two most ubiquitous wireless standards are WiFi and Bluetooth. Each has tens of billions of devices deployed worldwide. However, WiFi and Bluetooth standards were developed independently, and their waveforms and network layers are completely different. These differences, especially in the physical and MAC layers, prevent direct communication between the two. If such an intercommunication were possible, it would enable a new communication paradigm that benefits tens of billions of WiFi and Bluetooth device users. This would open enormous opportunities for more connected, more efficient and more ever-present communication systems and networks.

This thesis makes intercommunication between WiFi and Bluetooth a reality. We build several practical direct WiFi–Bluetooth communication systems and investigate various use cases. We identify a number of unique challenges and present effective solutions thereof to meet the design goal of each system. Each chapter of this thesis contains key innovations that enable direct communication between WiFi and Bluetooth. Finally, we also present practical system implementations and their thorough evaluations in real-world settings.

1.1 Background

1.1.1 WiFi

WiFi is the *de facto* wireless local area network (WLAN) technology providing Internet access for tens of billions devices around the world. The underlying technology used by WiFi devices is the IEEE 802.11 standard.[1] WiFi uses radio waves to extend Ethernet wirelessly, thus allowing users to conveniently access the Internet. WiFi chips are widely adopted in typical mobile devices such as laptops, tablets and smartphones. Many smart devices (e.g., TV, doorbell cameras, speakers, cooking appliances, etc.) also use WiFi for reliable and high-speed Internet connectivity.

WiFi's radio operates in the industrial, scientific, and medical (ISM) band. By using the unlicensed radio band, WiFi allows end-users to enjoy cheap and high-speed connections without paying the cost of licensed spectrum and cellular infrastructures. To provide WiFi connectivity for countless mobile and smart devices, WiFi *infrastructures* are ubiquitous for users to connect to.

[1]The term "WiFi" was created by the WiFi Alliance, which was originally created as an industry consortium to ensure the interoperability among IEEE 802.11b devices from different vendors. Over time, WiFi has become a general term for the technology and devices leveraging the IEEE 802.11 standard.

These infrastructures include WiFi access points (AP), routers and WiFi hotspots. Because WiFi uses the unlicensed spectrum, WiFi equipment can be easily installed and is commonly deployed in personal, residential, enterprise settings. WiFi is also available in outdoor environments and in public transportation.

Since WiFi was developed as a direct wireless extension to Ethernet, the design of WiFi prioritizes throughputs over power consumption. Specifically, WiFi uses 20MHz (or wider) channels and significant computation is required on conventional WiFi chips to process the waveform. Consequently, off-the-shelf WiFi chips are more complex and consume considerably more energy than other wireless technologies, which use simpler radio hardware.

1.1.2 Bluetooth

Bluetooth is a popular personal area network (PAN) technology initially developed by Ericsson. Bluetooth is designed to provide short-distance communication with very low power consumption. It is widely used in devices where power consumption and long battery life are critical. Its common applications include Bluetooth headphones, speakers, location beacons, smartwatches and wearables.

The low power consumption of Bluetooth is achieved with its extremely simple radio hardware. Bluetooth uses the frequency-shift keying (FSK) modulation, which is, in essence, using frequency modulation (FM) with digital waveforms to transmit bits. Such a simple circuit can be implemented with a very small chip area and with ultra-low power consumption.

Similar to WiFi, Bluetooth operates in the ISM band and end-users can easily use Bluetooth devices by leveraging the unlicensed spectrum. Since a primary design goal of Bluetooth is low power consumption, it has significantly lower bandwidth than WiFi. Specifically, a Bluetooth channel is either 1MHz or 2MHz. Processing such signals requires less power than processing WiFi signals.

1.1.3 Internet of Things

The Internet of Things (IoT) has grown in popularity in recent years. IoT aims to connect everyday objects, especially sensors and actuators, to the Internet and the Cloud. By collecting large-scale sensing data and running analytics in the Cloud, connected sensors and actuators provide more intelligent monitoring and automation for smart homes, buildings, factories and cities.

Connectivity is a key component in IoT, and enabling ubiquitous connected sensors and actuators comes with unique challenges. Since IoT is for everyday objects, its networks must support a very large number of devices. Furthermore, IoT devices are commonly powered by small batteries and are massively deployed in unattended settings. It is therefore crucial that IoT devices have an extremely long runtime with battery. Such a requirement is very different from the connectivity requirement of traditional mobile devices where devices can be recharged daily. Unlike mobile devices where high throughputs are more important, IoT devices have to prioritize ultra-low power consumption in order to operate for years without replacing or recharging the battery. Finally, IoT connectivity should be extremely cheap and with a very small footprint so that IoT devices can be deployed on a large scale.

These unique challenges prevent IoT devices from directly adopting off-the-shelf WiFi chips. Even though existing WiFi infrastructures in smart buildings and cities can readily provide global Internet access and routability, using typical WiFi chips on the device side results in relatively higher power consumption, higher device cost and larger circuit area. For IoT devices where battery life, device size and cost are of primary concern, Bluetooth is a more suitable alternative to provide connectivity since it enables very simple devices with ultra-low power consumption. However, Bluetooth devices cannot directly connect to existing WiFi infrastructures because of the incompatibility in their wireless technologies. Such devices must, therefore, rely on IoT *gateways* which bridge personal area networks and the Internet. Deploying and maintaining IoT gateways is a key obstacle of the IoT deployment. Furthermore, Bluetooth, as a PAN, does not have the direct global routability of Ethernet and WiFi. Thus, establishing, maintaining and routing IoT connectivity at scale adds additional complexity.

Enabling direct communication between devices of incompatible technologies can speed up the adoption and deployment of IoT devices. Specifically, if Bluetooth devices can directly communicate with WiFi access points, simple and power-efficient Bluetooth chips with years of battery life can be used on IoT devices; ubiquitous WiFi infrastructures already installed in smart homes, buildings, and cities can be directly used to provide Internet access for IoT sensors and actuators. Dedicated IoT gateways are no longer needed to bridge and route IoT connections.

1.2 Cross-Technology Communication

Connecting heterogeneous wireless devices is known as *Cross-Technology Communication* (CTC) in the literature of wireless systems research. Specifically, CTC research investigates the direct communication between mismatched wireless devices and radio.

This thesis investigates and contributes to the latest research of CTC, and especially Bluetooth–WiFi communication. Despite prior non-WiFi and non-Bluetooth CTC research exists, direct communication between WiFi and Bluetooth is still unique and highly challenging. Even though WiFi and Bluetooth are two of the most ubiquitous wireless technologies to date, very little has been done to directly address the intercommunication between them. This thesis overcomes these challenges and presents several innovative and practical solutions to enable this CTC communication.

1.2.1 Challenges of Direct Bluetooth–WiFi Communication

Direct communication between WiFi and Bluetooth is highly challenging for several technical reasons. First, WiFi and Bluetooth are significantly different in each layer of the 7-layer OSI reference model. In particular, the physical and data link (MAC) layers are of special importance to direct Bluetooth–WiFi communication because they are commonly implemented or accelerated in hardware on off-the-shelf WiFi and Bluetooth chips. These components cannot be trivially modified to support different wireless standards. Since WiFi and Bluetooth chips are fundamentally different starting from the hardware level, direct communication cannot be achieved by simply running different networking stacks. Instead, new system designs and signal processing techniques have to be used to make two chips compatible at the system level, even when different radio hardware are used.

Second, at the physical layer, WiFi and Bluetooth use completely different signal modulations. WiFi uses either OFDM or DSSS (with a PSK chip rate of 11MChip/s). Bluetooth uses FSK at the speed of 1Mbps or 2Mbps. These modulations are incompatible and cannot intercommunicate with each other out of the box. Specifically, different modulations encode information differently in the physical waveforms. OFDM encodes the information using FFT/IFFT. DSSS encodes the information by changing the polarity of the carrier. FSK encodes bits into the instantaneous frequency of the radio wave.

The differences in data and chip rates result in different bandwidths at which conventional WiFi and Bluetooth radio operate. Also, because WiFi and Bluetooth were developed independently, their packet formats and bit processing procedures are entirely different. In particular, the bit scrambling, bit interleaving and differential coding processes are very different in WiFi and Bluetooth standards. WiFi and Bluetooth also use different forward error correction algorithms. Furthermore, WiFi and Bluetooth differ in how radio channels are utilized and selected. Bluetooth uses frequency-hopping spread spectrum (FHSS) where the carrier frequency of each packet changes constantly (especially in the connected state). In contrast, WiFi connections always use the same RF frequency. In particular, a WiFi station always uses the WiFi channel at which the connected WiFi access point operates.

Third, WiFi and Bluetooth are also distinctly different in the MAC layer. Bluetooth adopted the TDMA (Time-Division Multiple Access) design with strict time slots. WiFi uses CSMA/CA (Carrier-Sense Multiple Access with Collision Avoidance) where each device senses whether the wireless medium is busy or not and opportunistically transmits packets when the medium is not busy. WiFi and Bluetooth also differ in terms of packet acknowledgement mechanisms and the timing of sending acknowledgements. In typical WiFi traffic, an acknowledgement packet is sent about 10μs after a packet is received. In contrast, the time gap between data and acknowledgement packets in Bluetooth are in the range of hundreds of microseconds. In the original Bluetooth standard, acknowledgements are piggybacked in the data packet of the subsequent time slot, occurring 625μs after the current time slot. The difference in the MAC layer is challenging for realizing direct communication between WiFi and Bluetooth. For example, typical Bluetooth chips have relatively long turnaround time between reception and transmission. Without special system design, Bluetooth chips cannot support standard WiFi operations in the MAC layer.

1.2.2 The State of the Art

Earlier CTC efforts [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] build on and expand the principle of energy sensing. In particular, although a receiver designed for one standard cannot directly decode signals of other standards, the receiver can still sense the wireless spectrum and indicate if the medium is busy or not. That is, the receiver can indicate if there are ongoing wireless transmissions in the vicinity. For example, wireless communication chips typically have RSSI (Received Signal Strength Indicator) or can perform CCA (Clear Channel Assessment). Leveraging this feature, early CTC work enables communication between heterogeneous devices by sending packets on one end and detecting the *presense* (instead of the content) of packets on the other end. The presence of a packet represents a ‘1’ and the absence of a packet indicates a ‘0’, which makes communication between different wireless devices possible.

Such communication is essentially a low-speed ASK (amplitude shift keying) where the entire packet the transmitter sends (or does not send) constitutes a single ASK symbol. On the receiver side, RSSI or CCA information is used as an envelope detector to recover the ASK symbol. Therefore, although such an approach is broad and can be largely agnostic of the wireless standard that the radio chip natively supports, it suffers from low data rates (typically <1000bps) and is susceptible to interference due to the nature of the amplitude modulation (AM). Additionally, the implementations must modify both the transmitter and the receiver to run the custom AM modulation on top of the chips’ native modulation.

Recent, state-of-the-art CTC research focuses on making a device directly support a mismatched device’s native modulation in the physical layer (instead of running an additional modulation on top of their respective modulations). This approach requires a deep understanding of the native modulation of both sides. It also requires sophisticated physical-layer, signal processing and system designs to make the transmitter and receiver compatible. However, this approach enables considerably higher throughputs than energy-sensing CTC because devices’ native throughput can be matched. Furthermore, by directly supporting the other device’s modulation, minimal (or no) modifications are necessary on its end. This approach is more challenging to design but is more practical because of its higher throughput and less device modifications.

One representative prior work with this approach is OfdmFi [14, 15, 16], which investigates the communication between LTE-U and WiFi. Note, however, that LTE-U and WiFi are both based on OFDM modulation (with different subcarrier spacings). Other prior CTC works [17, 18, 19, 20] focus on one-way WiFi–Zigbee communication. Some other CTC works [21, 22] focus on Bluetooth–Zigbee communication. LTE to Zigbee communication has also been demonstrated [23]. It is worth noting that Zigbee applies spread spectrum when receiving the signals, which enables Zigbee receivers to correct significantly distorted signals. Many prior works focus on Zigbee because this error-correcting process makes it easier to achieve CTC with other wireless standards. This spread spectrum step, however, comes at the cost of the reduction in throughput. (The symbol rate of Zigbee is 8× slower than the chip rate.)

What is absent from the state of the art is practical and direct communication between WiFi and Bluetooth. Specifically, the state of the art lacks direct WiFi–Bluetooth CTC systems that use off-the-shelf WiFi or Bluetooth chips (as opposed to expensive software-defined radio equipment such as USRP). In addition to the multiple technical reasons described in Sec. 1.2.1, communication between WiFi and Bluetooth is also more challenging than WiFi–Zigbee CTC because Bluetooth is 4× faster than Zigbee and does not have Zigbee's error-correcting process.

1.3 Thesis Statement

As outlined in the previous section, the state-of-the-art CTC research has yet to achieve practical direct communication between WiFi and Bluetooth, despite the fact that such communication would benefit a huge number of devices around the globe and would be immensely useful. This thesis aims to enable such CTC communications by addressing numerous important technical questions: *Is direct communication between WiFi and Bluetooth possible? What are the key challenges of such communication? What are the solutions and the theoretical insights that can be leveraged to overcome these challenges? What are the different scenarios (e.g., one-way or bidirectional communication) in which our designs can be deployed? How can we build practical systems that can be run on off-the-shelf chips? What are the performance of these systems in real-world settings? What applications can direct WiFi–Bluetooth communication enable?*

To answer these questions, we conduct state-of-the-art wireless communication research and present several system designs in the following chapters. The thesis can be summarized as follows:

Thesis Statement:

This thesis proposes several practical and high-performance designs that enable direct communication between WiFi and Bluetooth, and address the theoretical, computational, and implementational challenges associated with them.

1.4 Insights and Methodologies

A key feature and design goal of the systems presented throughout this thesis is making heterogeneous wireless devices compatible with each other *down to the physical layer and waveforms*. Our designs aim to enable two devices to communicate with each other without using any relays or gateways. This requirement makes our systems widely applicable to existing WiFi and Bluetooth devices because no additional or specialized hardware is needed. On the other hand, we have to ensure WiFi and Bluetooth can somehow correctly receive information from each other.

To achieve this goal, a key observation used in this thesis is that wireless devices communicate via the physical electromagnetic waveforms emitted by transmitters and demodulated by receivers, and therefore the electromagnetic waveforms are the ultimate bridge between a transmitters and a receiver. Furthermore, the electromagnetic waveforms are only analog signals transmitted over the air. They are agnostic of the generation and decoding processes. For example, generating a Bluetooth waveform does not necessarily require a Bluetooth radio transmitter. A WiFi transmitter, when configured carefully, can generate a waveform with a modulation pattern highly similar to a Bluetooth waveform. Such waveforms can be decoded correctly by a Bluetooth receiver, even when they are transmitted by a WiFi transmitter. Therefore, even when two devices modulate or demodulate signals differently using different hardware, they can still communicate if, after careful signal processing designs, their waveforms at the physical layer have enough similarity.

From the perspective of systems design, the waveform compatibility and similarity can be engineered at either the transmitter or the receiver end. Specifically, pre-processing can be leveraged on the transmitter side so that bits to be transmitted are carefully generated to maximize the waveform similarity (to the waveform of another wireless technology) after signal modulation. Alternatively, post-processing can be used on the receiver side so that the transmitter's information is fully recovered using radio and DSP hardware that do not match the transmitter's modulation. Both approaches are explored in this thesis, and we identify several different real-world use-cases and opportunities where the waveform compatibility can be practically engineered and applied.

In terms of research methodologies, we use rigorous signal processing analyses, extensive simulation as well as innovative embedded system design throughout this thesis. Using digital signal processing theories and results, we analyze and abstract the behaviors of radio blocks, discover opportunities to maximize waveform similarity, and make various critical design choices. Furthermore, we use simulation at both the block-level and system-level. We use both simulated and real traces so that the system behavior is predictable and the performance is optimized. Finally, to showcase that the system designs are practical and directly applicable to existing devices, various innovative mechanisms are devised for different WiFi and Bluetooth chips.

1.5 Thesis Contributions

The thesis starts by investigating if direct WiFi–Bluetooth communication is possible with careful signal processing design that achieves the waveform similarity. We present **BlueFi**, which is the first to achieve direct communication from off-the-shelf WiFi to Bluetooth chips. Next, we investigate the problem of bidirectional communication between WiFi and Bluetooth. We design **FLEW**, the first to achieve bidirectional direct communication between WiFi and FSK chips. We then look into the problem of whether such communication can be a single-chip solution. We address this with **Unify**, a system designed for SoC (System-on-a-Chip). Also, we tackle the problem of increasing the throughputs of WiFi–Bluetooth communication and present **DREW**, a system that doubles the throughputs on ultra-low-power Bluetooth chips. Finally, we extend the techniques of the above systems and investigate the problem of direct communication between Bluetooth Classic and BLE chips, which is solved by **BBC**.

The specific technical contributions and applications of each system are as follows:

1.5.1 BlueFi

Chapter 2 investigates the problem of one-way communication from modified WiFi chips to unmodified Bluetooth devices. We solve this problem by designing, implementing and demonstrating a novel system called **BlueFi**. It can readily run on existing, widely-deployed WiFi devices without modifying NIC firmware/hardware. BlueFi works by reversing the signal processing of WiFi hardware and finds special 802.11n packets that are decodable by unmodified Bluetooth devices. With BlueFi, every 802.11n device can be used simultaneously as a Bluetooth device, which instantly increases the coverage of Bluetooth, thanks to the omnipresence of WiFi devices. BlueFi is particularly useful for WiFi-only devices or environments. We implement and evaluate BlueFi on devices with widely-adopted WiFi chips. We also construct two prevalent end-to-end applications, Bluetooth beacon and audio, to showcase the practical use of BlueFi. The former allows ordinary access points to send location beacons while the latter enables WiFi chips to stream Bluetooth audio in real time.

1.5.2 FLEW

After addressing the one-way communication problem, the next step is to explore bidirectional communication. Chapter 3 investigates bidirectional direct communication between WiFi and

FSK chips. FSK is the modulation format of Bluetooth. We show that this bidirectional direct communication is actually possible and present a system called **FLEW** (Fully Emulated WiFi), which uses an FSK chip to fully emulate both transmission and reception of WiFi signals. Using FLEW, FSK-equipped IoT or mobile/wearable devices can directly communicate with unmodified WiFi APs, just like any WiFi device. FLEW combines the best of both technologies: extremely simple and low-power hardware and ubiquitous Internet access.

1.5.3 Unify

While FLEW provides several ground-breaking benefits, there are some practical limitations which are addressed in Chapter 4. In particular, FLEW requires an external microcontroller and requires slight hardware modifications to FSK chips. Chapter 4 describes a novel system called **Unify**, which is a single-chip solution and requires no hardware modifications. By just updating firmware, Unify transforms BLE/FSK SoCs into WiFi SoCs. It enables bidirectional communication between BLE/FSK devices and unmodified WiFi APs/devices, and is compatible with products of all major WiFi chip vendors. Thanks to their high integration, Unify devices are smaller, cheaper and consume significantly less power than both off-the-shelf WiFi chips and FLEW.

1.5.4 DREW

The systems presented in Chapter 3 and 4 only support BPSK demodulation, and thus their throughputs are limited to 1Mbps. Furthermore, these systems leverage the transmission mixers to generate BPSK modulation signals. However, newer ultra-low-power (ULP) BLE chips remove these mixers to conserve power. We address these problems and limitations in Chapter 5 by designing a system called **DREW**. DREW is designed to effectively transmit WiFi packets by only controlling the power amplifier (PA), and is thus applicable to mixer-less ULP BLE chips. We also propose an innovative use of BLE's IQ sampling capability to receive standard WiFi packets. We design efficient algorithms with SIMD (Single Instruction Multiple Data) acceleration to detect, synchronize and demodulate WiFi packets from IQ samples in real time. DREW also implements WiFi's CSMA/CA and timing, thus adding direct WiFi connectivity to ULP BLE devices. With IQ sampling, DREW uniquely supports QPSK and therefore doubles the downlink throughput. This $2\times$ throughput increase is crucial for new applications that prior work cannot support. In particular, DREW can stream lossless, HiFi-quality audio from WiFi to ULP BLE chips. Since stereo audio requires a throughput of 1.411Mbps, no prior work can support this important application due to their 1Mbps limitation.

1.5.5 BBC

The concepts, techniques, and domain knowledge developed after designing various WiFi–Bluetooth communication systems are valuable for direct communication between heterogeneous wireless technologies in general. In Chapter 6, we design a system, **BBC**, which enables bidirectional Bluetooth-Classic connectivity on BLE-only chips. BBC sends and receives raw FSK bits using BLE chips while emulating all other Bluetooth-Classic operations in the driver. By eliminating the need for Bluetooth-Classic hardware, BBC enables future devices to use BLE-only chips while maintaining the Bluetooth-Classic compatibility via emulation. BBC also enables new connectivity for current BLE devices to directly stream audio to Bluetooth-Classic headphones. BBC achieves a throughput of 557kbps and a PER of 4.86% at 10 meters, and provides the same audio quality as off-the-shelf Bluetooth-Classic chips.

1.6 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 describes the design and evaluation of **BlueFi**, which enables practical one-way communication from WiFi to Bluetooth devices. Chapter 3 introduces **FLEW**, a system achieving bidirectional communication between WiFi and FSK chips. Chapter 4 presents **Unify**, which further enables the bidirectional communication on single-chip BLE and FSK SoCs. Chapter 5 describes **DREW**, which doubles the throughput for Bluetooth–WiFi communications. Chapter 6 extends the idea of communication between WiFi and Bluetooth and **BBC** makes bidirectional Bluetooth-Classic communication possible on BLE single-mode chips. Finally, Chapter 7 concludes this thesis and discusses future directions.

CHAPTER 2

BlueFi: Bluetooth over WiFi

2.1 Introduction

The future of wireless communication is nothing short of heterogeneous technologies, as each technology comes with its own set of strengths and weaknesses. Tailored to its specific communication paradigm, each wireless standard/technology typically uses vastly different bandwidth, modulation/coding, and medium access control. This is bad news for supporting multiple heterogeneous wireless standards as each technology requires dedicated hardware, deployment and maintenance.

Bluetooth plays a key role in providing valuable functions such as location and automation services in business, industrial or public settings, such as retailers, buildings and airports. The number of Bluetooth location-service devices is projected to grow at an annual rate of 43% and reach 431M by 2023 [24]. Bluetooth is also the dominant technology used for personal audio streaming. 1.1B Bluetooth audio streaming devices were shipped in 2019 alone and the figure is expected to grow 7% per year [25].

On the other hand, more than 30B WiFi devices have already been shipped over the recent years, of which more than 13B devices are in active use [26]. Many of these devices are Access Points (APs) already deployed in the environments, providing pervasive coverage of WiFi signals. Cisco estimates that the number of WiFi hotspots in public alone will reach 628M by 2023 [27]. If WiFi hardware can be concurrently re-purposed as Bluetooth hardware, it will significantly increase the coverage of Bluetooth signals and provide useful Bluetooth functions in environments where only WiFi hardware is present. For example, to provide Internet connectivity for billions of devices, WiFi APs have been ubiquitously deployed, but almost none of them comes with Bluetooth hardware or Bluetooth connectivity. Dedicated Bluetooth infrastructures are also much less prevalent than WiFi infrastructures. Some desktops or low-cost mobile devices are only equipped with WiFi chips. Most USB WiFi NICs do not have Bluetooth functions. If WiFi–Bluetooth communication is possible, every AP can also function as a Bluetooth device, such as a Bluetooth beacon. Alternatively, users can use Bluetooth peripherals, such as Bluetooth headphones, with WiFi-only devices. With WiFi-Bluetooth communication and by leveraging the Broadcast Audio feature in the latest Bluetooth standard, it is even possible to use WiFi APs to broadcast audio streams to nearby Bluetooth headphones and provide interactive and immersive experiences in venues such as museums. Finally, thanks to the connectivity of WiFi devices, these emulated BT functions can be controlled remotely, even from cloud servers, which nicely fits the IoT paradigm.

In this chapter, we present `BlueFi`, a novel system that enables the transmission of legitimate BT signals using 802.11n-compliant hardware with simple driver updates. `BlueFi` requires no modification whatsoever to the hardware and firmware of Bluetooth receivers and of WiFi chips. Since newer WiFi standards, such as 802.11ac and 802.11ax, mandate the compliant hardware to be backward-compatible with 802.11n, `BlueFi` can run on 802.11ac and 802.11ax hardware as well. `BlueFi` carefully compensates and reverses the operations of WiFi hardware, and crafts special WiFi packets. These special packets, sent by our updated WiFi drivers, result in 802.11n-compliant waveforms which are also decodable by Bluetooth devices. Since it leverages the overall WiFi standard and vendor-agnostic hardware functions, `BlueFi` can run on *any* 802.11n-compliant chips, instead of specific chips from particular manufacturers.

However, transmitting BT signals using WiFi hardware is very challenging since the two wireless standards are very different from each other. At the highest level, Bluetooth encodes the information in the time domain whereas WiFi (specifically, 802.11a/g/n/ac/ax) uses OFDM and encodes the information in the frequency domain. Furthermore, most of the bit manipulation and signal processing will be automatically applied by WiFi hardware, and they cannot be bypassed. These operations will appear as signal impairments when a Bluetooth waveform is transmitted. We identify four major sources of impairments introduced by WiFi hardware.

I1. Cyclic Prefix (CP) Insertion: OFDM systems use CP to overcome inter-symbol interference (ISI). However, a small portion of the Bluetooth waveform we want to transmit will be overridden by this CP insertion, which is a copy of the tail of an OFDM symbol. Specifically, one WiFi symbol corresponds to approximately 4 Bluetooth bits. Therefore, part of the IQ waveform in bit 3 will overwrite the IQ waveform in bit 0. We find a Bluetooth receiver unable to pick up the signal without carefully compensating for the CP insertion process.

I2. QAM Modulation: OFDM encodes the information in the frequency domain before applying IFFT to generate the time-domain signal. Although we can use FFT to get the frequency-domain equivalent of a Bluetooth waveform, we cannot perfectly reconstruct the frequency-domain signal since a WiFi transmitter can only generate constellations with a very coarse resolution in the frequency domain. For example, using 64 QAM, samples at each subcarrier in the frequency domain must be selected from one of the 64 constellations. The difference between the selected constellation and the ideal value causes impairments in the frequency domain and subsequently in the time domain. Selecting the optimal constellations for the best Bluetooth performance can be formulated as an integer-programming (IP) problem and hence is NP-Complete. Solving the IP problem by exhaustive search is nearly impossible for this problem size.

I3. Pilots and Nulls: Not all subcarriers in one OFDM symbol are used for data transmission. Four of the subcarriers are for pilot signals and they are, on average, of higher magnitudes than those for data transmission. In contrast, some subcarriers, such as subcarrier 0, must be 0. These pilots and nulls will corrupt the transmission waveform if they are too close to the center frequency of a Bluetooth channel.

I4. FEC Coder: WiFi uses forward error correction (FEC) to combat communication errors. Since FEC encoders add redundancy to the bit-stream, some bits are related at the encoder's output. So, the encoder cannot generate arbitrary sequences. `BlueFi` must thus decide which bits are more important than others and find an input sequence that minimizes the important bits' hamming distance between the target output sequence and the reconstructed output sequence.

We have designed `BlueFi` to overcome the above impairments, and tested it on real, widely-adopted WiFi chips to find the transmitted signals are correctly decoded by conventional, unmodified Bluetooth devices. Although these impairments degrade the signal quality, the received signal strength is actually *higher* since WiFi is allowed to transmit at high power. We have also evaluated the effect of each impairment.

`BlueFi` enables tremendous opportunities for real-world applications. For example, `BlueFi` makes it possible to send Bluetooth beacons using WiFi infrastructures that have already been deployed almost everywhere. This will be very useful, especially in corporate, business or public environments, to provide useful features, such as way-finding, navigation, proximity marketing and more, all besides WiFi connectivity, simultaneously. Because of such market needs, various solutions have already been proposed. For example, the Cisco Virtual Beacon [28] adds the Bluetooth beacon functionality to existing Cisco APs. However, the Cisco solution requires a dedicated, purpose-built hardware to be installed on every AP and hence incurs hardware and deployment costs. (The word *virtual* refers to the fact that it is a networked solution and allows remote management and updates.) In contrast, we can implement such functionality readily on existing WiFi APs with `BlueFi` and no additional hardware is needed. In a sense, `BlueFi` is a true virtual solution that enables Bluetooth purely at the software level. To demonstrate this, we have built an end-to-end example in which an 802.11n-compliant AP is transformed into a Bluetooth beacon.

`BlueFi` can work with general and real-time Bluetooth apps as well. In particular, we are able to stream real-time audio with A2DP (Advanced Audio Distribution Profile) using WiFi chips. We envision that `BlueFi` will help eliminate the need for dedicated Bluetooth or combo chips in future devices, saving costs and the precious board-space, which is very important for small devices such as smart phones or watches. Alternatively, `BlueFi` can help users use Bluetooth headphones with laptops or desktops with old or no Bluetooth hardware.

2.2 System Design

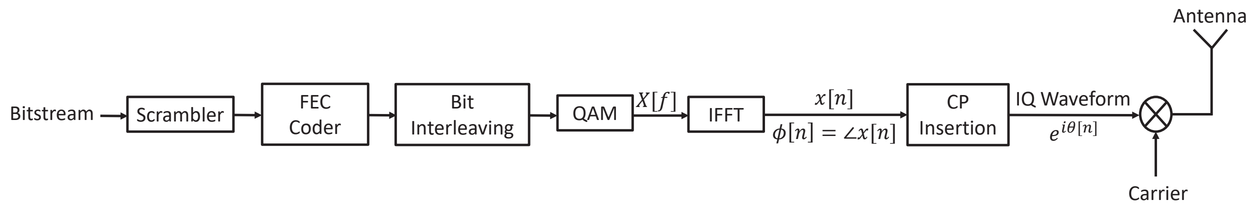


Figure 2.1: Block diagram of 802.11n transmitters

2.2.1 Primers

We first review the PHY specifications of Bluetooth and 802.11n. By comparing these technologies, we explore the opportunities of leveraging the functionalities of existing WiFi hardware to transmit Bluetooth signals.

2.2.1.1 Bluetooth

Bluetooth uses GFSK (Gaussian Frequency-Shift Keying), which is frequency-shift keying with a Gaussian filter applied to the input bit-stream to reduce spectral leakage. For FSK, the output has a positive frequency deviation for bit “1” and a negative frequency deviation for bit “0”. Since phases can be obtained by integrating frequencies, sending 1’s results in phases with a positive slope and sending 0’s results in phases with a negative slope. In addition, since no information is encoded in the amplitude of the time-domain waveform, a Bluetooth packet can be fully characterized by only the waveform’s phases. Bluetooth devices should support the basic 1Mbps data rate, so the bit duration in Bluetooth is 1000ns.

For Bluetooth beacons, advertisement packets are broadcast on 2402, 2426 or 2480 MHz and frequency hopping is not required for beacon operation. In fact, it is the receiver’s responsibility to scan all 3 advertisement channels and the transmitter can transmit at 1, 2 or 3 channels [29]. In contrast, frequency hopping is critical to the operation of connected devices and packets are transmitted in time slots. Each time slot is 625μs long and a device can only start transmission in every other time slot. Once the transmission starts, a single packet can occupy multiple (3 or 5) slots and the frequency stays the same during a multi-slot transmission.

2.2.1.2 802.11n

Officially known as High Throughput (HT) PHY in the 802.11 standard [30], 802.11n inherited the same OFDM structure as 802.11a and 802.11g. Fig. 2.1 shows the block diagram of a typical 802.11n transmitter. The bit-stream, sent from the MAC layer, is fed to a scrambler to remove long-running 1's or 0's. The scrambler XORs the input bit-stream with a pseudorandom sequence generated by shift registers. To enhance robustness, a forward error correction (FEC) encoder then adds redundancy to the scrambled bit-stream. Different code rates can be selected and are achieved by skipping the transmission of some encoded bits ("puncturing"). Instead of assigning adjacent bits to the same or nearby subcarriers, an inter-leaver enhances robustness further by evenly spreading nearby bits to subcarriers that are far apart. In the mandatory 20MHz mode, 52 out of 64 subcarriers are used for data transmission. Bits are grouped and placed on these subcarriers with BPSK, QPSK, 16-QAM or 64-QAM mapping. Subcarriers are separated by 20/64=0.3125MHz, and subcarriers -21,-7,7 and 21 are used for pilot tones. Subcarrier 0 is always 0. The samples on these 64 subcarriers are converted to a 64-sample-long time-domain signal via IFFT. The last 16 time-domain samples are copied and inserted into the front of the 64 samples. The inserted portion of the waveform is known as the *cyclic prefix* (CP). These 80 samples constitute one 802.11n OFDM symbol. The data portion of an 802.11n waveform (normally) consists of multiple OFDM symbols. To further reduce the spectral leakage caused by the discontinuity between OFDM symbols, the standard suggests application of windowing in the time domain. For two consecutive symbols, windowing can be achieved by appending the first symbol with the first sample of the IFFT results and then setting the first sample of the second symbol to the average between these two values. Sixteen 0's are inserted into the front of MAC layer bit-streams so that the receiver can determine the scrambler seed the transmitter is using. The data portion is appended to an 802.11n preamble, which contains various parameters used by the transmitter and signals for synchronization and CFO (carrier frequency offset) correction. We used the "Mixed Format" preamble since it is mandatory in 802.11n.

802.11n includes several key features. For `BlueFi`, the most important is the short guard interval (SGI), the only reason why `BlueFi` requires 802.11n hardware instead of 802.11g. With SGI, the length of CP is reduced from 16 samples (800ns) to 8 samples (400ns), and hence less impairment is introduced by the insertion of CP. SGI directly increases throughput by more than 10%, and therefore is implemented on all devices from all major vendors even though it is an optional feature. Any 802.11n NIC or router with an advertised speed of 150, 300, 450 or 600Mbps has the SGI feature.

Frame aggregation is a mandatory feature in 802.11n. Although the maximum length of a single MAC layer payload (MPDU) is 2,304 bytes, the PHY payload (PSDU) can be as long as 65,535 bytes. The 802.11n-compliant NIC's ability to transmit huge packets enables `BlueFi` to generate a very long waveform if needed. Supporting multiple antennas is a major focus of 802.11n, but it is an optional feature. Therefore, all devices should support using a single spatial stream.

2.2.2 Overview and Methodology of `BlueFi`

`BlueFi` starts with a simple principle: *As long as the IQ waveforms generated by a WiFi chip are close enough to those generated by a Bluetooth transmitter, Bluetooth devices will be able to correctly receive the signals*. Therefore, given a synthesized Bluetooth IQ waveform, we aim to find a corresponding WiFi bit-stream so that when it is fed into an 802.11n transmitter, the generated IQ waveform will be as close to the Bluetooth IQ waveform as possible. Finding the corresponding bit-stream is somewhat similar to simply decoding an 802.11n packet received from the radio. However, the former differs from the latter in that how "close" the reconstructed IQ waveform is to the target Bluetooth IQ waveform should be determined by the decoding process of a Bluetooth receiver, and a small signal deviation, from the WiFi hardware's perspective, can completely disrupt the decoding process of a Bluetooth receiver.

Therefore, we use the following methodology: just like decoding 802.11n packets, `BlueFi` tries to reverse the operation of each block in the transmitter one-by-one. However, the results of the reverse operation of each block are selected based on how close they can reconstruct the IQ waveform *from a Bluetooth receiver's perspective*.

2.2.3 Construction of IQ Waveform

For simplicity, we assume Bluetooth's GFSK bits, including the entire packet from the preamble to the CRC, are fed into `BlueFi`. We also assume the payload is properly scrambled with a correct seed. We have built a tool for converting Bluetooth payload to GFSK bits, which can also be done by other software tools. We construct the frequency signal by converting 1's and 0's with respective frequency deviations. Since typical WiFi hardware generates the IQ signal at the sampling rate of 20MHz, each 1 or 0 corresponds to 20 samples of the frequency signal. We also insert 0's to the front and to the back of the frequency signal since we observed such a pattern on commercial Bluetooth chips. We then convert the frequency signal to its phase signal by accumulating the frequency signal. Since the center frequency at which we wish to transmit the Bluetooth packet may not be exactly the same as one of the WiFi channels, we modulate the phase signal (sample-wise adding linearly increasing phases) so that the output is the phase signal with respect to the center of a WiFi channel. This modulating operation must be applied before CP insertion since these two operations are not commutative for phase signals. We denote this phase signal as $\theta[n]$.

2.2.4 CP Insertion

This process is illustrated with phase signals as we can always convert a phase signal $\theta[n]$ to its corresponding IQ waveform of $e^{i\cdot\theta[n]}$. The input of the CP insertion block can be mapped 1-to-1 to the output, and vice versa. Therefore, instead of seeking an input that will be mapped to the best-fitting IQ waveform, we first find an *output* IQ waveform $\hat{\theta}[n]$ that can be: (1) received by Bluetooth devices, and (2) generated by the CP insertion block.

The output of the CP insertion block always shows the first 8 samples being identical to the last 8 samples in every 72 samples. Therefore, the most basic waveform that satisfies (2) can be generated by copying the first 8 samples to the last 8 samples in every 72 samples. The CP insertion process technically copies the last 8 samples from the last 64 samples and inserts them to the front. However, since we have complete freedom in designing the last 64 samples, they can be generated in a way the last 8 samples appear to have been overwritten by the CP waveform inserted at the front.

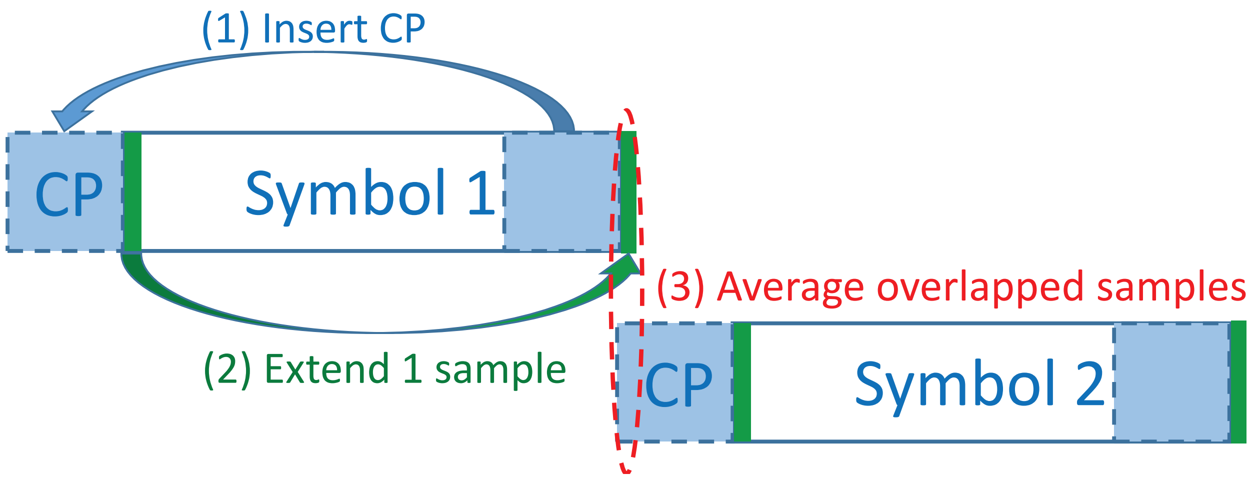


Figure 2.2: CP insertion and OFDM symbol windowing

Although the waveform of this simple method has shown acceptable performance in our simulations and when transmitted by USRP, it shows a very poor performance when transmitted by real WiFi chips and some Bluetooth receivers cannot pick up any signal at all. By transmitting various IQ waveforms with USRP and analyzing the responses of Bluetooth receivers, we found that this has something to do with windowing applied to each OFDM symbol, which is recommended by the standard to be implemented, dated all the way back to 802.11a, to reduce spectral leakage. The operation of OFDM windowing is illustrated in Fig. 2.2. According to the standard, the windowing works by extending each OFDM symbol by 1 sample (which is copied from the sample immediately following the CP) and then averaging the overlapped samples in the time domain. Since adding two phase samples in the time domain creates an erratic phase, the carefully-designed phase signal is corrupted in 1 of every 72 samples (on top of the CP corruption). We found this corruption alone enough to make the difference of reception/no reception on some devices. Therefore, we must consider one additional constraint, which can be summarized as the *continuity constraint*: for each OFDM symbol, the last few samples *along with the extended sample* must appear continuous with the first few samples in the next OFDM symbol.

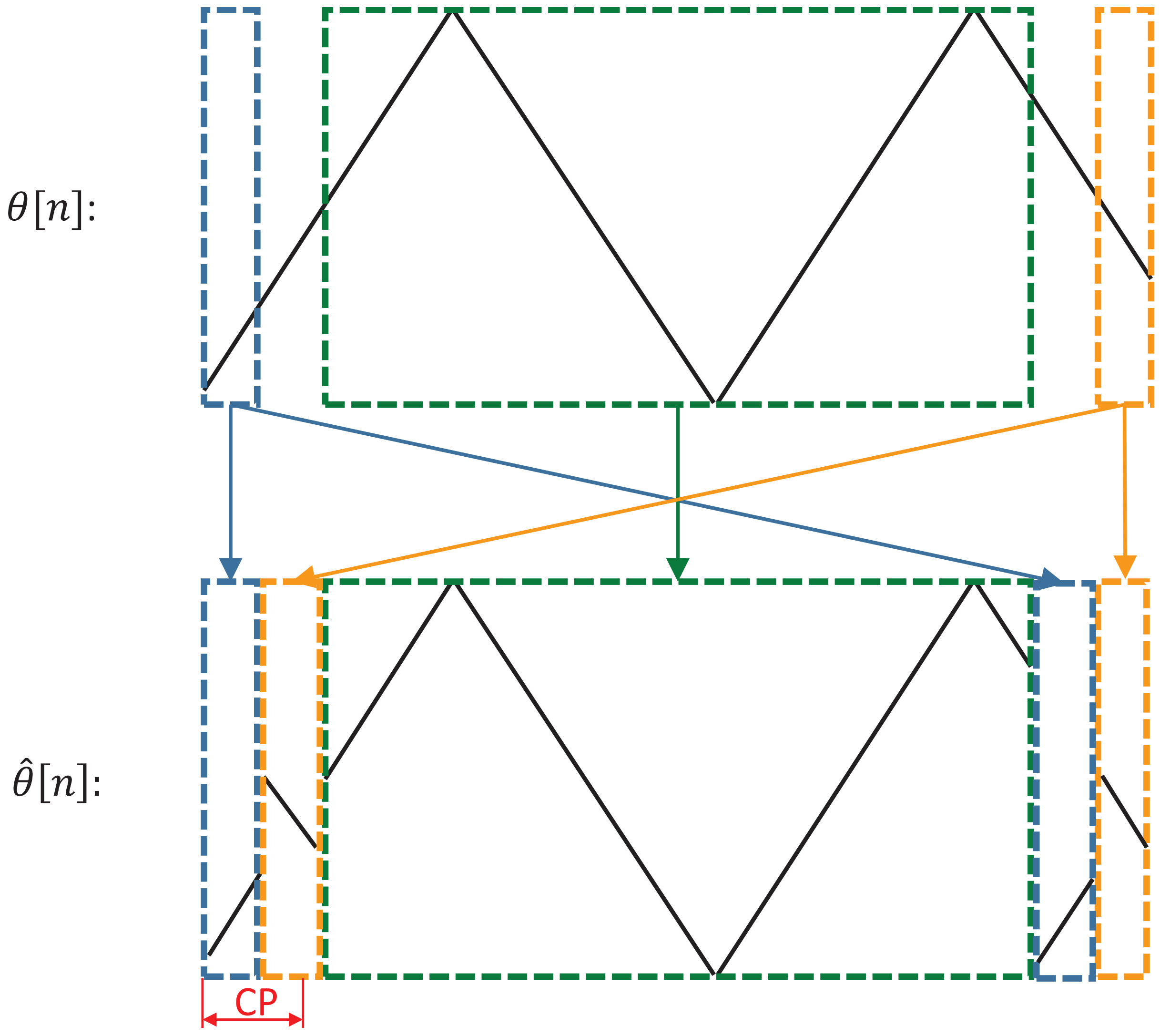


Figure 2.3: Constructing $\hat{\theta}[n]$ from $\theta[n]$ for every symbol

We found a way to construct an IQ waveform (whose phase is $\hat{\theta}[n]$) that satisfies all these constraints. The process is illustrated in Fig. 2.3. Mathematically,

$$\hat{\theta}[N+n] = \begin{cases} \theta[N+n], & 0 \le n \le 4 \\ \theta[N+n+64], & 5 \le n \le 8 \\ \theta[N+n], & 9 \le n \le 63 \\ \theta[N+n-64] = \hat{\theta}[N+n-64], & 64 \le n \le 68 \\ \theta[N+n] = \hat{\theta}[N+n-64], & 69 \le n \le 71 \end{cases}$$

where $N = 0, 72, 144, \cdots$.

Note that the CP ($0 \le n \le 7$) is exactly the same as the tail ($64 \le n \le 71$). Also, during the windowing operation, each OFDM symbol is extended by one sample $\hat{\theta}[N+72] = \hat{\theta}[N+8]$. Since $\hat{\theta}[N+8]$ is set to the first sample in the next symbol, $\theta[N+72]$, the windowing has no effect on the waveform. ($0.5 \cdot \theta[N+72] + 0.5 \cdot \hat{\theta}[N+72] = \theta[N+72]$.)

Since the CP insertion cannot be turned off in commercial chips, signal degradation is unavoidable. However, by designing the waveform this way, the signal degradation is spread out between the first and the last Bluetooth bits in every WiFi OFDM symbol. For these two bits, the degradation is less than 250ns, which is shorter than the bit duration of 1000ns. Furthermore, this short-term degradation will mostly appear as a high-frequency (1/250ns=4MHz) noise and is likely to be attenuated/removed by the band-pass filter on a Bluetooth receiver. The input, $\phi[n]$, that should be sent to the CP insertion block can be calculated by removing CPs in $\hat{\theta}[n]$.

2.2.5 QAM

The CP insertion block is immediately preceded by IFFT and QAM generator. Therefore, `BlueFi` first applies FFT to the reconstructed input to the CP block to obtain the frequency-domain samples that the QAM generator should generate.

Four possible modulation schemes (BPSK, QPSK, 16-QAM and 64-QAM) can be used in 802.11n to generate frequency-domain samples and a higher-order modulation scheme corresponds to a higher data rate. A higher-order modulation has more constellations and hence comparatively higher resolution in the frequency domain. However, even with 64-QAM, the resolution (8 levels or 3 bits in either the real or imaginary part) is very limited, so we must select each constellation carefully to minimize the error of quantizing the real or imaginary part to one of the 8 levels.

Owing to this limitation, it is hard to design an end-to-end algorithm that optimizes the reception performance. Specifically, the restriction of input assuming discrete values can be treated as an integer constraint. The reception performance can be measured by how close the phase of the reconstructed time-domain signal is to the original phase signal. The problem can thus be formulated as an integer-programming (IP) problem. Note that there is no simple formula relating the frequency-domain samples to the phases in the time domain. Obviously, an exact solution can be obtained by exhaustive search or branch-and-bound. However, the complexity of exhaustive search is $64^{52} = 2^{312}$ since we can control the samples on 52 frequencies (52 subcarriers for data in 802.11n). Even if we try to only optimize with samples at 8 subcarriers (corresponding to a bandwidth of $0.3125 \cdot 8$=2.5MHz), the complexity is $64^8 = 2^{48}$. Both are intractable on almost all computing platforms.

Therefore, `BlueFi` uses relaxation, a common practice for approximating the solution of an IP problem. In addition, we try to find the best fit for the time-domain waveform instead of the phase of this waveform since some analytical results can be derived. Suppose for a time-domain waveform $x[n]$, we want to find a least-square fit $\hat{x}[n]$ with the restriction that its frequency-domain counterpart, $\hat{X}[f]$, only assumes discrete values. (That is, $\hat{X}[f] \in \{a + bi \mid a \in \{\pm 1, \pm 3, \pm 5, \pm 7\}, b \in \{\pm 1, \pm 3, \pm 5, \pm 7\}\}$.) Since $\hat{x}[n]$ is the least-square fit, $\sum_n |x[n] - \hat{x}[n]|^2$ is minimized. Let $X[f] = FFT(x[n])$ and let $y[n] = x[n] - \hat{x}[n]$, then by Parseval's Theorem, $\sum_n |x[n] - \hat{x}[n]|^2 = \sum_n (y[n])^2 = \sum_f (Y[f])^2 = \sum_f |X[f] - \hat{X}[f]|^2$. Therefore, minimizing the time-domain residue is equivalent to minimizing the frequency-domain residue.

For any given $X[f]$, if we set $\hat{X}[f]$ to the constellation with the shortest Euclidean distance, then the objective function is minimized. Since only the phase of the time-domain waveform matters to a Bluetooth receiver, a scale factor A can be applied between the time-domain reference and the phase: $x[n] = A \cdot e^{i \cdot \phi[n]}$. We set the scale factor to $\frac{1}{5}$. This value is chosen such that if the energy of a Bluetooth waveform within one OFDM symbol is mainly concentrated on two subcarriers, each will have a magnitude of around 32 (=64/2) units, which is close to 35 ($=7 \cdot 5$). We tested using dynamic scale factors that further optimize the residue. The performance difference is negligible but the complexity is significantly higher as finding an optimal scale factor is still an IP problem. The process of selecting $\hat{X}[f]$ is illustrated in Fig. 2.4.

2.2.6 Pilots and Nulls

Not all subcarriers are modulated by the incoming data. Pilot subcarriers are modulated by known sequences whereas null subcarriers are always 0's. Since we cannot control these pilots and nulls, we solve the problem by frequency planning, leveraging the fact that we can switch WiFi channels and there are large overlaps between WiFi channels. For example, suppose we want to transmit on Bluetooth channel 38 (2426MHz), then this frequency is covered by WiFi channels 2, 3, 4 and 5, and corresponds to subcarriers 28.8, 12.8, -3.2 and -19.2, respectively. We can calculate its distance to any pilots or nulls and select the channel to keep the Bluetooth channel farthest away from pilots or nulls. In this example, we should use WiFi channel 3. Using channel 3, the closest pilot is 1.8125 (=5.8*0.3125) MHz away, which is significantly larger than half the bandwidth of Bluetooth signals.

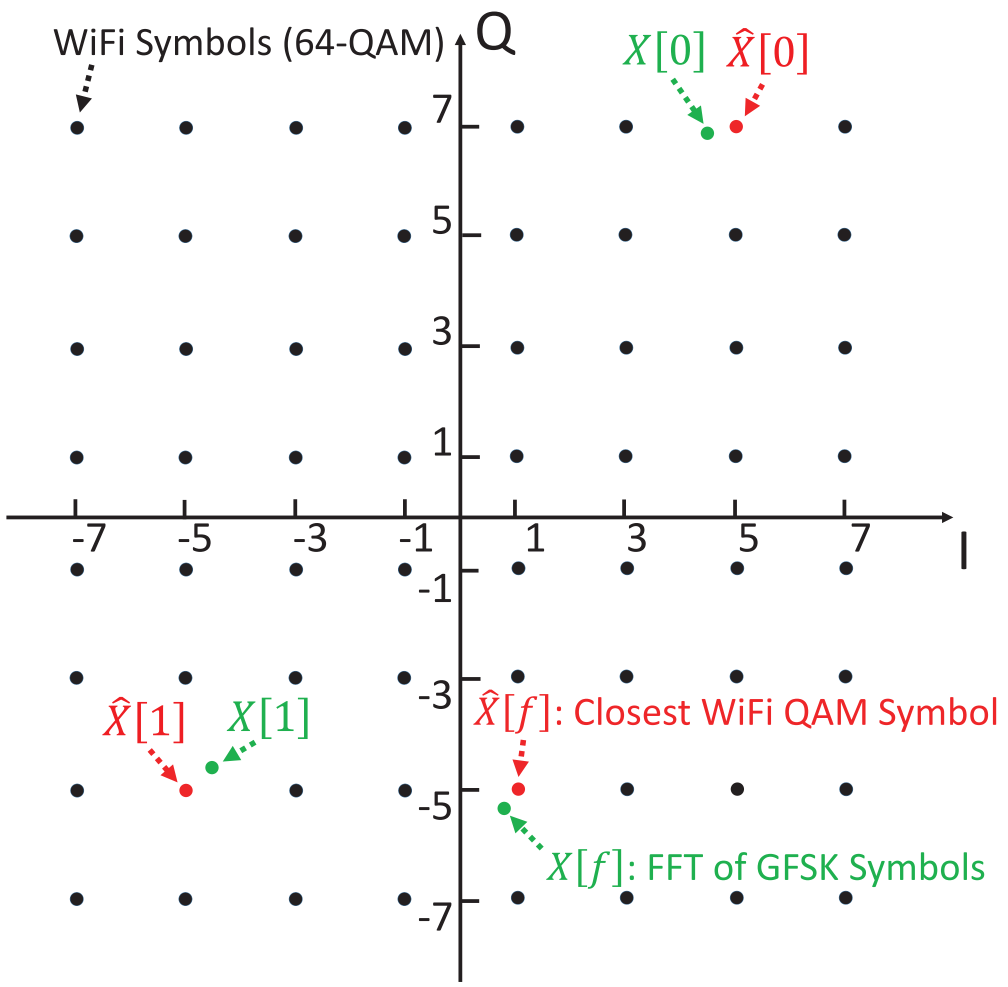


Figure 2.4: Selecting $\hat{X}[f]$ from $X[f]$. $X[f] = FFT(x[n])$. A scale factor A is applied ($x[n] = A \cdot e^{i \cdot \phi[n]}$) so that $X[f]$ is appropriately scaled w.r.t. origin.

2.2.7 FEC Coder

The FEC encoder adds redundancy to the bit-stream. Because of the redundancy in its output, an FEC encoder cannot generate arbitrary sequences. To reverse the operation of the encoder, we must build a decoder. We focus on convolutional codes since they are mandatory in 802.11n (as opposed to the optional LDPC codes). An FEC encoder can also be viewed as a decompressor whereas a decoder can be viewed as a lossy compressor. Consequently, when we try to reconstruct an output sequence from decoded bits, some of the bits will be different from the original sequence since information is lost when decoding the original sequence.

Convolutional codes can be optimally decoded by the Viterbi algorithm [31, 32]. Since we are not dealing with over-the-air signals that contain noise, we used hard decoding. The Viterbi algorithm uses dynamic programming to find the input bits corresponding to a sequence that has the least hamming distance (Euclidean distance for soft decoding) to the received sequence. For this decoder, we use the code rate of 5/6 as it has the minimal information loss in the decoding process.

In the conventional Viterbi algorithm, every bit-flip (except for the flip at punctured bits) has an equal weight and the algorithm finds an optimal survival path that minimizes the total weight. However, since Bluetooth signals only occupy part of the WiFi spectrum, bits on subcarriers corresponding to the main Bluetooth spectrum should have as few bit-flips as possible whereas the bit-flips on other subcarriers do not really matter. In addition, since bits are interleaved before being mapped to subcarriers, there will be no long runs of bits mapped to the same or nearby subcarriers since adjacent bits are mapped to subcarriers that are far apart. Consequently, it is possible to modify the Viterbi algorithm to further minimize the flips of bits that matter to Bluetooth reception. Specifically, we can assign higher weights to those important bits and the Viterbi algorithm will then find an optimal solution to reduce bit-flips. For example, in Table 2.1, we calculated the location the first few bits in an OFDM symbol will be mapped to. Assuming subcarriers 9 to 16 correspond to the main Bluetooth spectrum, we then assign the highest weight to bits on those subcarriers; a medium weight to those on 4 subcarriers immediately adjacent to subcarriers 9 to 16 on each side. The absolute value of these weights is not critical since the goal is to assign the priority of each bit. For example, the highest weight means that those bits will only flip if there is no alternative.

For apps that require real-time packet generation, we further simplify the decoding algorithm to significantly lower the complexity while guaranteeing no important bits to flip. For a real-time decoder, we use the code rate of 2/3 since it has the highest compression ratio, and hence we can reduce the length of the input bits required for an output sequence of given length. We also make several observations as follows. The bit inter-leaver in WiFi has an internal period of 13 and the same bit location in different cycles corresponds to the same or nearby subcarriers. So, important bits always appear in the same region in each cycle. We also found that the polynomials for the convolutional coder used in WiFi are chosen in such a way that we can design an algorithm to guarantee that at most 1/3 of bits will be flipped when we compare an arbitrary sequence with its reconstruction after decoding and encoding it with the code rate of 2/3. Specifically, we divide the original sequence into groups of 39 bits. For the first 13 bits, we pre-generate a lookup table of all possible 12-bit candidates that result in a given 9-bit pattern from bit 5 to 13. Because of the well-designed WiFi codebook, any 9-bit pattern has, and only has, eight 12-bit candidates and their first 3 bits are distinct. Note that in the normal process of continuous encoding, bit 0 to 13 are generated by feeding 9 bits into an encoder. We keep track of the last 3 bits of the decoded sequence we have so far (or use zeros for initialization). We select the candidate that has the same first 3 bits as these 3 bits and the remaining 9 bits are the decoded sequence for bit 0 to 13. These

12 bits together guarantee that bit 5 to 13 of the reconstructed sequence will not flip while the first 3 bits ensure that the solution for bit 0 to 13 is compatible with the sequence decoded in the last round. We use similar processes to decode bits 14–25 and bits 26–38. This solution guarantees that, after reconstruction/encoding, 2/3 of bits will not flip and bit-flips will only occur near the front for each 13-bit cycle. Using this algorithm, bit-flips can only happen on subcarriers -28 to -8, and hence we can use it for generating Bluetooth packets with a positive frequency shift, and it guarantees that important bits will never flip. For negative frequency shifts, we devise a similar algorithm so that bit-flips can only occur on subcarriers 8 to 28.

Table 2.1: Weight assignment for the Viterbi algorithm.

Bit	Mapped Location	Wt.	Bit	Mapped Location	Wt.
0	Subcarrier -28, bit 5	1	9	Subcarrier 12, bit 5	1000
1	Subcarrier -24, bit 3	1	10	Subcarrier 16, bit 3	1000
	⋮		11	Subcarrier 20, bit 4	100
7	Subcarrier 3, bit 3	1	12	Subcarrier 25, bit 5	1
8	Subcarrier 8, bit 4	100		⋮	

2.2.8 Scrambler

The feasibility of our solution depends on whether the mapping from the bit-stream to the IQ waveform is deterministic. The only operation in the WiFi Tx chain that might not be deterministic is the scrambling of bits as the standard suggests use of a "pseudorandom nonzero initial state." For testing and certification, however, the seed (i.e., the initial state of the scrambler) can usually be set to a constant by drivers, although public information on how to do it is very limited and not documented well. The datasheet and register map of almost all WiFi chips are not available without signing NDAs. We found that major vendors, such as Broadcom and Qualcomm, provide functions or register definitions in their drivers to set the scrambler seed to a constant. By capturing the radio signals, we also found that Realtek chips use fixed scrambler seeds, although the exact values are different for different chip generations (802.11n and 802.11ac).

Since the inverse of an XOR operation is simply the same XOR operation, we can obtain the de-scrambled bit-stream by applying the same scrambler with the same scrambler seed as that used in the WiFi chip.

2.3 Implementation

We have implemented `BlueFi` using Python and real, commercial off-the-shelf WiFi chips. We test the performance of using a GL-AR150 WiFi router, which is equipped with an (Qualcomm) Atheros AR9331 802.11n-compliant SoC and is pre-loaded with OpenWrt [33]. The AR9331 belongs to Atheros's widely-adopted ath79 product family. OpenWrt supports at least 272 routers with ath79 chips [34]. `BlueFi` does not use any OpenWrt-specific features. We use OpenWrt because its source code is available and we can modify the driver code (ath9k) for the ath79 chip.

We also test the performance of using a TP-Link T2U Nano WiFi NIC. At its core, T2U Nano uses the RTL8811AU chip from Realtek. The Realtek RTL88xx device family is popular among WiFi device makers and dominates the market of USB NICs. Although RTL8811AU supports 802.11ac, we did not use any of the 802.11ac modes. We chose this chip mainly because it is cheap and has better driver support in Linux.

The generation and transmission of a `BlueFi` packet starts in the user space. `BlueFi` first gathers Bluetooth payload. We use 30 bytes of data with 6 bytes of address as the payload. We use Python to implement the process described in Sec. 4.2. The Scipy library [35] is used for FFT computation. We also implement the modified Viterbi algorithm where the optimization takes the weight of each bit into consideration. The final results, in the form of WiFi packets, are sent to the WiFi hardware for transmission.

The required total number of bytes to be sent by WiFi hardware is in the range of a few thousand bytes, which is much smaller than the PSDU limits of 65,535 bytes defined in the WiFi physical layer (PHY) standard. We found that the Linux kernel typically fragments packets with the size exceeding the limit of an MPDU (2,304 bytes) or the MTU of Ethernet (1,500 bytes). Because of these limitations, `BlueFi` directly sends packets in the driver layer. `BlueFi` can support very long Bluetooth packets (even the longest, optional 5-time-slot packets) after driver modification. For AR9331, the transmission starts in the user space and packets are sent to the ath9k driver in the kernel space via netlink. A callback function will be invoked and set the transmit parameters such as MCS, SGI before invoking the normal transmit function in the driver. For RTL8811AU, we first remove the hard-coded limit of 2,304 bytes. (This does not affect normal WiFi traffic since Linux kernel fragments outgoing packets.) Packets are sent to the driver via a character driver interface. The driver then fills transmit parameters and sends the packets to hardware.

For the best performance, the value of the scrambler seed needs to be known. For Atheros chips, [14] suggests that similar to ath5k devices, the scrambler seed of earlier ath9k chips can be set to a constant of 1 by clearing the GEN_SCRAMBLER bit in the PHY_CTL register. However, we found that AR9331 uses an almost entirely different register map. We solved this problem by finding the new location of the register, which is not mentioned anywhere in the datasheet or the driver code. Alternatively, it is possible to determine the scrambler seed without setting registers since scrambler seeds are predictable (increment by 1 in Atheros's implementation) in most WiFi chips [36, 37]. Fixing the seed has no effect on normal WiFi operation and Realtek chips already use a constant by default. We find this constant (71 for RTL8811AU) by decoding the WiFi signals it sends.

2.4 Evaluation

2.4.1 Experimental Setup

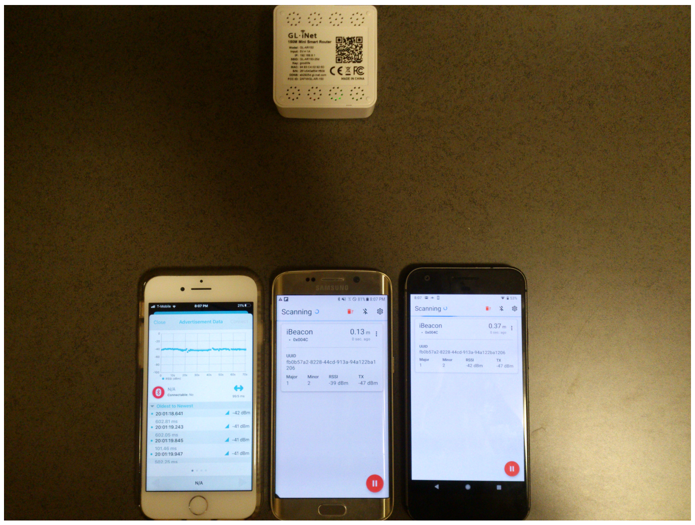
We use an iPhone, a Google Pixel and a Samsung S6 (Edge) as Bluetooth receivers. We use the nRF Connect app [38] on the iPhone and the Beacon Scanner [39] app on Android devices. We measured the signal strength under various conditions for 2 minutes, which is the default measuring duration of nRF Connect. For the `BlueFi` transmitter, the majority of tests are done on the GL-AR150 WiFi router as this represents the typical use-case (leveraging WiFi infrastructure for beacons) we envision. We can control (start/stop) or modify `BlueFi` packets remotely via SSH from either the Internet (e.g., cloud servers), local Ethernet or WiFi. To show that `BlueFi` is vendor-agnostic, we also test it on RTL8811AU. Both AR9331 and RTL8811AU can independently send `BlueFi` packets regardless of whether there is any connection to a station or AP or not.

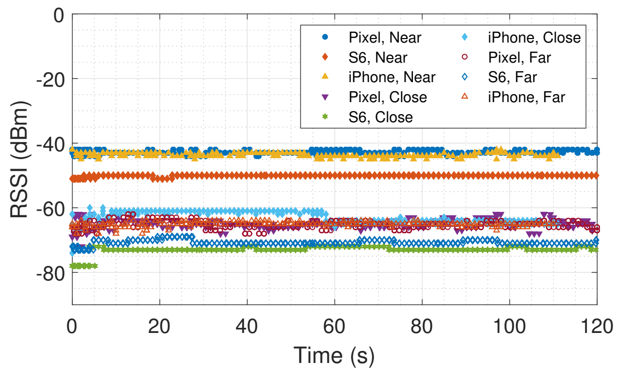
Since the 2.4GHz spectrum is very crowded and there are at least 2 other APs operating on the same WiFi channel in the test environment, we expect some interference typical of office environments. Except for Sec. 2.4.3, we use the default transmit power of AR9331 (18dBm) and RTL8811AU. We did not modify the firmware of RTL8811AU or the ART (Atheros Radio Test) partition of AR9331, which is required for regulatory compliance.

2.4.2 Performance vs. Distance

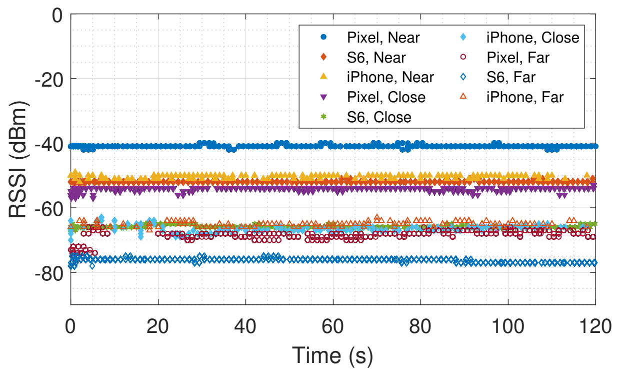
We place the phones under test near (~20cm), close (~1.5m), and far (4~5m) from a WiFi transmitter on which `BlueFi` runs, and collect the received signal strengths of packets (RSSI) reported by Bluetooth hardware.

Fig. 2.5b plots the results of using AR9331, showing that different smartphones can receive Bluetooth packets with consistent performance. Although the measuring duration of nRF Connect is 2 minutes, the iPhone's power-saving mechanism kicks in after approximately 110 seconds elapsed, and therefore iPhone's traces are typically 10 seconds short. We observe different RSSI levels on different phones placed at the same distance. The RSSI of S6 is generally 6~10dB less than the counterparts. This is most likely due to the fact that the underlying Bluetooth chips


(a) Experimental setup



(b) AR9331



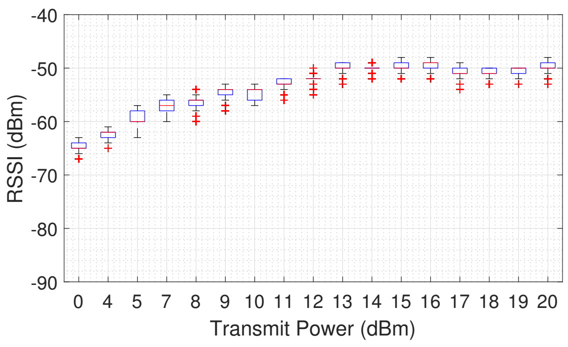
(c) RTL8811AU

Figure 2.5: Evaluation of `BlueFi`

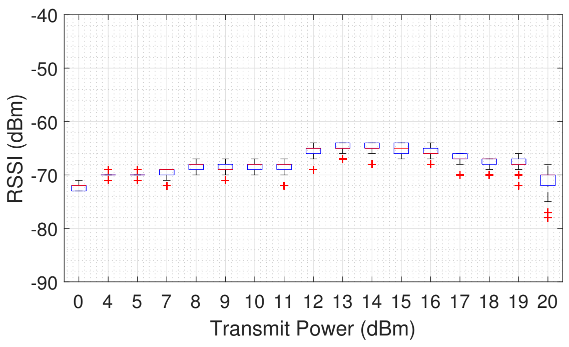
have different sensitivity. We observe the same behavior even when dedicated Bluetooth hardware is used (Sec. 2.4.4). We found (by transmitting `BlueFi` signals using USRP) that smartphones can pick up Bluetooth signals of as low as -90 to -100 dBm. Therefore, the margin is around 10~20dB, which is theoretically equivalent to 3~10x in range. Fig. 2.5c shows the results of using RTL8811AU under the same condition. Compared to Fig. 2.5b, there are some variations in terms of RSSI, but devices can still steadily receive Bluetooth packets using `BlueFi`.

2.4.3 Performance vs. WiFi Tx Power

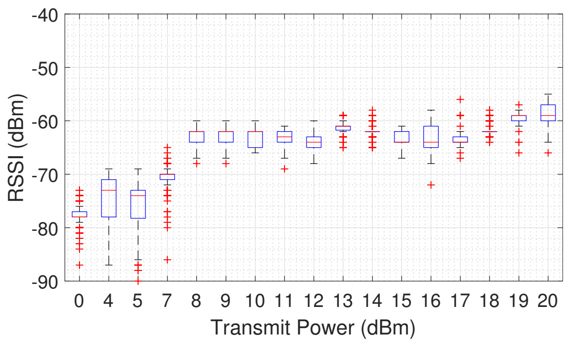
OpenWrt provides a convenient way to control the transmit power, and hence we also measure the received signal strength with respect to different transmit power levels. We placed the phones 1.5m away from the WiFi router. Fig. 2.6 shows the results. The RSSI is very high on the Pixel and



(a) Pixel



(b) S6



(c) iPhone

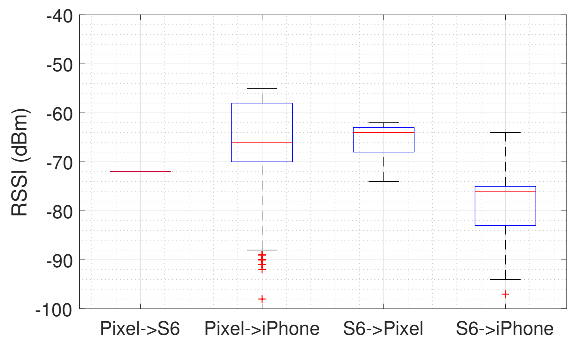
Figure 2.6: Performance vs. Transmit power

gradually decreases with the transmit power. Even at the router's lowest transmit power of 0dBm (=1mW), the RSSI is still significantly higher than -90dBm. In contrast, such a trend is not so obvious on S6. Its RSSI values may be more sensitive to the waveform impairments than to the absolute power. Although the iPhone's RSSI shows a similar trend as Pixel's, it fluctuates more, which may be the result of multipaths or interference from the environment.

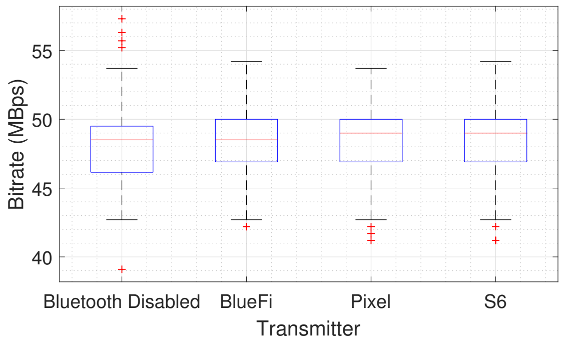
2.4.4 Comparison with Bluetooth Hardware

To compare `BlueFi` with dedicated Bluetooth hardware, we also measure the performance of using conventional Bluetooth transmitters. Beacon packets are sent using the Beacon Simulator app [40] on Android. We set the Bluetooth Tx power to high and set the broadcasting frequency to 10Hz. All other conditions are exactly the same as those in Sec. 2.4.3.

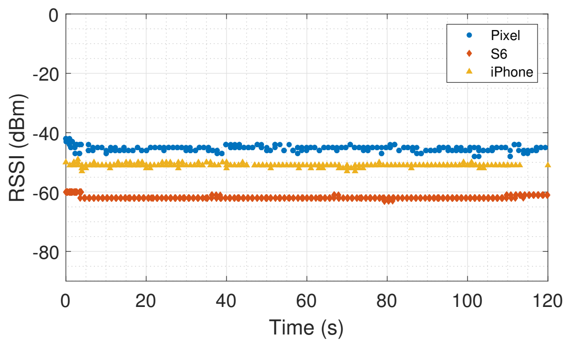
The results are plotted in Fig. 2.7a, where the first two and the last two columns represent using Pixel and S6 as the transmitter, respectively. Note that the same fluctuating behavior on iPhone can be observed here, and hence we conclude that the transmitter design does not cause such behavior in Sec. 2.4.3. We can also see that the RSSI is lower on S6 than on iPhone under the same condition.



(a) Using dedicated Bluetooth hardware



(b) WiFi throughput measurements



(c) RSSI with background WiFi traffic

Figure 2.7: Comparison with dedicated hardware and effect of background WiFi traffic

Since conditions are exactly the same as those in Sec. 2.4.3, we can directly compare Figs. 2.6 and 2.7a. At the Tx power of 8dBm, the performance of `BlueFi` is found comparable to those of using a dedicated Bluetooth chip. Therefore, with WiFi chips that nominally come with a default Tx power of 18dBm, one could expect better performance with `BlueFi`.

2.4.5 Effect on Concurrent WiFi Traffic

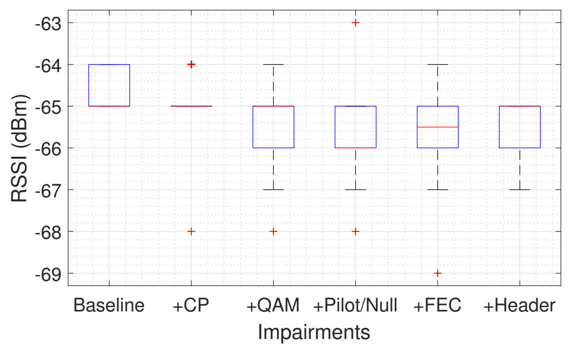
We also evaluate the effect of `BlueFi` on concurrent WiFi traffic. For this, we use iPerf3 [41], a standard tool for benchmarking network throughput. We install iperf3 on the WiFi router and configure it as an iPerf3 server. We then connect a Ubuntu laptop to the router over WiFi and run an iPerf3 client on the laptop. We make the throughput measurements, reported by iPerf3 every second, for 120s.

As shown in Fig. 2.7b, we test four scenarios. We establish the baseline by measuring the throughput without any Bluetooth transmission. Then, we run `BlueFi` on the same WiFi router that also runs the iPerf3 server simultaneously. For comparison, we also test the throughput when we use dedicated Bluetooth hardware on Pixel and S6 instead.

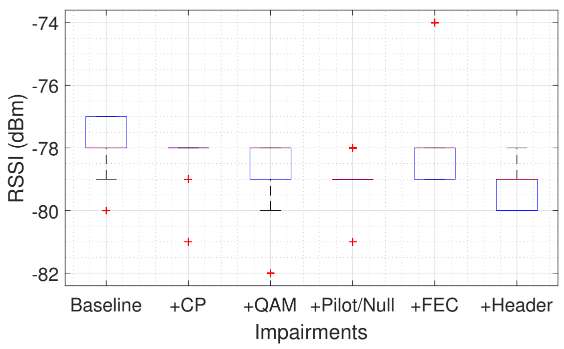
The figure shows the throughput difference in each scenario to be very small. Although the baseline has the lowest median throughput, it has the highest average throughput (UL: 48.8Mbps, DL: 48.7Mbps). With `BlueFi`, the average throughputs are 47.8Mbps UL and 47.7Mbps DL, which are only 1Mbps lower than the baseline. For comparison, using Pixel and S6 yields average throughputs of 48.6/48.6 and 48.4/48.3Mbps, respectively. Note the contention for airtime is not the only factor that limits the throughput. Since we send `BlueFi` packets by the single-core microcontroller in the AR9331, it consumes a tiny amount of the CPU and the memory (0% of the CPU and 1% of the virtual memory), which most likely contributes to the reduction in throughput. This slight reduction in throughput may be a worthy trade-off for WiFi infrastructures to support various Bluetooth apps.

Background WiFi traffic has little effect on `BlueFi` packets. As Fig. 2.7c shows, all phones can still steadily receive Bluetooth packets even when we saturate the WiFi channel. The WiFi traffic only causes the Pixel's RSSI to fluctuate by a small amount. As usual, the power-saving mechanism causes anomalies in the iPhone's trace near the end.

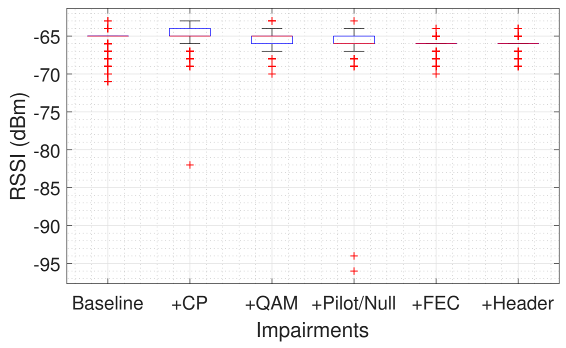
2.4.6 Effect of Each Impairment



(a) Pixel



(b) S6



(c) iPhone

Figure 2.8: Effect of each impairment

To see the effect of the impairment caused by each block in a WiFi transmitter, we generate various waveforms and transmit them using USRP.

In Fig. 2.8, we generate a standard FSK waveform as the baseline and cumulatively apply each impairment in each column. The last column represents sending a complete 802.11n PSDU. As the figure shows, each impairment degrades signal quality by approximately 1dB and the overall degradation is around 2dB. Note that BlueFi reverses the WiFi operation block-by-block and does not aim to globally optimize the process. Therefore, some bit-flips, caused by adding the FEC and the header, may slightly enhance the signal quality.

2.4.7 Bluetooth Audio

Other apps can also use BlueFi as their Bluetooth physical and link layers. We demonstrate this by building an audio transmitter with A2DP. For general apps, Bluetooth devices transmit packets at the start of predetermined time slots and hops to different frequencies for different time slots. Therefore, BlueFi must follow a strict frequency hopping sequence and transmit the generated packets within the targeted time slot.

On the other hand, WiFi hardware has a few limitations, making it harder to follow the frequency hopping sequence. Bluetooth hops to a different frequency every 1.25ms and WiFi chips are not designed to constantly hop at such a pace. Also, Bluetooth hops randomly across 79 channels, spanning 79MHz, which is much larger than the bandwidth of a single 802.11n channel. Finally, the process of generating Bluetooth GFSK bits from a higher-layer payload depends on the Bluetooth clock value in the transmission time slot. Thus, packets need to be generated shortly before the transmission and then released precisely at the desired time slot.

We use several strategies to overcome these limitations. Instead of constantly changing the physical WiFi channels, we only use a single WiFi channel and implement frequency hopping by using different subcarriers within a WiFi channel. Since one WiFi channel only has a bandwidth of 20MHz, it cannot cover the 79-channel hopping sequence. We solve this by using Bluetooth's adaptive frequency hopping (AFH) feature to only use the 20 channels corresponding to the single WiFi channel we select. AFH simply remaps the channel outside of these channels to one of the 20 channels and has no effect on the theoretical throughput. AFH is available on all Bluetooth devices we tested. BlueFi thus covers all types of Bluetooth channels, since data channels can be specified with AFH and one advertisement channel is well-covered by WiFi channel 3. Finally, we use the high-resolution timer [42] in the Linux kernel to precisely schedule the transmission of each BlueFi packet.

We run BlueFi locally on an i5-3210M laptop and transmit packets using RTL8811AU. We test BlueFi with Sony SBH20 Bluetooth headphones and also quantitatively measure the performance using FTS4BT [43] from Frontline, a standard tool used by industry leaders like CSR and Broadcom. Note that the tool uses CSR's widely-adopted BlueCore chips as the underlying hardware, and hence the results are representative of reception using off-the-shelf Bluetooth chips. We report the FTS4BT's PER and throughput measurements.

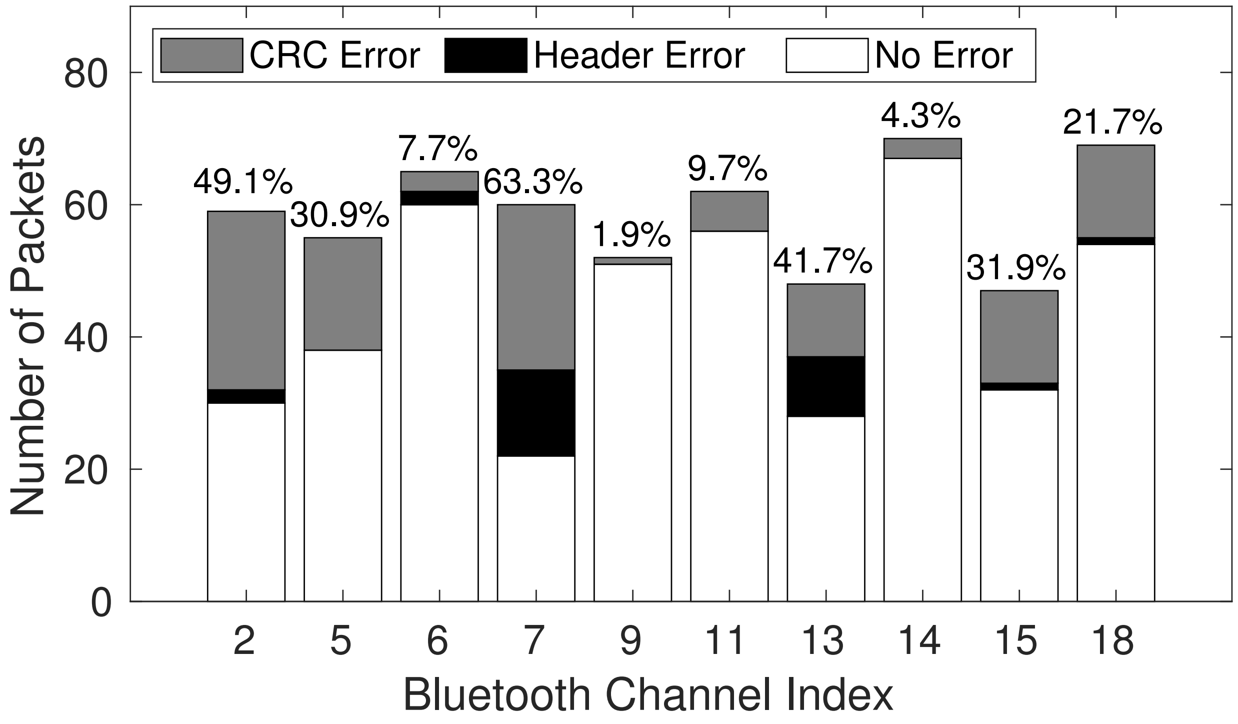


Figure 2.9: PER with single-slot packets

Due to nulls and pilots, the performance of transmission on each Bluetooth channel is different within a single WiFi channel. For example, Fig. 2.9 shows the packet error rate (PER) reported by FTS4BT of `BlueFi` transmitting single-slot packets on 10 different channels. PER is shown to be as low as 1.9% on good channels whereas it is much higher for channels adjacent to WiFi pilots. The measured throughput for the upper layer is 37.5kbps, since single-slot packets have significant overhead and we only use half of the channels. Note that Bluetooth's frequency hopping algorithm does not guarantee uniform assignment from time slots to channels.

The throughput and goodput are increased vastly by using multi-slot packets, which incur much less overhead. More importantly, since the frequency will remain the same for multiple slots, we effectively cover nearly 2x or 3x the number of time slots with the same number of Bluetooth channels.

To keep PER low for multi-slot packets, we select 3 best channels to transmit audio packets. We re-route PulseAudio and send A2DP audio streams to `BlueFi`, which then allocates a time slot and calculates its hopping frequency. If it matches the channels we use, `BlueFi` additionally

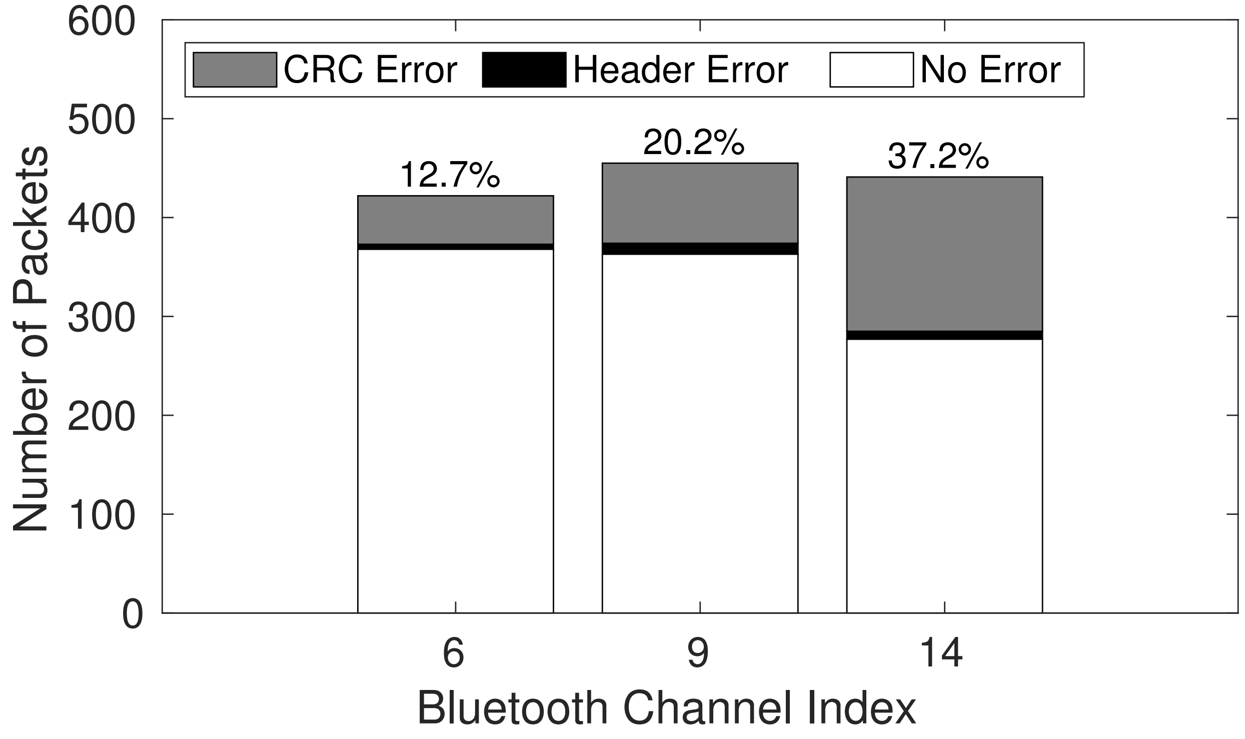


Figure 2.10: PER with 5-slot packets (audio)

allocates 4 subsequent time slots for an audio packet. The clock value of the allocated slots is used to convert the audio stream, which is a standard L2CAP stream, into Bluetooth GFSK bits. L2CAP is a universal layer on which almost all Bluetooth apps rely. With these bits and a desired frequency offset, `BlueFi` then performs various signal processing tasks and generates a WiFi packet. The packet is marked with the clock value and sent to the driver. Inside the driver, we construct a high-resolution timer to schedule the packet to be transmitted at the precise instant specified by its clock value.

We are able to use `BlueFi` to stream real-time stereo audio to Sony SBH20 Bluetooth headphones. In addition, we use FTS4BT to measure the throughput and PER. We did not modify the Bluetooth headphones in any way. Without any firmware modification, a connection token is needed in order for the headphones to accept incoming audio data, and we first create the token by making a connection with Bluetooth hardware. Once the connection token is created, `BlueFi` can stream audio on its own. Fig. 2.10 shows the PER when streaming audio. Longer packets increase PER. The overall PER is 23% and the upper-layer throughput is measured at 122.5kbps, corresponding to a goodput of 93.4kbps. Throughput and goodput can be increased, at the expense of higher PER, by filling unoccupied time slots with single- or multi-slot packets. Conversely, PER can be drastically decreased by using fewer channels or shorter packets. We leave further optimizations as future work.

We use the SBC (sub-band coding) codec as it is the mandatory and the only codec supported by Sony SBH20. Advanced codec shouldn't cause any difficulty working with `BlueFi` since `BlueFi`, like any other BT radio and PHY, is only responsible for sending 1's and 0's and upper layers are oblivious to how the radio and PHY are actually implemented.

2.4.8 Execution Time and Complexity

Our first prototype uses Python and generating a single packet using Python takes around 2.60s, which includes IQ generation (0.01s), FFT and QAM (0.18s), FEC decoder (2.39s), scrambler (¡0.01s) and file operation (0.01s). We drastically improved the runtime by porting `BlueFi` to C. The C version produces identical outputs as the Python prototype and generating a single packet takes 46.88ms, more than 55x faster. Almost 100% of the execution time is spent on the FEC decoder. The Viterbi algorithm uses dynamic programming and has a pseudo-polynomial runtime of $O(Tn^2)$ where T is the length of a sequence and n is the number of states. The relatively long runtime when applying the algorithm is the result of long sequences and a high (64) number of states.

By replacing the Viterbi algorithm with our real-time decoder (with a complexity of $O(T)$) and by using the FFTW [44] library, the execution time of `BlueFi` can be reduced by approximately another 50x. On an old (Ivy Bridge) i5 laptop, the execution time is around 0.954ms (with the standard deviation of 0.122ms), which is less than the minimal interval (1.25ms) of two consecutive Bluetooth packets. Therefore, `BlueFi` can run in real time and the delay incurred is around 0.954ms. The timeliness is important since Bluetooth payloads are scrambled with the clock value at the time of transmission and real-time generation greatly simplifies the design. More importantly, the throughput is not limited by the computation. We expect the execution time to be even lower if newer hardware, SIMD, hardware acceleration or multithreading is used.

For applications where devices use wall power, such as APs and desktops, the power consumption is less of a concern. For mobile devices, instead of processing the signals locally, edge or cloud servers can be used to offload the computation. When run locally, the signal processing draws moderate power. Using PowerTOP [45], we measure the power consumption of continuously generating `BlueFi` packets for every possible Bluetooth time slot in real time on an i5-1135g7 laptop. The steady-state power consumption is 1.11W, which represents the case of maximum throughput (100% duty cycle). This power consumption scales proportionally with the duty cycle.

2.5 Discussion

2.5.1 Different 802.11 Generations

Although supporting 2.4GHz band is not strictly required for 802.11ac, we found that most 802.11ac devices do support the dual band operation since operating at only the 5GHz band makes the device incompatible with 802.11b, g and 2.4GHz 802.11n devices. 802.11ac supports 256-QAM and some chips even support 1024-QAM. Higher-order modulation means higher resolution in the frequency domain, and therefore we expect less quantization error in the QAM process. In 802.11ax, 1024-QAM becomes mandatory. New modes in 802.11ax use longer guard intervals, and thus they are not particularly useful to `BlueFi`.

It is possible to modify `BlueFi` so as to work on 802.11g, the predecessor of 802.11n, hardware as they are very similar. Both standards use OFDM and the maximum allowable PSDU length for 802.11a/g is 4,095 bytes, which is still sufficient for containing Bluetooth packets. However, the main challenge is that we cannot use SGI. We found a way to solve the CP insertion problem and the signal can be picked up by Bluetooth receivers, but the performance is spotty. Since 802.11g was standardized nearly 20 years ago, we feel that it is too old and most existing WiFi hardware uses newer standards, such as 802.11n/ac/ax. Therefore, we opted not to support 802.11g hardware.

2.5.2 Fine-grain Cooperation and Scheduling

`BlueFi` enables the possibility of fine-grain cooperation and scheduling between WiFi and Bluetooth. Previously, the solutions for Bluetooth and WiFi coexistence were complex. For example, we found that the codes in the RTL8811AU driver for dealing with Bluetooth coexistence are nearly 6000 lines long. By converging two standards on one hardware, `BlueFi` simplifies the coexistence problem by eliminating the inter-chip messaging and delay. We also note that conventional WiFi and Bluetooth cooperation works by disabling WiFi during Bluetooth transmission. Therefore, from the transmitter's perspective, using `BlueFi` does not sacrifice the amount of information transmitted over the air within the same amount of time, since the standard cooperation mechanism already forgoes the whole WiFi spectrum during Bluetooth transmission.

In the current implementation, `BlueFi` packets are assigned to queues just like typical WiFi packets. It is possible to further optimize the priority assignments of both WiFi and Bluetooth packets so that time-sensitive packets, such as audio data, are given priority regardless of whether they are sent over WiFi or Bluetooth.

2.6 Related Work

Shadow Wi-Fi [46] allows Broadcom's 802.11ac chips to transmit arbitrary waveforms. However, its method is vendor-specific, non-real-time and would need hardware recertification. Several cross-technology communication (CTC) systems [3, 4, 5, 6] modulate the power of a transmitter and a receiver senses the amplitude to recover embedded information. The use of this basic modulation leads to very low bit rates (all of which are less than 700bps) and requires modifications on both ends.

OfdmFi [14, 15, 16] enables the transmission of LTE-U waveforms using WiFi's OFDM hardware. Unfortunately, it is not applicable to Bluetooth since LTE uses OFDM whereas Bluetooth uses GFSK, which is completely different from OFDM. ULTRON [47] emulates WiFi CTS frames using LTE-U waveforms. Interscatter [48] uses WiFi to transmit amplitude-modulated waveforms for RFID communication. WEBee [17] enables WiFi-to-Zigbee communication. As described in Sec. 3.1.3 in [17], it relies on the error correction from Zigbee's direct sequence spread spectrum, which is not available on any Bluetooth systems. Bluetooth also has 4x higher bit rates, making it more challenging. Zigbee uses PSK and Bluetooth requires a completely different waveform and symbol boundary design. WEBee requires hundreds of big (288×216) matrix inversions for every Zigbee packet, which is computationally expensive. Timeliness is important for BT data transmission since its waveform is time-variant, even for the same payload. Bluetooth uses time slots and frequency hopping, a communication pattern very different from Zigbee's. Finally, Bluetooth is much more widely deployed and covers unique apps, such as location beacons and audio. Based on a similar principle, LTE2B [23] focuses on LTE to Zigbee communication. LongBee [18] extends the range of WEBee. TwinBee [19] applies additional channel coding on top of WEBee to improve reliability. WIDE [20] is also similar to WEBee but uses a different pulse-shaping waveform and uses USRP as the transmitter. A recent CTC work [49] explores the communication between USRP-emulated WiFi transceivers and modified Bluetooth devices and mainly focuses on Bluetooth-to-WiFi communication. There are also several critical differences in WiFi-to-Bluetooth communication. First, the prior work strictly requires modification of firmware on each Bluetooth device in order to implement the decoding of a two-layer error correction algorithm, which first drops $\frac{1}{4}$ Bluetooth bits and then decodes the remaining $\frac{3}{4}$ bits with the Hamming(7,4)-code. Our system directly overcomes the impairments introduced by WiFi's signal processing and does not rely on such error correction algorithms. Therefore, our system directly works with commodity Bluetooth devices that are not modified at all. Using unmodified Bluetooth devices is highly preferable since most users do not have the tools for firmware updates and most device vendors do not share their firmware source codes. Furthermore, one WiFi device may interface with multiple Bluetooth devices (e.g., using APs as Bluetooth beacons) and requiring modifications on every single Bluetooth device severely limits the use cases. We also note that employing two error correction algorithms significantly ($\frac{3}{4} \cdot \frac{4}{7} = \frac{3}{7}$) decreases the throughput. Second, our system is designed and shown to work with real, widely-deployed WiFi chips, not just with SDR equipment. As we show in Sec. 4.2, COTS WiFi chips behave differently from SDR devices, notably in terms of OFDM symbol filtering and the bit-stream scrambler. From experiments, we found that the differences are so critical that a design could work perfectly on SDR devices but fail to work on COTS WiFi chips at all. Finally, we design and demonstrate practical, real-world applications, such as Bluetooth beacons and Bluetooth Audio, running on our system in real time, and not just sending physical-layer packets.

Recitation [50] examines implementations of WiFi to predict bit-prone locations. Several 802.11 security studies [51, 36, 37] found that the scrambler seeds in most 802.11p or 802.11n/ac chips are predictable (constant, using arithmetic sequences or selecting from a few values).

2.7 Conclusion

We have presented a novel system, called `BlueFi`, that transmits legitimate Bluetooth packets using commercial off-the-shelf WiFi hardware. `BlueFi` overcomes all signal impairments and enables the signals to be received by unmodified Bluetooth devices. By re-purposing existing WiFi hardware, `BlueFi` broadens the coverage of Bluetooth and enables the use of Bluetooth functions in WiFi-only environments or with WiFi-only devices. `BlueFi` can be controlled from the cloud, and its convergence of the underlying hardware simplifies the cooperation between WiFi and Bluetooth. We have evaluated `BlueFi` on real, widely-adopted WiFi chips and shown that it supports real-world and real-time Bluetooth apps. We believe that `BlueFi` will accelerate the adoption of rich and valuable Bluetooth frameworks and applications (such as beacons and audio streaming) already developed using omnipresent WiFi devices, and will help tens of billions of WiFi devices communicate with tens of billions of Bluetooth devices.

CHAPTER 3

FLEW: Fully Emulated WiFi

3.1 Introduction

WiFi is the *de facto* standard for tens of billions of devices [26] to access the Internet. Designed to support all types of Internet applications, WiFi chips require a considerable amount of DSP circuitry for processing various WiFi waveforms, including high-throughput waveforms, leading to physically larger chips, higher energy consumption and higher chip costs.

Typical IoT applications only require a low data rate but energy consumption and chip size/cost are of paramount importance. Owing to these requirements, many IoT devices are built with FSK chips, also commonly known as *2.4 GHz proprietary wireless chips*. FSK chips are also the hardware of Bluetooth/BLE. In fact, many Bluetooth/BLE chips are essentially FSK chips with a Bluetooth software stack. FSK is arguably the simplest modulation scheme that offers decent enough throughput and noise immunity. Using FM circuits to transmit/receive digital waveforms, FSK chips are extremely energy-efficient and low-cost as well as occupy less board-space than WiFi chips. For example, we find the smallest WiFi chip available is Silabs' WF200 [52], whereas the smallest FSK chip available is TI's CC2500 [53]. Table 3.1 shows a comparison between these two chips. Although they differ in many ways [1], FSK chips are, in general, much smaller and cheaper, and consume less power than WiFi chips.

However, using protocols other than the ubiquitous WiFi means that IoT devices cannot use the WiFi infrastructure for Internet connectivity. Instead, they must rely on additional IoT gateways to relay the data to/from, and access the Internet indirectly. This gateway reliance implies that, to use any IoT device (with FSK or protocols other than WiFi), their corresponding IoT gateways must be installed in the environment. This hampers the adoption of IoT since using these IoT devices

[1]WF200 is much newer than CC2500 and therefore allows lower supply voltage (1.8V) for baseband and Rx circuits compared to CC2500 (3.0V). CC2500 still consumes less power after factoring in this difference. Newer FSK chips have even lower power consumption and can use lower supply voltage. For example, CC2650 [2] allows 1.8V operation and consumes only 6.1mA (Tx, BB) and 5.9mA (Rx) at 3.0V. An end-to-end power measurement with the same power condition is present in Sec. 5.3.7.

Table 3.1: Comparison of WiFi and FSK chips.

	Tech.	Package	Tx current (mA)	Rx current (mA)	Price (USD)
WF200	WiFi	QFN32	108 (PA) + 44.6 (BB)	41.6	3.28
CC2500	FSK	QFN20	100 (PA) + 21.5 (BB)	19.6	1.18

requires not only buying the devices but also investing/deploying/managing the IoT gateways. This ”gateway problem” [54] is worsened by the non-existence of a universal protocol that dominates the market, and hence different IoT devices may require separate gateways.

This poses an important question: *What if we can still use the small, simple, low-power FSK chips on the IoT devices, but we somehow allow FSK chips to directly communicate with WiFi infrastructures?* If such communication is possible/enabled, these FSK IoT devices can leverage existing WiFi infrastructures and eliminate the need for IoT gateways. They can stay connected wherever there is WiFi coverage!

To answer this question, we propose `FLEW`, which turns FSK chips into WiFi chips and enables them to directly communicate with unmodified WiFi APs.

Although several CTC (cross technology communication) efforts have explored communications *from* WiFi devices, `FLEW` represents a very different design philosophy and targets different use-cases. In particular, prior WiFi CTC work is *WiFi-centric* whereas `FLEW` is *FSK-centric*. Specifically, prior work enables one-way communication from modified WiFi APs/devices to unmodified FSK devices (e.g., Bluetooth), whereas `FLEW` focuses on enabling **bi-directional** communication between unmodified WiFi APs/devices and modified FSK devices. `FLEW` complements existing CTC work well by covering scenarios where users are not permitted to modify the firmware of WiFi APs/devices (e.g., WiFi APs in public or enterprises), or where one FSK device may connect to many APs (e.g., roaming) arbitrarily without needing to modify the firmware of every single AP. Furthermore, many IoT applications have uplink (i.e., FSK to WiFi) traffic, which is not possible with the prior WiFi-to-FSK work.

Even though several CTC studies have demonstrated the communication from WiFi devices to Bluetooth devices, their WiFi-to-FSK solutions are still not directly applicable because of more stringent system requirements. In particular, prior work requires modification of WiFi transmitters to generate FSK waveforms. However, since the goal of `FLEW` is for FSK chips to directly communicate with unmodified WiFi devices, it must be able to decode *any* standard WiFi packets at one data rate at least; not just WiFi packets with a magic payload (a payload that results in FSK-look-alike waveforms). To this end, we show how FSK hardware can be used to receive *any* WiFi DSSS waveforms. The difference can also be explained with the waveforms transmitted by the WiFi devices. Prior work modifies WiFi devices to transmit FSK-look-alike waveforms. In contrast, under FLEW, WiFi devices transmit standard 802.11b DSSS waveforms, like the conventional 802.11b WiFi operations, thus allowing the use of *unmodified* WiFi devices.

Furthermore, we show how FSK hardware can be effectively used to transmit 802.11b waveforms, thus enabling bi-directional communication between FSK and unmodified WiFi devices. FSK-to-WiFi communication is of great significance because any useful WiFi operation requires bi-directional communication at the physical layer. Even with a unilateral transport layer traffic, the physical layer requires the transmission of ACK packets in the opposite direction. Bi-directional communication is also needed for a client to join a WiFi network.

The different philosophy of FLEW also leads to different trade-offs. FLEW tries to modify FSK devices so that FSK devices work like WiFi devices as much as possible, whereas prior CTC work tries to modify WiFi devices so that they work like FSK/Bluetooth devices as much as possible. For this purpose, we utilize the underlying FSK hardware, instead of the full BLE/Bluetooth stack, to ensure a FLEW device behaves as close to a WiFi device as possible, since a full BLE/Bluetooth operation imposes unnecessary software limits. To ensure maximum compatibility with different unmodified WiFi devices, a FLEW terminal is designed to directly appear like a WiFi device, not a WiFi and Bluetooth device, during FLEW operation. This does not imply that new hardware is needed, however, as we have shown that FLEW can be implemented on existing FSK devices and chips. While prior CTC works are compliant with both Bluetooth and WiFi waveforms, they cannot support full operations of either standards. For example, the (COTS) WiFi chips in prior CTC work cannot receive Bluetooth packets and the Bluetooth chips cannot receive arbitrary WiFi packets. Although FLEW directly uses conventional WiFi waveforms and therefore does not have the "dual-compliance" during FLEW operations, we feel that this is a worthy trade-off because using conventional WiFi waveforms achieves the goal of enabling full WiFi operations and is the only way to ensure maximum compatibility with unmodified WiFi devices.

On the technical side, FLEW leverages the insights that a) at its core, WiFi (802.11b DSSS) encodes the information in the form of PSK (Phase Shift Keying), b) PSK modulation is similar to, and therefore can be demodulated by FSK receivers with a frequency shift, and c) PSK with DSSS signals can be transmitted by directly connecting digital waveforms to the mixer. These high-level principles are relatively simple, but are extremely powerful, and can bridge the gap between DSSS and FSK modulations.

In contrast to the conventional IoT topology where gateways and devices employ similar radio circuitry and chips, the hardware of WiFi APs and FSK devices in FLEW is highly asymmetrical. This asymmetry provides an opportunity to use simple and energy-efficient FSK chips while still providing good performance by leveraging powerful PAs and LNAs in WiFi APs. In addition, WiFi's DSSS modulation at 1Mbps has a higher coding gain than Zigbee or Bluetooth, and is intrinsically robust. Finally, to support higher data rates in newer 802.11 standards, many APs come with multiple antennas and advanced MIMO signal processing can further enhance the performance in both directions. Specifically, WiFi APs are allowed to transmit at high power and some APs support transmit beamforming for 802.11b [55], which enhances signal strength and overall mixed-client throughput. Also, many APs use diversity or MIMO processing (e.g., RAKE or MRC for 802.11b) to boost reception performance. For example, for 1Mbps, modern WiFi APs has a sensitivity as low as -102dBm [56], which outperforms the latest Zigbee offerings from TI (-100dBm [57]) and Microchip/Atmel (-101dBm [58]) even though WiFi is 4x faster than Zigbee (250kbps). Compared to Bluetooth/BLE chips, the difference is even greater (TI: -97dBm [59], Qualcomm/CSR: -95dBm [60], Broadcom/Cypress: -96dBm [61]).

Without WiFi encryption, `FLEW` is as secure as existing FSK protocols. If an existing protocol encrypts the payload, `FLEW` can simply transmit the encrypted payload over open WiFi networks. On the other hand, since it allows devices to directly communicate via WiFi, `FLEW` can provide stronger, enterprise-grade security protection by directly using the tried-and-true WiFi security framework on which billions of devices currently rely.

We implement `FLEW` with COTS FSK chips. With `FLEW`, these FSK chips are emulated as WiFi chips and can communicate with conventional, unmodified WiFi devices/APs. We extensively evaluate the performance of `FLEW` with multiple WiFi devices equipped with many different widely-adopted chips from all major WiFi chip-makers. We note that FSK hardware is the foundation of Classic Bluetooth and BLE (Bluetooth Low Energy). Therefore, with `FLEW`, it is possible to simultaneously support Bluetooth, BLE and WiFi using a single FSK chip!

`FLEW` enables connectivity and use-cases that were previously deemed impossible. `FLEW` allows IoT devices to directly connect to already-deployed WiFi APs and eliminates IoT gateways altogether. Alternatively, since FSK chips are cheaper, smaller and more energy-efficient than WiFi chips, `FLEW` can provide general Internet access for mobile devices where cost, area and/or power are of great importance. As an example, we showcase using `FLEW` for steaming high-quality music or streaming 720p YouTube videos in real time.

In `FLEW`, we focus on transmitting/receiving data at 1Mbps, which is on par with BLE 4 and 4x faster than Zigbee, and is sufficient for IoT operations. For WiFi, the 1Mbps data rate has a special significance. 1Mbps has the most robust performance among all possible WiFi modulations and many APs use 1Mbps for management (beacon, association, authentication, etc.) frames regardless of the data rates of data frames. In a multi-rate environment (which is almost always the case for typical WiFi networks), the transmit data rate is controlled by the rate adaptation algorithm (RAA), which will reduce the transmit data rates (i.e., use more robust modulation) if transmitted packets are not acknowledged. APs will try 1Mbps modulation if transmitting with higher data rates is unsuccessful. Therefore, implementing 1Mbps ensures that the WiFi-FSK connection will converge to a steady state using 1Mbps. If only a higher data rate is supported instead, then a connection may not be able to be established because of the packet loss of management frames. In addition, even if the higher data rate is negotiated, any transient behavior in the network may cause two devices to diverge from the agreed-on data rate and thus cause disconnection.

For the WiFi AP, a `FLEW` terminal will appear as a device that needs the most robust modulation and only 1Mbps modulation can get through. Such a scenario can legitimately happen with a conventional WiFi terminal (e.g., with a weak signal or with strong interference). Therefore, rate adaptation algorithms should always support `FLEW` operations regardless of their actual implementation.

The techniques used in `FLEW` may also help develop future low-power WiFi transceivers. For example, we show that instead of using a full-blown multi-rate PSK receiver with complicated phase synchronization, the relatively simple FM/FSK demodulator can be used to demodulate WiFi waveforms at 1Mbps very effectively. Since 1Mbps is frequently used to transmit management frames, the receiver can be completely turned off and only use low-power FM circuits to monitor the management traffic. The FM circuit can be used to wake up the main WiFi receiver after certain management frames (e.g., those containing traffic indication map (TIM)) are received. We note that Bluetooth is also using FM circuits, so it is even possible to use a Bluetooth receiver to wake up WiFi.

The insights gained from designing `FLEW` are also useful for understanding and mitigating the interference between FSK (such as Bluetooth) and WiFi. For example, Bluetooth devices should avoid using certain bit sequences (e.g., 0x05AE4701) as their access codes, since legitimate WiFi waveforms can cause accidental packet detection on Bluetooth receivers with such access codes.

3.2 System Design

3.2.1 Primer

3.2.1.1 FSK

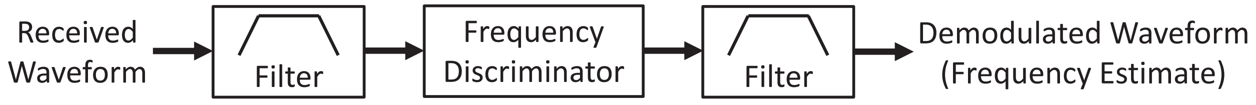


Figure 3.1: General model of FM/FSK receivers.

At the bit level, FSK is simply FM with digital data. In other words, FSK sends digital (high or low) data into an FM modulator. The instantaneous frequency of an FSK waveform depends

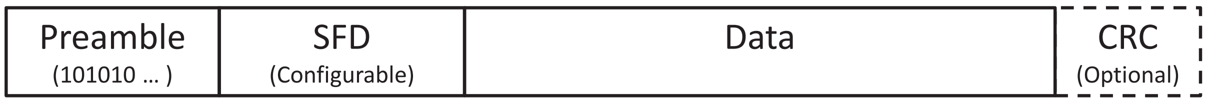


Figure 3.2: FSK packet format.

on the input level. For bit '1', the frequency is higher than the center frequency by one frequency deviation. For bit '0', the frequency is lower than the center frequency by one frequency deviation. An FSK receiver uses an FM demodulator to recover the bits. The FM demodulator tries to estimate the received signal's instantaneous frequency, which corresponds to the original digital data.

Fig. 3.1 shows the general model of FM/FSK receivers. The first filter is used to extract signals near the receive frequency. The filter after the frequency discriminator is also essential since the frequency discriminator is nonlinear and the filtering on the signal itself does not guarantee that its instantaneous frequency is properly filtered. Fig. 3.1 is the general model, regardless of whether the filters or demodulator are implemented in analog or digital domain, or whether there is down-conversion between these components (e.g., zero-IF or low-IF receivers).

Fig. 3.2 shows the packet format of FSK/proprietary protocols. The preamble consists of alternating 1's and 0's so that the demodulator in the receiver can be stabilized. The preamble is followed by a configurable SFD,[2] which signifies the receiver that it should expect and start collecting actual data after it receives SFD. If CRC is enabled, the CRC sequence is appended to the data field for detecting bit errors.

3.2.1.2 802.11b

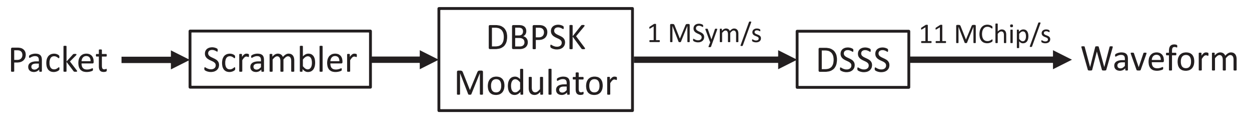


Figure 3.3: 802.11b modulation process.

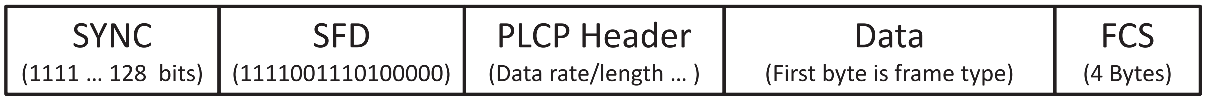


Figure 3.4: 802.11b packet format.

[2]To avoid confusion, here we use the WiFi nomenclature. This field is commonly referred to as the sync word in FSK protocols.

Fig. 3.3 shows the modulation process of 802.11b. The incoming bitstream is first scrambled with a different scrambler from other 802.11 standards. The scrambled bits are modulated by differential binary PSK, which either rotates the phase of the carrier by π for bit ‘1’ or keeps the phase unchanged for bit ‘0’. This waveform is then modulated using DSSS, which further toggles the phase within each bit duration using the 11-chip Barker sequence.

Fig. 3.4 shows the packet format of 802.11b. The SYNC field consists of 128 bits of ‘1’, which are used to stabilize the receiver. A constant SFD follows the SYNC field and is used by the receiver to detect the start of a WiFi packet. The PLCP header contains vital information about the modulation and the total length of subsequent fields. The PLCP header also contains a 16-bit CRC for detecting errors in the header. Upper-layer packets are put into the data field. The FCS (Frame Check Sequence) uses CRC32 and is calculated over the data field. A WiFi transmitter constructs a physical-layer packet this way before sending the entire packet into the scrambler.

3.2.2 Overview

For any useful WiFi operations, bi-directional communication is required. Therefore, Secs. 5.2.1 and 5.2.2, respectively, show how the FSK hardware can be used to receive and transmit WiFi signals with arbitrary payloads. In addition, since WiFi devices work in a half-duplex manner, the FSK chip must switch between transmission and reception appropriately and timely to avoid missed reception or transmission collisions. These MAC layer issues are addressed in Sec. 5.2.3.

3.2.3 WiFi to FSK

3.2.3.1 Bit Level

We devise a key method that enables `FLEW` to use FSK hardware to receive WiFi frames with arbitrary payloads. Its underlying insight is: **With an appropriate frequency shift, conventional FM/FSK receivers can work as a DSSS *plus* DBPSK demodulator.**

The implication of this insight is significant. With this method, instead of using a conventional PSK demodulator (which involves much more complicated phase synchronization), a simple, low-power FM demodulator can be used. Furthermore, the addition of DSSS requires conventional demodulators to run at a significantly higher speed (e.g., 11MHz). In `FLEW`, the FM demodulator can run at a much lower speed (1MHz). Therefore, this method allows `FLEW` to demodulate conventional WiFi frames in a much simpler and more power-efficient fashion.

The intuition behind this method is that the DSSS modulation process is essentially, in the frequency domain, convoluting the PSK spectrum with the spectrum of a repeated 11-length barker sequence. Note that an 11-length barker code has a white spectrum and the spectrum of a repeated

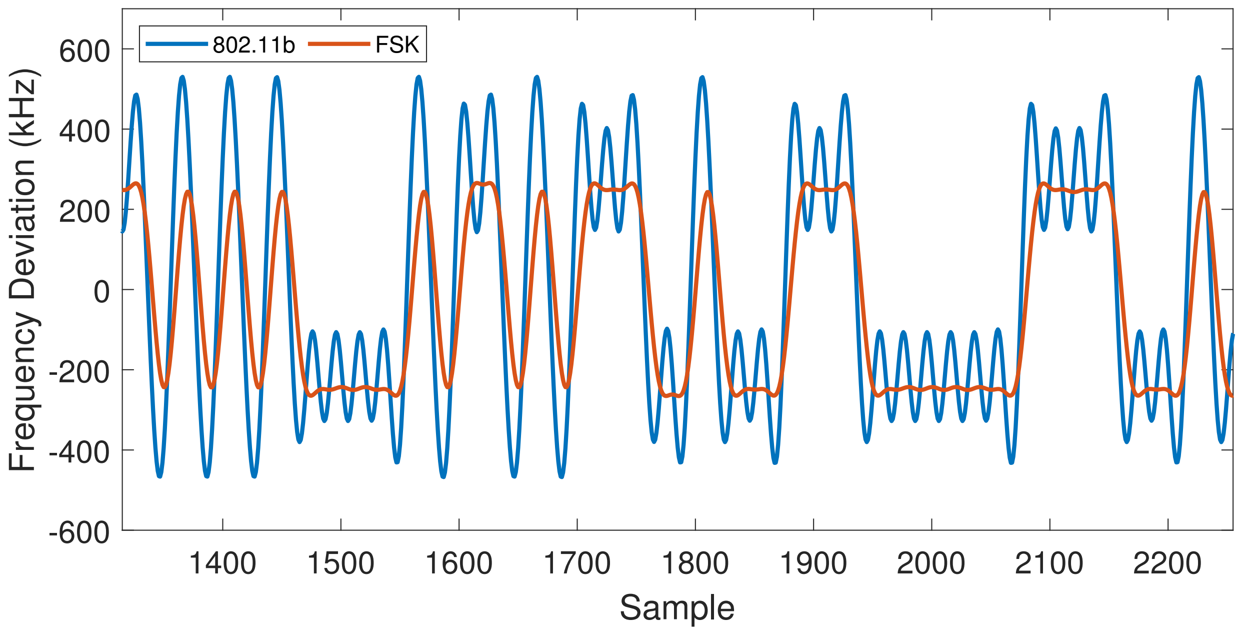


Figure 3.5: Demodulation outputs of bits modulated by 802.11b and FSK.

11-length barker sequence is only non-zero at ±1, ±2, ±3, ±4, ±5 MHz. The convolution process simply copies the PSK spectrum and places the replicas at these frequencies. Therefore, if we employ a relatively narrow filter to the DSSS waveform near one of the frequencies, the result is approximately the main lobe of the PSK spectrum.

Differential PSK is somewhat similar to FSK. Conceptually, the difference of phase is frequency. One major difference is that, in FSK, the phase is constantly increasing or decreasing for the duration of each bit, whereas in DPSK, the phase remains constant and only changes *between* bits. In FSK receivers, this difference can be mitigated by the filter after the frequency discriminator since the filter smoothens the step-like phase waveform of PSK modulation to become a constantly increasing or decreasing phase waveform.

Finally, we also need a small frequency shift in addition to, say 1 MHz, because an FSK receiver expects the frequency deviation of each bit to be either positive or negative. However, with DBPSK waveforms, the frequency deviation is either zero (no phase change) or non-zero (phase change). This can be corrected using a small amount of frequency shift, which equivalently adds a constant bias to the frequency deviation of each bit, and converts the frequency waveform to be non-return-to-zero.

Applying frequency shifts requires no additional hardware. In practice, we can simply change the center frequency of the receiver.

Let us illustrate this method with an example. In Fig. 3.5, we show the output waveforms of a FSK receiver demodulating an 802.11b waveform and a standard FSK waveform.

Specifically, we generate an 802.11b waveform and use the general model of FSK receivers to demodulate the waveform. The receiver has a frequency offset of 1.22MHz and two low-pass filters are used. The first trace shows an 802.11b waveform segment demodulated using the FSK model. The segment corresponds to bit 67 to bit 113 of DBPSK bits, which are 10101010000010110101110010001110000000111100010. The second trace shows the FSK demodulation results when these bits are instead modulated with FSK (using exactly the same modulation parameters as BLE), and demodulated without the frequency offset. Although the first trace tends to have higher overshoot, the correct bit sequence is still clearly visible, indicating that FSK receivers can indeed be used as WiFi receivers.

With the FSK hardware replacing DSSS and DBPSK demodulator, the only step left for recovering the WiFi bits is descrambling the bit stream. The descrambling process is extremely simple. Furthermore, the descrambling can be done in batch to each byte or word, and does not require extracting/reassembling bits to process them one-by-one. Specifically, the descrambling in 802.11b can be simplified as XOR'ing the input with two shifted versions of the input. With least-significant-bit-first ordering, the descrambling process involves only 4 lines of code:

```
reg = (descrambling_in<<8) | lastbyte;
reg2 = reg ^ (reg>>3) ^ (reg>>7);
descrambling_out = 0xFF & (reg2);
lastbyte = descrambling_in;
```

3.2.3.2 Packet level

Even with bit-level communication from WiFi to FSK hardware, we still need to address the differences in packet formats in order to successfully receive a WiFi packet.

The 802.11 standard [30] (Sec. 17.2.3.2) explicitly specifies the constant seed that an 802.11b transmitter should use. Because the scrambler seed is always the same, the scrambled SYNC sequence is always the same bit sequence. We verify that this is also the behavior of actual WiFi chips. For all WiFi devices we tested, including (Qualcomm) Atheros, Broadcom, Intel, Marvell, Mediatek/Ralink and Realtek chips, their (scrambled) SYNC is exactly the same bit sequence.

During reception, conventional WiFi chips detect a WiFi packet by matching the SFD pattern (which follows the SYNC field) in the descrambled DBPSK sequence.

In `FLEW`, the FSK demodulator outputs the scrambled WiFi DBPSK sequence. Because the (scrambled) WiFi SYNC field is always the same sequence, `FLEW` detects WiFi packets by directly matching a pattern in the scrambled sequence instead of the descrambled sequence. This design allows use of the SFD matching hardware in FSK chips, which is significantly more efficient. Since the matching circuit in FSK chips expects alternating 1's and 0's preceding the SFD to stabilize the demodulator, `FLEW` matches a bit sequence within the scrambled WiFi SYNC field instead of matching the scrambled WiFi SFD. Specifically, as illustrated in Fig. 3.5, bit 67 to bit 73 of the DBPSK bit stream is [1,0,1,0,1,0,1] and bit 74 to bit 105 is 0x05AE4701. Therefore, we configure FSK chips to search for 0x05AE4701. Once this sequence is detected, FSK chips continuously put subsequent bytes into the receive FIFO. We can thus periodically retrieve these bytes and descramble them to recover the WiFi packet. The reception is terminated once the number of bytes received reaches the length specified in the PLCP header.

While testing `FLEW`, we unexpectedly discovered a major bug in Realtek chips: their WiFi SYNC field is 2 bits shorter (126 bits of 1 instead of 128 bits of 1) than required. This is a clear deviation from the 802.11 standard. Even so, we further devise a simple and effective solution so that `FLEW` uses the same routine for communicating with both Realtek and non-Realtek transmitters. `FLEW` dynamically detects and fixes the bug for the former. Specifically, after descrambling, `FLEW` checks the last byte of the WiFi SYNC field, which should be 0xFF, as specified in the standard. If, instead, it is 0x3F, this indicates that the SYNC field is 2 bits shorter and 2 bits of SFD are shifted to this byte (since WiFi transmits least significant bits first). If the bug is detected, we simply apply a shift of 2 bits to all subsequence bytes.

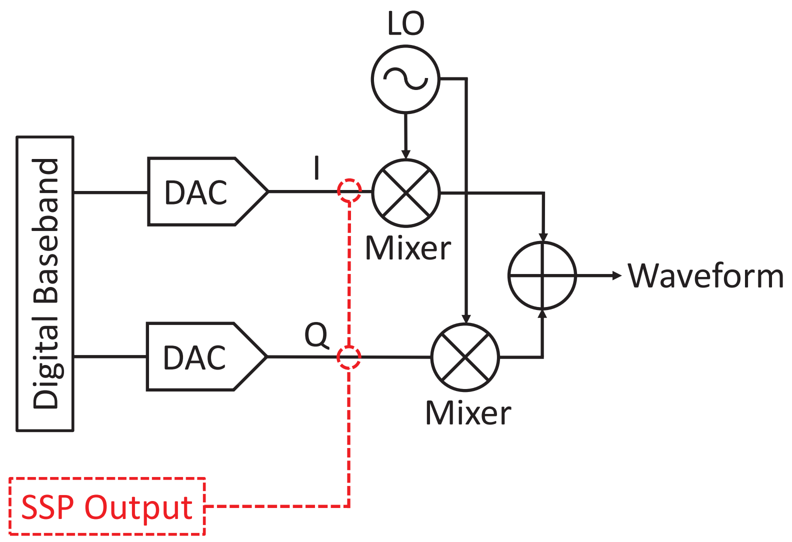
The tail of each WiFi packet is 4 bytes of FCS, which is the CRC32 of the data field. FCS is used to check the integrity of the received packet and if the calculated FCS does not match the received FCS, the receiver should not acknowledge this packet and the transmitter will re-transmit the packet. The implementation of FCS in `FLEW` is straightforward. We add a few optimizations, such as using table-based calculation and updating the CRC immediately after receiving each byte.

3.2.4 FSK to WiFi

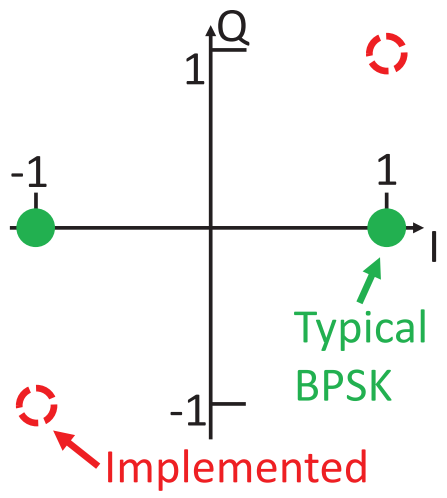
In a conventional PSK system, the PSK receiver is usually complicated due to phase synchronization, but the PSK transmitter is extremely simple. In fact, when a digital bit stream is directly fed into a mixer, the output is the PSK waveform, since bit ‘0’ has a negative voltage and inverts the carrier, while bit ‘1’ has a positive voltage and leaves the carrier unchanged.

By convention, BPSK constellations are (1,0) and (-1,0), as illustrated in Fig. 3.6b. When these constellations are fed into the IQ modulator, we essentially feed the digital bit stream (that swings to either 1 or -1) into the I-branch mixer and turn off the Q-branch mixer. Note that (1,0) and (-1,0) are 180° apart. However, this constellation involves three voltages (-1,0,1). A simpler implementation is actually tying the I-branch and Q-branch together. The constellations become (1,1) and (-1,-1), which are still 180° apart but only involve two voltages. Furthermore, the output gets 3dB stronger using both branches.

To transmit PSK waveforms using FSK hardware, we turn off the digital baseband and DAC



(a) FSK transmitter & waveform injection



(b) Constellation

Figure 3.6: FSK to WiFi Design

completely and directly inject the signals into the mixers, which can be achieved using the analog pins on FSK chips.

802.11 uses DSSS after PSK modulation. A PSK signal with DSSS is still a PSK signal, only faster. Conceptually, before DSSS, we are injecting either 1 or -1 into the mixer every 1μs. After DSSS, we are injecting either 10110111000 or 01001000111 every 1μs, which translates to a chip rate of 11MChip/s.

To generate the bit stream at 11Mbps, the serial interface (such as SPI, SSP, USART or even I2S) in microcontrollers can be used. These serial interfaces are also commonly double-buffered, ensuring that bits are transmitted continuously and precisely. In fact, on the microcontroller we use, the SSP has 8 transmit buffers.

At the packet level, `FLEW` assembles packets according to the 802.11b format.

3.2.5 FSM and MAC Layer

3.2.5.1 CSMA/CA

In WiFi, the transmission and reception of signals operate in half-duplex. A WiFi device should avoid transmitting signals when other devices are transmitting. WiFi uses CSMA/CA in the MAC layer, which senses the spectrum before transmission and waits if a wireless carrier is present. Typical FSK chips are also capable of sensing the spectrum. In particular, when in receive mode, FSK chips give the RSSI estimates. Combined with the timing design, presented in Sec. 3.2.5.3, CSMA/CA can be thus implemented.

3.2.5.2 Packet Handling

In typical WiFi systems, most of packet handling is implemented in the driver or software layers. Two exceptions are ACKs and RTS-CTS, which are subject to a very tight timing constraint, and hence usually handled by hardware.

Except for special packets, normal unicast packets in WiFi need to be acknowledged immediately. Failure of acknowledging a packet results in the sender constantly re-transmitting the same packet, which severely decreases the goodput. Furthermore, when a client tries to join a network, it must acknowledge the association response sent by the AP. Otherwise, the AP de-authenticates the client and a connection cannot be established.

According to the 802.11 standard, an ACK frame should be transmitted by the receiver one SIFS after it receives a packet that passes the FCS check. 802.11b has a very short SIFS (10μs). The standard also specifies the ACKTimeout, which is SIFS (10μs)+aSlotTime (20μs)+Preamble/Header (192μs). This timeout value is measured with respect to the end of the header of the ACK packet. Thus, when measured with respect to the start of the preamble, the time interval between a packet and its ACK packet should ideally be less than 30μs.

According to our testing, the Rx-to-Tx turnaround time of the FSK chip we use is around 30~40μs. We found that for chips of industry leaders (Atheros, Broadcom, etc.), the ACK packets sent can be successfully detected without any further design or modification. No workaround is needed for connecting to APs with Atheros, Broadcom and Marvell chips.

On the other hand, certain chips are sensitive to the ACK timing. An extreme example is the Intel chips. We found Intel chips can only detect ACK frames transmitted within a very short time (ideally just less than 10μs). The timeout specified in the standard is determined by the reception of the ACK header, not by the start of the ACK preamble. So, we can, in theory, start the preamble late but transmit less scrambled 1's to meet the deadline. However, such a design has little effect with Intel chips. Without detecting the ACK, the Intel chip re-transmits the same packet over and over again, essentially reducing the goodput to 0.

We, therefore, design a general solution that allows `FLEW` to meet the timing requirement of all WiFi products we tested. Our solution leverages the facts that a) the tail of WiFi packets is the 4-byte FCS and not the actual payload, b) higher layer either has additional error checking (e.g., TCP, even UDP, has checksums), or naturally anticipates occasional errors. Specifically, we terminate the reception after receiving 1 byte of FCS, thus reserving more time for the FSK chip to transition to transmit mode for ACK. The 1 byte of FCS is still used to check the integrity of the packet and determine whether an ACK packet should be transmitted.

An alternative design is only acknowledging the retransmitted packets and performing a full FCS check on the first packet. However, this will decrease the throughput by half due to re-transmissions.

Since ACK packets are always the same for a given (source) MAC address, instead of generating the ACK on the fly, we pre-generate the ACK bits before sending the authentication packet to the AP and re-use those ACK bits for all subsequent packets. We choose this time instance since the authentication packet signifies an FSK chip's intent to join the AP's network, and the FSK chip is expected to acknowledge traffic from the AP afterwards.

In the opposite direction, once `FLEW` sends a normal packet, AP should transmit an ACK packet. We use this packet to implement the re-transmit logic. Specifically, once a packet is sent, `FLEW` turns the FSK chip into receiving mode immediately. If a valid ACK is received, `FLEW` releases the transmit buffer, turns the chip into receiving mode and copies more data from the upper layer. If no packet is received after a timeout, `FLEW` re-transmits the packet. If a unicast (to FSK) packet is received, this indicates that the AP and the FSK chip might be transmitting simultaneously. In such a case, `FLEW` instructs the FSK chip to acknowledge the incoming packet, does not release the transmit buffer, and turns the chip into receiving mode for more incoming packets.

We noticed that the RTS-CTS mechanism is enabled by default on some APs. Thus, we also implement that mechanism. This is important for those APs since without receiving CTS, the AP will constantly re-send RTS without sending any data. The implementation of RTS-CTS is mostly the same as ACK handling, since CTS is also expected to be transmitted one SIFS after RTS.

3.2.5.3 Timing and FSM

We can now put all components together and design the main control logic, which can be simplified as a Finite State Machine (FSM), shown in Fig. 5.10. Once in Idle state, `FLEW` transitions to either Tx or Rx immediately, depending on the value of `txnext`. Rx indicates that the FSK chip is in receive mode, but the SFD (0x05AE4701) is yet to be detected. The CSMA/CA is implemented in this FSM because the only way that `txnext` changes from 0 to 1 (thus initiating a new transmission), is that a) the FSK chip is in Rx and reaches timeout, b) no WiFi packet is detected, and c) no carrier is present in the medium. `txnext` turns to 0 or remains unchanged (e.g., during re-transmission) for all other paths.

The Rx Packet state indicates that a WiFi SYNC is detected and the FSK chip is actively collecting the data. Depending on the packet type, transmission of ACK or CTS may follow. In the case of packet errors, either in the PLCP header or the data field, `FLEW` goes to Idle and restarts the process.

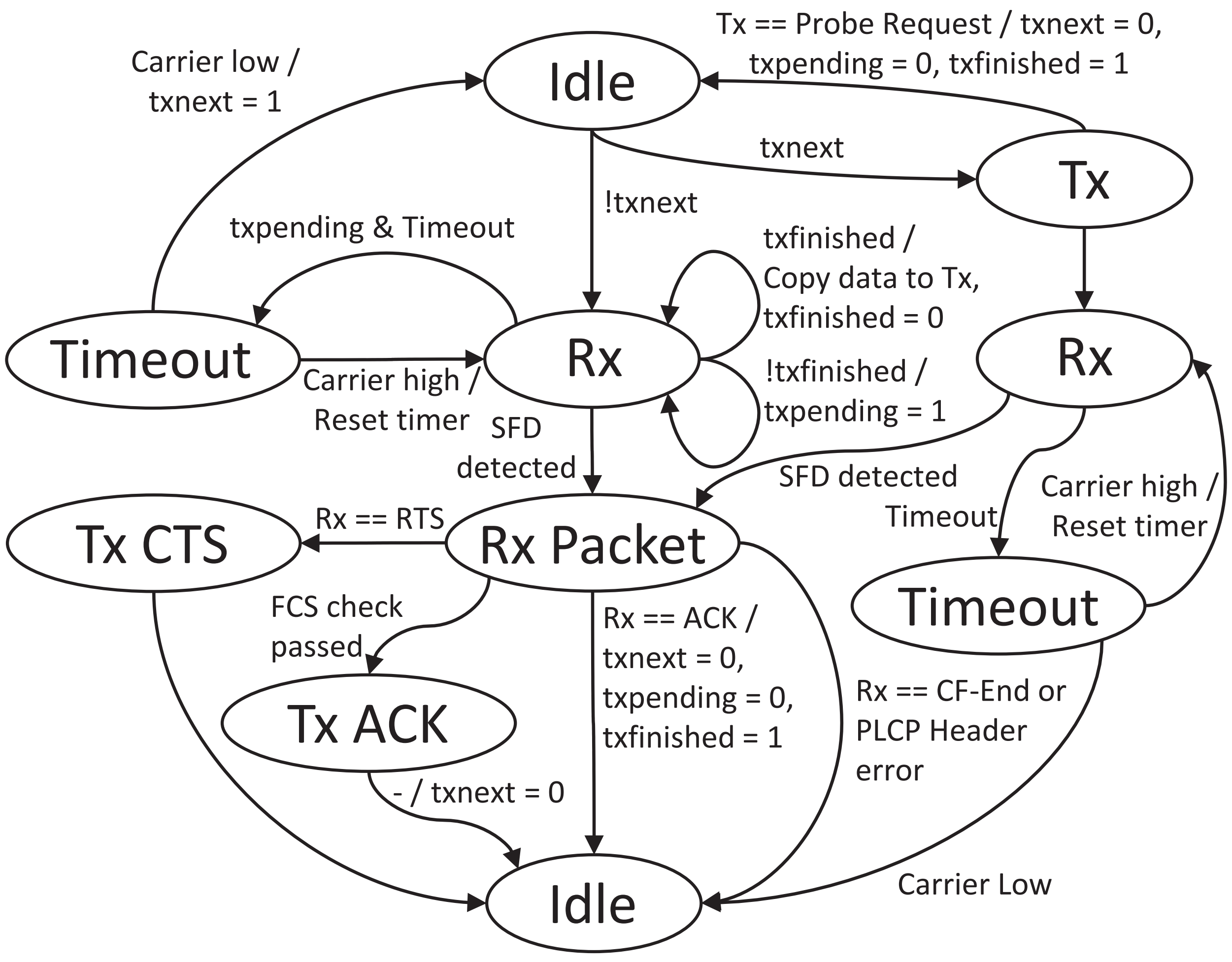

Carrier low / txnext = 1
Idle
Tx == Probe Request / txnext = 0, txpending = 0, txfinished = 1
txnext
Tx
!txnext
txpending & Timeout
txfinished / Copy data to Tx, txfinished = 0
Timeout
Rx
Rx
Carrier high / Reset timer
!txfinished / txpending = 1
SFD detected
SFD detected
Timeout
Carrier high / Reset timer
Tx CTS
Rx == RTS
Rx Packet
Timeout
FCS check passed
Rx == ACK / txnext = 0, txpending = 0, txfinished = 1
Rx == CF-End or PLCP Header error
Tx ACK
- / txnext = 0
Idle
Carrier Low

Figure 3.7: FSM

3.3 Implementation

3.3.1 Hardware and Firmware

We use COTS FSK chips to implement `FLEW`. In particular, we use Ubertooth [62] as the underlying hardware. At its core, Ubertooth uses TI's CC2400 [63], which is a standard FSK chip designed for low-power, low-voltage applications. The CC2400 family is also widely used in wireless sensor networks (WSNs), which require considerably lower power consumption than WiFi networks. Ubertooth was originally conceived as a Bluetooth *and* BLE debugging tool. With an appropriate firmware, it can communicate with Bluetooth and BLE devices. For this purpose, an optional radio front-end is hard-wired to CC2400, although it is not necessary for the operation of CC2400.

Ubertooth uses NXP's LPC1756 [64] as the main MCU, which also acts as a bridge between the USB and CC2400. The MCU is down-clocked to 88MHz via the built-in PLL. We use the LPC1756's SSP module to transmit the DSSS bits into CC2400's mixers. The IO pins of LPC1756 run at 3.3V whereas the core voltage of CC2400 is 1.8V. We thus use two resistors to convert the voltage. We tie the inputs to I and Q mixers together and with voltage conversion, 3.3V corresponds to 1 and 0V corresponds to -1 to the mixers in the constellation plane.

LPC1756 has an ARM Cortex-M3 core and we implement the FSM using C codes. Our custom firmware also handles the USB traffic and controlling CC2400 via SPI commands and hardware pins. Outgoing packets arrive at LPC1756 in the form of PSK bits, encapsulated in USB packets. The SSP module either sends 10110111000 or 01001000111, depending on each PSK bit. For reception, LPC1756 performs descrambling and FCS check in real time using the Cortex-M3 core. If the check passes, the transmission of ACK is directly initiated by LPC1756. Complete WiFi packets are sent to the host via USB packets.

3.3.2 WiFi Driver

We write a custom driver (a Linux kernel module) to interface FSK chips to mac80211, which sits on top of WiFi drivers in modern Linux WiFi architecture. Our custom driver is a very thin layer (less than 1k lines of code) that handles various mac80211 function calls, most notably ieee80211_tx and ieee80211_rx. The driver also manages a queue that buffers outgoing packets. Packets are popped off from the queue sequentially. The driver then converts the packet to WiFi PSK bits by adding the PHY header and FCS, scrambling the entire packets and applying differential codings. All these steps are simple bit operations and do not require any floating-point or DSP computation. For received packets, the driver polls USB packets and checks the FCS of the WiFi packet. If the check passes, the driver passes the packet to mac80211.

Even with such a small footprint, our driver supports monitor mode *and* packet injection. Furthermore, we ensure that the driver does not drop packets even when they are injected at the maximum speed, unlike certain other (most notably Realtek and Ralink) drivers.

It is possible to reuse existing WiFi drivers instead of writing a custom one. Such a design is only implementation variations and is not our focus.

3.3.3 MLME, WiFi Security and Upper Layers

We write the driver and the firmware in such a way that they can directly work with unmodified mac80211 and upper layers. WiFi's MLME (MAC subLayer Management Entity) operations, such as scanning via Probe Requests, authenticating and associating with an AP, are already implemented in mac80211. IP/ICMP and TCP/UDP have already been implemented in the Linux kernel. Many Linux (and Android) distributions come with wpa_supplicant, the *de facto* open-source WiFi security implementation. All these components work without modification. Therefore, the Internet works out-of-the-box once the driver is implemented.

3.4 Evaluation

The evaluation of `FLEW` can be divided into *physical-layer* and *system-level* evaluations. The former measures the physical-layer performance, such as PER (Packet Error Rate), in either WiFi-to-FSK or FSK-to-WiFi direction. Since we need to monitor the PHY traffic for physical-layer evaluation, these experiments are conducted with several WiFi NICs operating in monitor mode. Although these PHY metrics provide insights into synchronization, modulation and demodulation performance, such constant unilateral traffic never exists in real WiFi operations. Specifically, WiFi data packets must be acknowledged using ACK packets in the opposite direction. Therefore, the end-to-end performance depends on the PHY performance in both directions. Furthermore, the MAC layer also affects the end-to-end performance since wireless devices use a shared medium. Therefore, for system-level evaluation, we test `FLEW` with real WiFi APs and measure performance at the transport layer, which represents a more realistic performance characterization in the real world. The transmit power of `FLEW` is set to 20dBm for both evaluations.

3.4.1 Experimental Setup

Table 3.2: WiFi chips and APs used in experiments.

Chip Maker	PHY Evaluation	System Evaluation
Atheros	AR9462	GL.iNet GL-AR150 (AR9331)
Broadcom	BCM4313	ASUS RT-AC66U (BCM4331)
Ralink/ Mediatek	RT3072	TP-Link TL-WR841N (MT7628NN)
Marvell	-	Linksys EA3500 (88W8366)
Intel	AC 7260	TP-Link Archer AX3000 (WAV654A0)
Realtek	RTL8811AU	D-Link DIR-619L (RTL8192ER)

To test the performance and compatibility, we comprehensively evaluate `FLEW` with numerous commodity WiFi APs and devices with chips from all major WiFi chip-makers. The tested devices and chips are listed in Table 5.2.

(Qualcomm) **Atheros** and **Broadcom**[3] have long been considered industry leaders and each has a full WiFi portfolio from low-cost 1T1R to high-end, enterprise-grade chips. The overwhelming majority of Apple's products uses Broadcom's WiFi chips.

Ralink/Mediatek[4] chips are commonly found in low-cost APs. Note that Qualcomm and Mediatek

[3]Broadcom sold part of its WiFi portfolio to Cypress

[4]Mediatek acquired Ralink in 2011

SoC are widely used in high-end and low-cost Android smartphones/tablets, respectively. Therefore, although smartphones are not the main focus of `FLEW`, testing these chips ensures the compatibility of `FLEW` with the majority of smartphones. For example, we have verified that `FLEW` works with an unmodified iPhone acting as a WiFi hotspot.

Marvell[5] chips are mainly used in niche markets (high-end or enterprise APs) and are rarely used as general NICs nowadays. They lack Linux drivers or the drivers do not support monitor mode. So, we evaluate their performance at the system level. Marvell chips are commonly found in Cisco's 802.11ac APs, although Cisco's newer APs use Qualcomm chips. We use a Linksys EA3500 (which was launched during the Cisco era and internally uses a Cisco PCB) to evaluate the performance. On the other hand, **Intel** and **Realtek** chips are mostly used in client NICs and have much less presence in APs. Even so, we include their results for completeness.

In what follows, we refer to each system with its WiFi chip-maker. We do not modify the WiFi devices. In fact, we use the original firmware supplied with each equipment and we do not flash any new firmware. For WiFi APs, we use their browser interface to configure the WiFi channels and WiFi passwords. We use WiFi channel 4 and all APs use WPA2-PSK, unless specified otherwise.

3.4.2 PHY Layer and PER

Table 3.3: PER Evaluation (%)

Direction	WiFi-to-FSK			FSK-to-WiFi		
Distance	5m	10m	20m	5m	10m	20m
Atheros	1.93	1.46	2.29	0.00	0.02	0.07
Broadcom	3.10	3.15	2.59	0.27	0.78	0.17
Ralink	3.56	4.88	4.22	0.63	1.32	5.83
Intel	5.59	3.03	2.49	2.39	2.17	1.90
Realtek	3.32	4.47	4.69	6.52	8.54	8.86

We evaluate the PHY performance in both FSK-to-WiFi and WiFi-to-FSK directions. The transmitter (either FSK or WiFi) continuously sends a 1508-byte (1512-byte including FCS) packet to the receiver. For all (including FSK) receivers, all 4 bytes of FCS are received and compared; FCS-error frames, if any, are discarded. Since 4 bytes of FCS are used, it is practically impossible ($\frac{1}{2^{32}} = 2.33 \cdot 10^{-10}$) for an error frame to have the correct FCS.

To measure the PER, the transmitter marks these packets with unique sequence numbers. We iterate all possible sequence numbers and thus 4096 packets are sent for each test. We collect the sequence numbers of packets received at the other end. Because the sequence numbers sent are unique, any number missing within [0, 4095] indicates packet losses/errors.

[5] Marvell sold its WiFi portfolio to NXP

Table 3.3 shows the PER in the WiFi-to-FSK direction. For signal transmission, the Atheros chip shows superior performance. We use exactly the same set of typical WiFi antennas for Atheros, Broadcom and Intel chips, and their performance is largely the same. Even at 20m, the PER is around 2.5% or lower. For Broadcom and Intel chips, the PER actually increases slightly with shorter distances. This may be because the signal is slightly stronger than the optimal ranges, and they potentially have a slightly less accurate waveform. It is possible to further tweak the AGC (Automatic Gain Control) in the FSK chip, instead of using the default values, to extend the optimal range. For Ralink and Realtek chips, the NICs are in the USB form (since the mini PCI-E versions are hard to find). Because of the form factor limitation, USB NICs typically do not have the best performance, as evident in the table. `FLEW` still achieves a PER of less than 5% at 20m. Although Realtek chips do contain the SYNC length bug, to the FSK chip, this bug only affects the packet format and should be mostly unrelated to the PER.

Table 3.3 also shows the PER in the FSK-to-WiFi direction. The Atheros chip again shows phenomenal performance. At a distance of 5m, it is even possible to achieve 0% PER. Even at 20m, the PER is still much less than 1%. The Broadcom chip also has great performance with less than 1% PER under all conditions. For the Ralink chip, we believe the performance is mostly limited by the USB form factor, and thus the PER steadily increases with distance. The Realtek chip has the worst receiving performance among all chips (including receiving using FSK chips!), which may be partly contributed by its tiny USB form factor. It is also possible that bugs in Realtek chips affect the performance (e.g., expecting 126 bits of SYNC instead of 128 bits).

3.4.3 TCP/UDP Throughput

Table 3.4: Throughput Evaluation (kbps)

Direction	Uplink						Downlink					
Transport	TCP			UDP			TCP			UDP		
Distance	5m	10m	20m	5m	10m	20m	5m	10m	20m	5m	10m	20m
Atheros	646	661	671	721	700	697	766	771	768	839	840	857
Broadcom	616	602	654	697	698	697	703	686	707	813	818	800
Ralink	654	639	615	695	692	708	767	749	734	846	842	834
Marvell	645	636	656	693	701	698	679	678	694	792	803	791
Intel	651	636	471	699	578	673	743	721	586	811	814	816
Realtek	350	357	449	375	346	386	588	525	544	700	658	667

For system-level evaluation, we measure the transport-layer throughputs with unmodified WiFi APs. We use iperf3 [41], the standard tool for measuring network performance. An iperf3 server is either run on the AP itself (Atheros), or on a host connected (via LAN) to the AP (Broadcom, Ralink, Marvell, Intel, Realtek). At the other end, a Ubuntu laptop with FLEW driver installed is running an iperf3 client. The laptop connects to these APs via the FSK chip. We measure both TCP and UDP throughputs in both directions with the following commands:

```
TCP Uplink: iperf3 -c <server's IP>
UDP Uplink: iperf3 -c <server's IP> -u
TCP Downlink: iperf3 -c <server's IP> -R
UDP Downlink: iperf3 -c <server's IP> -R -u
```

We record the throughput reported by the receiving end.

Table 5.5 shows the uplink throughputs at different distances. All AP chips of four popular chip-makers (Atheros, Broadcom, Ralink, Marvell) have similarly good performances. The Atheros chip is still slightly better than others. UDP throughputs are higher than TCP throughputs as TCP requires additional TCP ACKs sent at the transport layer. To the physical layer, these TCP ACKs are simply packet payloads traveling in the opposite direction. Although it is redundant to have ACKs at both the physical layer and the transport layer, it is the way the Internet works. Because WiFi is half-duplex, these TCP ACKs consume some bandwidth, and thus decrease the "real" throughput. In addition, these TCP ACKs create more contention as the AP now also tries to access the channel to transmit TCP ACKs. Nevertheless, FLEW's CSMA/CA works well and the throughput is only 5~10% lower. The Intel chip has equally good performance but is less stable at longer distances.

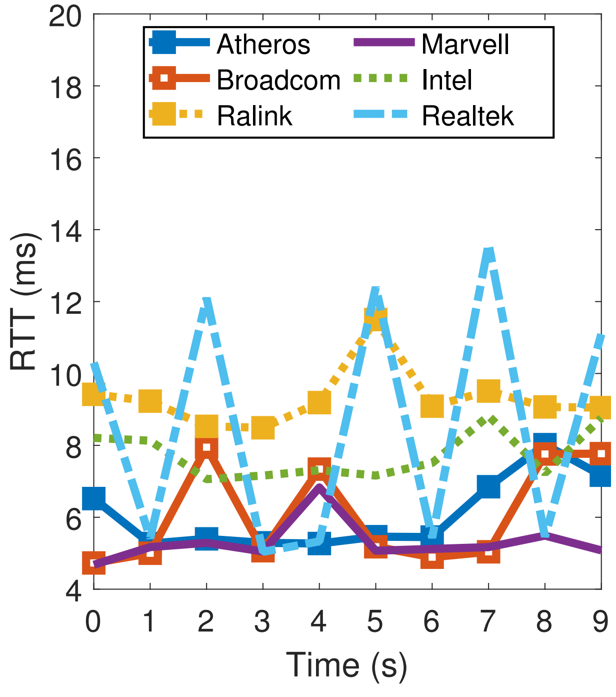
The only outlier in the uplink experiments is Realtek. This may be attributed to its inferior receiving performance, potentially due to the SYNC or other bugs. We would like to stress that Realtek chips are much less commonly used in commodity APs, and are especially rare in enterprise APs. They can still sustain a transport-layer throughput of at least 350kbps. We leave specific optimizations for Realtek chips as future work.

Table 5.5 also shows the downlink throughputs. Downlink throughputs are, in general, higher than uplink throughputs because, in the FSM (Fig. 5.10), a small amount of time is spent on copying transmit data between each transmission. That is, even if the channel is clear/available, the FSK chip will not transition to Tx mode if the data is not ready. This design simplifies the re-transmit and USB logic, but the throughputs are slightly lower due to unused airtime. The difference is smaller for TCP because the unused airtime can be reclaimed by the AP to transmit TCP ACKs.

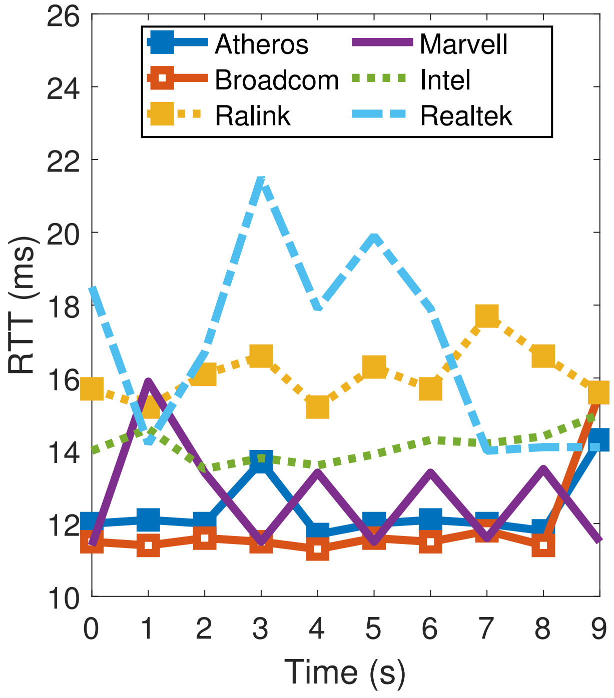
Downlink has a similar trend to uplink. The first four chip-makers have most consistent results across different distance ranges. The Intel chip can be equally good but is less consistent. The Realtek chip performs much better, since downlink involves mostly transmission, but is still inferior to other chips.

Unlike uplink, downlink throughputs show higher variations among different chip-makers. For example, Atheros and Ralink chips consistently outperform Broadcom and Marvell chips. We believe the rate adaptation algorithms (RAAs) used largely contribute to this since Atheros and Ralink (and even Intel) have their own proprietary RAAs [65, 66, 67]. When data packets are sent by the AP, the AP may briefly try a different rate before falling back to 1Mbps. These RAAs seem to better balance the throughput and exploration of other rates. As discussed in Sec. 5.1, using 1Mbps allows RAAs to converge and not diverge or cause disconnection. Even at distance of 20m, `FLEW` stays connected throughout all the experiments.

3.4.4 RTT



(a) LAN



(b) WAN

Figure 3.8: RTT

We also measure the round-trip time (RTT) of each setting. Furthermore, we measure the RTT over both LAN and WAN. For LAN, we `ping` the AP to which the FSK chip is connected. For WAN, we connect the AP to the Internet and `FLEW` tries to `ping` 8.8.4.4, Google's public DNS server.

For each system, we perform 10 pings. We measure RTTs at 20m, since the RTTs of shorter distances are largely the same as those of 20m. Even at 20m, no ping was lost for every configuration.

Figs. 4.6a and 4.6b plot the measurement results. Atheros, Broadcom and Marvell generally have the lowest RTTs. Intel and Ralink lag behind, but their RTTs are still stable. Realtek has the worst RTTs that jump all over the place. This can be partly attributed to Realtek chips' non-ideal wireless performance. WAN RTTs are mostly the same as LAN RTTs. The only difference is the ~6ms delay for traveling across the Internet.

3.4.5 Coexistence

3.4.5.1 Coexistence with WiFi Devices

Table 3.5: Coexistence with multiple WiFi devices (bps)

# of Devices		TCP UL	UDP UL	TCP DL	UDP DL
1	FLEW	663 k	708 k	730 k	841 k
2	FLEW	412 k	698 k	293 k	603 k
	AC7260	13.1 M	1.05 M	25.0 M	1.05 M
3	FLEW	123 k	676 k	117 k	517 k
	AC7260	10.5 M	1.05 M	19.8 M	1.05 M
	AR9271	19.3 M	1.05 M	11.5 M	1.00 M
4	FLEW	92.7 k	616 k	178 k	504 k
	AC7260	8.27 M	1.05 M	12.5 M	1.06 M
	AR9271	7.73 M	1.05 M	6.2 M	1.04 M
	RTL8811AU	10.9 M	1.05 M	12.5 M	1.05 M
5	FLEW	90.9 k	621 k	166 k	580 k
	AC7260	6.98 M	1.05 M	10.3 M	1.05 M
	AR9271	1.95 M	1.05 M	3.09 M	1.03 M
	RTL8811AU	2.80 M	1.05 M	10.4 M	1.05 M
	BCM43602	15.2 M	1.05 M	7.12 M	1.05 M

To evaluate `FLEW`'s performance when it coexists with other WiFi devices, we also use iperf3 to measure the performance when up to 5 WiFi clients are simultaneously sending or receiving data. To accommodate multiple iperf3 connections, multiple iperf3 servers simultaneously run on the Atheros AP and listen on different ports.

Table 4.4 shows the results. By default, iperf3 injects UDP data to and from each client at around 1.05Mbps. In these cases, the channel is not saturated and all clients access the channel efficiently. For UDP uplink, the throughput of `FLEW` does not decrease much, indicating that the MAC layer of `FLEW` functions properly and allows different WiFi clients to access the spectrum efficiently. Throughputs decrease more for downlink, which may be due to slight delays for a single AP to prepare and send multiple streams.

For TCP, iperf3 injects the data at the maximum speed, which saturates the channel. In these situations, any throughput gain at one client comes at the expense of the throughput decrease at another client. For FLEW, throughput decreases are more pronounced as 2 or 3 devices coexisting and plateau out as more devices are added. We believe that from FLEW 's perspective, interference increases drastically from 1 to 2 or 3 devices. Although interference does continue to increase for more devices, the increase in ratio is comparatively lower. We observe that even though WiFi devices follow the WiFi MAC standard, throughput imbalance still occurs, even among COTS WiFi cards from different vendors, when the WiFi channel is saturated to the max. Therefore, we conclude that the WiFi MAC alone cannot guarantee a perfect throughput distribution among WiFi terminals in practical wireless environments. However, the key takeaway is that even under saturation, throughput of FLEW does not starve to 0 and is still sufficient for many IoT applications. Furthermore, a WiFi channel may only truly saturate occasionally during bursty data transmission. In such cases, FLEW can still provide decent overall throughput by utilizing the channel idle time.

A fairer throughput distribution can be achieved by rate limiting or a better load balancing at the AP. Alternatively, PCF can be used to allow the AP to arbitrate the traffic and eliminate spectrum contentions. It is also possible to fine-tune the MAC layer to access the spectrum more aggressively. We leave these as future work.

3.4.5.2 Coexistence with FSK Devices

Table 3.6: Throughputs with multiple FLEW devices (kbps)

# of Devices		TCP UL	UDP UL	TCP DL	UDP DL
1	FLEW #1	663	714	753	833
2	FLEW #1	357	385	390	295
	FLEW #2	358	386	307	396

Table 3.7: Coexistence with background BT traffic (kbps)

# of BT Devices	TCP UL	UDP UL	TCP DL	UDP DL
0	655	715	761	819
1	667	716	758	820
2	650	714	748	821

FLEW coexists with other FSK devices well. Table 3.6 shows the results of multiple FLEW nodes sending/receiving data simultaneously. For uplink, throughputs are almost perfectly divided by multiple nodes, indicating that the MAC layer design allows each node to access the spectrum equally and efficiently. Furthermore, the aggregate uplink throughputs of multiple nodes are slightly higher, which is a result of the utilization, by the second node, of the small unused uplink airtime (described in Sec. 5.3.3) of the first node. For downlink, the throughputs are split in about 57:43

between nodes, which may be due to the different channel conditions and packet processing in the AP. The aggregate downlink throughputs of multiple nodes are slightly lower because of more spectrum contention.

FLEW achieves excellent performance in the presence of coexisting Bluetooth devices. We evaluate the performance of FLEW when there are multiple Bluetooth connections in the background streaming music. Table 4.6 shows that the throughputs of FLEW are virtually unaffected by the number of Bluetooth audio streams in the environment. We attribute this result to the frequency-hopping design and the adaptive frequency-hopping mechanism in Bluetooth protocols. Specifically, Bluetooth transmitters are constantly switching channels over the span of 79MHz, thus making a Bluetooth transmitter unlikely to hop to the same frequency as the FSK chip. Furthermore, Bluetooth systems use the adaptive frequency-hopping mechanism, which automatically avoids Bluetooth channels within active WiFi channels. Additionally, Bluetooth systems typically operate at a lower transmit power, which, combined with robustness of DSSS, mitigates the impact on WiFi performance even if a collision does occur. Since FLEW uses the standard WiFi waveform and WiFi MAC design (i.e., timing), the existing results of WiFi–Bluetooth coexistence and interference-reduction mechanisms are directly applicable to FLEW.

3.4.6 Mobile and Outdoor Environments

3.4.6.1 Mobile Environments

Table 3.8: Throughputs in mobile environments (kbps)

Condition	TCP UL	UDP UL	TCP DL	UDP DL
Stationary	641	700	682	809
Walking	651	642	673	747
Running	647	677	670	733

To evaluate the performance under mobile conditions, we measure the throughputs when a person with the FLEW terminal is walking or running back and forth toward/away from the AP at a distance of 10~15 meters. Interestingly, Table 3.9 shows that mobile environments have more impacts on UDP than on TCP throughputs, although UDP still has higher throughputs. From the results, we conclude that the extra robustness and throttling provided by TCP may mitigate the throughput variations in certain conditions.

Further optimizations for mobile environments are possible. For example, we use the default packet size, which has a relatively long time duration. If shorter packets are used (e.g., by fragmentation), the channel response may be more similar within the duration of each packet, making it closer to the (piecewise) stationary condition.

3.4.6.2 Outdoor Environments

Table 3.9: Throughputs in outdoor environments (kbps)

Distance (m)	TCP UL	UDP UL	TCP DL	UDP DL
25	654	661	734	821
50	646	702	731	844
75	652	709	761	862
100	674	716	739	828
125	652	710	719	763
150	677	710	610	689

We also evaluate `FLEW` in practical outdoor environments. Table 3.9 shows the performance in a typical university campus environment with very few interference sources around the Atheros AP. Uplink throughputs maintain consistently good performance for at least 150m, which validates that `FLEW` is ideal for IoT applications where sensor data travels in the uplink direction. Downlink throughputs are consistently high within 100m and are reduced gradually for longer ranges, which is likely a result of the higher PER. Even so, at 150m, throughputs of `FLEW` in real-world outdoor environments are at least 2x the maximum throughput of Zigbee. (In addition, Zigbee at 2.4GHz typically only has a range of 10~100m [68].)

Outdoor and industrial WiFi APs are allowed to transmit at a higher power than indoor APs. For example, a typical Cisco outdoor AP [69] has a transmit power of 30 dBm, which is at least 10~12 dB (corresponding to 3~4x in range) higher than the AP we use. This additional transmission power can significantly increase downlink range. The Cisco outdoor AP also has a better sensitivity at -103 dBm. Therefore, we expect an even better performance when `FLEW` is paired with APs designed for outdoor and industrial use.

3.4.7 Secured vs. Open Network

Table 3.10: Throughputs w/wo WPA2-PSK (kbps)

WPA2-PSK	TCP UL	UDP UL	TCP DL	UDP DL
Enabled	661	700	755	850
Disabled	676	697	780	858

All system evaluations presented so far are done with WiFi security (WPA2-PSK) enabled, since it is the recommended setting. Enabling WiFi security incurs only a very small amount of overhead. For example, WPA2-PSK incurs an overhead of 16 bytes per packet, which is about 1%

of a typical WiFi data packet (~1500 bytes). Throughput may increase very slightly using open networks. Using the Atheros AP as an example, we measure the throughputs with and without WPA2-PSK and leave other parameters, including device placement, intact. Table 3.10 shows slightly higher throughputs using open networks. In practice, other factors, such as background traffic and interference, can easily outweigh the effect of WiFi security settings.

3.4.8 Application Examples

To the network stack and the AP, `FLEW` behaves just like a conventional WiFi chip. Network applications can use `FLEW` for Internet access without even recognizing the use of an FSK chip, instead of a WiFi chip. General web browsing works normally as well. In addition, `FLEW` can support streaming high-quality audio in real time using Spotify. `FLEW` is also shown to be able to support streaming 480p Youtube videos in real time. When little/no background interference is present, it can even stream 720p Youtube videos in real time. Higher resolution videos can be supported with enough buffering (or potentially in real time with better video compression).

3.4.9 Power Consumption

Table 3.11: Comparison of power consumption

	Idle (A)	Tx (A)	Rx (A)	Rx-Idle (A)
AR9271	0.02	0.49	0.07	0.05
RTL8811AU	0.01	0.32	0.07	0.06
RT3072	0.04	0.28	0.13	0.09
`FLEW`	0.08	0.16	0.11	0.03

FSK chips are very power-efficient. CC2400 only draws 19mA in transmit mode (at 0dBm) and 24mA in receive mode. For a fair comparison (and since the transmit current of conventional WiFi chips is typically measured at near 20dBm), we consider that an external PA [70] is used, which is also the case in Ubertooth. The external PA boosts the signal to 20dBm but draws 100mA (at 3.3V). Even with the PA, the overall power consumption is considerably lower than the WF200 in Table 3.1.

Table 5.8 shows the overall power consumption of different USB WiFi cards, including Ubertooth with `FLEW`. We measure their USB (5V) supply current when in idle, in continuous transmission or reception of 1Mbps WiFi waveforms. `FLEW` has the lowest power consumption in Tx mode. Ubertooth has a higher idle current, even when CC2400 and the PA are turned off, than other WiFi cards. If we adjust the Rx current with the idle current, `FLEW` also has the lowest power consumption in Rx mode.

3.5 Discussion

3.5.1 OFDM

The 802.11 standard has always stressed backward compatibility, which enables FLEW to communicate with newer systems, even though they might primarily use OFDM modulation. Specifically, 802.11g is a superset of 802.11b and 802.11a and the standard explicitly specifies that any 802.11g device should fully support 802.11b operations. Similarly, the standard states that every 802.11n device should also be an 802.11g device; every 802.11ax device should also be an 802.11n device, etc. Therefore, even if new devices use OFDM, they should still support FLEW operations. We have verified that FLEW works with 802.11b, 802.11g, 802.11n, 802.11ac and 802.11ax APs and devices.

Compared to OFDM waveforms, DSSS waveforms are more suitable for IoT applications, where device complexity, power consumption and cost are more important than throughput. In addition, DSSS waveforms are much more robust than OFDM (a 9dB [56] higher link budget). OFDM waveforms are also known to have high PAPR (Peak-to-Average Power Ratio), which prevents the use of simple, power-efficient PA and LNA. In contrast, PSK waveforms are generally considered as a form of constant-envelope modulation since information is encoded entirely in the phase of the signal [71]. Therefore, DSSS is much more suitable for the PA and LNA inside FSK chips, since FSK is also a constant-envelope modulation.

The data rate of 802.11b overlaps with common FSK data rates (1Mbps), which allows FLEW to directly reuse demodulation hardware, including the packet handling circuits, on common FSK chips, thus enabling low-power operations. The data rate of WiFi OFDM does not overlap with common FSK data rates. Consequently, the demodulator and packet handling circuits on FSK chips are unlikely to work with OFDM waveforms. Furthermore, processing OFDM waveforms requires significantly more complex and power-hungry circuits, including much wider radio front-end, faster A/D conversion and FFT circuits. These essential OFDM building blocks are simply not present on FSK chips.

Finally, we found that 802.11g/802.11n/802.11ac/802.11ax APs, by default, use 802.11b waveforms to transmit beacons and management frames. Therefore, even if there exists a solution that enables FSK chips to receive/transmit OFDM waveforms, the support of 802.11b is still essential, especially if the APs cannot be modified. In this regard, FLEW will still be integral in such solutions to handle management functions, such as association with APs, and to ensure maximum compatibility with unmodified WiFi devices.

From the practicality perspective, if existing devices already use FSK, Bluetooth or BLE protocols, then the throughput of FLEW is sufficient since FLEW matches their mandatory data rate. Thus, for these devices, the benefits of OFDM waveforms are not critical, especially considering DSSS waveforms are more robust and FSK chips lack the OFDM building blocks.

The limitation of emulating OFDM mainly comes from the fundamental hardware limits of FSK chips. It might be possible to use the radio circuits on FSK chips and emulate OFDM building blocks in firmware using a high-speed micro-controller. However, such a design would require a significant amount of computing power and would likely require floating point units (FPUs). These requirements will likely prevent the adoption of such a solution to simple, low-power and low-cost IoT devices. In contrast, `FLEW` does not require any floating point calculations. In fact, the Cortex-M3 micro-controller on Ubertooth is not equipped with any FPU.

3.5.2 Security Implementation

WiFi security algorithms can be implemented in software. However, modern WiFi security frameworks (e.g., WPA2 and WPA3) use AES and most CPUs (Intel since Sandy Bridge, AMD since Bulldozer, modern ARM processors) support AES instructions. Many micro-controllers, especially ARM TrustZone MCUs and even low-end MCUs designed for IoT, also have hardware AES accelerators. These AES hardware help efficiently encrypt and decrypt WiFi payloads.

3.6 Related Work

Multiple systems [17, 18, 19, 20] have demonstrated the communication from modified WiFi devices to Zigbee chips. However, they aim to turn WiFi devices into a Zigbee transmitter, and in those systems, Zigbee chips do *not* work as a conventional WiFi device.

Several prior studies explore communication from modified WiFi devices to Bluetooth/BLE devices. BlueFi [72] modifies WiFi devices to transmit Bluetooth signals, such as BLE beacons or audio packets. WiBeacon [73] modifies WiFi APs to transmit BLE beacons. These systems only allow one-way broadcast (beacons) or one-way unicast (audio) communication *from* WiFi *to* Bluetooth devices. In addition, modifications must be made to the WiFi device. NBee [74] shows the possibility of bit-level communication from 802.11b (with QPSK formulation) to BLE receivers, NBee uses this to construct the magic packets that a modified 802.11b transmitter should send so that the BLE receivers can decode the packet as a normal BLE packet. In contrast, `FLEW` analyzes BPSK waveforms and constructs a fully functional 802.11b receiver for packets with arbitrary payloads, transmitted by unmodified WiFi transmitters. In addition, `FLEW` presents a general model of FSK demodulation and we show, with theoretical insights and simulations, how such a communication will work on the general, architectural level. The core analysis in NBee is under the condition of BLE receivers being tuned to the same frequency as the WiFi channel. `FLEW` uses a frequency offset and we explain, with theoretical reasoning, why such an offset plays a critical role in converting DSSS to PSK waveforms and in demodulating DBPSK waveforms with

FSK circuits. With the general FSK receiver model, the WiFi-to-FSK result only holds when the frequency offset is introduced. FLEW also demonstrates how the packet handling hardware on FSK chips can be used to detect and efficiently decode conventional WiFi packets, not just certain packets that are precoded in the BLE form. FLEW also addresses the subtle issues of waveforms transmitted by different WiFi chips, such as the bug of Realtek chips. NBee only evaluates the performance using USRP whereas FLEW extensively evaluates the performance with real, unmodified WiFi chips from all major WiFi chip makers. Therefore, FLEW targets a very different use-case and presents a practical design and results under real-world conditions.

Although another prior work [49] does enable bi-directional communication between SDR-emulated WiFi devices and modified Bluetooth devices, not only does it require modifications to both Bluetooth and WiFi devices, it also does not validate the running of such a method on real commercial WiFi chips (instead of SDR devices). Moreover, a custom encoding is used on top of Bluetooth modulation, thus significantly reducing its throughput. All these systems aim to use WiFi to transmit Bluetooth waveforms and their Bluetooth devices do not operate as a conventional WiFi client. Their Bluetooth devices can only receive WiFi packets with specifically-crafted payloads. Inter-scatter [48] enables bi-directional communication between a backscatter, which uses a modified Bluetooth transmitter as the RF source, and modified WiFi devices. It also requires the RF source to be placed very close (¡1m) to the backscatter.

BlueBee [21] enables communication from modified Bluetooth transmitters to Zigbee receivers and XBee [22] enables communication from modified (for generating access codes) Zigbee transmitters to modified (for cross-coding) Bluetooth receivers. Zigbee uses MSK [75], which is a special form of FSK. In contrast, WiFi uses PSK with DSSS, which is much more different than FSK. In addition, WiFi is 4x faster than Zigbee and it is theoretically impossible (250kbps¡1Mbps) for a standard Zigbee chip to receive WiFi packets with arbitrary payloads. Therefore, their methods are not directly applicable to our system. Furthermore, even at the transport layer, FLEW is still much faster than Zigbee CTCs (250 kbps). XBee configures the access address of BLE receivers to detect Zigbee packets. The contribution of FLEW here is that we methodologically find a specific SFD and show via extensive experiments that the SFD works well for WiFi-to-FSK communication. In addition, the process of selecting SFD in FLEW can be generalized for other WiFi applications, such as reducing the interferences between WiFi and FSK devices, or waking up WiFi receivers with BLE chips. These contributions are very different from XBee, which is designed for Zigbee-to-BLE communication.

3.7 Conclusion

We have presented FLEW that enables low-power FSK chips to directly communicate with unmodified WiFi APs. We have leveraged several insights on FSK and WiFi modulation. We have evaluated FLEW extensively to demonstrate its compatibility with chips from all major WiFi chip-makers with good performance.

CHAPTER 4

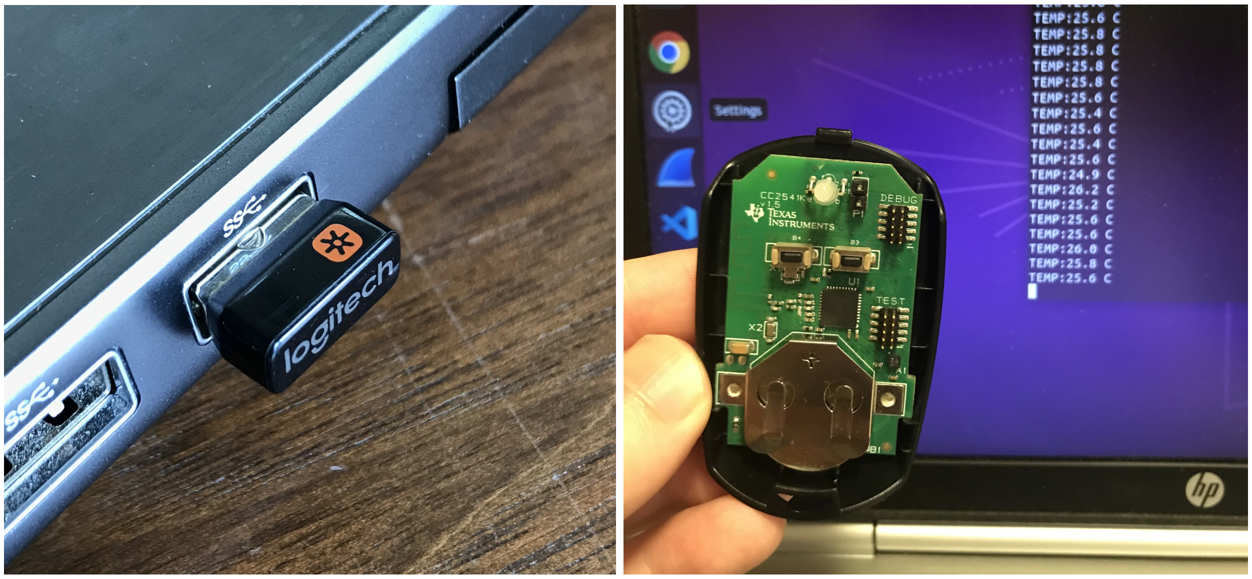
Unify: Turning BLE/FSK SoC into WiFi SoC

4.1 Introduction

WiFi and Bluetooth are the two leading wireless technologies that connect tens of billions of devices. Conventionally, these two technologies are deemed incompatible and treat signals from/to mismatched devices as interference. Recently, however, researchers in the field of cross-technology communication (CTC) have shown that this "interference" can actually become valid communication signals by leveraging digital signal processing to pre-process signals and/or by reusing various radio and digital blocks.

It is both important and practical to enable bi-directional communication between WiFi and Bluetooth devices. It allows a device to communicate in environments without any other device of the same wireless technology, which was previously impossible. For example, Bluetooth devices can directly connect to WiFi APs and access the Internet, eliminating the need to install any Bluetooth gateways. Furthermore, such a communication can combine the strengths of both WiFi and Bluetooth. For example, Bluetooth has lower cost, lower device complexity, and better power efficiency than WiFi, while WiFi infrastructure is omnipresent and provides direct Internet access. BLE chips are typically cheaper than WiFi chips, and the much simpler hardware of many BLE system-on-chips (SoCs) enables their operation for several years with a single coin cell battery. Bi-directional communication between WiFi and Bluetooth opens a new opportunity of connecting low-cost, energy-efficient Bluetooth devices to omnipresent WiFi APs and provides direct Internet connectivity to Bluetooth devices without requiring gateways.

This envisioned WiFi–Bluetooth bi-directional communication is particularly suitable for Internet of Things (IoTs) and wireless sensor networks (WSNs) in which nodes upload sensor data to, and receive actuation (control) messages from, the cloud. In these use-cases, the sensor data and the actuation messages are usually small but should be sent/received in a timely manner in order to accurately reflect and control real-world events/behaviors. Considering the large number of IoT/WSN nodes that have already been, or are expected to be, deployed in the real world, each



```
TEMP:25.6 C
TEMP:25.8 C
TEMP:25.8 C
TEMP:25.8 C
TEMP:25.8 C
TEMP:25.6 C
TEMP:25.4 C
TEMP:25.6 C
TEMP:25.4 C
TEMP:25.6 C
TEMP:24.9 C
TEMP:26.2 C
TEMP:25.2 C
TEMP:25.6 C
TEMP:25.6 C
TEMP:26.0 C
TEMP:25.8 C
TEMP:25.6 C
```

(a) Unifiying dongles

(b) CC2541 BLE modules

Figure 4.1: (a) `Unify` turns a Logitech Unifiying dongle into a fully operational WiFi dongle without any hardware modification. The dongle directly connects to a standard WiFi AP. (b) `Unify` turns a CC2541 BLE keyfob into a WiFi thermostat. CC2541 uploads sensor readings directly to the Internet via a WiFi AP.

of them should be cheap, small and highly power-efficient so that they can "operate" as intended without frequent battery changes/charging. BLE and FSK chips are ideal for meeting these requirements as they are very cheap, small, and suitable for "small but frequent" (as opposed to "large and bursty") communications with ultra low power consumption. However, IoT and WSN nodes require connection to the Internet, but BLE or FSK chips by themselves do not support such connectivity and hence require gateways. Bi-directional communication between WiFi and Bluetooth can fill this gap by equipping BLE/FSK chips with WiFi connectivity, thus allowing IoT or sensor nodes to use cheap and energy-efficient BLE/FSK chips while also enabling their direct use in conventional WiFi environments.

On the other hand, state-of-the-art (SOTA) CTC solutions have a number of limitations and have not yet fully realized this vision. Specifically, some CTC solutions only enable *one-way communication* whereas others require use of *multiple chips and hardware modifications*.

Numerous SOTA CTC solutions [73, 72, 76, 74, 77] enable one-way communication by modifying WiFi transmitters to send "magic packets" that can be received by Bluetooth devices. Specifically, these solutions carefully select the bits within a WiFi packet so that, after modulating these bits in WiFi modulation, the physical waveform resembles a legitimate Bluetooth waveform. However, since this approach can only provide one-way communication from WiFi Tx to Bluetooth

Rx, it cannot be used when IoT/WSN nodes (using Bluetooth chips) need to upload sensor data to the cloud via WiFi infrastructure. Also, device discovery and encryption cannot be implemented with one-way communication without side channel information, which significantly complicates system deployment. As a result, the use of such solutions is limited to Bluetooth beacon broadcasting or one-way "downlink" (from WiFi to Bluetooth) communication.

FLEW [78] introduced a fundamentally different design that addresses the above shortcomings and is shown to work with actual WiFi APs. Other CTCs enable WiFi-to-Bluetooth one-way communication by modifying WiFi transmitters to emulate Bluetooth transmitters. Unlike these other CTCs, FLEW enables bi-directional communication between WiFi and Bluetooth by directly enabling conventional WiFi operations on FSK devices (i.e., the hardware of Bluetooth devices). It makes an FSK device appear as a standard WiFi device and the FSK device adheres to the conventional WiFi protocol. The modified FSK device can then connect to unmodified WiFi APs and use the standard WiFi encryption. Because the FSK device behaves like a standard WiFi device, users do not need to modify the WiFi AP or infrastructure, and no IoT gateway is needed either.

Despite its many salient features, FLEW also has a few shortcomings. In order to transmit WiFi waveforms using an FSK device, FLEW requires hardware modifications to inject the WiFi waveform into the mixers of the FSK chip. Furthermore, this waveform injection (along with all other packet processing) requires an external microprocessor. So, FLEW needs at least 2 chips. (In fact, all evaluations in FLEW use 3 chips.)

In this chapter, we present a novel CTC solution, called `Unify`, which transforms modern BLE/FSK SoCs into fully operational WiFi SoCs **without any hardware modification**. `Unify` transmits and receives conventional WiFi packets. Because `Unify` behaves just like an off-the-shelf WiFi device, it can directly connect to unmodified, conventional WiFi APs. Unlike FLEW that relies on multiple chips, `Unify` is a **single-chip solution**, and hence is smaller, cheaper and more power-efficient.

`Unify` can directly run on very popular and widely-deployed BLE/FSK SoCs (CC254x) made by Texas Instruments (TI). For example, the CC2544 SoC is widely used in Logitech's Unifying dongles. Fig. 4.1a shows that `Unify` turns a Unifying mouse dongle into a proper WiFi dongle, which provides direct WiFi connection for the entire laptop. In another example, the CC2541 SoC is widely used in numerous highly popular BLE modules, such as the HM-10, HM-11, JDY-06 and JDY-08 modules. In Fig. 4.1b, `Unify` is shown to transform a CC2541 keyfob into a WiFi thermostat, which periodically uploads temperature readings directly to the Internet/cloud.

Enabling bi-directional Bluetooth–WiFi communication without any hardware modification is a significant advantage of `Unify`. Specifically, hardware modifications require either manufacturing brand-new devices, or retrieving and modifying the circuit board of the device already deployed in the field. In contrast, `Unify` only needs a firmware update and the circuit board need not be modified, which greatly simplifies and reduces the deployment difficulty. More importantly, some CC254x SoCs are capable of over-the-air (OTA) firmware upgrade [79]. That is, this firmware update process could be done wirelessly and even remotely, without ever physically accessing the device.

The single-chip operation of `Unify` allows for significantly smaller, cheaper, and more power-efficient devices, even when compared to FLEW. For example, CC2544 uses the QFN32 [80] package with a dimension of 5mm×5mm, which is already smaller than the FSK transceiver chip (QFN48 [63]) used in FLEW. FLEW also requires an additional microprocessor (LQFP80 [64], 12mm×12mm) for a complete operation. Also, CC2544 is as cheap as $1.2 USD [81] and the HM-11 module is available for less than $2 USD. Both are considerably cheaper than the FSK chip used in FLEW. Finally, as we will show in Sec. 5.3.7, `Unify` consumes much less power than not only standard WiFi chips but also FLEW.

`Unify` is groundbreaking in that *WiFi connectivity can be achieved with the same device complexity/cost and with the same power budget as BLE connectivity*. We achieve this vision with three key technical concepts: a) streaming DAC IQ samples with an innovative use of DMA, b) capturing FSK signals with SPI, and c) satisfying WiFi timings with power overrides. These concepts are directly applicable to the CC254x SoC. In addition, outputting FSK signals and using power overrides are common on BLE SoCs for the purpose of BLE certification. Therefore, these techniques can be easily applied to other BLE SoCs if the register map of the SoC is known. Some vendors consider the register maps proprietary information and thus the configuration procedure is not publicly available. However, the vendors themselves or trusted developers should have access and can thus apply the techniques. `Unify` transmits WiFi waveforms with an innovative DMA scheduling. This concept can be generalized as using the DMA to control the phase of the carrier on BLE SoCs. For example, we can apply our concept to other BLE SoCs by using and scheduling the DMA to efficiently control the radio configuration, such as the PLL or the AFE (analog front end) settings.

We develop `Unify` entirely from publicly available information and we do not use any proprietary information.

4.2 System Design

4.2.1 SoC: Challenges and Opportunities

There are numerous technical challenges in transforming BLE/FSK SoCs into WiFi SoCs. Unlike the highly configurable FSK transceiver chip used in FLEW, BLE/FSK SoCs abstract away low-level configurations and have less flexibility in customizing radio protocols. For example, CC254x

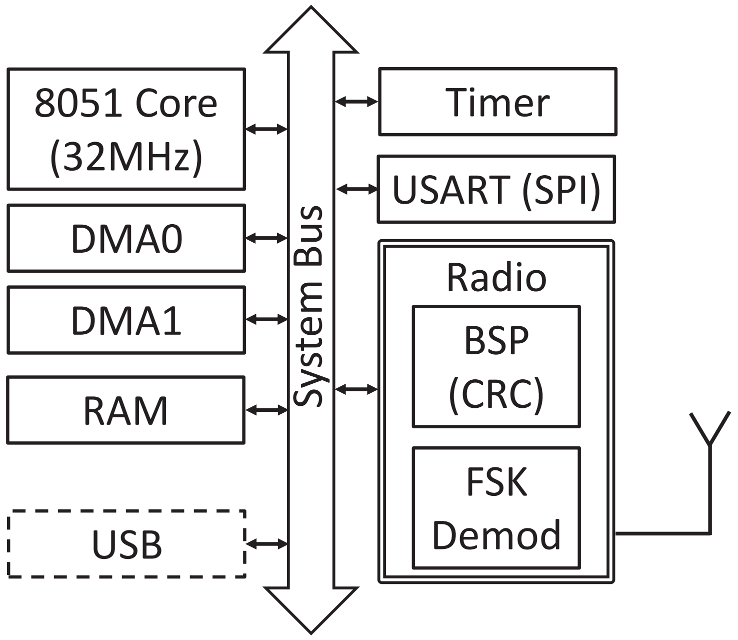


Figure 4.2: Simplified block diagram of CC254x SoC.

does not have the infinite receive mode used in FLEW. Moreover, since SoCs integrate everything on a single chip, the connections between a processor and the radio circuits are internal, thus preventing use of certain techniques that are applicable only when the processor and the radio circuits are two separate chips. For example, FLEW implements WiFi transmission by using an external processor to inject digital waveforms into the analog pins on the FSK transceiver. Such a transmission design is not feasible on CC254x.

On the other hand, the SoC includes a processor core, a memory bus, and a number of peripherals. Fig. 4.2 shows the components (utilized by `Unify`) on a CC254x chip. By leveraging the different functions provided by the peripherals and because of the much faster access to the radio or peripheral registers, these resources create a unique opportunity for `Unify` to overcome numerous hardware challenges and perform standard WiFi operations on SoCs without any hardware modification.

4.2.2 Design Goal

The ultimate goal of `Unify` is to make BLE/FSK SoCs behave like standard WiFi chips. To achieve this goal, the SoC must **transmit** and **receive** standard, un-coded WiFi packets. We take a bottom-up approach to designing both the WiFi transmission (Sec. 4.2.3) and the WiFi reception (Sec. 4.2.4). We first design transmission/reception of the physical waveform of a single WiFi bit. We then design robust mechanisms for transmitting and receiving complete packets. Sec. 4.2.5 presents an innovative way of drastically reducing the turnaround times between transmission and reception, which is critical for meeting the WiFi timing requirements. Finally, Sec. 4.2.6 puts all components together and schedules the Tx and the Rx to transform BLE/FSK SoCs into WiFi SoCs.

4.2.3 Transmitting WiFi Packets

To transmit a WiFi packet, `Unify` divides its physical waveform into segments, each of which is copied to the DAC registers for radio transmission. The segment copying is done by precise, robust scheduling of two DMA controllers.

4.2.3.1 DAC Registers

Similar to earlier TI transceivers, CC254x chips have registers for controlling transmission DACs. When set appropriately, these registers allow the transmitting IQ samples to be overridden by software. Although these registers are not documented for CC254x chips, we were able to pinpoint them among the undocumented memory locations and verify that the IQ overrides function correctly. Specifically, the IQ overrides are activated by loading a value of 41 to location 0x61AA (in the 8051 XDATA region), and the actual I and Q overrides are located at 0x61A7 and 0x61A8. We also find that the DACs do not seem to be clock-gated. That is, any I/Q overrides are directly reflected in the transmitted waveforms after memory stores, and no synchronization between the DAC and the memory bus is required. Conceptually, DACs are always synchronized with memory bus.

For CC254x, these DAC registers, unlike those in older TI transceivers, provide a unique opportunity to transmit WiFi waveforms. Specifically, since CC254x are SoCs, DAC registers can be updated at a high frequency. In contrast, older TI transceivers are not SoCs and register writes require using much slower inter-chip communications, such as an SPI bus, and thus the register updates on older transceivers are too slow to be useful for WiFi waveforms.

To transmit proper WiFi waveforms, the DAC registers must be updated fast enough. Moreover, to reliably transmit WiFi packets from CC254x to WiFi devices every time, these register updates should have precise, predictable timings and any glitch should be eliminated. Transmitting one WiFi packet involves thousands of consecutive writes to the DAC register. Missing or duplicating one memory write results in the shifting of all subsequent memory writes (and therefore, the transmitted WiFi waveforms), which will cause bit misalignment between CC254x and WiFi devices, resulting in packet errors.

A straightforward way to update the DAC registers is to use the 8051 on the SoC. The 8051 CPU on CC254x is a TI-enhanced variant with a typical 8051 memory layout (RAM, SFR, XDATA, etc.) and reduced clock cycles for the instructions (as opposed to 12 cycles of the original 8051 [82]). The CPU runs at 32MHz. We have extensively explored use of the 8051 CPU for writing DAC registers that transmit WiFi waveforms. Although we find it possible to transmit a WiFi packet by our highly-optimized assembly code and with the CPU set to the real-time mode, glitches are observed and a less precise WiFi waveform needs to be used because of longer cycles of accessing the XDATA memory. So, we conclude that using the CPU does not meet the speed and reliable timing requirements for transmitting WiFi packets.

4.2.3.2 DMA Controllers

Fortunately, CC254x is equipped with DMA controllers. Conceptually, a DMA controller is hardware that implements `memcpy()`. Therefore, a WiFi IQ waveform can first be stored in memory and then a DMA controller copies (and therefore transmits) the IQ waveform to the DAC registers. With this method, no CPU intervention is needed throughout the transmission of WiFi packets. Also, since the memory reads and writes are implemented in hardware, using the DMA controller enables much faster DAC register updates, and thus the transmission of precise WiFi waveforms. Moreover, the memory access priority of the DMA controllers on CC254x is configurable. With the priority set to high, the DMA memory access always gets priority over the CPU access, thus guaranteeing precise, glitch-free timing. We find that the DMA controllers actually have a hidden level ("absolute") with an even higher priority and use this level for copying IQ samples to DAC registers.

Similar to FLEW, `Unify` transmits DSSS BPSK WiFi waveforms, which is the most robust WiFi modulation format. The DSSS BPSK modulation can be viewed as typical BPSK modulations at a higher speed, which can be implemented by updating I samples only and keeping the Q samples to midpoint (i.e., Q-branch is always 0 on the constellation plane). While it is possible to use a single DMA controller to update both I and Q samples, we choose to implement BPSK by only updating the I samples. This is because the I and Q samples are in separate registers and updating both samples reduces the update rate by half.

By only modulating the I-branch, the spectrum of the transmitted waveform is symmetric. The standard DSSS WiFi waveform can be generated by modulating the I-branch at 11MHz. This corresponds to the 11-bit Barker sequence the WiFi DSSS uses. That is, after scrambling and differential coding, a WiFi packet is encoded in an 1Mbps bitstream. For any 1's in the bitstream, the Barker sequence [1,-1,1,1,-1,1,1,1,-1,-1,-1] is sent to the I-branch at 11Mbps. For any 0's, its complement ([-1,1,-1,-1,1,-1,-1,-1,1,1,1]) is used.

In `Unify`, the DMA updates the I samples at 8MSps. Therefore, a sequence, resampled in the phase domain from the Barker sequence, is used. Specifically, for 1's in the 1Mbps bitstream, [1,-1,1,-1,1,1,-1,-1] is sequentially sent to the I registers using DMA. For 0's, [-1,1,-1,1,-1,-1,1,1] is used. Let the maximum of the autocorrelation of the Barker sequence be 100%, then the maximum of the cross-correlation between the Barker sequence and the resampled sequence (when both are upsampled to 88MHz) is 81%, indicating that they are highly similar. Technically, the resampled sequence has a smaller bandwidth (modulating at 8MHz instead of at 11MHz). However, since conventional WiFi systems are designed to withstand interference from nearby channels and since the high frequency components near channel boundaries are already attenuated either by the circuits or the channel filter in practice, this resampled sequence works properly and reliably with chips from all major WiFi vendors.

In theory, WiFi packets can be transmitted by updating the DAC registers using a single DMA transfer. However, this is infeasible in practice due to memory and processing constraints. CC254x only has 2~8kB of SRAM. In `Unify`, each WiFi bit is represented by 8 bytes of I data. For sending a 200 byte WiFi packet, a contiguous memory of $200 \cdot 8 \cdot 8 = 12.8$kB is required, which far exceeds the available memory on these low-cost chips. Furthermore, even if an unlimited amount of SRAM is available, generating and putting all the I data in the memory in the first place would take too much time for the 8051 core, resulting in low throughputs.

4.2.3.3 Multiple DMA Scheduling

To overcome these limitations, we must reuse the memory regions, meaning that we must configure and trigger multiple DMA transfers during the transmission of a single WiFi packet. Conceptually, each DMA transfer sends a small segment of the WiFi packet and the whole WiFi packet is transmitted by concatenating all DMA-transferred segments.

Using multiple DMA transfers and concatenating them introduce new challenges. Instead of configuring and then triggering one DMA transfer, we need to configure multiple DMA transfers on the fly, and each transfer should be triggered precisely on time. If any DMA transfer is not triggered precisely at the scheduled instant, the waveform segment sent during the DMA transfer is shifted and corrupted. If a DMA transfer is not correctly configured before its triggering, an incorrect waveform segment will be sent.

We address these challenges by devising a robust mechanism that schedules the DMA transfers without any CPU involvement during transmission. The scheduling is very tricky to design but works elegantly and reliably in `Unify` as described below.

Our solution uses a timer and two DMA controllers on CC254x. One DMA controller (DMA0) periodically copies a waveform segment from a memory region to the DAC register. Since waveform segments must be copied to the DAC register starting at precise time instants, each transfer issued by DMA0 must start at a precise time instant. This is guaranteed by triggering DMA0 in hardware using a timer on CC254x. Note that a waveform segment does not have to contain the waveform of just one WiFi bit. In fact, `Unify` triggers DMA0 every 4μs, corresponding to the duration of 4 WiFi bits. The basic idea is that, depending on the WiFi bits to transmit, DMA0 will copy a waveform segment from different memory locations (by modifying the DMA0 configuration). The memory region is pre-loaded with all possible waveform segments corresponding to all bit permutations within the duration of a waveform segment.

After a segment (corresponding to a group of WiFi bits) is transferred via DMA0, the configuration of DMA0 must be updated to transmit the next segment. Specifically, the source memory address of DMA0 must be updated to point to the starting location of the next waveform segment. On CC254x, DMA is configured by specifying a DMA descriptor. The DMA descriptor itself is actually 8 bytes of memory in the XDATA space. Another key idea is that, since the descriptor of DMA0 is also a memory region, we can use a second DMA (DMA1) to keep updating the source memory addresses of DMA0. Additionally, because DMA1 can be hardware-triggered upon completion of each DMA0 transfer, the address update happens exactly once after each DMA0 transfer with guaranteed ordering. While the 8051 core, instead of DMA1, could theoretically be used for updating addresses, we find the timing is not reliable using the 8051. Also, the DMA0 triggering is too fast to allow interrupt-based designs for updating the descriptor.

Since DMA0 transfers are the most time-critical, the priority of DMA0 is set to"absolute" while the priority of DMA1 is "high". DMA0 is also set to repeated block mode where DMA0 transfers a block of data once triggered, and after each transfer, DMA0 automatically reads the new descriptor and waits for the next trigger.

A small turnaround time is required between DMA0 transfers. (This turnaround time is still present if, alternatively, DMA0 and DMA1 are interleaved to update the DAC register.) Therefore, although 4μs corresponds to 32 memory accesses, the block size of DMA0 is set to 28. During the remaining 4 cycles, the DAC register is not updated and the waveform stays unchanged. To account for this, DMA0 transfers are not aligned with the bit boundaries of WiFi bitstreams. Each transfer spans 5 WiFi bits where the first cycle in each transfer copies the last I sample of the first WiFi bit, and the last 3 cycles copy the first 3 I samples of the last WiFi bit. That is, the DMA0 transfer is repeated every 4 WiFi bits and the waveform for one of these 4 bits is transmitted by the last 3 cycles of a transfer and the first 1 cycle of the next transfer. The 4 cycles of idle time in between are used for turnaround.

4.2.3.4 Waveform Segments

DMA0 copies I samples from a memory region in which every possible waveform segment within 4μs can be found. A naive approach to storing all these segments is to store each segment separately in a non-overlapping manner. Since every possible 5 WiFi bit permutation needs an entry, the overall required memory is $2^5 \cdot 28 = 896$ bytes. While this memory size is within the limits of CC254x chips, it is still relatively large. More importantly, since a memory region of 896 bytes cannot be indexed using a single byte, DMA1 needs to update the source address (in the DMA0 descriptor) in two bytes after each DMA0 transfer and each address waiting to be copied to DMA0 descriptor by DMA1 needs to be stored in two bytes of the main memory.

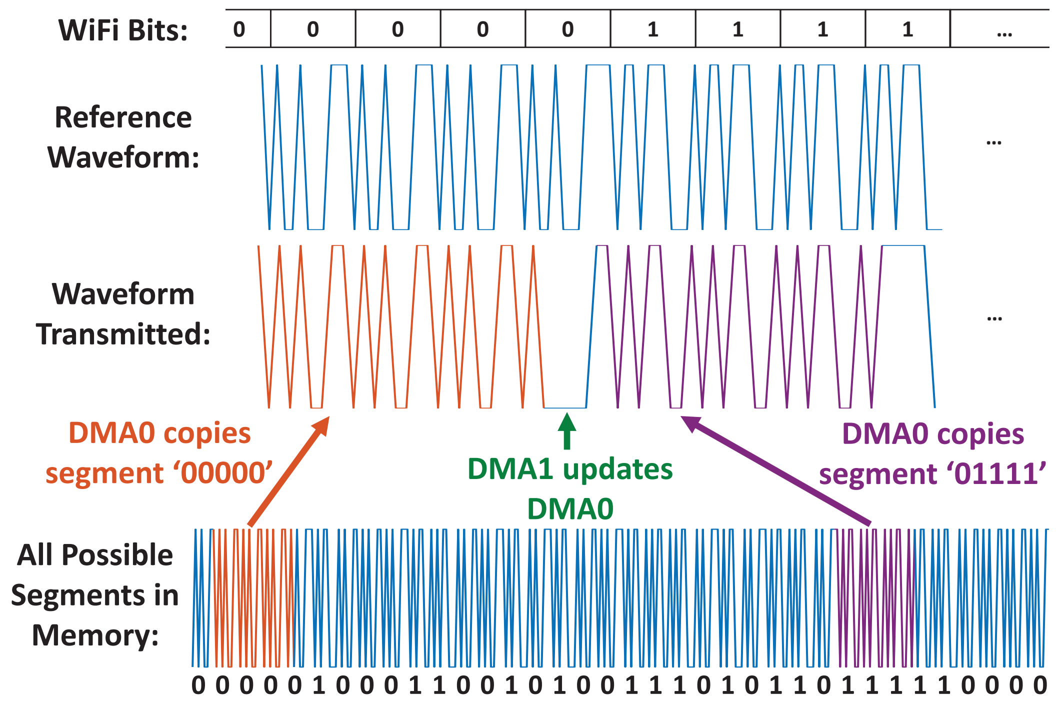


Figure 4.3: Transmit waveforms using 2 DMAs.

A better approach is to store all segments in the memory with maximum overlaps between segments, resulting in the minimum required space. This approach makes use of the de Bruijn sequence [83]. Specifically, every 5-bit permutation is a subsequence of "000001000110010100111010110111110000" and we can store all possible segments in this manner. Each '0' or '1' in this sequence represents one WiFi bit, which actually occupies 8 bytes (representing 8 I samples) in the memory. (That is, [1,-1,1,-1,1,1,-1,-1] for '1' and [-1,1,-1,1,-1,-1,1,1] for '0'.) With these stored segments, we can transmit a WiFi bitstream by setting, using DMA1, a series of appropriate offsets as the source address of DMA0. Let us consider Fig. 4.3 as an example and suppose the last WiFi bit in the last iteration is '0' and we want to transmit WiFi bits '0000', then we need to specify the offset for '00000', which is 0. However, since we only transmit the last I sample (i.e., skipping the first 7 samples) of the first WiFi bit, the offset is 7. Suppose the WiFi bits after '0000' is '1111', we need to specify the offset for '01111', which is $7+26\cdot 8=215$. Since the maximum offset is $7+31\cdot 8=255$, each offset can be represented with just one byte. Therefore, DMA1 only needs to update a single byte of the DMA0 descriptor each time and each offset waiting to be copied by DMA1 occupies only 1 byte in the main memory.

The ability to use a single byte to store offsets is also the reason why DMA0 is triggered every 4μs, since 4μs is the longest interval that all possible waveform segments for every bit permutation can still be addressed by a single byte.

4.2.4 Receiving WiFi Packets

To receive standard, un-coded WiFi packets, `Unify` uses the radio circuit and the FSK demodulator to receive raw WiFi bits. `Unify` also reuses the SFD detector on CC254x, which is intended to detect FSK/BLE packets, to detect standard WiFi packets. Upon detection of a WiFi preamble, `Unify` starts collecting raw WiFi bits and descrambling the incoming bit stream to recover each byte in the WiFi packet. CRC checks are performed on every WiFi packet, which is critical for standard WiFi operations. All this processing needs to be done in real time on CC254x, which is only equipped with a low-power, 8-bit 8051. Although receiving standard WiFi packets is usually assumed to be computationally-intensive, `Unify` achieves this goal by leveraging recent signal processing discoveries, by reusing certain hardware accelerators on CC254x and by optimizing the codes run on 8051.

4.2.4.1 Receive a raw WiFi bit

To receive a raw WiFi bit, `Unify` leverages a simple and known technique used in the latest WiFi–FSK CTC works [78, 74, 77]: FSK demodulators, operated at a relative frequency offset, can receive WiFi DSSS bits. `Unify` operates the FSK receivers at -3MHz (relative to the center frequency of the selected WiFi channel) with an intermediate frequency of -1MHz. The frequency offset is chosen based on the receive performance. Although the latest WiFi–FSK CTC results show that a fractional frequency offset should ideally be used, the frequency can only be set in 1MHz increments/decrements on CC254x. To address this, the frequency offset compensation feature on CC254x is used, which provides additional fractional frequency offsets based on the received waveforms. A similar technique can be found in [74]. `Unify` configures CC254x to continuously track and compensate for any frequency offset before an SFD is detected. Once an SFD is received, the frequency offset is frozen and this estimate is used to demodulate the rest of a packet. This setting yields the best performance among all different frequency compensation configurations.

Although the above techniques are useful in demodulating raw WiFi bits, they do not enable CC254x to directly receive WiFi packets. In particular, since the bit decoding and packet logic on CC254x are designed for BLE/FSK packets, they cannot be directly used for conventional WiFi packets. The prior work of [74], which enables one-way WiFi-to-BLE communication, circumvents these limitations by pre-processing and pre-coding WiFi packets in the form of BLE packets. However, this method would require modifications on every WiFi device from which the BLE chip receives packets.

We design `Unify` to directly detect and decode standard WiFi packets. This eliminates the need for pre-coding WiFi packets so that `Unify` can work with conventional WiFi devices. To achieve this goal, we must address other challenges imposed by the hardware.

4.2.4.2 Packet Detection

To detect WiFi packets using FSK/BLE packet logics, CC254x is configured to detect the standard WiFi preamble from the received raw WiFi bitstream. Although detecting WiFi preambles can, in theory, be implemented in software using sliding windows, it is computationally prohibitive, especially on an 8-bit 32MHz CPU.

`Unify` exploits the BLE/FSK SFD matching hardware on CC254x for the detection of WiFi packets. The SFD to be matched is set to a bit sequence within the scrambled WiFi SYNC field. This technique can be found in FLEW. However, `Unify` uses different SFD and SFD length because of the frequency compensation requirements. The SFD used in FLEW is 0x05AE4701. In standard WiFi packets, the 8-bit pattern before 0x05AE4701 is '11010101'. On the other hand, CC254x uses the SFD, along with the 8 bits received before SFD, to estimate frequency offsets. For CC254x, the 8-bit pattern should be either '01010101' or '10101010', and receiving '11010101' lowers the accuracy of the frequency estimates.

We meet this requirement by using 0x0B5C8E03 as the SFD, since the WiFi bit pattern is '10101010' before 0x0B5C8E03. CC254x is configured to expect this bit pattern before SFD. 0x05AE4701 is 0x0B5C8E03 shifted right by 1 bit. Therefore, the length of SFD is set to 31 so as to align the subsequent bytes to the appropriate byte boundaries.

With appropriate bit descrambling, the WiFi PLCP starts 5 bytes (40 bits) after the SFD (0x05AE4701). This holds for packets conforming to the 802.11 standard [30]. As described in [78], Realtek's chips are buggy in that their PLCP (and subsequent bytes) are 2 bits early (38 bits after bit descrambling). For connecting to Realtek devices, `Unify` sets the length of SFD to 29 in order to automatically align the bytes without any software processing.

4.2.4.3 Packet Length

The length of a WiFi packet can be determined after parsing the PLCP header. `Unify` also performs sanity checks on the PLCP to prevent erroneous or unsupported PLCP affecting the packet reception.

The hardware limitations of CC254x make it very difficult to decode standard WiFi packets, especially due to the packet-length limitation. Since we are not pre-coding WiFi packets with BLE packet format, we cannot use CC254x's packet-length parsing, which determines the packet length and adjusts the reception duration accordingly. If the length parsing is not used, we need to use the fixed length packet format, which requires specifying the length before starting each packet reception. This is infeasible because the WiFi packet lengths are variable and we do not know the length of each packet until its PLCP is received.

In addition, the packet logic (with or without length parsing) only supports packets up to 255 bytes and CC254x immediately terminates the reception afterwards. Conventional WiFi packets easily exceed this size.

We use a special design that solves the size limitation and dynamically adjusts the reception duration. The key idea is to prevent the packet logic from terminating the reception before a complete WiFi packet is received. From extensive experiments, we find this achievable by forcing the packet logic "stuck" at its initial state. Specifically, we find that after an SFD is received, we can arbitrarily prolong the reception by periodically overwriting one register in an undocumented RAM region the packet logic uses. The register is mapped to 0x607E in the XDATA space when the RF memory page is set to 5 (RFRAMCFG = 5). By constantly overwriting 0 to 0x607E, it prevents the packet logic from terminating the reception. Furthermore, by controlling the total duration of overwriting 0, the reception duration of a WiFi packet can be adjusted dynamically after its PLCP is received.

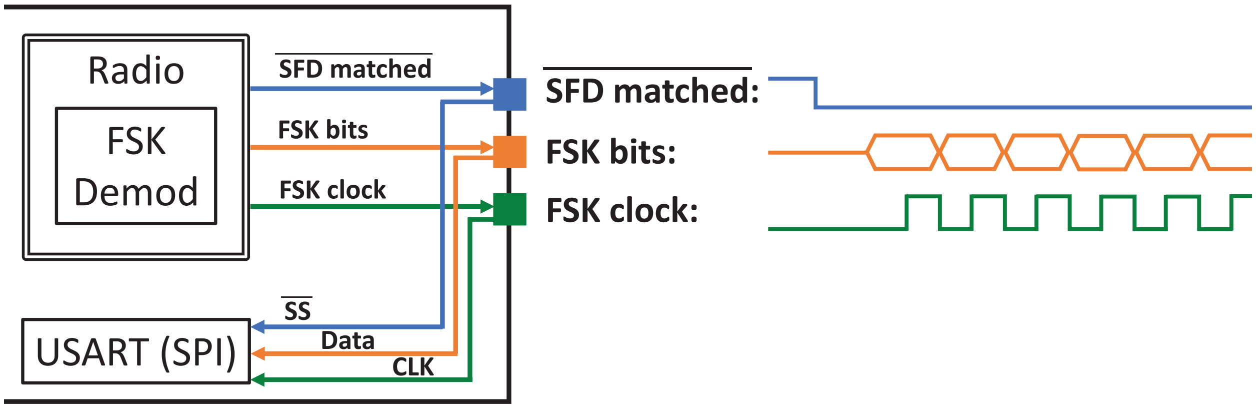


Figure 4.4: Receiving FSK bits with SPI.

One side effect of overwriting the packet logic's register is that the packet buffer goes haywire. Consequently, the raw WiFi bits cannot be read out from the packet buffer. To address this problem, we utilize CC254x's RF observation signals, which are various signals from the FSK demodulator. By selecting "demodulated bits", "clock" and "SFD matched", these signals emulate an SPI sender. We can then use an SPI receiver (USART hardware on CC254x) to collect the demodulated bits (the raw WiFi bit stream), as shown in Fig. 4.4.

While it might seem that external wires are needed to connect the SPI sender to the SPI receiver, `Unify` does not require such external connections. We find that the peripheral inputs are (implicitly) always connected to the pins, even when those pins are configured as outputs. Therefore, by using the IO matrix to route the RF observation and SPI receiver signals to the same pins, they are connected without any external connections.

4.2.4.4 Packet Decoding & CRC

After raw WiFi bits are received from SPI, `Unify` descrambles the bit stream. `Unify` performs descrambling immediately upon receiving each byte. The WiFi descrambling can be simplified as performing XOR on 3 shifted versions of the bit stream. We optimize the descrambling as 8-bit computations (as opposed to shifting and XOR'ing three 16-bit integers). We devise the following C code to produce each WiFi byte (`output`):

```
sb = <one byte received from SPI, LSB first>;
output = lb ^ (lb>>7) ^ (lb>>3) ^ (sb<<1) ^ (sb<<5);
lb = sb.
```

Further optimization is possible. Unlike ARM's shift instructions, rotate left/right on 8051 only shifts 1 bit at a time and extra cycles are required to mask bit-rotations to get logical shifts. Consequently, the second line in the above snippet takes approximately 31 cycles (depending on the register allocation) using the IAR compiler. This line can be replaced by an optimized assembly routine. The key idea is to construct two lookup tables in the 8051 CODE space to store the results for `lb ^ (lb>>7) ^ (lb>>3)` and `(sb<<1) ^ (sb<<5)`. During descrambling, we can load `lb` (or `sb`) into `A` and use the `MOVC` instruction to directly fetch the results. The number of cycles is reduced to 17 using this method. We process the WiFi PSDU (excluding the CRC) with this optimization.

CRC must be performed on WiFi packets, since the CRC result is used to determine whether an ACK should be transmitted after 10 μs. However, CRC32 is simply too computationally intensive to run on the 8-bit 8051 at WiFi speed. Fortunately, CC254x has a configurable hardware CRC module (known as the BSP co-processor). We devise an appropriate initialization sequence for the CRC module. Specifically, the CRC32 polynomial is configured once at startup. After the SFD (0x0B5C8E03) is received *and just before the start of WiFi PLCP*, `Unify` initializes the CRC32 shift register (to 0xFFFFFFFF). Then, for each byte (excluding the last 4 bytes) in the WiFi PSDU, `Unify` sends the descrambled byte to the BSP. The 4 bytes received last are matched with the CRC calculated by the BSP. An ACK should be transmitted if the CRC matches.

The CC254x has a register called `FREQTUNE`. Contrary to its name, we find that it has little effect on the RF frequency. Instead, it fine-tunes the crystal oscillator. We find this register particularly useful in fine-tuning CC254x to receive long WiFi packets, since it compensates for the small timing offsets between CC254x and a WiFi device. This fine-tuning is needed only when large packets are expected and it only needs to run once for each WiFi device. We iterate 5 possible values and choose the best `FREQTUNE`.

4.2.5 Transitions between Tx and Rx

4.2.5.1 Rx to Tx Turnaround

Normal WiFi operation requires a device to respond (i.e., with ACK or CTS packets) almost instantly (10μs) after receiving packets. Meeting this WiFi turnaround time is highly challenging for `Unify`, since CC254x is not designed with such a short turnaround time. In particular, we find that the shortest turnaround time of CC254x, using the standard Rx to Tx transition, is around 130μs. This is far too long for any WiFi devices.

After extensive experiments, we fundamentally resolved this challenge. Our solution starts from a key insight that, since `Unify` transmits waveforms by directly controlling the DAC registers, we can initiate a transmission with minimum delay by directly turning on the PA, mixer and DAC. The power signals of these components can be overridden by modifying the "power down" registers. Although the locations of these registers on CC254x are completely undocumented, we are able to pinpoint the two registers after extensive experiments. Specifically, the transmit chain is turned on if 0x61AC (in the XDATA space) is set to 8.

By directly controlling the power signals, the Rx to Tx turnaround is reduced to approximately 10μs, thus satisfying the timing requirement of transmitting an ACK (or CTS) after receiving the corresponding packet. Unlike FLEW, `Unify` meets this timing requirement without truncating packet reception, and thus full 4 bytes of CRC32 are received, just like any conventional WiFi chip. We observe that terminating packet reception affects the RF carrier signal, and hence do not terminate the reception until an ACK is transmitted. That is, the reception is prolonged to receive the full WiFi packet, plus 10μs, plus the duration of an ACK.

4.2.5.2 Tx to Rx Turnaround

In the opposite direction, after `Unify` sends a packet to an AP, the AP will send an ACK after 10μs if the AP properly receives the packet. While it is technically possible that a WiFi system is functional without ACK detection, such a design would have very poor performance because a reliable re-transmission mechanism cannot be implemented without detecting ACKs. However, 10μs is also a far too short turnaround time for CC254x.

To address this challenge, we use a similar idea, except we change the order of Tx and Rx. That is, to send a packet, `Unify` initializes reception, sets the Tx frequency, and then immediately overwrites the power registers (which block the signal from going into the receive chain). In addition to loading 8 to 0x61AC, 32 is loaded to 0x61AB to reduce power consumption. Once the transmission is completed, DMA1 becomes inactive. After detecting this, the 8051 cancels DMA0 transfers and resumes the highest access priority. Afterwards, it sets the Rx frequency, sets both 0x61AB and 0x61AC to 0, and waits for a packet (which should be an ACK). The key idea is that by the time `Unify` finishes transmission, the receive circuit will be primed and ready for the next packet. All we need to do is ”unblock” the receive signal by removing the power overrides, and a fast Tx to Rx transition is achieved. The timing of this control logic is not very tight since 10μs leaves enough room for the 32MHz 8051. (The main source of delay was the receive circuit, not the 8051.)

The Tx to Rx turnaround can be followed by an Rx to Tx turnaround. For example, if a unicast packet (instead of an ACK) is received after sending a packet, `Unify` will receive the unicast packet and transmit an ACK (after $\sim$10μs).

4.2.6 MAC Layer

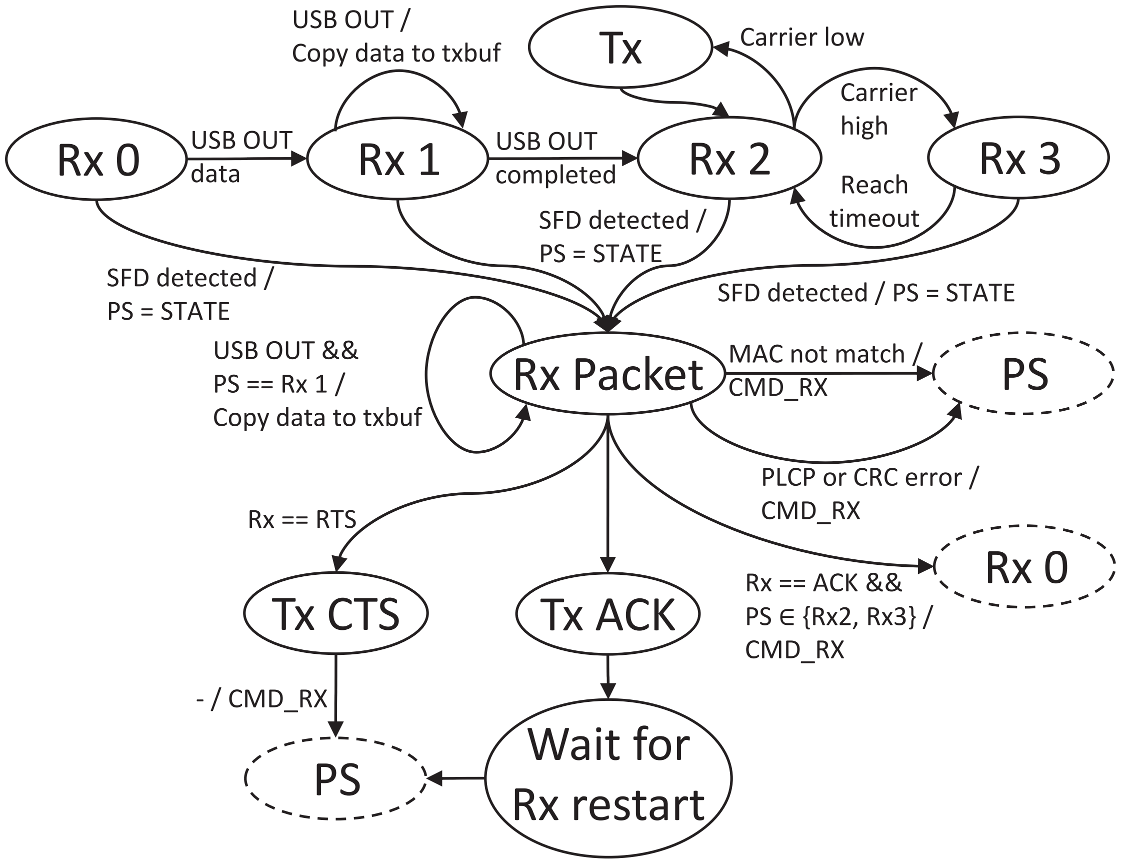


Figure 4.5: The FSM of `Unify`.

The finite state machine (FSM) of `Unify` consolidates all important components, schedules transmission and reception in a half-duplex manner, and implements the MAC-layer functions of WiFi.

Fig. 5.10 shows the high-level FSM of `Unify`. After initialization, a `CMD_RX` is issued and `Unify` enters the Rx 0 state. In Rx 0~3, the hardware is in receiving mode and waiting for a valid SFD. These states differ only in software. From Rx 0 to Rx 2, Unify copies and prepares the WiFi data until a complete WiFi packet (in the form of offsets) is ready to be transmitted in the memory. State Rx 3 represents that the packet is ready but the channel is busy.

Unify follows CSMA/CA, which is the fundamental medium access mechanism in WiFi. Before each packet transmission, the Rx 2 state senses the channel by monitoring the RSSI value. If an idle channel is observed, Unify transmits a packet. After Tx, Unify returns to Rx 2. If the connecting AP sends an ACK, Rx 2 can detect the SFD and transition to Rx Packet. Otherwise, Unify initiates re-transmissions.

Rx Packet is the main state that collects and processes (descrambling and CRC) the bitstream and extends the reception to a complete packet. Rx Packet provides information (the frame type, address and CRC results) so that necessary actions are performed thereafter. For packets with a wrong MAC address or other errors, Unify simply returns to the last state ("PS") before Rx Packet. If an ACK is received after packet transmission, Unify goes to Rx 0 to prepare the next packet for transmission. CTS is transmitted if RTS is received. If a unicast packet with a matched MAC address is received, Unify transmit an ACK. Then, Unify does *not* issue a new CMD_RX because each CMD_RX is configured as repeated with fast warm-up. Since Rx Packet to Tx ACK is the normal reception flow, the receiver will become ready faster by simply waiting for the repeated reception to start.

We also include an optimization in Rx Packet. When Unify is processing the incoming WiFi bit stream from the radio, the 8051 also copies the outgoing WiFi bytes from the USB buffer to the memory. This allows Unify to go to Rx 2 much faster after reception.

4.3 Implementation

4.3.1 Hardware

To directly compare Unify with SOTA CTC, we implement Unify on the CC2544 since it has built-in USB. We use Logitech Unifying dongles as the hardware. We do not modify the hardware of Logitech Unifying dongles. We simply load the Unifying dongle with Unify firmware and WiFi connectivity is achieved.

4.3.2 Firmware

The firmware of Unify is developed from scratch. This includes in-house USB firmware codes that are completely developed from the ground up. The firmware is compiled with IAR EW8051

[84].

After USB initialization and exchanging USB standard requests, the firmware initializes the radio hardware. The 8-bit Timer 3 is initialized to provide a constant 4μs trigger. The radio is set to the fixed length mode (with a length of 120) and repeated reception with the "synthesizer on" option. The actual reception duration will be dynamically adjusted at run time. The AGC is off and the radio is manually set to the maximum gain. The firmware initializes 1 DMA0 and 3 DMA1 descriptors. Three descriptors are for transmitting a normal packet, an ACK and an CTS, respectively. This allows `Unify` to switch between different transmissions by simply specifying a different descriptor for DMA1.

The firmware also initializes an XDATA region that stores all possible waveform segments to be sent by DMA0. Using the IAR compiler, this XDATA region always starts at 0x0001. A small optimization is used here. In Sec. 4.2.3, the lowest possible offset (index) is 7. Therefore, we generate all possible waveform segments, truncate the very first 7 bytes and store them in XDATA starting at 0x0001. All offsets are adjusted (i.e., subtracting 6) accordingly.

After initializations, the firmware enters the FSM loop. If any USB error (e.g., USB reset) is detected, the firmware jumps to 0x0000 and the USB initialization follows.

4.3.3 Firmware Update

The `Unify` firmware can always be loaded to CC254x using a CC254x programmer. The official CC Debugger [85] from TI can be used. Alternatively, an Arduino board can be used as the debugger [86]. We also design the `Unify` firmware so that Unifying dongles with a compatible bootloader can be directly updated via USB. Specifically, the bootloader on the Unifying dongles in our possession occupies 0x0000~0x03FF and 0x7400~0x7EBF (in the CODE space). A compatible `Unify` firmware is first generated by placing the `Unify` codes starting at 0x052C. Then, all instructions between 0x0400~0x052B are replaced by jumps to 0x056D, which is the entry point to the `Unify` codes. CRC16 is calculated over 0x0400~0x6BF9 and is placed at 0x6BFA. Finally, CRC16 is followed by a magic string (0xFE,0xC0,0xAD,0xDE). With this code layout, `Unify` firmware can be flashed to compatible dongles using USB flashing tools for Unifying dongles. Finally, our firmware is also designed to be able to revert back to the original Unifying firmware.

The `Unify` firmware focuses on WiFi communication. TI developed a "Boot Image Manager", which allows running multiple firmware [87]. Alternatively, BLE functions can later be added to our firmware. Therefore, BLE and WiFi communication can both be supported with a single SoC.

4.3.4 WiFi Driver

The WiFi driver of `Unify` is similar to FLEW and interfaces the firmware with the `mac80211` module in the Linux kernel. Similar to FLEW, the transmission path converts the WiFi packets to WiFi (phase) bit streams. Then, every 4 bits (and the one bit that precedes them) are converted, using a simple lookup table, to an offset that points to the corresponding waveform segment. All offsets are sent to the firmware for transmission. In the receiving path, the firmware sends descrambled WiFi packets, along with two status bytes for each packet, to the driver. The status bytes include a CRC flag indicating if there is a packet error. The driver simply checks this flag and either relays a packet to `mac80211` or ignores an error packet. In the driver, we set the MTU to avoid overflowing CC2544's memory and to limit the sampling offsets. For these purposes, we can use an MTU of 256, which is the minimum MTU setting on the modern Linux. However, we use a higher MTU (552) because it increases throughputs and 552 was the minimum MTU of Linux for a long time [88, 89, 90, 91].

4.4 Evaluation

4.4.1 Experimental Setup

Table 5.2 shows the WiFi devices and APs used in the evaluation of `Unify`. The experimental setup is similar to that of FLEW. Various NICs from major WiFi chip makers are used for measuring physical-layer performance. A difference between our setup and that of FLEW is that all NICs use exactly the same antennas (on an HP 2570p laptop), making their performance directly comparable. Marvell does not seem to manufacture any standard half-length mini PCIe card and its chip does not support monitor mode well. Therefore, the evaluation is done at the system level. Various commodity WiFi APs are used for evaluating system-level performance. These system evaluations represent real-world use-cases where `Unify`, just like any typical WiFi device, directly connects to conventional WiFi APs. They characterize end-to-end performance where all aspects of WiFi protocol (e.g., re-transmissions, CSMA/CA, etc.) are taken into account.

All WiFi devices and APs are unmodified and all NICs use their default driver supplied with Ubuntu 20.04 LTS. We use WiFi channel 9 and the system evaluations (Sec. 5.3.3~4.4.5) use the WPA2-PSK encryption. This WiFi channel is chosen because it has the lowest activity in the test environment.

Table 4.1: WiFi chips and APs used in experiments.

Chip Maker	PHY Evaluation	System Evaluation
Atheros	AR9462	GL.iNet GL-AR150 (AR9331)
Broadcom	BCM4313	ASUS RT-AC66U (BCM4331)
Intel	Advanced-N 6205	TP-Link Archer AX3000 (WAV654A0)
Marvell	-	Linksys EA3500 (88W8366)
Ralink/ Mediatek	RT3290	TP-Link TL-WR841N (MT7628NN)
Realtek	RTL8188CE	D-Link DIR-619L (RTL8192ER)

Table 4.2: PER Evaluation (%)

Direction	WiFi to `Unify`			`Unify` to WiFi		
Distance	5m	10m	20m	5m	10m	20m
Atheros	0.44	0.73	2.64	2.47	3.83	4.00
Broadcom	5.25	6.30	6.40	2.88	3.10	3.81
Intel	5.52	8.35	9.74	1.78	3.37	3.74
Ralink	5.27	7.52	10.91	5.71	7.89	9.01
Realtek	1.25	3.08	5.40	12.82	13.21	13.26

4.4.2 PHY Layer and PER

We set `Unify` and NICs to the monitor mode and continuously send/receive standard WiFi packets with a PSDU of 564 bytes (including 4 bytes of CRC). In each setting, 4096 packets are sent so that each packet has a unique sequence number. On the receiver side, CRC checks are performed. We calculate the number of received packets with a correct CRC and the PER is defined as $1 - \frac{\text{\# of correct packets}}{4096}$.

Using `Unify` to receive standard WiFi packets transmitted by standard WiFi chips, Table 4.2 shows that transmission by (Qualcomm) Atheros has the best performance with less than 0.5% PER at 5m. The performance with Broadcom at 5m is similar to that of Intel and Ralink but the PER increases less with longer distances. Intel and Ralink have very similar performance across different distances with Intel being slightly better at 20m. Interestingly, Realtek has the second best PER performance. This is in part due to the fact that `Unify` uses a shorter SFD length (to compensate for Realtek's packet format bug), which distinguishes Realtek's packets from background interferences and yields good performance.

In the opposite direction, `Unify` transmits conventional WiFi packets and standard WiFi NICs receive them. Atheros and Broadcom have similar performance and the PER is $\leq$4% even at 20m. Intel actually shows the best receive performance in our testing. Ralink and Realtek chips have noticeably worse receive performance. This is consistent with the observations, reported in the prior work, of the inferior receive performance of Ralink and Realtek NICs. We believe these are in part due to the comparatively worse circuit or DSP designs. It is also possible that Realtek uses a non-standard WiFi SYNC field in the Rx chain (as it does in the Tx) and leads to the worst performance. `Unify` uses a shorter SFD length for Realtek tests in both directions, and this increases the possibility of packets transmitted by `Unify` collide with background interference since the shorter SFD length is for receiving Realtek's packets.

These physical-layer evaluations measure one-way, fixed-data-rate and no-retransmission performance. For packet transmission (i.e., WiFi to Unify), WiFi cards may have IQ imbalance, constellation imperfections, or timing or frequency drifts. For packet reception (i.e., Unify to WiFi), different vendors may use different preamble detection, bit demodulation, and RF circuit implementations, which can lead to noticeable performance differences. On the other hand, higher layer issues such as the rate adaptation algorithm and MAC layer implementations will not contribute to the performance differences at the physical layer.

4.4.3 TCP/UDP Throughput

Table 4.3: Throughput Evaluation (kbps)

Uplink						
Transport	TCP			UDP		
Distance	5m	10m	20m	5m	10m	20m
Atheros	421	420	412	471	479	459
Broadcom	397	405	391	461	455	444
Intel	390	388	369	446	438	415
Marvell	395	385	310	435	427	419
Ralink	401	395	387	438	441	437
Realtek	405	399	360	458	454	410
Downlink						
Transport	TCP			UDP		
Distance	5m	10m	20m	5m	10m	20m
Atheros	552	530	428	655	652	609
Broadcom	507	491	473	621	597	535
Intel	501	494	474	586	576	544
Marvell	425	424	395	512	520	479
Ralink	469	448	422	571	555	553
Realtek	475	468	451	570	560	527

In the system-level evaluation, `Unify` directly connects to unmodified WiFi APs. Using the standard `iperf3` [41] tool, we evaluate the transport-layer throughputs in both the uplink (`Unify` to AP) and downlink (AP to `Unify`) directions. We run an `iperf3` server on a Ubuntu laptop, which is connected (via Ethernet) to the first LAN port of the testing AP. `Unify` joins the AP's WiFi network and runs an `iperf3` client to measure the TCP and UDP throughputs in both directions. The measurements are always taken from the receiving end. Besides the MAC layer and packet retransmissions, physical-layer performances in both directions affect the throughputs since data packets are ACKed in the opposite direction. Different rate adaptation algorithms also affect data packets sent by an AP.

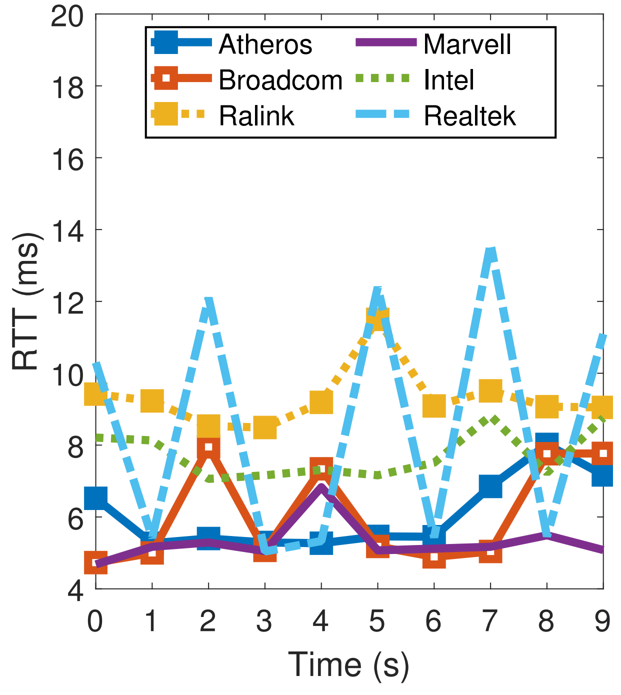
For uplink, Table 5.3.3 shows Atheros has the best performance, which is due to the superior overall (Tx and Rx) physical-layer performance. Broadcom also has good performance and similar to Atheros, the throughputs do not decrease much even at 20m. Intel and Marvell have similar performance but their throughputs tend to decrease more at 20m. The newer Mediatek chip inside the Ralink AP seems to have a better receive performance with little throughput decrease at 20m. Realtek is similar to Ralink at shorter distances but the throughput drops more at 20m. These decreases are caused by the inferior physical layer performance that requires more re-transmissions.

For downlink, Atheros shows the best performance in Table 5.3.3. Broadcom and Intel have almost identical TCP throughputs but have different UDP throughputs. This could have been caused by different rate adaptation algorithms where the Intel AP tries to use a different data rate more proactively after multiple successful transmissions. Ralink and Realtek have similar throughputs and some throughput drops can be observed at 20m. Marvell has the lowest downlink throughputs, which could be caused by the radio performance and the rate adaptation algorithms. We observe that during rate exploration, the Marvell's AP simply iterates all data rates from high to low, which takes more air time.

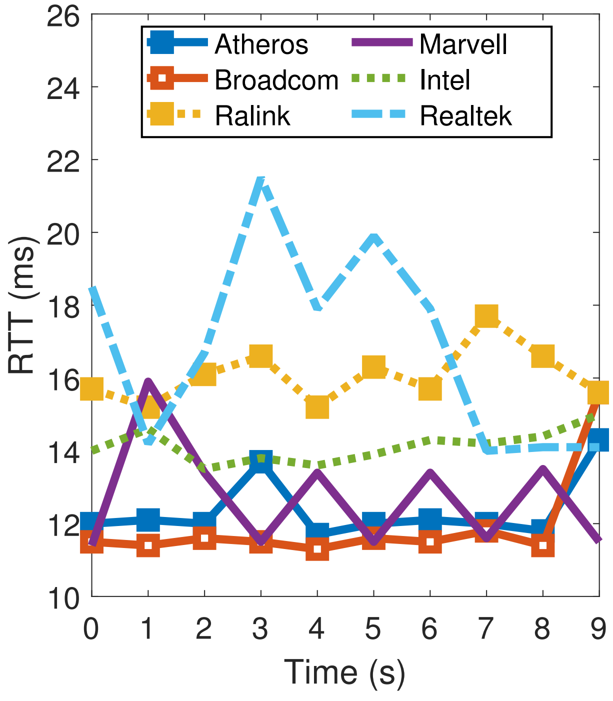
4.4.4 Round-Trip Time

We connect each AP to the Internet and measure the RTT of `Unify` over LAN (pinging the AP) and over WAN (pinging 8.8.4.4). The distance between the AP and `Unify` is 20m.

The wireline RTT between our location and 8.8.4.4 is approximately 6.80ms. This explains that the WAN RTT (Fig. 4.6b) is similar to the LAN RTT (Fig. 4.6a) except for a roughly 7ms offset. Broadcom and Atheros show the best performance, followed by Marvell and Intel. Ralink and Realtek have inferior RTT performances. We observe that Realtek RTT tends to oscillate between high and low. This might be due to its rate adaptation design or a bug. Overall, the RTT is 4$\sim$10ms for LAN and 11$\sim$21ms for WAN, making `Unify` highly suitable for latency-sensitive applications.



(a) LAN



(b) WAN

Figure 4.6: RTT

4.4.5 Coexistence with Multiple Devices

We conducted WiFi coexistence tests, demonstrating `Unify`'s ability to coexist with conventional WiFi devices without "starvation" even when the WiFi channel is fully saturated. We use the Atheros AP and concurrently run multiple `iperf3` servers. `Unify` and different conventional WiFi chips will simultaneously upload or download via TCP or UDP. By default, each UDP link is rate-limited by `iperf3` to 1Mbps whereas each TCP link is not.

For UDP, the channel is less saturated than for TCP and Table 4.4 shows that the throughput decrease of `iperf3` is relatively moderate as the number of active transmissions increases. For TCP, the channel is maximally saturated and each device must contend for the channel and share the overall bandwidth. For TCP uplink, the `Unify` throughput is roughly halved for 2 devices and is 39% (of the one device throughput) for 3 devices, etc. This validates that the MAC layer is working well and accesses the channel properly. For TCP downlink, the `Unify` throughput has an initial drop but stays relatively similar thereafter. We believe this is because the downlink arbitration is mostly done by the AP whereas the uplink arbitration is mostly achieved by multiple devices contending for the channel.

Table 4.5 shows the results of evaluating the coexistence of multiple `Unify` nodes. For two `Unify` nodes with active traffic, the throughputs are roughly halved for TCP or UDP downlink. For uplink, the throughputs are actually more than 50%. This is because when one `Unify` is

Table 4.4: Coexistence with multiple WiFi devices (bps)

# of Devices		TCP UL	UDP UL	TCP DL	UDP DL
1	Unify	432 k	484 k	554 k	654 k
2	Unify	218 k	417 k	193 k	634 k
	Intel 6205	8.07 M	1.05 M	26.3 M	1.07 M
3	Unify	170 k	442 k	118 k	588 k
	Intel 6205	18.2 M	1.05 M	13.3 M	1.05 M
	AR9462	1.88 M	1.05 M	17.6 M	1.00 M
4	Unify	92.3 k	411 k	92.4 k	553 k
	Intel 6205	12.6 M	1.05 M	14.4 M	1.05 M
	AR9462	5.18 M	1.05 M	14.5 M	1.05 M
	BCM4313	2.03 M	1.05 M	5.36 M	1.05 M
5	Unify	67.0 k	423 k	115 k	523 k
	Intel 6205	14.0 M	1.05 M	11.9 M	1.05 M
	AR9462	1.84 M	1.05 M	7.79 M	1.05 M
	BCM4313	1.13 M	1.05 M	6.30 M	1.05 M
	RTL8811AU	13.2 M	1.05 M	3.91 M	1.05 M
6	Unify	55.8 k	330 k	101 k	471 k
	Intel 6205	5.85 M	1.05 M	6.68 M	1.05 M
	AR9462	5.68 M	1.04 M	7.84 M	1.05 M
	BCM4313	554 k	1.04 M	5.95 M	1.05 M
	RTL8811AU	4.61 M	1.05 M	5.69 M	1.05 M
	MT7612U	6.06 M	978 k	3.38 M	1.04 M

copying the uplink packet, the other `Unify` can use the time to transmit packets. For three nodes, the throughputs are affected more, particularly in the uplink direction, because of the increased possibility of collision. However, this occurs only when all `Unify` nodes saturate the channel, and even so, no node is starved. Note that throughput variations can be observed even among standard WiFi vendors in Table 4.4. The bandwidth control, a common feature on WiFi APs, can be used to guarantee minimum throughputs for all devices. This approach should be better than relying purely on MAC layer implementations, which come with lower guarantees and can be slightly different among vendors.

We also tested the coexistence of `Unify` with Bluetooth devices. Table 4.6 shows `Unify`'s throughputs when 0, 1 or 2 Bluetooth headphones are simultaneously streaming audio. The results show that Bluetooth traffic has virtually no effect on `Unify`, because Bluetooth is designed to adaptively avoid active WiFi traffic and `Unify` is designed to behave exactly like a WiFi device.

Table 4.5: Throughputs of multiple `Unify` devices (kbps)

# of Devices		TCP UL	UDP UL	TCP DL	UDP DL
1	Unify #1	432	483	567	684
2	Unify #1	233	297	280	346
	Unify #2	229	296	292	347
3	Unify #1	107	86.7	94.7	242
	Unify #2	123	112	224	171
	Unify #3	90.1	150	118	276

Table 4.6: Coexistence with background BT traffic (kbps)

	TCP UL	UDP UL	TCP DL	UDP DL
0 BT Devices	431	485	551	672
1 BT Devices	432	484	551	672
2 BT Devices	432	485	547	671

4.4.6 Power Consumption

We measure the power consumption of `Unify` and compare it with off-the-shelf WiFi cards and prior work. For every device, currents are measured from the USB 5V power line when the device is constantly sending or receiving DSSS WiFi packets. Table 5.8 shows that `Unify` has significantly lower power consumption than standard WiFi cards and even FLEW. For a comparison with the same transmit power, we can first compare FLEW and `Unify`. Specifically, FLEW uses an FSK transceiver (0dBm) and an additional PA (+20dB, 100mA at 3V [70]). Just like FLEW, `Unify` could use the same PA (+20dB) for higher power. In such a case, the transmit power of `Unify` would be greater than 20dBm, and the overall power consumption (0.04+0.1 = 0.14A) is still lower than all other devices after considering the power consumption of the additional PA. (In practice, the PA would draw less than 0.1A at 5V.) `Unify` is set to the maximum Rx gain in all evaluations. Using a medium Rx gain consumes even lower power. Medium gain has a practical range of about 5~10m.

4.4.7 Applications

`Unify` enables CC254x to behave just like a WiFi chip and its low cost, tiny footprint and low power consumption make `Unify` ideal for IoT applications, such as the thermostat shown in Fig. 4.1b. Other applications that normally use WiFi can directly use `Unify` as well. For example, `Unify` is capable of streaming 360p Youtube videos in real time. Alternatively, `Unify` can stream high-quality Spotify audio in real time. `Unify` can also directly access web pages, such as weather websites. The novel communication paradigm of `Unify` can pave the way for innovative IoT

Table 4.7: Comparison of power consumption

	Tx (A) (Peak Power)	Rx (A)
AR9271	0.49 (19dBm)	0.07
RTL8811AU	0.32 (≤20dBm)	0.07
RT3072	0.28 (20dBm)	0.13
FLEW	0.16 (~20dBm)	0.11
Unify	0.04 (4dBm)	0.04
Unify (Med. Rx gain)	0.04 (4dBm)	0.03

applications in the future.

4.5 Discussion

We select the DSSS waveform since it is the most robust WiFi modulation and offers as high as 10dB higher sensitivity than OFDM. The benefit is that `Unify` maintains the same distance with a much lower transmit power (or allows a longer distance with the same transmit power).

`Unify` works with old and new WiFi devices, because of WiFi's backward compatibility. Another benefit is that DSSS does not have the problem of OFDM's high PAPR where the linearity requirement precludes low-power implementations. DSSS also allows for reusing existing FSK hardware, which is more difficult for OFDM. `Unify` thus achieves very low power consumption, which is particularly useful when IoT nodes are listening for an extended period but need to take immediate actions once a packet arrives.

Chatty DSSS connections may affect coexisting OFDM traffic. However, the advantage of `Unify` here is that since every device follows the WiFi signal and protocol, it provides good coexistence between them. This is supported by our coexistence test where no device is starved even under network saturation. In contrast, strong OFDM signals may drown out concurrent BLE connections under network saturation. Also, for narrow band signals like BLE, it has been shown that OFDM is susceptible to narrow band interference [92] and narrow band interference may greatly affect OFDM timing synchronization and detection [93]. `Unify` avoids starving and other coexistence issues by directly leveraging the WiFi design that already exists in access points.

4.6 Related Work

Earlier CTC works [7, 6, 8, 9, 10, 11, 12, 13] modulate packets' transmit power and a receiver measures the signal strength to decode information. These designs have considerably lower throughputs than the SOTA and require modifications on both ends. Numerous recent works demonstrate communication from modified WiFi transmitters to non-WiFi receivers, by carefully constructing magic WiFi packets. WiBeacon [73] broadcasts Bluetooth beacons by modifying WiFi APs to generate and send 802.11b waveforms that also resemble Bluetooth waveforms. BlueFi [72] transmits Bluetooth beacon or audio packets by sending special 802.11n packets. TransFi [76] works similarly but in a MIMO setting. NBee [74] and WiBle [77] modify WiFi devices to send DSSS packets that encode BLE packets. OfdmFi [14, 16, 15] enables communication between WiFi and LTE-U. Interscatter [48] uses WiFi chips to send a custom AM signal. WiFi-to-Zigbee [17, 19, 18, 20, 94, 95] and WiFi-to-LoRa [96, 97] are also possible by selecting WiFi packets.

Two prior studies enable communication between Bluetooth and WiFi, but none of them can work without modifying the WiFi receivers. Due to the extent of modifications required for WiFi receivers, they are only shown to work in simulation or with software-defined radios. The approach in [49] receives BLE packets using the FFT signals within the WiFi demodulation process. It is unclear whether such a feature is available across different WiFi vendors, or if collecting the FFT signals from standard WiFi chips is fast enough to provide continuous and seamless baseband samples. The WiFi implementation uses software-defined radios. Another study [98] tries to detect BLE packets using the preamble detector on WiFi receivers but concludes that the default WiFi preamble detection cannot be used. It thus proposes an extended preamble detection and demodulates BLE by collecting WiFi payloads (once a packet is detected). Its evaluation was done in simulation only.

FLEW [78] enables two-way communication between unmodified WiFi devices and modified FSK devices. However, it requires hardware modifications and using multiple chips, including a configurable FSK chip. In contrast, `Unify` is a single-chip solution for popular BLE/FSK SoCs.

4.7 Conclusion

We have designed and evaluated `Unify`, which transforms popular BLE/FSK SoCs into WiFi SoCs. `Unify` overcomes numerous significant challenges imposed by the hardware so that standard, bi-directional WiFi operations are achieved in SoCs without hardware modification. Without modifying WiFi devices, `Unify` is compatible with all major WiFi vendors and has low latency, good throughput and coexistence performance, and low power consumption.

CHAPTER 5

DREW: Double-Throughput Emulated WiFi

5.1 Introduction

Cross-technology communication (CTC) enables direct communication between heterogeneous wireless devices, such as between WiFi and Bluetooth devices. With the ubiquitous usage and wide deployment of WiFi and Bluetooth around the world, the WiFi–Bluetooth CTC creates highly useful connectivity applicable to tens of billions of devices. The state of the art (and DREW) enable end-to-end WiFi connectivity by transforming Bluetooth chips to fully-operational WiFi chips.

One practical application of this new connectivity is for the Internet of Things (IoT). For IoT, each device should be low-cost and highly energy-efficient in order to facilitate massive deployment and support ultra-long battery life. To meet these requirements, Bluetooth Low Energy (BLE) is a popular, widely-deployed technology. BLE chips cost as low as \$0.99 apiece [99, 100]. In addition, the latest BLE chips use ultra-low-power (ULP) designs that consume as little as 6mA [1, 2], and 10 years of battery life can be achieved with a single coin-cell battery [101]. However, the BLE protocol alone cannot provide Internet connectivity, and additional gateways are required to bridge the communication and forward the packets to/from the Internet. These IoT gateways incur additional costs to users, which are substantially higher than the price of individual IoT devices. Moreover, unlike the WiFi infrastructures (such as WiFi APs and routers), these IoT gateways have a much smaller installed base and do not have the same global adoption and coverage as WiFi. The necessity to use IoT gateways is thus a major obstacle to the widespread adoption of IoT [54, 102].

WiFi–Bluetooth CTC eliminates the need for gateways and thus effectively overcomes this obstacle. The state-of-the-art CTC solutions, FLEW [78] and Unify [103], achieve bidirectional communication between FSK (BLE modulation) and WiFi. With these solutions, IoT devices can use low-cost and energy-efficient FSK/BLE chips while having direct Internet access and global routability via the ubiquitous WiFi infrastructures.

However, FLEW and Unify are both based on older FSK chips, which have several key differences from the latest ultra-low-power (ULP) BLE chips. Specifically, older FSK chips use transmission

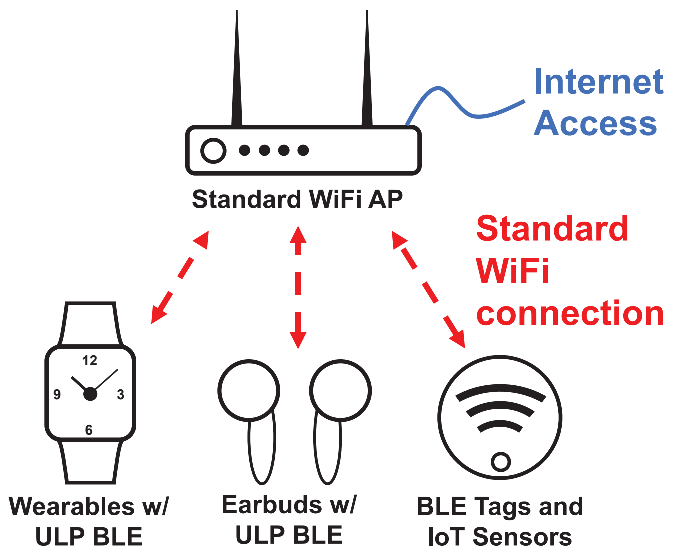


Figure 5.1: `DREW` is a end-to-end (e2e) system for ULP BLE devices to have standard WiFi connection.

mixers to modulate signals. Mixers have high insertion loss (passive mixers) or have high power consumption (active mixers). Newer BLE chips instead use **direct modulation** [104, 1, 2], where the signal is directly modulated by the PLL (phase-locked loop) and then amplified by the power amplifier (PA) (Fig. 5.3). Eliminating mixers between the PLL and PA is crucial for enabling ultra-low power consumption with more than 10 years of battery life. Such ULP chips only consume 6.1mA at 3.6V (KW41Z [1]) or 6.1mA at 3V (CC2650 [2]), whereas the CC2541 [105] chip used in Unify consumes ~12mA at 3.6V. On the other hand, FLEW and Unify require using mixers to flip the phase of the carrier for implementing the phase-shift keying modulation of WiFi. Therefore, they are not applicable to ULP BLE chips.

Another limitation of prior WiFi–Bluetooth CTC solutions is the 1Mbps throughput limitation. In particular, these WiFi–Bluetooth CTC works [78, 103] rely on the similarity between the WiFi BPSK and 1Mbps FSK waveforms in order to demodulate standard BPSK waveforms using FSK hardware. However, this similarity does not exist in WiFi's QPSK waveforms. In particular, QPSK is *not* BPSK clocked at twice the speed. In fact, QPSK is two orthogonal BPSK transmissions while keeping the same symbol rate. It is much more challenging for FSK hardware to receive WiFi's QPSK modulation, since QPSK cannot be demodulated by clocking the receiver twice faster. As a result, the state-of-the-art CTCs [78, 103, 74, 77, 72, 73, 76] focus exclusively on BPSK waveforms, which limits their maximum physical-layer throughput to 1Mbps.

In this chapter, we propose `DREW` (**D**ouble-th**R**oughput **E**mulated **W**iFi), which enables conventional WiFi connectivity on newer, ULP BLE chips. Because their hardware is significantly different, `DREW` addresses new challenges while exploring and leveraging new opportunities. `DREW`

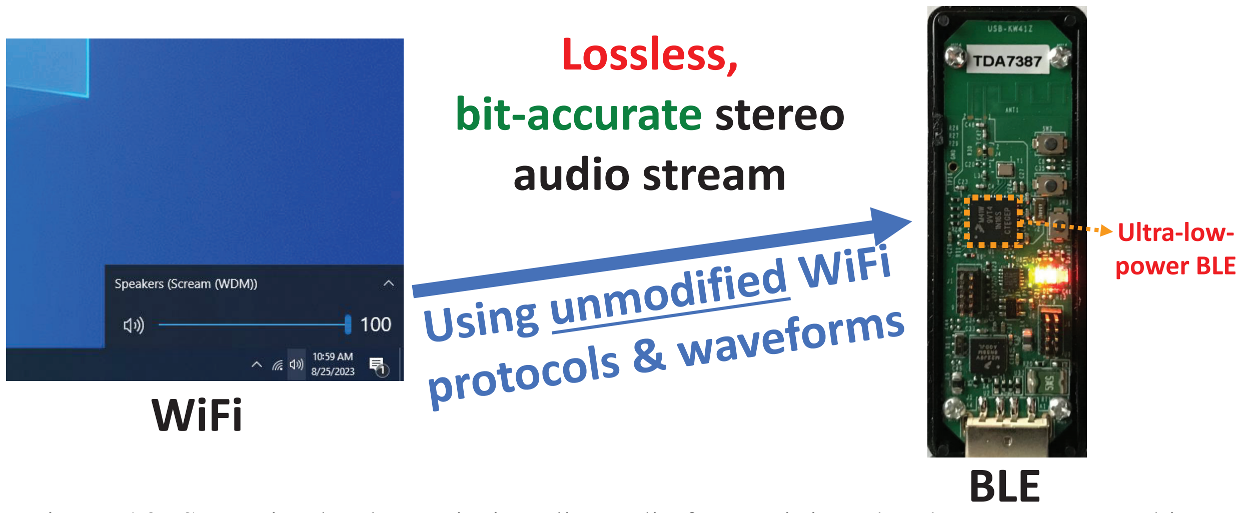


Figure 5.2: Streaming lossless, Hi-Fi quality audio from WiFi to ultra-low-power BLE chips

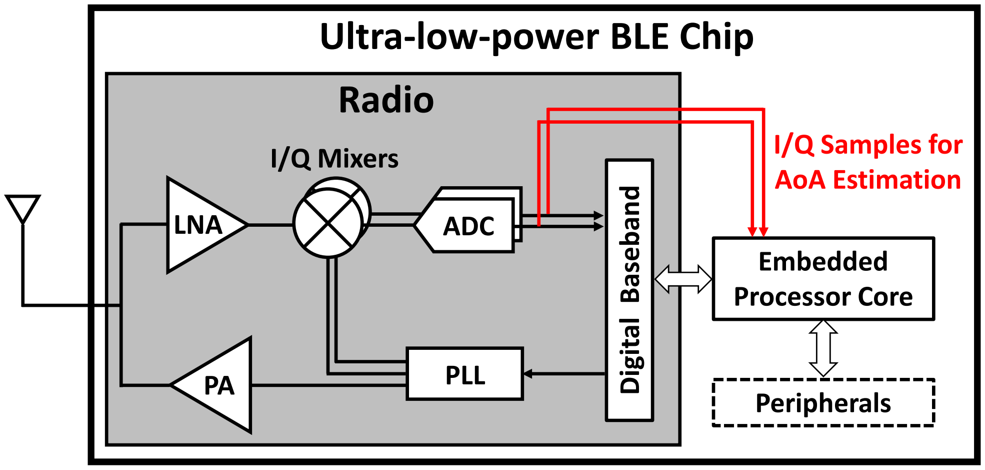


Figure 5.3: Architecture of ULP BLE chips [1, 2].

is an end-to-end WiFi system containing several key technical innovations: Specifically, we

- Propose and demonstrate new use of BLE's IQ sampling to receive standard, unmodified WiFi packets;
- Devise efficient algorithms, with SIMD parallel processing, for detecting and demodulating WiFi packets;
- Double the downlink throughput (compared to prior work) by uniquely supporting QPSK demodulation;
- Transmit WiFi packets by carefully controlling the PA on mixer-less BLE chips, which consumes ultra-low power; and
- Coordinate Tx/Rx to emulate WiFi's CSMA/CA and timing, specifically on ULP BLE chips.

The key design requirements – QPSK demodulation and mixer-less transmission – lead to brand-new designs for both WiFi-to-BLE and BLE-to-WiFi communications, which are very different from all prior work. Our WiFi-to-BLE design is based on an innovative use of the IQ sampling feature on modern BLE chips. Specifically, Bluetooth *localization* is a key feature on modern BLE chips, and the localization relies on processing the Angle-of-Arrival (AoA) of the incoming signal. The AoA is estimated by processing the IQ samples from the receiver. Therefore, modern BLE chips are capable of *IQ sampling* where the IQ samples are exposed to the processor core for post-processing. This feature has been incorporated into the standard since Bluetooth 5.1 [106]. For `DREW`, the key idea is that we can use this feature for data communication (instead of localization) to capture standard WiFi signals.

The IQ samples collected are relatively narrow-band and we design special algorithms to detect, synchronize and demodulate standard WiFi packets. Furthermore, we aim to process the IQ samples in ”real time” with an ARM Cortex-M0 core at 48MHz, since Cortex-M0 is the smallest ARM processor [107] and its nominal speed is 48MHz. To meet this goal, we come up with a special design that leverages the SIMD parallel processing. We also design algorithms to support QPSK and double the downlink throughput.

This real-time processing of IQ samples is important for `DREW` to behave and coexist like a typical WiFi chip. If the processing is not done in real time, the demodulation lags behind the waveform coming from the antennas, which results in processing delays at the end of a WiFi packet. Any such delays directly decrease transport-layer throughputs. More importantly, since the WiFi standard requires immediately sending an acknowledgement packet after receiving a packet, the processing delay prevents sending ACKs within the time limit, which makes such a design incompatible with off-the-shelf WiFi devices.

The challenge in the opposite direction is enabling BLE-to-WiFi communication with ULP BLE chips. Since ULP BLE chips do not have transmission mixers, we must generate the waveforms using only the PLL or the PA. We overcome this challenge by proposing a novel way of controlling the PA for BLE-to-WiFi communication. It is applicable to ULP BLE chips as it only relies on changing PA’s power level.

In order for BLE chips to interoperate with unmodified WiFi devices, `DREW` must follow the timing and medium access mechanism of the WiFi standard. `DREW` also coordinates packet transmission and reception, and overcomes the timing challenges of ULP BLE chips.

As a complete system, `DREW` follows the WiFi standard, including probing, authentication, association and encryption. `DREW` uses the 4-way handshake and WPA2-CCMP (AES encryption), and is secured against eavesdropping.

`DREW` uses WiFi DSSS waveforms because DSSS provides the highest reliability and robustness. Since no other WiFi waveforms match the same reliability and because of WiFi’s backward compatibility, DSSS is an important baseline modulation for current and future WiFi standards and the latest WiFi devices are still required to support it.

DREW's novel algorithm and use of IQ sampling achieve a physical-layer downlink bit rate of 2Mbps (= 2x the rate of FLEW and Unify). This enables new applications and benefits a wide variety of BLE systems, including IoT, wearable, and other ULP devices. For example, DREW can be used for delivering audio and short video messages to wearable devices. With the new transmission design, DREW (908kbps) also provides about 26% higher uplink throughput than FLEW (721kbps), which is particularly useful for IoT applications.

Another important application is high-quality audio streaming. We demonstrate that DREW can directly stream lossless stereo audio from WiFi to BLE devices (Fig. 5.2). This standard audio stream has a bit rate of 1.411Mbps, which well exceeds the throughputs of FLEW and Unify. In fact, even conventional Bluetooth headphones cannot stream lossless, uncompressed audio because Bluetooth can only support compressed audio streams (e.g., SBC, AAC, aptX), which degrade audio quality (typically 4x compression) and introduce latencies. In contrast, DREW leverages audio over IP (i.e., WiFi), and thus audio is not compressed by Bluetooth transmitters. Without compression, DREW offers a bit-accurate, wire-equivalent audio experience with ultra-low power consumption of BLE chips.

5.2 System Design

5.2.1 WiFi to BLE

5.2.1.1 Observation

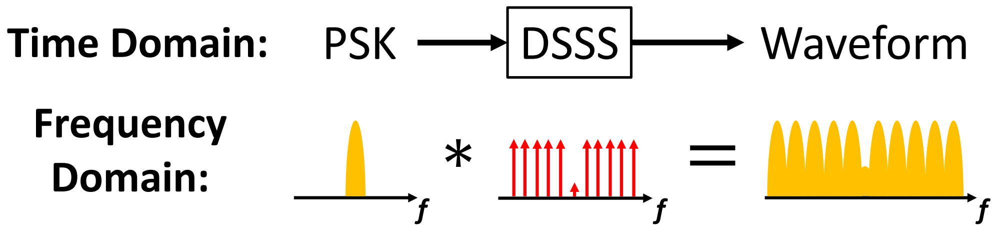


Figure 5.4: The DSSS modulation process in WiFi.

To design the WiFi-to-BLE communication, we start by looking into a key block in the WiFi transmitters. For an 802.11 waveform, a WiFi packet (after scrambling and differential coding) is converted to a BPSK or QPSK bitstream with a symbol rate of 1MSym/s. Each PSK symbol is then multiplied by the 11-chip Barker sequence to generate the waveform, as shown in Fig. 5.4. The spectrum of a periodic 11-chip Barker sequence is 11 impulses located at $\pm1, \pm2, \ldots$MHz. Since multiplication in the time domain corresponds to convolution in the frequency domain, the result of the DSSS process is replication of the original PSK spectrum at $\pm1, \pm2, \ldots$MHz. If we capture the waveform at -1MHz and calculate its phase, we can recover the PSK symbols. By leveraging the close relationship between BPSK and FSK, prior work [78] subsequently recovers the BPSK symbols with an FSK demodulator operating at an additional frequency shift. However, since there is no similar relationship between QPSK and FSK, this design cannot be extended to QPSK demodulation.

Our key idea is to leverage the IQ sampling capability on modern BLE chips to capture the PSK spectrum. By processing the IQ samples, both BPSK and QPSK demodulations become possible, and thus we can double the throughput. Since BLE chips are designed to receive FSK waveforms at 1MSym/s, the IQ sampling works well for receiving BPSK/QPSK waveforms at 1MSym/s.

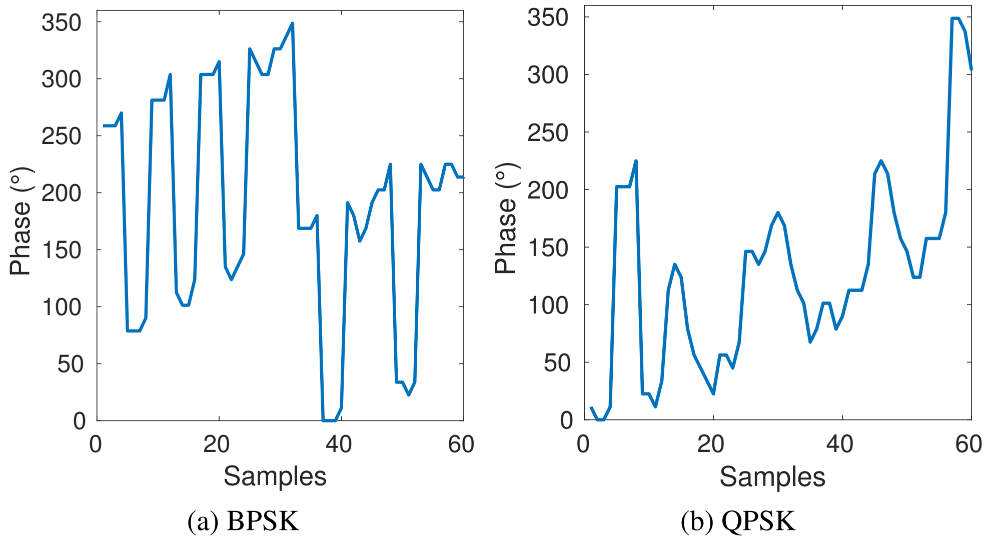


Figure 5.5: Utilizing BLE's IQ sampling to receive standard WiFi waveforms.

As a real example, we use an off-the-shelf BLE chip to perform IQ sampling on BPSK and QPSK WiFi packets. The WiFi packets are sent at 2452MHz and the BLE chip operates at 2451MHz. Figs. 5.5(a) and 5.5(b) show the phase of the IQ capture. The IQ sampling runs at 4MHz, and thus each WiFi symbol is represented by 4 data points. In Fig. 5.5a, the phases change by either 0° or 180° in groups of four. Similarly for QPSK, the phases change by 0°, 90°, 180° or 270° in Fig. 5.5b. The figure also illustrates we should ideally select the phases near the center of each symbol for accurate sampling.

5.2.1.2 Processing Phases

To process the collected phases, we first pack them into 32-bit words, as illustrated in Fig. 5.6. We use 32-bit words because they are the intrinsic unit for most arithmetic and logical operations on ARM microcontrollers. Since each WiFi symbol spans 4 phases and WiFi uses differential coding, we calculate the phase difference between WiFi symbols. Let the phases be $\theta[0], \theta[1], \theta[2], \cdots$. We calculate 4 sets:

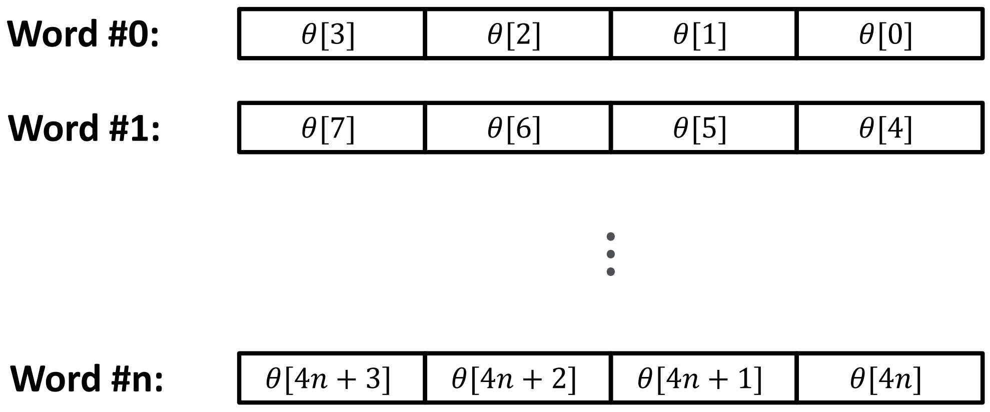


Figure 5.6: Packing phases into 32-bit words.

- $\{\theta[4n] - \theta[4n-4], n \in \mathbb{N}\}$
- $\{\theta[4n+1] - \theta[4n+1-4], n \in \mathbb{N}\}$
- $\{\theta[4n+2] - \theta[4n+2-4], n \in \mathbb{N}\}$
- $\{\theta[4n+3] - \theta[4n+3-4], n \in \mathbb{N}\}$.

These sets represent the phase differences sampled at slightly different time instants within WiFi symbols, and the one with the optimal sampling instant will have the best estimates. Fig. 5.7 shows an example of using IQ sampling to capture an actual QPSK WiFi packet (with a BPSK header and a QPSK payload). In Fig. 5.7, $\{\theta[4n+2] - \theta[4n+2-4]\}$ has the best estimates. For the BPSK header ($n < 127$), the phase differences between symbols are 0° (bit '0') or 180° (bit '1'). For the QPSK payload ($n \geq 127$), the phase differences are 0° (bits '00'), 90° (bits '01'), 180° (bits '11') or 270° (bits '10').

For each phase within a word, we use 5 bits to represent the range $[0, 360)$. The overflow and underflow properties of integers map nicely to the fact that the phase wraps around every 360°. That is, if we add or subtract two phases, the lower 5 bits will represent the principal angle of the result (even when an overflow or underflow occurs).

By packing multiple phases in a 32-bit word, we can use one instruction to perform the same arithmetic operation on 4 phases simultaneously. This type of parallel computation is known as *Single Instruction Multiple Data* (SIMD). Although ARM has an official SIMD implementation ("NEON" [108]), it is usually not available on low-end ARM microcontrollers. However, we can still apply SIMD with only conventional instructions (such as `ADD` or `SUB`). In Fig. 5.6, if we calculate `Word[1]-Word[0]`, the upper 8 bits will be $\theta[7] - \theta[3]$ and the next 8 bits will be $\theta[6] - \theta[2]$, etc. However, if $\theta[6] < \theta[2]$, the upper 8 bits will be $\theta[7] - \theta[3] - 1$ because of borrowing. To

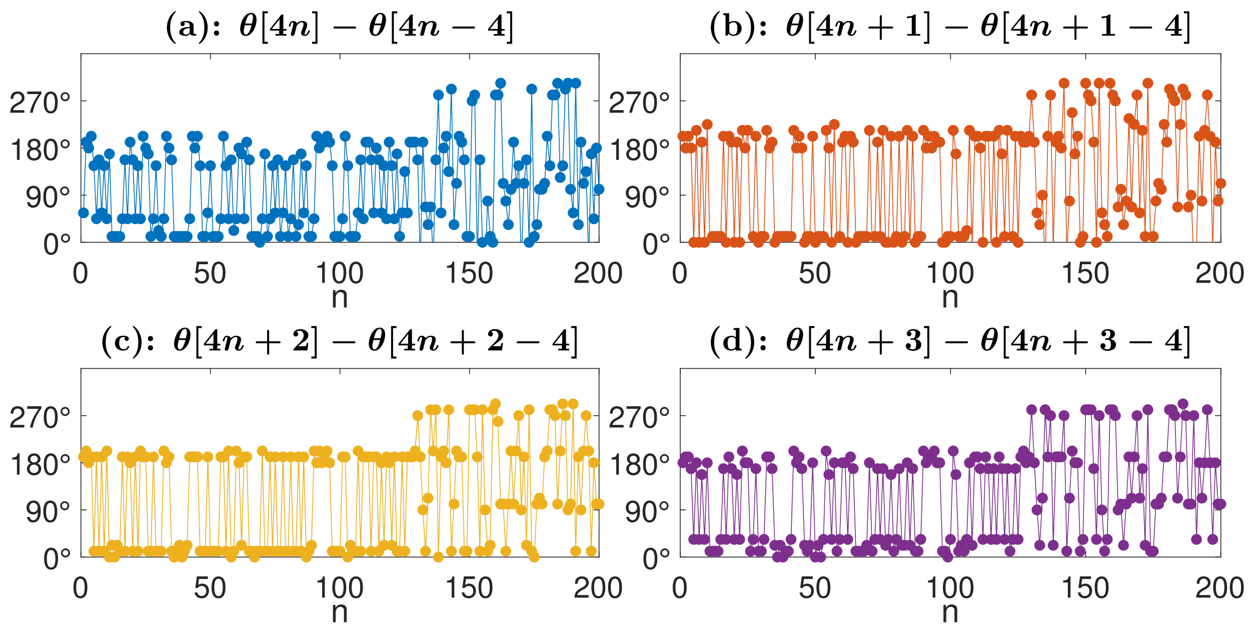


Figure 5.7: Phase changes between WiFi symbols.

account for this, we can instead calculate `Word[1]+0x80808080−Word[0]` so that the borrowing will never occur across byte boundaries and we will take the lower 5-bits of each byte as results.

5.2.1.3 BPSK Demodulation

For BPSK demodulation, we need to determine whether the phase difference is 0° or 180°. Taking the first half of Fig. 5.7(c) as an example, we can make a bit decision by first adding a constant 90° to the phase difference and then determining whether the result has a principal angle in $[0°, 180°)$ or in $[180°, 360°)$.

This process can be implemented very efficiently using SIMD and leveraging the property of binary representation. With the phase format we use, 360° corresponds to a value of 32, and thus adding 90° to each phase difference is adding 8 to each byte. We use SIMD and add 0x08080808 to each 32-bit word. We can further merge two additions and simply calculate `Word[i]−Word[i-1]+0x88888888`. Within this 32-bit result, the principal angles are the lower 5 bits of each byte and the most significant bit (of these 5 bits) will indicate whether the angle is in $[0°, 180°)$ or $[180°, 360°)$. Therefore, we can recover the WiFi BPSK bitstream by continuously calculating `Word[i]−Word[i-1] +0x88888888` and extracting that bit from every 32-bit result.

SIMD is a critical part of `DREW` to enable real-time IQ processing. Furthermore, the SUB and ADD instructions used in our SIMD method are generic 32-bit subtract and addition. Thus, this method is not specific to ARM and is thus applicable to 32-bit processors in general. Using BPSK demodulation as an example, Table 5.1 compares processing phases with and without SIMD.

Table 5.1: Processing with and without SIMD

`DREW`	Without SIMD
SUB r1, r0, r1 ADD r1, r1, r2	SUB r2, r0, r1 ADD r3, r2, #0x8; first byte LSR r0, r0, #8 LSR r1, r1, #8 SUB r2, r0, r1 ADD r4, r2, #0x8; second byte LSR r0, r0, #8 LSR r1, r1, #8 SUB r2, r0, r1 ADD r5, r2, #0x8; third byte LSR r0, r0, #8 LSR r1, r1, #8 SUB r2, r0, r1 ADD r6, r2, #0x8; fourth byte

`DREW` takes 2 cycles whereas processing without SIMD takes 14 cycles. With a processor running at 48MHz, new sets of IQ samples are generated every 48 cycles, and other processing steps (loading phases, extracting bits, pattern matching) take about 39 cycles. Therefore, without SIMD, `DREW` cannot process the IQ samples in real time. Also, processing without SIMD requires using additional registers (r3~r6), and additional cycles are needed for managing these lower registers of the ARM processor.

5.2.1.4 QPSK Demodulation

We extend the above process to demodulating QPSK symbols. The major difference is that the phase change of QPSK symbols can be 0°, 90°, 180° or 270°. In Fig. 5.7(c), QPSK demodulation can be achieved by first adding a constant 45° to the phase difference and then determining the quadrant of the phase difference.

To demodulate QPSK bits, we calculate `Word[`i`]−Word[`$i-1$`]+0x84848484`. (Note that the value '4' corresponds to 45°.) The quadrant of the phase difference is naturally represented by the upper 2 bits of the lower 5 bits of each byte. Finally, since WiFi uses Gray code for the QPSK symbol mapping, we convert the "dibits" from binary code to Gray code. We build a simple lookup table for this conversion.

5.2.1.5 Packet Detection and Time Synchronization

To receive an entire WiFi packet, we need to locate the beginning of an incoming WiFi packet. This process is known as *packet detection* in which the receiver constantly monitors the incoming waveform and triggers the packet decoding when a valid packet arrives. Packet detection relies on the principle of pattern matching. The WiFi standard defines special BPSK bit patterns (i.e., PLCP preamble) that the transmitter should use, and the receiver searches for the bit pattern to detect the start of a valid WiFi packet.

At the start of a standard packet, a WiFi transmitter sends the PLCP preamble. However, since WiFi transmitters later apply bit-scrambling to the entire packet, the over-the-air packet contains the scrambled PLCP preamble. Thus, a possible implementation is to first descramble the demodulated BPSK bits and then search for the original PLCP preamble.

Prior work [78, 103] point out that the WiFi standard [30] defines the exact scrambler seed all transmitters should use for BPSK packets, and hence the descrambling step can be eliminated by directly searching for the scrambled PLCP preamble among the demodulated bits.

Different from prior work, however, `DREW` must achieve a precise synchronization suitable for QPSK. Furthermore, this synchronization should be at the IQ-sample level, since the FSK demodulator (that prior work used) cannot decode QPSK payloads. At a finer time granularity, we sample each WiFi BPSK symbol with 4 phase samples, and thus there are 4 possible sampling time offsets, as shown in Fig. 5.7. Equivalently, each 32-bit word contains 4 bytes and provides 4 possible BPSK estimates at slightly different sampling time instants. If the optimal sampling time instant is known, we can directly extract the BPSK bit from one of the four possibilities. However, there is no intrinsic time synchronization between a transmitter and a receiver, and a WiFi waveform could start at $\theta[0]$, or at $\theta[1]$, $\cdots$. Thus, in addition to packet detection, we need to establish precise time synchronization when a valid packet arrives.

We design an algorithm that solves packet detection and time synchronization simultaneously. We form 4 independent bitstreams from the 4 BPSK estimates of every word and search for the scrambled PLCP preamble from each bitstream. We use one register as one 32-bit FIFO for each bitstream and the BPSK estimates are constantly shifted into the FIFO. This can be efficiently implemented by shifting a demodulated BPSK bit into the carry bit with `LSR` (logical shift right) and shifting the carry bit into the corresponding FIFO with `LSL` (logical shift left) and `ADC` (add with carry). FIFO #0 contains BPSK bitstream estimated from $\{\theta[4n] - \theta[4n-4]\}$; FIFO #1 contains BPSK bitstream estimated from $\{\theta[4n+1] - \theta[4n+1-4]\}$, etc. Next, we compare each FIFO with the bit pattern of the scrambled PLCP preamble using `CMP` (compare). An exact match indicates that the start of a packet is detected at a fine-grain sampling time instant.

Since there are 4 BPSK estimates for every WiFi symbol, one valid PLCP will ideally trigger multiple exact matches. In Fig. 5.6, if a WiFi packet starts at $\theta[0]$, exact matches will be triggered at $\theta[4k], \theta[4k+1], \theta[4k+2]$ and $\theta[4k+3]$ (for some k). In such a case, we will subsequently use the phase samples near the center of symbols (e.g., $\{\theta[4n+2], n \in \mathbb{N}\}$) for header and payload decoding.

If a WiFi packet starts at $\theta[2]$, exact matches will be triggered at $\theta[4k+2], \theta[4k+3], \theta[4k+4]$ and $\theta[4k+5]$ and we will select $\{\theta[4n+4], n \in \mathbb{N}\}$. In the general case, if any exact match occurs at $\theta[4k], \theta[4k+1], \theta[4k+2]$ or $\theta[4k+3]$, we consider a valid packet is detected and will perform the demodulation and the FIFO matching for one extra round to see if there are more matches at $\theta[4k+4], \theta[4k+5]$ or $\theta[4k+6]$. Based on whether the preamble is detected or not at $\theta[4k \sim 4k+6]$, we can establish fine-grain time synchronization and select an appropriate sampling time offset l such that $\{\theta[4n+l], n \in \mathbb{N}\}$ are near the center of symbols. After the synchronization, only $\{\theta[4n+l], n \in \mathbb{N}\}$ will be used for decoding.

In our design, the result of preamble matching at $\theta[4k \sim 4k+6]$ is stored as an integer with one-hot encoding. We build a lookup table that directly converts this encoding to the best sampling time offset l. In our implementation, this sampling time offset l is stored in the form of the number of right shifts to be applied after performing SIMD.

5.2.1.6 Long Preamble and Short Preamble

The WiFi standard defines two possible PLCP preambles, so a transmitter may use long (144 bits) or short (72 bits) preambles. However, the long preamble is always used for BPSK payloads. Since prior work focused on BPSK only, their packet detection is for long PLCP preamble only.

For `DREW`, a transmitter may use either the long or short preamble for QPSK payloads. Technically, the short preamble is an optional (and typically configurable) feature and a receiver is not strictly required to support it. However, we find that some off-the-shelf WiFi APs use the short preamble by default. To ensure the best compatibility, we aim to support both long and short preambles simultaneously.

Since both the long and short PLCP preambles use BPSK modulation, the packet detection and time synchronization are applicable to both. However, the short preamble is a different bit pattern scrambled with a different seed. Two types of preambles thus have different bit patterns, and `DREW` has to search for both simultaneously. This can be implemented efficiently since we already have the demodulated BPSK bits in the FIFOs. We simply add 4 extra compares (`CMP`) to search for the short preamble from 4 independent bitstreams.

The pattern that the `CMP` instruction tries to match can be any 32-bit sub-sequence of the scrambled (long or short) PLCP preamble. We use `0x78869b04` (for the long preamble) and `0xfa51c63f` (for the short preamble), which are the scrambled bit patterns near the end of the preambles. With such selections, `DREW` can detect a packet as long as the last few bytes of the PLCP preamble are received, even when the receiver starts sometime later than the beginning of the packet.

5.2.1.7 Supporting Realtek's Non-standard Preamble

Prior work [78, 103] observed that Realtek has a wrong WiFi implementation that produces incorrect long preambles. Consequently, Realtek's packets also have a different (scrambled) bit pattern near the end of the preamble. To detect Realtek's packets, we instead search for `0x1e21a6a5`, which is a 32-bit sub-sequence using Realtek's PLCP format.

A unique feature of `DREW` is supporting QPSK packets, which may use the short preamble format (unlike BPSK packets that always use the long preamble). We find that Realtek's short preamble is also wrong. Using simulation and analyzing waveform captures, we have traced the issue down to the scrambler seed. Sec. 16.2.3.9 of the WiFi standard [30] specifies that a transmitter should initialize the scrambler with `0b0011011` for all short PLCP packets. In contrast, Realtek always uses `0b0110010` as the scrambler seed. The scrambled short PLCP preamble is thus a different BPSK pattern. We solve this issue by using `0xda1503e9` (the last few bytes of this pattern) to detect such packets.

5.2.1.8 Parsing PLCP Header and Bit Descrambling

The PLCP header follows the PLCP preamble and contains the duration and modulation of the payload. The header uses BPSK (for the long PLCP) or QPSK (for the short PLCP). We demodulate BPSK or QPSK bits using the time-synchronized phases ($\{\theta[4n + l]\}$). We then descramble the bitstream to recover the actual header bytes. We also convert the duration of the payload to the number of bytes of the payload.

5.2.1.9 Decoding Payload and Checking CRC

The WiFi payload begins after the PLCP header. Depending on the "SIGNAL" byte in the header, `DREW` dynamically selects BPSK or QPSK demodulation. Payload is recovered after demodulation and descrambling. The payload is then stored in the memory.

The last 4 bytes in the payload are the preceding bytes' CRC32 values. When receiving the PLCP header, we initialize a 32-bit register to hold the CRC32 result. After recovering each WiFi payload byte, we calculate the updated CRC32 using a lookup-table-based algorithm. The updated CRC32 is a function of the current CRC32 value and the WiFi byte received. At the end of packet reception, we check the CRC32 value and discard any packet with a mismatched CRC.

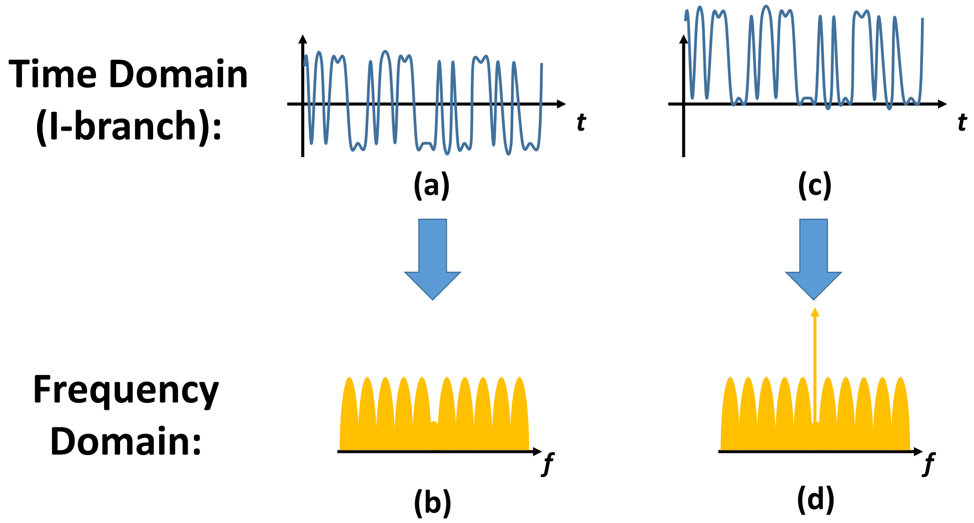


Figure 5.8: Comparing BPSK and BPSK with a DC offset.

5.2.2 BLE to WiFi

5.2.2.1 Observation

To design the BLE-to-WiFi communication, we look into the standard WiFi waveform. Fig. 5.8(a) shows a typical WiFi BPSK waveform. We plot the time-domain waveform in the baseband (IQ form) and only the I-branch is shown because the Q-branch is always 0 for BPSK. We can see that the time-domain waveform is a PSK modulation (the I-branch is either 1 or -1) and each WiFi bit is represented by a Barker sequence. Therefore, each WiFi bit ultimately becomes either $\{1, -1, 1, 1, -1, 1, 1, 1, -1, -1, -1\}$ or $\{-1, 1, -1, -1, 1, -1, -1, -1, 1, 1, 1\}$. The Barker sequence results in a corresponding spectrum (Fig. 5.8(b)).

Fig. 5.8 also shows a waveform related to (a). When a constant offset is added, (a) becomes (c). In the spectrum, this DC is manifested as an arrow at $f = 0$, as shown in (d). Note that spectrum (b) and (d) are identical except at DC.

The DC ($f = 0$) in the baseband corresponds to the carrier frequency (e.g., $f = 2412$MHz) in the passband. A constant DC level (or a constant carrier) does not carry information and receivers commonly remove DC prior to demodulation because the RF circuitry can produce unwanted DC residue and cause an uncertain DC level in the baseband. For example, the bias voltage at which the mixers or ADCs operate becomes DC in the baseband. Also, the LO leakage in zero-IF receivers will introduce an uncertain amount of DC in the baseband. Specifically, the LO signal (e.g., $f = 2412$MHz) will have some leakage that couples (along with the desired signal) into the RF port, and such a leakage becomes a DC uncertainty in the baseband after downconversion.

Suppose a WiFi device receives the waveform in Fig. 5.8(c), the DC will be removed in the baseband before demodulation. As illustrated in Fig. 5.9, Fig. 5.8(c) becomes Fig. 5.8(a) after DC removal. Therefore, if we substitute Fig. 5.8(a) with Fig. 5.8(c) at the transmitter, the receiver still sees the same baseband signal after DC removal and can therefore decode the packet correctly.

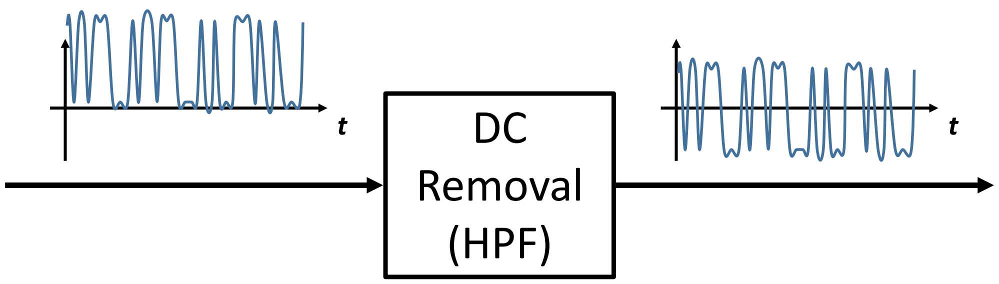


Figure 5.9: DC removal in receivers.

This observation is key to the design of BLE-to-WiFi communication. Fig. 5.8(a) is difficult to implement on mixer-less BLE chips as it requires precisely inverting the phase of the carrier at a relatively high speed. In contrast, Fig. 5.8(c) can be implemented by switching the carrier on and off.

5.2.2.2 Sending Bits

This switching can be realized by turning on and off any component along the transmission signal path. A straightforward and effective design is to use assembly to control the power level of the PA. Since the `STR` (store) instruction takes 2 cycles in ARM microprocessors, we update the power level of the PA at 8MHz, which gives enough headroom for microprocessors running at 16MHz or higher (which is typical for BLE chips). The Barker sequence also scales well at 8MHz (and becomes $\{1, 0, 1, 0, 1, 1, 0, 0\}$). To send a WiFi bit, the power levels are $\{1, 0, 1, 0, 1, 1, 0, 0\}$ for phase 0° and $\{0, 1, 0, 1, 0, 0, 1, 1\}$ for phase 180°, assuming that '1' is the maximum power level of the PA.

5.2.2.3 Sending Packets

By repeating the same process for every bit in a packet, we can send a complete WiFi packet. Instead of spelling out all power levels of a WiFi packet with thousands of instructions, we design a compact routine that sends a WiFi packet with less than 20 instructions. Specifically, the sketch of the transmission routine is:

```
txloop:
    R1 → [PA power register]
    R0 → [PA power register]
    R1 → [PA power register]
    R0 → [PA power register]
    R1 → [PA power register]
    R1 ← Next WiFi bit (from memory)
    R0 → [PA power register]
    R0 ← $\overline{R1}$
```

```
Jump to txloop if there are more bits
```

The idea is that since `'10101100'` (or `'01010011'`) contains consecutive duplicates, instead of setting the power level twice, we can use those cycles to get the phase of the next WiFi bit and prepare the power levels. Note that R0 $\leftarrow \overline{\text{R1}}$ is not a memory operation, and thus we can insert a conditional jump afterwards without affecting the timing. In our implementation, we also insert `NOP` at appropriate locations so that each `txloop` takes exactly 1μs.

5.2.2.4 Zero-Wait Packet Transmission

The above assembly routine works well for sending WiFi packets. However, it requires copying the entire packet to the memory before transmission, which causes delays and decreases throughputs. This issue is manifested on FLEW [78] and Unify [103].

We design another assembly routine that eliminates this waiting time, thus increasing throughputs. We start transmission as soon as the first byte is ready. During the cycles where the power level stays the same, we collect the WiFi bits (from UART), update R0 and R1 accordingly, and copy the WiFi bits to memory. We still copy the WiFi packet to memory so that we can run the routine in Sec. 5.2.2.3 to retransmit the packet in case of unsuccessful transmissions.

5.2.3 MAC Layer

`DREW` is to make a ULP BLE chip *indistinguishable* from a WiFi chip (and thus ensuring compatibility). To achieve this goal, the ULP BLE chip should emulate WiFi's MAC, including CSMA/CA, Tx/Rx switching (timings) and ACK/CTS handling. Since the emulated MAC layer of prior work [78, 103] is shown to be effective and highly compatible across WiFi vendors, `DREW`'s MAC layer follows the same design principles with several differences to accommodate the entirely different physical-layer design and QPSK packets.

To emulate CSMA/CA, `DREW` uses RSSI from the radio to perform Clear Channel Assessment (CCA) before transmission, similar to prior work. Although `DREW`'s IQ sampling could, in theory, alternatively be used to implement CSMA/CA, we chose to directly read the RSSI register because it is simple and effective. Random backoff is also implemented.

Applying the same design principle, we create `DREW`'s finite-state machine, shown in Fig. 5.10. The FSM governs the state transitions and associated actions. The actions performed after receiving ACK, RTS or unicast packets closely follow the WiFi standard. However, the FSM has several differences from prior work. First, because of the *zero-wait packet transmission*, the FSM is actually simpler and has higher transmission throughputs. `DREW` can directly go to the the Tx state once the channel is clear (without copying the entire packet first), whereas prior work have several wait states and flags in order to reuse part of the downlink (Rx) airtime to copy uplink (Tx) data. `DREW`'s design especially benefits highly asymmetrical traffic (e.g., UDP uplink). There is still one flag associated with the Tx path to ensure that each unicast packet is properly acknowledged before a new packet is used in the `Tx` state. In `Tx`, `DREW` will only fetch a new packet if `flag` is 0.

Second, since `DREW` supports QPSK, the `Rx` state searches for both long and short preambles simultaneously, which is efficiently implemented by adding 4 compare instructions. Also, the FSM handles 3 combinations (long-BPSK, long-QPSK, short-QPSK) that an incoming packet may have.

As prior work pointed out, WiFi's Tx/Rx timings can be particularly tricky for BLE chips to emulate. Surprisingly, we found that using the IQ sampling actually helps relax some of the timing requirements, such as the Tx–Rx turnaround. In particular, direct processing of IQ samples eliminates any delay or timing uncertainty that the FSK demodulation logic might have. More importantly, IQ sampling allows a greater flexibility in selecting the bit patterns for packet detection. Prior work has more restrictions in selecting the bit patterns because the FSK demodulator has to be stabilized with the ”1010101” pattern first. For `DREW`, we can use the bit patterns near the very end of a WiFi preamble and thus relax the Tx–Rx turnaround requirement. We can invoke the normal Rx warm-up sequence and still satisfy WiFi timing.

Because `DREW` only relies on the PA to transmit packets, we can leverage the power override idea (proposed in [103]) to satisfy the Rx–Tx turnaround. Specifically, for ACK or CTS transmission, `DREW` keeps IQ sampling running (so that the PLL is still locked) while powering on the PA (and signal buffers along the Tx path). The carrier is modulated by controlling the power level. After ACK or CTS, the power overrides are turned off and another Rx warm-up is invoked.

5.2.4 Implementation

We have implemented `DREW` on COTS BLE chips. We use the Freescale (now NXP) Kinetis KW41Z ultra-low-power [1] BLE chip as the hardware platform because its reference manual [109] and register maps are publicly available. The KW41Z is a single-chip SoC that has an ARM Cortex-M0+ core [110] running up to 48MHz. In the evaluations, we use the USB-KW41Z development board [111] from NXP. This KW41Z SoC can also be found in industrial IoT modules made by Panasonic Industry [112] and u-blox [113].

The firmware is written in C and is developed in the MCUXpresso IDE [114] with the GNU ARM Embedded Toolchain [115]. Time-sensitive tasks (such as packet detection, demodulation or transmission) are written in ARM assembly (with the Thumb instructions). The USB-KW41Z development board has an on-board debugger, which is a separate microcontroller running the OpenSDA [116] debugging functions. We also use this microcontroller as the USB-UART bridge. We rewrote the USB-UART bridge firmware ourselves because we found that SEGGER's UART implementation does not work properly at higher (¿1MBaud) speeds.

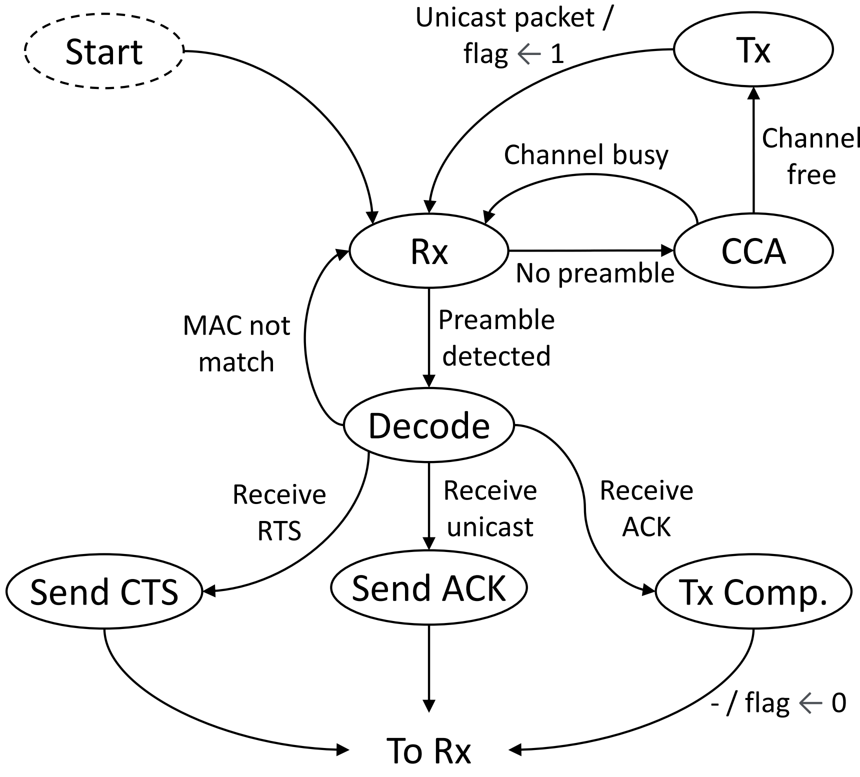


Figure 5.10: Finite state machine.

5.2.4.1 IQ Sampling

The reference manual of KW41Z [109] provides detailed instructions of IQ sampling using DMA. We configure KW41Z to convert IQ to phases and pack them in the 4-byte format as shown in Fig. 5.6. We also enable the channel filter that runs a 2-sample moving mean.

Instead of using the DMA, we use the ARM core to fetch the phases and process them directly. A new 32-bit word (containing 4 phases) is fetched every 1μs. To help the ARM core distinguish a new word from the old one, we slightly increase (about 0.1MHz) the RF frequency, ensuring the phases change slightly each time and consecutive samples are different. This frequency bump is later compensated for when calculating the phase differences.

5.2.4.2 Clocking

We found that setting the CPU and peripheral clocks on the KW41Z chip requires special attention. This is because KW41Z uses a legacy clock module (MCG) from the Motorola (later Freescale) 68HCS08 family [117]. The KW41Z has two major clock sources: a 32MHz oscillator and a 32.768kHz RTC (real-time clock). The BLE radio circuitry always uses the 32MHz clock, whereas the ARM core must use the FLL (frequency-locked loop) inside the MCG to reach the advertised speed of 48MHz.

We design the receive codes to run at 48MHz. For transmission, however, we find that the 48MHz clock from the FLL has too many jitters for the transmission routine. Broadcom and Ralink WiFi chips can actually tolerate this waveform jitters. However, the jitters have a noticeable performance impact with Atheros chips.

We solve this issue by dynamically switching the clock source so that the transmission routine is clocked from the 32MHz oscillator. Because WiFi only allows a very short time between Tx and Rx, clock switching also has to be swift. We find that this is possible when the clocks are *coherent*. Therefore, we use the FLL to generate a 48MHz clock from the 32MHz clock and use it for the Rx routine. For the Tx routines (including sending ACK and CTS), we directly switch on the PA, LO and various signal buffers with the `OVRD1` and `OVRD3` registers. Since both the ARM core and the PA use the same clock source during transmission, we set `GASKET_BYPASS` to 1 to bypass unnecessary clock synchronization.

5.2.4.3 Driver

We write a custom driver and `DREW` is directly compatible with the Linux kernel and the `mac80211` module. In addition to normal Tx/Rx paths, monitor mode and packet injection also work properly.

5.3 Evaluation

5.3.1 Experimental Setup

We first run microbenchmarks on the PHY layer. The microbenchmarks evaluate the performance of communication between BLE and COTS WiFi chips. We measure the packet error rate (PER) at different distances in both WiFi-to-BLE and BLE-to-WiFi directions. We use WiFi cards from major chip-makers that are widely used in WiFi infrastructures (APs), including Atheros (now Qualcomm), Broadcom, Ralink (now Mediatek) and Realtek. All chipsets are unmodified in terms of hardware, firmware or driver. We used the default firmware and driver provided by Ubuntu 20.04. We use WiFi channel 9 to avoid background interference.

In the microbenchmarks, we put a WiFi card (or `DREW`) in monitor mode and injected 4096 packets in each test case. Each packet has 1508 bytes of payload and has a unique sequence number. The receiving WiFi card (or `DREW`) is also in monitor mode and we collect the packets in Wireshark. CRC check is enforced and the sequence numbers of correctly-received packets are put into a set. The PER is derived by calculating the percentage of the sequence numbers missing out of 4096. To showcase `DREW`'s unique ability to receive QPSK packets, we perform PER microbenchmarks of BPSK and QPSK demodulation. For Atheros, Broadcom and Ralink, the injected packets are standard WiFi packets with an 802.11 frame type of 0x08. For Realtek, we found that injection of QPSK data (0x08) packets does not work with the logic of the default driver (rtlwifi). We solve this issue by injecting packets with the frame type 0x00, which forces the driver to use the correct modulation specified in the radiotap header.

Table 5.2: WiFi APs and chipsets used in evaluations.

Chipset Maker	Router	Chipset	Preamble
Atheros	ASUS RT-AC55U	QCA9557	Long
Broadcom	ASUS RT-AC66U	BCM4331	Short
Ralink/ Mediatek	TP-Link TL-WR841N	MT7628NN	Long
Realtek	Edimax BR-6478AC	RTL8192CE	Long
Marvell	Linksys EA3500	88W8366	Short

To evaluate end-to-end performance, we directly connect DREW to various off-the-shelf WiFi APs. We measure system-oriented metrics, such as TCP/UDP throughputs and round-trip time (RTT). Table 5.2 summarizes the WiFi APs and the chipsets inside. All APs are unmodified and WiFi encryption (WPA2) is enabled. Table 5.2 also lists the default preamble setting. This preamble setting only applies to QPSK packets since BPSK packets always use the long preamble. We use the AP's setup webpage to select WiFi channel 9, and in Realtek's webpage we select QPSK. We use iperf3 [41] to measure throughputs. Specifically, a Windows laptop running an iperf3 server is connected via LAN to the AP. Then, we use DREW and run the iperf3 client on the BLE side. We further set the UDP bitrate to 2Mbps for downlink since DREW exceeds iperf3's default of 1Mbps.

5.3.2 Microbenchmark

Table 5.3: Packet Error Rate (WiFi to BLE) (%)

Chipset	5m		10m		20m	
	BPSK	QPSK	BPSK	QPSK	BPSK	QPSK
Atheros AR9462	0.02	0.00	0.00	0.05	0.07	0.05
Broadcom BCM4313	0.10	0.76	0.07	1.32	0.63	1.83
Ralink RT3290	0.00	0.00	0.00	0.27	0.12	0.49
Realtek RTL8188CE	0.05	0.46	0.05	1.42	0.15	1.44

Table 5.3 shows DREW's excellent performance in receiving standard WiFi packets. The PER with Atheros is very close to 0.00% across all distances. Thanks to DREW's good reception, the overall packet error comes from very occasional background interferences (e.g., 0.07% PER is 3 packet misses out of 4096 packets) and the performance difference between BPSK and QPSK is insignificant. Ralink also has good performance, achieving 0.00% PER for both BPSK and QPSK

at 5m. The PER increases with distance, and QPSK has a higher PER than BPSK because QPSK has smaller decision regions. Broadcom and Realtek have similar performance numbers and are slightly worse than Atheros and Ralink. Even so, all PERs are less than 2% at 20m, corroborating the effectiveness of our algorithms and using IQ sampling. Several variables can impact PER. Actual WiFi cards may have a slight frequency deviation from the ideal WiFi channel, which affects the optimal decision boundaries. Additionally, IQ imbalance and circuit non-linearity can cause slight variations in PER.

Table 5.4: Packet Error Rate (BLE to WiFi) (%)

Chipset	5m	10m	20m
Atheros AR9462	4.86	7.30	10.03
Broadcom BCM4313	1.42	2.56	7.50
Ralink RT3290	4.57	6.23	12.21
Realtek RTL8188CE	10.86	11.65	10.21

Table 5.4 shows the PER in the BLE-to-WiFi direction. Because the BLE chip does not have an external PA, its lower transmit power incurs a higher PER. Broadcom shows the best performance. From experiments, we found that Broadcom's chips are capable of correcting even substantial timing jitters and LO leakage. These qualities translate into better PER in realistic settings. Atheros and Ralink show comparable trends, with Ralink having higher PER at greater distances. Realtek shows the worst PER, which is consistent with the prior report [78] of an inferior performance. To investigate Realtek's chip further, we have tested sending packets at longer intervals but the results are consistent. There might be other issues associated with Realtek's monitor mode or its driver, such as unintended WiFi scan.

5.3.3 TCP and UDP Throughputs

Table 5.5: TCP/UDP Throughputs (kbps)

Direction	Uplink (BLE to WiFi AP)						Downlink (WiFi AP to BLE)					
Transport	TCP			UDP			TCP			UDP		
Distance	5m	10m	20m	5m	10m	20m	5m	10m	20m	5m	10m	20m
Atheros	757	746	639	891	881	753	1234	1246	1010	1388	1399	1260
Broadcom	770	739	660	872	856	758	1332	1324	1231	1515	1516	1454
Ralink	785	752	672	908	892	765	1344	1337	1258	1608	1607	1509
Realtek	767	735	617	865	843	708	1300	1233	1147	1520	1502	1336
Marvell	693	543	515	779	582	569	1325	1187	1140	1535	1455	1403

Table 5.5 shows `DREW`'s transport-layer throughputs. There are multiple factors, besides the PHY performance, that can impact the overall system goodput. The 802.11 standard allows different

implementation options and variations, particularly in the MAC layer. For example, devices can choose different CCA methods and they are all 802.11-compliant. Furthermore, the standard does not specify the details of rate adaptation, which dynamically changes the modulation of transmission. WiFi APs may use different or proprietary *rate adaptation algorithms*, leading to throughput variations.

For uplink, the zero-wait packet transmission design provides high throughputs. Atheros and Broadcom perform very similarly with TCP and with UDP. We found that the MAC layer implementation of Ralink works really well when consecutively receiving a series of packets. Combined with our zero-wait packet transmission, Ralink achieves even better performance than Atheros and Broadcom. Realtek's throughputs are mostly comparable but exhibit a more significant decrease at 20 meters due to its worse PHY layer. `DREW`'s throughputs are considerably higher than prior work (since they must copy the entire packet into the memory before transmission). (In general, the difference is ~100kbps for TCP and ~200kbps for UDP.) The Marvell AP is found more sensitive to the residue DC, thus having a lower uplink throughput than others. However, `DREW`'s zero-wait packet transmission still yields higher throughputs at 5m than FLEW.

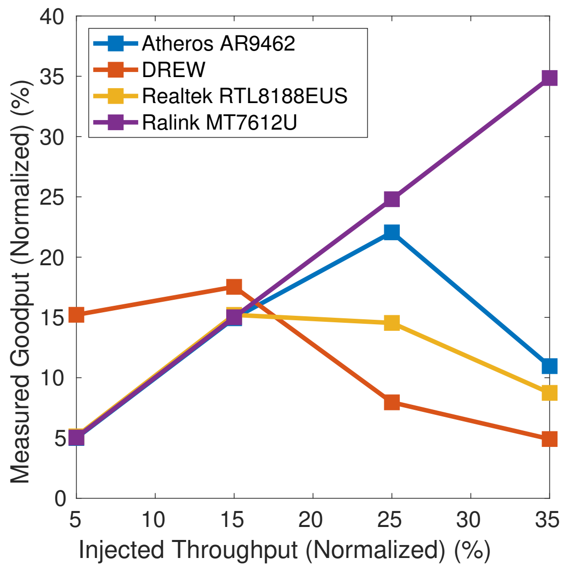
For downlink, QPSK demodulation doubles the throughput. Ralink performs best and Broadcom also performs well. Broadcom devices exhibit good compatibility and stability but with slightly lower throughputs. Since TCP requires transport-layer ACKs and UDP requires PHY-layer ACK packets, the BLE-to-WiFi performance can affect downlink throughputs. So, Realtek and Marvell have good throughputs at shorter distances but their throughputs decrease as the distance increases. Finally, the Atheros AP we use is based on Atheros' LSDK, and LSDK is found to have a peculiar behavior: RTS/CTS is always used regardless of the configuration. This extra overhead lowers the throughputs.

Table 5.6: Round-trip Time (ms)

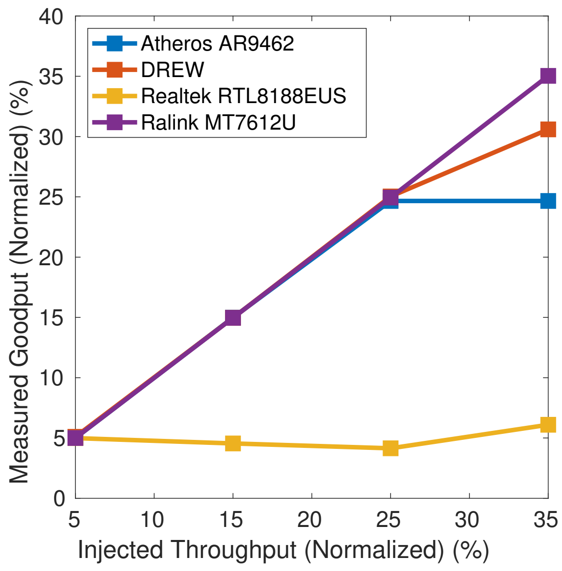
	LAN		WAN	
	μ	σ	μ	σ
Atheros	4.56	0.36	11.02	0.15
Broadcom	2.94	0.37	10.68	0.89
Ralink	6.35	0.84	12.62	0.70
Realtek	3.26	0.52	10.65	0.34
Marvell	3.38	0.82	11.38	0.99

5.3.4 RTT

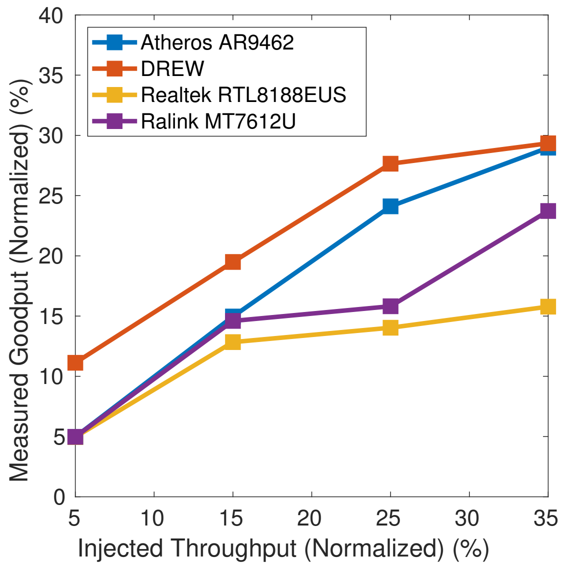
We measure the round-trip time (RTT) through LAN and WAN using `ping`. We ping the AP (LAN) or 8.8.8.8 (WAN) 10 times and calculate the mean (μ) and standard deviation (σ), as shown in Table 5.6. The ethernet connection between each AP and 8.8.8.8. has an RTT of 6.68ms.



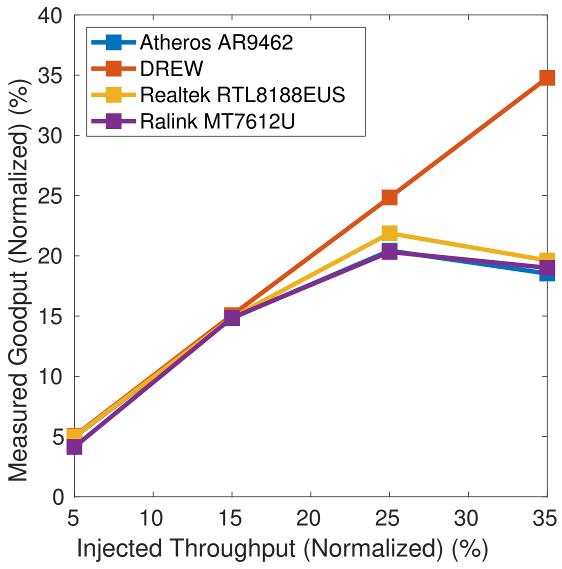
(a) TCP UL



(b) UDP UL



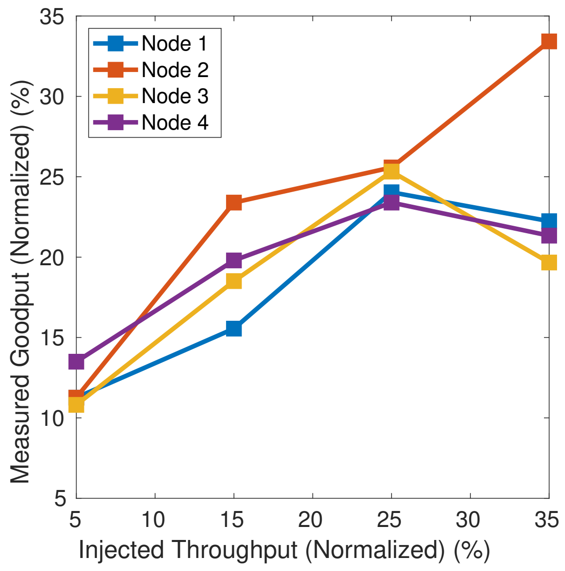
(c) TCP DL



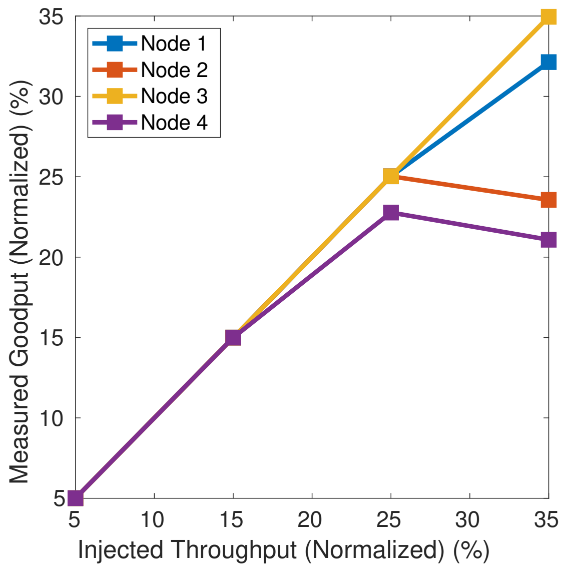
(d) UDP DL

Figure 5.11: Coexistence with COTS WiFi devices

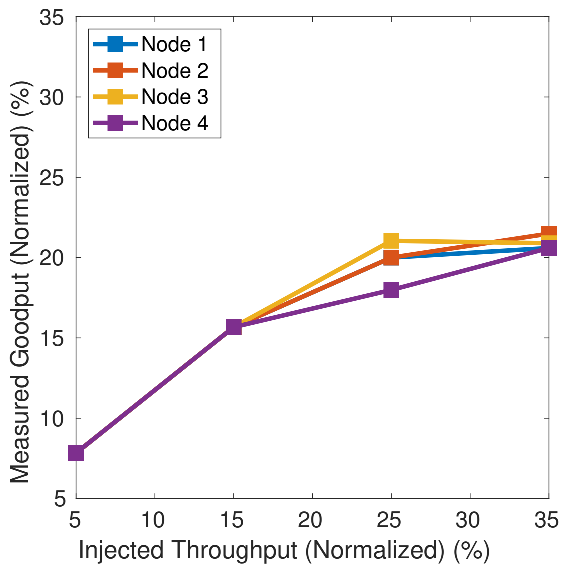
Table 5.6 shows that `DREW` has good RTTs of 3~6ms for LAN and 11~13ms for WAN. The RTT is affected by how fast the AP responds and forwards packets and it is dependent on the processing speed of each AP. The Broadcom AP has the fastest processor (1GHz) and the lowest RTT. The Ralink AP has comparatively higher RTT because it uses the slowest processor (575MHz).



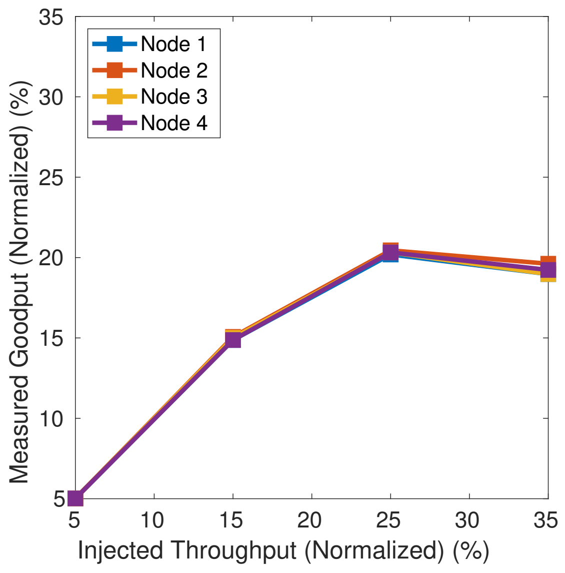
(a) TCP UL



(b) UDP UL



(c) TCP DL



(d) UDP DL

Figure 5.12: Coexistence with DREW devices

5.3.5 Coexistence

To demonstrate DREW's ability to coexist with other devices like a normal WiFi device, we measure the throughputs of concurrent connections with multiple devices.

5.3.5.1 Coexistence with WiFi Devices

To evaluate DREW's coexistence with others, we use the Broadcom AP and maintain 4 active connections with Atheros, DREW, Ralink, Realtek WiFi cards. The AP's LAN port is connected to a laptop running 4 iperf3 servers listening on different ports. We first measure the maximum throughputs of individual devices, and then simultaneously run 4 iperf3 clients and inject 5, 15, 25 or 35% of the throughputs per device into the network.

We tested TCP and UDP in both directions and plotted the measurement results in Fig. 5.11. For each test-case, the network was not saturated when injecting 5% or 15% of throughputs per link. The network is at maximum capacity when injecting 25% per link. The network is oversubscribed and spectrum contention is high when injecting 35% per link. As shown in Fig. 5.11(a), the devices coexist nicely with each other when the network is not saturated and each throughput matches the injected throughput. The throughput of DREW is actually higher because of the TCP packet size iperf3 uses. For 25% per link, the network is saturated and the throughputs are lower than the injected throughputs, and it is more so for the oversubscribed case. Of all the clients, the uplink throughput of Ralink is least affected. This is consistent with the findings in Sec. 5.3.3 where the MAC layer of Ralink utilizes and accesses the spectrum proactively. With a more dynamic protocol like TCP, we observe the throughput of DREW is affected more under saturation. However, it does not completely drop to zero and still allows the data to be transmitted successfully. In the UDP case where the WiFi traffic is more unilateral, Fig. 5.11(b) shows that the throughputs of Ralink, DREW and Atheros coexist nicely, although they react differently in an oversubscribed environment. Realtek performs badly in this test and cannot access the spectrum efficiently in the presence of other unilateral UDP traffic. For downlink, the data packets are mostly sent by the AP, and thus the results are different from uplink. Figs. 5.11(c) and (d) show that DREW is least affected in saturated conditions under AP's arbitration.

5.3.5.2 Coexistence with DREW Devices

To evaluate the coexistence performance of multiple active DREW devices, we use 4 DREW clients and repeat the coexistence evaluation. Fig. 5.12 shows a similar trend for all nodes since they have the same timing and MAC-layer implementation. The throughputs increase linearly before saturation and taper off at 25% and 35% injections per node. There are some variations in throughputs due to the randomness in the WiFi MAC layer. For uplink, the 4 nodes contend for the spectrum at the same time, so the uplink throughputs vary more than the downlink (since the AP transmits most of the traffic for downlink).

Table 5.7: Throughputs with active BT connections

	TCP UL	UDP UL	TCP DL	UDP DL
No BT	780 kbps	896 kbps	1359 kbps	1552 kbps
1 BT	786 kbps	896 kbps	1324 kbps	1566 kbps
2 BT	792 kbps	898 kbps	1314 kbps	1586 kbps

5.3.5.3 Coexistence with Bluetooth Devices

We measure the throughputs of DREW when simultaneously using Airpod earbuds and Sennheiser headphones. Table 5.7 shows that these active Bluetooth connections do not have significant effect on the throughputs. Furthermore, we do not observe any audio stuttering or discernible defects with these Bluetooth headphones when DREW is running the tests. This good coexistence performance comes from the fact that Bluetooth devices will minimize interference by avoiding the channels used by nearby WiFi APs.

5.3.6 End-to-End Applications

DREW enables new applications that no prior work could support. Specifically, the bit rate of a real-time, stereo, Hi-Fi quality audio is 1.411Mbps (= $44100 \cdot 16 \cdot 2$). DREW's unique support of QPSK (2Mbps) is crucial for streaming audio, since the throughputs of prior work (BPSK) are insufficient (FLEW: 857kbps, Unify: 655kbps). DREW (~1.6Mbps transport-layer throughput) can directly stream uncompressed, bit-accurate audio, which guarantees lossless, wired-equivalent audio quality and completely eliminates algorithmic latencies (¿100ms for SBC) of audio compressions. In fact, even existing Bluetooth headphones ("Bluetooth Classic") are not capable of streaming such an audio stream, since the Bluetooth standard **only** supports lossy audio formats (e.g., SBC, AAC and aptX) with heavy (~4x) compressions. Furthermore, BLE is not compatible with Bluetooth Classic. Streaming audio over BLE has even higher compression (4~8x) due to BLE's lower throughput and is not supported on most (Bluetooth 5.1 or older) devices.

The direct WiFi connectivity enables DREW to leverage **audio over IP** solutions, thus circumventing the limitations of the Bluetooth stack. Of the many audio over IP solutions available, we choose Scream [118] to demonstrate audio streaming. Scream is a simple Windows driver that creates a virtual sound card and streams uncompressed sound samples over UDP. We use the unicast mode and the UDP packets are sent over WiFi. On the BLE side, bit-accurate audio samples are outputted to the speakers using Scream's Linux receiver. The bit rate is measured at 1.44Mbps. Interestingly, the BLE chip we used (KW41Z) only supports Bluetooth 4.2[1] and does not support Bluetooth Classic or BLE audio. With DREW, however, we can directly stream uncompressed audio via WiFi and offer higher audio quality than Bluetooth.

This throughput advantage also enables fetching high-quality multimedia contents directly from the Internet. We have used DREW to watch 720p (and some 1080p) Youtube videos without buffering. We also used DREW to stream Netflix at 720p, the maximum resolution under DRM restrictions.

With WiFi's global routability, DREW is also well-suited for IoT applications. With DREW, IoT

[1]KW41Z's stack was not certified for all Bluetooth 5 features but the hardware supports IQ sampling. The IQ sampling was incorporated in Bluetooth 5.1, and thus newer chips' hardware support it. The standard does not specify circuit implementation and ULP designs commonly eliminate mixers.

devices can directly retrieve information from various websites or upload sensor readings to the cloud. Furthermore, DREW's direct interoperability with WiFi allows ULP BLE devices to use normal web services without requiring custom packet translation or gateways. For example, we have used DREW to send prompts and receive responses to/from the ChatGPT server. We also use DREW to perform Google Voice Search. These use-cases are particularly useful for adding valuable features to ULP wearable devices, such as smartwatches.

5.3.7 Power Consumption

Table 5.8: Power-consumption measurements

	Tx (A)	BPSK Rx (A)	QPSK Rx (A)
Atheros AR9271	0.49	0.07	0.07
Ralink RT3572	0.42	0.15	0.15
Ralink MT7612U	0.42	0.15	0.15
Realtek RTL8811AU	0.28	0.08	0.08
FLEW [78]	0.16	0.11	Not supported
Unify [103]	0.04	0.04	Not supported
DREW	0.07	0.07	0.07
DREW w/ K22F sleep	0.05	0.05	0.05

The KW41Z chip used in our implementation has ultra-low power consumption. According to its datasheet, the active currents (at 3.6V) are as low as 6.1mA (Tx) and 6.8mA (Rx), which are considerably lower than those of a typical WiFi chip. For example, a QCA9377 module consumes 538mA (Tx) and 155mA (Rx) at 3.3V [119].

Table 5.8 shows the power consumption of DREW, various common WiFi dongles, and prior work. We measure the USB (5V) current during active transmission or reception (and every device uses the same modulation format). DREW has the lowest power consumption among chips that support QPSK. Also, the USB power on the USB-KW41Z board is shared by the BLE chip and the debugger. The debugger itself is a powerful Cortex-M4 K22F microcontroller running at 120MHz, which contributes to the overall power consumption. We further tried downclocking the K22F and using sleep modes (with the WFI instruction) whenever possible, and the power consumption is reduced to 0.05A for all cases. Further power saving may be possible by enabling KW41Z's buck converter or by removing the debugger completely. The KW41Z is operated at 3.5dBm. Since the PA is switched on 50% of the time, the transmit power is approximately $3.5-3 = 0.5$dBm, which is within the power range specified in the WiFi standard. If a higher power is needed, an external PA can be used. Assuming a nominal efficiency of 33%, the PA will add $(0.1\text{W}-0.001\text{W})/0.33/5\text{V} = 0.06\text{A}$ to the transmit current.

5.4 Discussion

5.4.1 Advantage over using WiFi 64-QAM

Although a prior work [120] indicates that utilizing WiFi's 64-QAM modulation ***at the same transmit power*** may consume less energy than BLE during packet transmission, the goal of `DREW` is to enable WiFi connectivity on **existing** hardware that comes with simple, low-cost hardware and without high-speed DSP or wideband RF ADCs/DACs. Furthermore, a typical BLE chip has a sensitivity of -97dBm [2], which is 19dB better than -78dBm [56] of MCS7 (64-QAM). Therefore, without affecting the range, the transmit power of BLE should be 19dB lower than MCS7, which would have resulted in much lower power consumption for BLE. (Also, DSSS waveforms can achieve an even better sensitivity at -102dBm [56] thanks to the advantage of PSK over FSK.) `DREW` is significant in that a ultra-low-power WiFi connectivity is possible by reusing the BLE's hardware, since BLE chips have ultra-low continuous currents (Tx: 6.1mA, Rx: 6.8mA) with low-power ARM processors. (In contrast, the WiFi chip used in [120] is known for its high current draw in the industry (DSSS Tx: 240mA, DSSS Rx: 95mA) and the chip consumes more than 20mA even with WiFi disabled [121].)

5.4.2 IQ Sampling

IQ sampling is available on a wide range of new BLE chips to support localization and AoA features defined in the Bluetooth standard. Different BLE chips have different procedures for enabling the IQ sampling. We set up IQ sampling by following the *transceiver DMA* procedure described in NXP's manual [109]. Other parts of the Bluetooth firmware and software stack are not utilized in `DREW`. By directly fetching IQ samples with the ARM core (Sec. 5.2.4.1), the length of IQ capture need not be known in advance and is not limited by the buffer size or by the DMA configuration. However, this design requires real-time IQ processing, thus making SIMD a critical component. Finally, we have tested using IQ sampling with OFDM waveforms, but the challenge is that the IQ sampling is relatively narrow-band and is insufficient for decoding a packet without errors.

5.4.3 Spectrum

In terms of spectrum, `DREW` is at least as efficient as prior CTC designs, and is twice as efficient when QPSK is used. Furthermore, typical WiFi systems still commonly use DSSS for critical traffic, such as management packets, beacons, RTS/CTS, and ACKs. Spectrum efficiency can also be improved by spatial diversity and power control between devices.

5.4.4 Packet Detection

The packet detection of `DREW` is optimized for both speed and memory. We directly use ARM registers as 32-bit FIFOs for the fastest access. Updating 4 FIFOs (for 4 IQ samples) takes 12 cycles and comparing these 4 FIFOs with both long and short preambles takes 16 cycles. In contrast, the correlation-based method requires 128 (= 32 μs $\cdot$ 4 samples/μs) complex multiplications and additions for every IQ sample received. Additional loads and stores for memory access are also needed. Such a method exceeds the capability of typical ARM microcontrollers.

5.5 Related Work

Earlier CTC works [7, 6, 8, 9] use one packet to represent one or a few modulation symbols for a heterogeneous device to receive. These designs have lower throughputs and they need to modify both Tx and Rx. Recent CTC works directly transmit compatible waveforms at the PHY layer and achieve higher throughputs. These works include WiFi-to-Bluetooth [72, 73, 76, 74, 77], WiFi-to-Zigbee [17, 18, 19, 20, 94, 95], and WiFi-to-LoRa [96, 97]. However, they only enable one-way communication and require modifying the transmitter. Furthermore, they focus on generating 1Mbps FSK [72, 73, 76, 74, 77], 250kbps Zigbee [17, 18, 19, 20, 94, 95] or slower waveforms. Some other CTCs [122, 123] modify WiFi devices to receive non-WiFi waveforms. Other CTCs focus on Bluetooth–Zigbee [22, 21] or LTE [14, 16, 15, 23]. CTC is also related to concurrent-communication systems [124, 125, 126].

FLEW [78] and Unify [103] achieve bidirectional WiFi–FSK communication without any modification on the WiFi side. However, they require older FSK chips that have mixers, and the PHY throughput is capped at 1Mbps. Other software-defined-radio-based [49] and simulation-only [98] systems have also been proposed but are not directly applicable to COTS WiFi systems.

Many prior work [127, 128, 129, 130, 131, 132, 133, 134, 135] use BLE's AoA feature and propose various post-processing algorithms for localization. To the best of our knowledge, `DREW` is the first that uses the AoA feature for data communication (specifically WiFi communication) instead of localization.

5.6 Conclusion

`DREW` enables direct communication between ULP BLE chips and WiFi APs. We proposed innovative use of the PA and IQ sampling, and devised various algorithms with SIMD acceleration. `DREW` is applicable to mixer-less BLE chips and uniquely supports QPSK. It is shown to have good coexistence, low power consumption and much higher throughputs.

CHAPTER 6

BBC: Enabling BLE to Support Bluetooth Classic

6.1 Introduction

Since its invention more than 20 years ago, Bluetooth has been widely used for short-range, low-power communications. One major Bluetooth application is streaming audio to wireless headphones and earbuds. Each year, more than one billion (1.46 billion [136] in 2023) of Bluetooth audio streaming devices have been shipped worldwide and Bluetooth is the dominant wireless technology for audio. To stream high-fidelity audio, the overwhelming majority ($> 85\%$ [136]) of Bluetooth headphones use the "Bluetooth-Classic" protocol.

In recent years, the Bluetooth SIG (Special Interest Group) has been heavily promoting the transition from Bluetooth Classic to Bluetooth Low Energy (BLE), since the latter enables simpler hardware implementation and is more energy-efficient. However, BLE is an entirely separate and different protocol from Bluetooth Classic. Specifically, BLE uses different bit-processing, frequency hopping, timing, packet format from Bluetooth Classic and does not support A2DP (Advanced Audio Distribution Profile [137]) at all. Consequently, BLE chips cannot directly communicate with Bluetooth-Classic devices. BLE-only chips are readily available (e.g., in ultra-low power BLE tags and BLE locks) in the market and are smaller, cheaper and highly energy-efficient (supporting years of battery life). However, they cannot be used with Bluetooth-Classic headphones. On the other hand, because the vast majority of headphones use Bluetooth Classic, multimedia devices must use *dual-mode* Bluetooth chips, which contain both Bluetooth Classic and BLE hardware components. As shown in Bluetooth SIG's block diagram [138, 139], Bluetooth Classic and BLE are very different in each layer and share very little in common. As a result, dual-mode chips have to maintain two full and separate communication stacks (from PHY to application profiles), thus increasing their cost, size and power consumption.

This incompatibility between BLE and Bluetooth Classic is becoming a major obstacle as the number of BLE devices continues to grow rapidly. As the industry shifts towards more and more BLE devices, billions of Bluetooth (Classic) headphones users have already purchased (and the

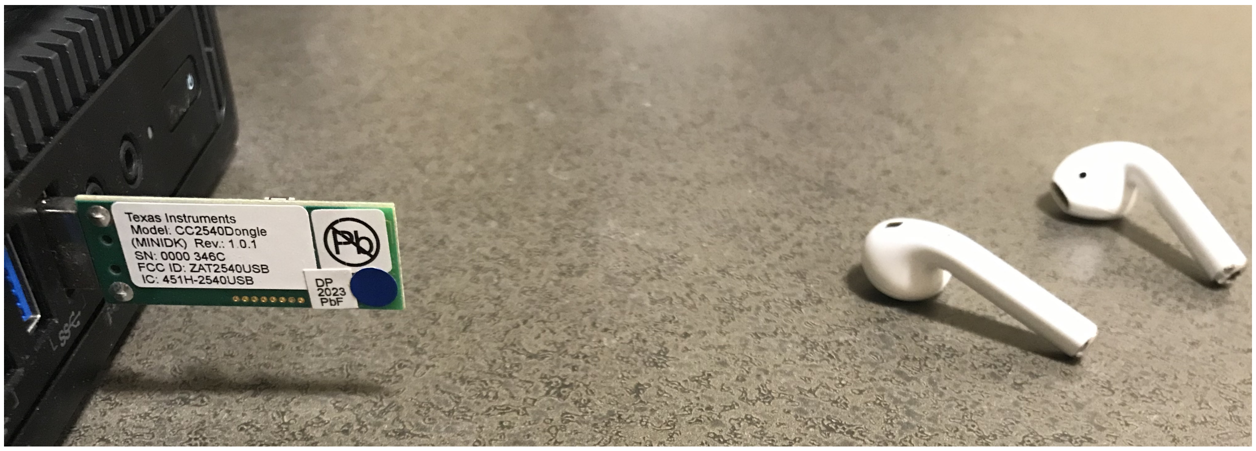


Figure 6.1: BBC enables direct communication between the BLE chip and Bluetooth Classic headphones

overwhelming majority of headphones being sold in today's market) may become unusable when most devices use BLE.

To address this important and practical problem, we propose BBC, which enables direct communication between BLE and Bluetooth-Classic chips. BBC emulates full operation of Bluetooth Classic using BLE chips so that conventional, unmodified Bluetooth headphones can be used with BLE-only chips. BBC offers several important benefits. First, BBC is an ideal solution for backward compatibility on newer or BLE-only chips without the need for "dual-mode" chipsets that require two full stacks (e.g., separate physical/MAC/network layers). BBC is especially attractive during the transition from Bluetooth Classic to BLE when billions of headphones are still relying on Bluetooth Classic. Second, even without considering this transition, BBC creates a new mode of communication and adds useful audio connectivity on current BLE-only devices, such as BLE beacons or BLE fitness trackers. Finally, BBC exactly matches the audio quality of Bluetooth Classic. Blind tests [140, 141] have shown that high-quality audio codecs (AAC or aptX) used in Bluetooth Classic offer noticeably better sound quality than Bluetooth SIG's LE audio proposal.

For hardware vendors, BBC enables using simpler, cheaper and smaller radio chips to support both BLE and Bluetooth Classic, thereby encouraging its adoption. BBC relieves hardware complexity using software. Also, because modern devices have considerably greater computational power than 20 years ago, this emulation only incurs very minimal overhead on modern processors. Our evaluation shows that BBC only requires about 1.6% of the CPU time. For users, BBC addresses the problem of compatibility between different Bluetooth standards without modification to the hardware. Furthermore, BBC also enables new applications that were not possible on existing hardware. For example, audio streaming becomes possible with existing BLE devices using BBC.

BBC matches the operational range of Bluetooth-Classic devices with the same transmit power.

Using BLE chips with a transmit power of 4dBm, BBC can stream music without glitching at 10 meters, which is the same operational range of Class 2 (4dBm [142]) Bluetooth-Classic devices. Note that Bluetooth-Classic devices *can* have higher transmit power, which gives a greater advertised range (e.g., 100m). However, BLE devices can also use higher power. In particular, both Bluetooth Classic and BLE have "Class 1" devices, which have a transmit power of up to 20dBm. Note, however, that Class 1 devices are for industrial applications and most consumer electronic devices are Class 2 or Class 3 [143]. BBC simply matches the operational range of Bluetooth Classic under the same transmit power, and does not increase or decrease the communication range. Similar to the case of industrial applications, higher transmit power (e.g., by power amplifiers) can be used if a greater range is desired.

BBC utilizes a hardware timer and BLE's raw transmission and reception routines whereas all other lower layers of Bluetooth Classic are emulated in the driver. This architecture is significantly different from conventional Bluetooth-Classic chips. In fact, BBC's architecture closely resembles WiFi's architecture in that the driver directly sends or receives the over-the-air (OTA) FSK bits to/from the radio chip. In contrast, typical Bluetooth-Classic chips use the HCI (Host-Controller Interface) specification [142] for the communication between the driver ("Host") and the radio chip ("Controller"). Note that Bluetooth's HCI traffic (which is the capture Wireshark provides) is never the OTA FSK bits. Instead, the HCI traffic is either high-level commands or transport-layer payloads (which require multiple layers of processing to produce the OTA FSK bits and the controller may delay commands or payloads arbitrarily). This layer of indirection and complexity makes diagnostics difficult, and because lower-layer processing and logics (such as connection establishment and link management transactions) are hard-baked into the chips, it is inflexible and prone to incompatibility. For example, many Bluetooth devices run into difficulties when establishing an initial connection with peers, or when negotiating the configuration of wireless links, because the driver can only indirectly control the connection establishment by issuing commands whereas the actual radio messages and exchanges are autonomously handled by the Link Manager (LM), which is hard-baked in Bluetooth chips.

In contrast to this conventional architecture, BBC offers several crucial benefits. First, since the driver directly sends and receives FSK bits, it has complete knowledge of OTA transmissions and receptions, and thus has much better diagnostic capabilities. This is akin to WiFi systems in which OTA transmissions and receptions are directly visible to the driver. Second, because the lower-layer processing is emulated in the driver, BBC does not require Bluetooth-Classic-specific blocks and components (such as the Link Manager) on the radio chip. Thus, BBC can use BLE chips as the hardware. Finally, BBC is highly flexible as the lower-layer processing is easily upgradable by simply compiling new drivers. Unlike WiFi systems where the connection method, authentication and encryption have gone through many iterations, the original (1999) Bluetooth paging/authentication/encryption [144] is still in use, because with the conventional architecture, these designs are hard-baked in the chip.

The key question BBC aims to answer is: *Instead of using dedicated hardware blocks for Bluetooth Classic, can we build corresponding blocks in software so that simpler hardware, such as a BLE chip, can be emulated as a Bluetooth-Classic chip?* BBC shows that such software-based emulation is actually possible, even though many processing blocks in Bluetooth Classic are highly dependent on hardware. (For example, the bit processing, connection establishment, and encryption are directly dependent on the Bluetooth Classic's native clock.) In addition to enabling Bluetooth Classic using BLE chips, BBC also provides insights into the inner-workings of Bluetooth Classic by presenting a fully-functional, emulation-based Bluetooth system. In the literature, Bluetooth Classic chips are treated as black boxes, and the research into the processing between the raw FSK bits and the HCI layer is scarce. BBC directly bridges this gap and emulates every step between the two layers.

Emulating Bluetooth Classic's operation is very challenging because its lower layers have numerous highly-intricate components, and implementing the full emulation is highly error-prone. To the best of our knowledge, no one in academia or the open-source community has successfully built end-to-end, fully functional (including both link establishment and audio streaming) Bluetooth-Classic systems from the ground up. BBC is unique in that it accurately emulates various Bluetooth-Classic components (especially below the HCI layer) and enables off-the-shelf BLE chips to establish, negotiate, authenticate, encrypt Bluetooth-Classic connections and then stream audio to Bluetooth headphones, all without the help of any Bluetooth-Classic chip.

Although the design of Bluetooth is documented in the Bluetooth Core Specification [142], it isl challenging to build several key hardware components in software and emulatie them accurately. One of the challenges is parsing the unconventional use of terminologies in the standard. For example, "baseband", in the Bluetooth standard, includes controlling carrier frequencies, connection establishment between two devices, connection authentication, payload encryption, packet acknowledgement/retransmission, addressing and flow controls. These components are *not* considered baseband in other (WiFi or cellular) standards. Another challenge is that because the frequencies are constantly changing in Bluetooth, the frequency hopping and packet generation/processing must both be absolutely correct in order to communicate with Bluetooth headphones. Furthermore, the frequency-hopping sequences are complicated and are both address- and state-dependent. (The sequences are completely different before and after a connection is established, and are also different for Central and for Peripheral.) The packet generation and processing involves numerous fields (e.g., headers, scrambler seeds, error checking, encryption) that are time-varying (dependent on the Bluetooth clock) and/or address-dependent. All of them have to be emulated correctly for successful communications. Finally, debugging Bluetooth communication is difficult because every packet uses a different frequency. This is especially challenging for debugging the connection establishment process because the frequency-hopping pattern changes once Peripheral receives a paging packet from Central (and changes again once the connection is established). It is thus almost impossible for a narrow-band sniffer to reliably follow the entire process because such a sniffer cannot simultaneously capture traffic on two different frequency-hopping sequences. Also, during connection establishment, Bluetooth devices use random frequencies from the entire 2.4GHz spectrum, which is larger than the bandwidth of most SDR (Software-Defined Radio) equipment.

6.2 System Design

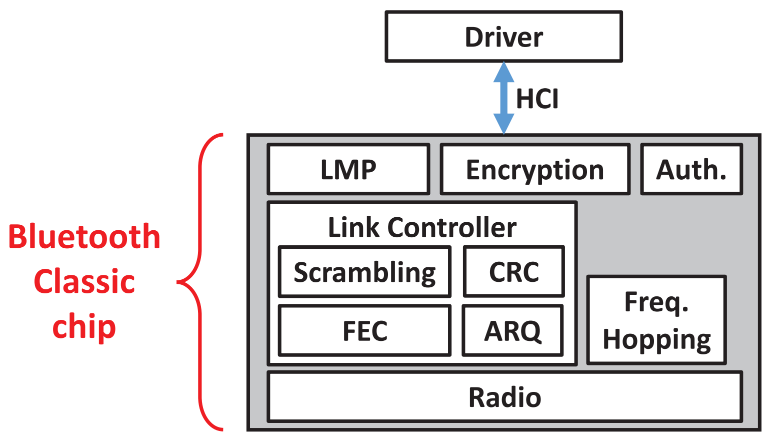
6.2.1 Primer and Overview of Bluetooth Classic

We first review several key hardware components of Bluetooth Classic. Fig. 6.2a shows the block diagram of a typical Bluetooth Classic chip. In a Bluetooth connection pair, one device assumes the role of Central while the other is Peripheral. Bluetooth Classic is a strict time-slot based protocol and Central assigns the time slots. Central and Peripheral both maintain a Bluetooth clock for precise time-slot communication. The initial synchronization of two clocks is achieved via paging, where Central informs Peripheral its clock value in a 6-packet exchange. In each time slot, packets are sent over a different RF frequency carrier. The RF frequency depends on the Bluetooth clock and address, and thus Central and Peripheral hop to the same RF frequency once synchronized.

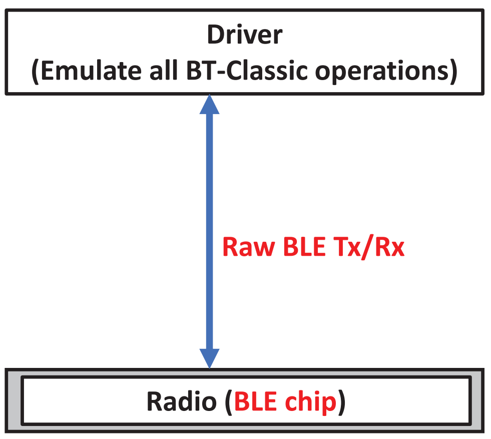
Bluetooth-Classic chips also apply a significant amount of bit processing for each packet, which is done by the Link Controller (LC). The LC's bit processing is time-variant because many computation steps depend on the Bluetooth clock. To ensure reliable communication, the LC also handles the ARQ (automatic repeat request). The component above the LC is the Link Manager (LM), which runs the Link Manager Protocol (LMP) to establish, negotiate and control a logical Bluetooth connection. The LM also controls link authentication and encryption, which are also implemented in hardware on typical Bluetooth-Classic chips. Bluetooth's authentication runs custom hash functions for identity verification. The calculated hash is also used in encryption, which runs a custom cipher and applies additional bit processing to the data stream.

6.2.2 Architecture

Fig. 6.2b shows the architecture of BBC. BBC uses the raw transmit and receive commands on BLE chips to send and receive raw FSK packets. These packets are directly relayed to and from the driver. The driver will emulate all necessary processing (as described in the following sections) of Bluetooth Classic.



(a) Traditional Bluetooth



(b) BBC

Figure 6.2: Comparison of Architectures.

Specifically, BBC processes the raw FSK bitstreams in software. Since Bluetooth Classic is a strict slot-based system, we devise a slotted communication method that allows software-based bit generation and reception without relying on the dedicated hardware. For each slot, the packets are further processed by the Link Controller, emulated in software. The Link Controller handles various bit processing, including bit scrambling, FEC, CRC, and generating various headers. BBCalso emulates the Link Manager in software, which runs the Link Manager Protocol (LMP) to create, negotiate and modify the physical-layer and MAC-layer behaviors of a Bluetooth connection. LMP is also used to authenticate a connection and to turn on the encryption. The authentication and encryption functions are also emulated in software in BBC. The custom hash function runs directly inside the driver for authentication. The driver also performs bitstream encryption. Finally, with all these components emulated, a transport layer connection is established, which allows the complete Bluetooth function.

Compared to the traditional architecture (Fig. 6.2a), BBC has *lean* hardware. Without relying on several hardware blocks (i.e., the Link Controller, the Link Manager, authentication/encryption, frequency hopping), BBC can use BLE chips as hardware.

6.2.3 Bluetooth Clock

In the Bluetooth-Classic protocol, each device maintains a 28-bit clock. The 28-bit clock monotonically increases every 312.5μs, and is crucial for the lower layers of the Bluetooth operation. Each device uses the clock to switch between transmission and reception (every 625μs) and hops to a different frequency calculated (with pseudorandom algorithms) from the Bluetooth clock. The bit scrambling and encryption are also dependent on the instantaneous clock value. So, two

Bluetooth devices must be precisely synchronized to communicate with each other correctly. This synchronization is first achieved by the *paging* process (described in Sec. 6.2.5). Two devices must then continuously update this clock and stay synchronized.

To implement the Bluetooth clock, we use the hardware timer on BLE chips, which runs at 1MHz and generates a hardware interrupt every 1250 cycles (=1250μs). After chip initialization, all BLE transmit and receive routines in BBC run within the interrupt handler. This pure interrupt-based design ensures a precise and deterministic timing required by the Bluetooth standard.

6.2.4 Slotted Communication

Bluetooth Classic uses frequency hopping and changes the RF frequency 1600 times per second when connected. Since the frequency is constantly changing, Bluetooth divides communication into time slots so that Central and Peripheral will both hop to the same frequency at each time slot (after paging). Slots are scheduled alternately, i.e., Central-to-Peripheral communication takes the even slots and Peripheral-to-Central communication takes the odd slots.

As described in Sec. 2.2, the timer interrupt is triggered every 1250μs, which is the duration of exactly one pair of Central-to-Peripheral and Peripheral-to-Central slots. From BBC's perspective, Central-to-Peripheral is transmission and Peripheral-to-Central is reception. At the start of the interrupt routine, BBC cancels any outstanding reception and then starts a new transmission immediately. After the transmission, BBC immediately switches to reception. Note that BBC does not wait until the Peripheral-to-Central slot to start the reception because BLE chips can infer the beginning of an incoming packet via its preamble and sync word.

The alternating 625μs Tx and 625μs Rx is the mandatory arrangement in the Bluetooth standard. The standard also defines multi-slot transmission, where transmission takes 1875μs (or 3125μs), followed by a 625μs reception. This is an optional feature and must be negotiated through the LMP (Link Manager Protocol) after the paging process. However, we observed that it is used in all conventional Bluetooth headphones because it significantly improves throughput.

We further design BBC to support both single- and multi-slot transmissions. This is achieved by executing two versions of the interrupt routine alternately. The first 1250μs routine always starts a new transmission. For single-slot transmissions, the first 1250μs routine starts reception after transmission and the second 1250μs routine collects the received bytes. For multi-slot transmissions, the second 1250μs routine waits until the transmission is finished, switches to reception, and starts collecting received bytes at the 938μs mark. At the end of the second routine, BBC shuts down the reception.

The received bytes (and bytes to be transmitted) are sent to the driver. Additionally, we prepend the packets with auxiliary information. In the second 1250μs routine, we calculate and send the clock values of the reception slot and of the next transmit slot to the driver. These clock values are critical for the driver to decode the incoming packet and to encode the next transmission correctly. The driver also uses the clock value to calculate the frequency hops of the next transmit slot and the following receive slot. (The latter will also depend on whether the transmission is single- or multi-slot.) The driver sends the two frequency hops, the number of time slots, the number of bytes to be transmitted, and the actual bytes to the BLE chip for transmission.

The length of the transmission is variable and configured when the transmission starts. For the reception, we use a fixed length of 42 bytes after a single-slot transmission and 14 bytes after a multi-slot transmission. This design allows BBC to receive a full packet from Peripheral after a single-slot transmission and to receive the acknowledgement status (the header bits) after a multi-slot transmission.

6.2.5 Paging

Upon powering on the BLE chip and headphones, their Bluetooth clocks are not synchronized at all. In order to establish a connection with Peripheral, Central must first *page* Peripheral so that Peripheral can learn Central's clock and follow it after paging.

In the Bluetooth standard, paging is a 6-packet process (and each packet uses a 625μs slot). Central first sends an ID packet and Peripheral replies with an ID packet. Central then sends an FHS packet, which contains Central's clock value, and Peripheral again replies with an ID packet. These 4 packets have to use Peripheral's access code and frequency hopping sequences. Starting from the 5-th packet, however, both Central and Peripheral switch to Central's access code and frequency-hopping sequence (in the connected state). Typically, the 5-th packet is a POLL packet from Central and the 6-th packet is a NULL packet from Peripheral.

The major complexity of Bluetooth's paging, however, is that the frequency hopping and bit scrambling are enabled at all times, and thus two devices will hop to different frequencies and scramble packets randomly. To successfully exchange the first 4 packets, Central has to precisely **guess** Peripheral's clock.

We found that implementing the Bluetooth Classic's paging process in software is extremely difficult and highly intricate. Bluetooth's paging design makes the connection process inherently random. Furthermore, the description of this guessing process in the standard is convoluted. (The standard defines two separate random sequences during the paging: the page and page response hopping sequences. However, these sequences also change when Central or Peripheral receives a response and progresses to a different stage. The sequences depend on the Peripheral's address, the estimated clock, and a value X. In particular, the description of X in the standard is complicated. X is somewhat related to the Peripheral's clock but the clock value has to be frozen when the first

ID packet is received. For the first 4 packets, the X value itself is not frozen, but it is incremented differently from Peripheral's clock. In addition, X is also used as the scrambler seed for the FHS packet, and thus the entire paging will fail if X or any of the hopping sequences is incorrect.) Essentially, the standard's guessing process is (pseudo)randomly trying different X values.

To debug and implement the paging correctly and to make the problem tractable, we minimize the randomness in the paging process. We achieve this based on several insights. First, the frequency hops of the first 4 packets are entirely deterministic if the Peripheral clock, Peripheral address and X are known. Second, in the pseudorandom algorithm for selecting frequency hops, only 1 bit of the Peripheral clock is used and this bit is also deterministic for the first 4 packets. Therefore, for a given Peripheral address, the first 4 hops depend entirely on the 5-bit X value, which only has 32 possibilities. Third, the 5-th packet is POLL from Central at a frequency hop that depends solely on Central's address and clock, which are sent to Peripheral by FHS (i.e., the 3-rd packet). Since Central assembles and sends the FHS packet, the address and clock can be set arbitrarily. However, FHS must be scrambled with the correct X value.

From these insights, we can see that the paging process ultimately depends on X only. For a given X, the entire 6-packet paging process can be made deterministic and we can generate both frequency hops and every bit in each packet from the X value. Furthermore, the correct X, at the time of the first packet, is simply 5 of 28 bits of the Peripheral clock. Thus, the correct X monotonically increases and wraps around every 40.96s.

We implement a simple and easy-to-debug paging method. For a given X, we generate the frequency hops and the first, third, & fifth packets of the paging process. `BBC` keeps transmitting these three packets while listening on the corresponding frequency hops on the second, fourth, and sixth slots. If the X we choose happens to match headphones' current X, the BLE chip should receive two ID packets and a NULL packet. If these packets are received, `BBC` initializes its Bluetooth clock (with the clock specified in the FHS packet) and updates the clock continuously afterwards.

In the Bluetooth standard, the Peripheral should always respond to the 5-th packet (POLL) with a NULL packet to complete paging. However, when experimenting with Bluetooth headphones, we find that some headphones do not strictly follow this. To handle this problem, we further send two additional POLLs at the next two transmit slots, and `BBC` keeps sending these 5 packets (in total). Note that the bitstream and the frequency of these POLLs packets are different because their Bluetooth clocks are different. We found Apple's headphones usually do not respond to the first POLL and these two additional POLLs are particularly useful.

6.2.6 Slot Management after Paging

If paging is successful, an initial connection with headphones is established and BBC sends the clock of the next scheduled slot to the driver. In BBC, we schedule the first packet transmission 30 slots after the last POLL packet of paging. Each transmission is then sent at 4-slot intervals. Every 4 slots map exactly to the first and second interrupt routines described in Sec. 6.2.3. After paging, the driver starts bit-, packet- and protocol-processing for the connected state (Sec. 2.6~2.12). BBC also monitors whether there is a response for each transmission. If no responses are received for 100 consecutive transmissions, BBC considers the connection lost and restarts the paging process.

6.2.7 Generate Raw FSK Bits

6.2.7.1 Packet Types

The Bluetooth standard defines various packet types. BBC utilizes ID, FHS, POLL, DM1 and DH3 packets. The first three are used in the paging process while the last three are used in the connected state. Each type of packet comes with different encodings and number of fields. ID only contains the access code whereas POLL has the access code and header. FHS, DM1 and DH3 further contain the payload field. Forward-error correction (FEC) is applied to the payload of FHS and DM1. DM1 and DH3 are used for general data communication. DH3 can use up to 3 slots and enables much higher throughputs.

6.2.7.2 Access Code

Each Bluetooth packet begins with a 72-bit access code. This access code does *not* correspond to the address of the receiver of a single transmission. Instead, the access code depends on Peripheral's address during paging, and on Central's address once connected. For example, in the connected state, both Central and Peripheral will use the same access code generated from Central's address.

Additionally, the access code and the address do not have the same bit pattern. Instead, the access code is generated from the lower 24-bit of the address via a complicated process involving two XOR (exclusive OR) with a random sequence and applying the BCH (Bose-Chaudhuri-Hocquenghem) code between XOR operations. For ease of debugging, we generate the access codes at compile time.

The beginnings of Bluetooth Classic and BLE packets are very different. Bluetooth Classic uses 72-bit access codes whereas BLE uses 4-byte access addresses. We set the access address of BLE to the byte 0 to byte 3 of the calculated access codes. This ensures that BLE starts packet reception when a Bluetooth Classic packet arrives, and correctly transmits bits in Bluetooth Classic's format (with the BLE's data field calculated as described in the following subsections).

Note that two distinct access codes are used during the paging process, so BBC has to switch access codes at different slots. Specifically, BBC uses Peripheral's access code when transmitting ID and FHS packets, and uses Central's access code when transmitting POLL packets. If paging is successful, the Central's access code will be used for subsequent traffic.

6.2.7.3 Header and HEC

Some Bluetooth Classic packets contain an 18-bit header, which consists of 10 bits of information and 8 bits of *Header Error Check* (HEC). The 10 bits include the packet type (e.g., DM1), a unique address (AM_ADDR) within Central's network, 1 bit (ARQN) for acknowledgement, and 1 bit (SEQN) for data toggling.

The AM_ADDR of Peripheral is directly tied to the paging process. Specifically, AM_ADDR is explicitly specified by Central in the FHS packet and will be used for all subsequent traffic. In the FHS packet, we set AM_ADDR to 1, and thus the AM_ADDR of all other packets is also 1. The FLOW bit is always set to 1 and the packet type bits are set accordingly. The ARQN and SEQN bits are used for the packet retransmission logic and are calculated as described in Sec. 6.2.9.

The HEC is an error-checking code calculated from these 10 bits using a linear feedback shift register. Furthermore, the shift register has to be initialized with bit 25 to bit 32 of Central's address (except for the FHS packet during paging where Peripheral's address is used).

6.2.7.4 Payload and CRC

For DM1 and DH3 packets, the payload field follows the header. Confusingly, a Bluetooth payload also contains a header known as the *payload header*, which is the first byte (DM1) or the first two bytes (DH3) of the payload. The payload header contains *LLID* and the length of the remaining payload.

The LLID is important for managing the fragmentation at the lowest layer and marking special payload. An LLID of 0b10 indicates the start of a logical packet, while an LLID of 0b01 indicates the payload is a continuation fragment. An LLID of 0b11 indicates the payload is a LMP message used for controlling the Bluetooth link and negotiating configurations. An LMP message is always contained in a single DM1 packet.

The CRC (cyclic redundancy check) of the entire payload field (including the payload header) is appended at the end of the packet. Similarly to HEC, CRC is based on linear feedback shift registers and is initialized with bit 25 to bit 32 of Central's address. However, CRC has a different register length (16 bit) and feedback polynomial.

6.2.7.5 Scrambling

After BBC generates the header and payload (including CRC) in the driver, the entire bitstream has to be processed by the scrambler. The Bluetooth scrambler is a 6-bit linear feedback shift register. The scrambler is initialized by bit 6 to bit 1 of the Bluetooth clock. Then, the scrambler will continuously XOR each bit (of the bitstream) with the output of the shift register.

6.2.7.6 Forward Error Correction (FEC)

After scrambling, FEC is applied to the bitstream. However, the header and the payload must use different FEC codes.

The header portion (the first 18 bits) should be encoded with the (3,1) repetition code (so the output is 54 bits). Bluetooth's order of scrambling and FEC is counterintuitive. Specifically, a scrambler is commonly used to remove long-running 1s or 0s, but the repetition code will again repeat each 1 and 0 three times. However, this is the actual design of Bluetooth and is thus followed.

For packet types such as FHS and DM1, the payload (including CRC) is also encoded. Although the Bluetooth standard describes using the "shortened Hamming code" [142], we find that the specified structure is actually a BCH encoder. We implement such an encoder using linear feedback shift registers that generate 15 coded bits for every 10 information bits. The input to the encoder is padded so that its length is a multiple of 10. For DH3 packets, the scrambled payload is directly used since the payload is uncoded.

6.2.7.7 Pack FSK Bits as Payload

After the FEC, BBC prepends the bitstream with the last 5 bytes of the access code. Note that BLE's access address is set to the first 4 bytes of the access code, and thus the first 4 bytes are automatically transmitted by the chip. These bits are packed as the BLE payload and sent to the chip for transmission.

6.2.8 Decode Raw FSK Bits

When a Bluetooth Classic packet (addressed to Central's network) arrives, its access code will match BLE's access address and raw FSK bits will be collected. These raw bits, along with the two clock values (as described in Sec. 6.2.4), will be sent directly to the driver for decoding. The decoding process is, in principle, the reverse process of Sec. 6.2.7, including removing the FEC, descrambling the entire bitstream, and checking the HEC and CRC.

When the driver receives the bits from the BLE chip, it first removes the first 14 bytes (which includes two clock values, one byte of total length, and 5 bytes of the tail of the access code). The next 54 bits (which use the repetition code) are decoded using majority vote. The next 255 (=15·17) bits correspond to the payload portion of a packet. We extract the first 10 bits of every 15 bits and concatenate all output bits. Note DM1 is the only data-bearing packet type that is mandatory for all Bluetooth devices, and the maximum payload length of a DM1 packet is 17 bytes.

The entire bitstream is descrambled using the clock value of the receive slot. Albeit counterintuitive, the processing order (FEC, descrambling, then parsing the fields) must be strictly followed. Specifically, the header portion has to be descrambled but the header's raw bits must first be reduced from 54 bits to 18 bits. Also, the payload is a continued descrambling of these 18 bits, even though the header and the payload are encoded very differently.

After descrambling, BBC checks the HEC by comparing the calculated and received HEC. If the header check passes, BBC further parses the information bits (e.g., the acknowledgement bit) and the packet type. If the incoming packet is DM1, BBC decodes the length of the payload (via the first byte of the payload), which is necessary for further calculating the CRC of the packet. If the CRC passes, BBC checks the LLID. If the LLID is LMP, the payload is processed by the BBC's LMP logic (Sec. 6.2.10). If the LLID is 0b10, BBC starts collecting a new upper-layer (i.e., L2CAP) packet. A logical upper-layer packet can be larger than the maximum size of DM1, and thus a DM1 packet may be followed by many continuation packets (LLID=0b01). To ensure correct defragmentation, we first calculate the total length of the upper-layer packet when a DM1 with LLID=0b10 is received. Since the length of the upper-layer packet is encoded in the first two bytes of message, BBC continuously concatenates subsequent bytes within every continuation packet until the length is reached.

6.2.9 Tx FIFO, SEQN and ARQN

So far, BBC enables Bluetooth Classic transmission (Sec. 6.2.7) and reception (Sec. 6.2.8). However, the packet delivery is not reliable since packets may be received with error or lost entirely. In either case, a packet retransmission is required for reliable delivery. In Bluetooth Classic, this is possible with the SEQN (sequence number) and ARQN (acknowledgement) bits in the header of each packet.

Since the upper layer may send data packets at arbitrary times, we build a circular FIFO to queue the outgoing messages. Furthermore, we also queue outgoing LMP messages generated by the LMP logic (Sec. 6.2.10). The read and write pointers of this FIFO are reset when a paging is completed. Each entry in the FIFO contains a pointer to the payload and a flag. The flag indicates whether the payload is a LMP message, whether the payload should be transmitted in a DH3 packet, and whether the payload is a start of a logical L2CAP packet. When the upper layer sends a large packet to BBC, BBC first fragments the packet into a start fragment and many continuation fragments so that each fragment is no larger than the maximum payload size of the DM1 or DH3. All fragments are queued in the FIFO.

At every available transmit slot, `BBC` checks the FIFO. If the FIFO is not empty, `BBC` generates the raw bits (Sec. 6.2.7) and transmits them (with either DM1 or DH3 packets). Otherwise, `BBC` generates and sends a POLL packet.

6.2.9.1 Processing SEQN and ARQN

After sending DM1 or DH3, Peripheral should always respond with a packet that contains a header (e.g., DM1 or NULL). This header contains the ARQN bit, indicating whether the last data reception is successful. If ARQN is set, `BBC` advances the read pointer of the FIFO. Otherwise, the read pointer remains and the next transmit slot retransmits the same payload.

The SEQN bit in the header is used to detect duplicate transmission. The SEQN is toggled for every new message. When `BBC` receives a data packet (typically in response to a POLL), `BBC` checks the SEQN. If the SEQN is the same as the last data reception, `BBC` ignores the current reception and does not send the payload to the upper layer or the LMP logic.

6.2.9.2 Setting SEQN and ARQN

SEQN and ARQN are also contained in packets sent by `BBC` and they have to be set correctly. If `BBC` receives a data packet with a correct CRC, ARQN of the next `BBC` transmission is set to 1. For SEQN, we make a key observation. Since SEQN is toggled for every new message, the correct SEQN always forms a deterministic pattern within the FIFO. In fact, the correct SEQN is the inversion of the least-significant bit of the read index (read pointer), as the initial SEQN should be 1 after paging and the size of the circular FIFO is a multiple of 2.

6.2.10 Link Manager Protocol (LMP)

Upon a successful paging, the LMP must take over to negotiate the various properties of the physical link. In addition to paging, two Bluetooth devices must also establish a connection at the LMP layer before higher layers exchange any messages. LMP is also used for authenticating the connection and for turning on the link encryption.

To start the LMP message exchange, `BBC` queues a `LMP_VERSION_REQ` after a successful paging. This triggers several message exchanges about LMP features and eventually `BBC` sends `LMP_HOST_CONNECTION_REQ` and subsequently `LMP_SETUP_COMPLETE`. At this point a very basic LMP connection is established but no link properties have been set. When `BBC` receives a `LMP_SETUP_COMPLETE` from headphones, it informs the upper layer of a successful LMP connection via an HCI message (`HCI_Connection_Complete`).

Note that these LMP exchanges are slightly different from the reference message chart in the Bluetooth standard [142]. In the reference chart, after LMP_HOST_CONNECTION_REQ, the Central and Peripheral will negotiate various link capabilities, authenticate and encrypt the link before finally exchanging LMP_SETUP_COMPLETE. However, we observe that commercial Bluetooth headphones tend to first establish a very simple connection before further negotiating Bluetooth capabilities, authenticating and encrypting the link. (Headphones spontaneously send LMP_SETUP_COMPLETE immediately after accepting LMP_HOST_CONNECTION_REQ.) For compatibility, we design LMP as described above because these intermediate steps are not necessary for a simple LMP connection.

After LMP_SETUP_COMPLETE, Bluetooth headphones typically start negotiating link capabilities in detail by sending a lot of LMP requests. We implement various LMP exchanges according to the Bluetooth standard so that each request is responded correctly.

6.2.11 Authentication

After the upper layer (e.g., BlueZ on Linux systems) receives HCI_Connection_Complete, it also sends various HCI requests to the driver. After several miscellaneous requests (e.g., HCI_Remote_Name_Request), the upper layer starts authentication by issuing HCI_Authentication_Requested.

While Bluetooth authentication and encryption are technically optional, we find that Bluetooth headphones will refuse to open the audio port if the link is not authenticated and encrypted. Since the design goal of BBC is to both establish a connection and stream audio data, we implement both the Bluetooth authentication and encryption.

At the core of Bluetooth authentication is a hash function. This hash function consists of two block ciphers: a SAFER+ cipher and another SAFER+ cipher with a slight modification to the third round. The input of the first cipher is a 128-bit key and a 128-bit random number. Within the SAFER+ cipher, the key is first expanded into 17 "sub keys" by applying bit rotation, byte permutation and byte addition. The SAFER+ cipher has 8 rounds and each round uses two sub keys. The input to each round is first added (or XOR'ed) with the first sub key, applied nonlinear mapping, and then added (or XOR'ed) with the second sub key. The bytes are further processed through 4 sets of Pseudo Hadamard Transform with 3 permutations in between. After 8 rounds, the 17-th sub-key is applied to generate the output. The output of the first cipher is then XOR with its input, and combined with the third input to the hash function. The second SAFER+ cipher takes this result as the input and uses an offseted key. The output of the hash function is provided by the second SAFER+ cipher. We implement the entire hash function and the two SAFER+ ciphers from the ground up.

When BBC receives HCI_Authentication_Requested, it first sends HCI_Link_Key_Request to retrieve the link key, which is used as the key input to the hash function. BBC also sends LMP_AU_RAND to Peripheral. The LMP_AU_RAND packet contains a 128-bit random number (AURAND), which is another input to the hash function. Finally, the third input of the hash function is Peripheral's address. Peripheral responds to LMP_AU_RAND with LMP_SRES, which contains the first 4 bytes of the hash function's output. If LMP_SRES is received, BBC notifies the upper layer with HCI_Authentication_Complete. The remaining 12 bytes of the hash are known as ACO, which is essential for calculating the actual encryption key after authentication.

6.2.12 Encryption

After authentication, the upper layer starts encryption by issuing HCI_Set_Connection_Encryption. When this command is received, BBC sequentially sends 3 LMP packets (encryption mode, encryption key size, start encryption) to Peripheral. The second LMP packet specifies the size of the encryption key and BBC specifies a size of 16 bytes (128 bits), the highest value possible.

6.2.12.1 Generating the Key

In Bluetooth, the actual encryption key differs from the link key. The actual key is generated, using the hash function, from the link key, a random number EN_RAND, and ACO (calculated during authentication). The 16-byte random number EN_RAND is specified by Central in the third LMP message (LMP_START_ENCRYPTION_REQ). The entire 128-bit output of the hash function is used as the encryption key. We implement the key generation by reusing the hash function we built for authentication.

6.2.12.2 Bit Processing of Encryption

The Bluetooth SIG built its own encryption algorithm known as E0. Implementing E0 is highly error-prone. Although an open-source implementation of E0 [145] can be found online and although it actually passes all test vectors included in the Bluetooth standard, we find that the implementation is actually incorrect and produces wrong encrypting bitstreams for most Bluetooth addresses. (The test vectors are special cases where part of the address is always zero.) Additionally, this code also incorrectly implements the "parallel load" phase specified by the Bluetooth standard. Since the open-source implementation is unreliable, we decide to implement the E0 algorithm entirely on our own.

E0 is a symmetric cipher where both Central and Peripheral generate the same pseudorandom sequence. The ciphertext is the XOR of the plain text and the pseudorandom sequence (and the plain text is recovered using the same XOR operation at the receiver).

For generating the pseudorandom sequence, E0 uses 4 linear feedback shift registers with different sizes (25, 31, 33, 39). The shift registers are initialized with Central's address, current Bluetooth clock, and the 128-bit encryption key. The outputs of the shift registers are fed into an auxiliary logic including a summation and a "blend" logic. The output of E0 is generated by combining (with XOR) the outputs of all shift registers and the auxiliary logic.

The shift registers and the auxiliary logic are first run for 240 cycles. During the last 128 cycles, the E0 output is temporarily stored as a variable Z. At the 240-th cycle, the content of the 4 shift registers is instantaneously updated ("parallel load") with Z. At that instant, E0 also starts outputting the pseudorandom sequence that is actually used for encrypting the payload. However, the auxiliary logic has internal states, which have to be carefully maintained in order to generate the E0 output correctly. Specifically, we must first pause the update of the auxiliary logic, load 4 registers with Z, and generate the first pseudorandom bit before updating the auxiliary logic again. This order has to be strictly followed to ensure the first bit is correct. E0 then keeps on generating the pseudorandom sequence until the end of a packet.

In Bluetooth, only the payload portion of a packet is encrypted. From the bit-processing's perspective, E0 is applied after the payload is generated but before scrambling. We add the function call to E0 at the corresponding locations in `BBC`. A peculiar design of Bluetooth, however, is that the CRC is also encrypted.

We leverage this peculiar design to detect whether the payload is encrypted or not. (Note that the Bluetooth header does not have any field to indicate encryption.) Specifically, after `BBC` sends the third message (`LMP_START_ENCRYPTION_REQ`), Peripheral should turn on encryption and reply with an encrypted `LMP_ACCEPTED` packet. When this packet is received, `BBC` first tries, without decryption, to decode the packet and check CRC. If the packet is encrypted, the CRC check fails, and `BBC` then tries to decode the same packet with decryption enabled. If the CRC check (after decryption) passes, `BBC` sets an internal flag (`encryptionenabled`) to 1. If this flag is set, `BBC` enables encryption for any outgoing payload. After the encrypted `LMP_ACCEPTED` packet is received, `BBC` informs the upper layer that encryption is turned on.

This two-step CRC check ensures that `BBC` is resilient when the packet exchange deviates from the ideal case. For example, Peripheral may be temporarily unable to turn on encryption and send an unencrypted `LMP_NOT_ACCEPTED` packet. Or, Peripheral sends other unencrypted packets before `LMP_START_ENCRYPTION_REQ` is processed. Our design ensures that these packets can still be received correctly regardless of Peripheral's encryption state.

We perform an optimization to minimize the E0 computation during packet transmission. (After various negotiations, the majority of traffic is `BBC`-to-headphones and thus transmission optimization is the most effective.) Note that the pseudorandom sequence can be pre-generated because the address and the encryption key are known and the Bluetooth clock of the next transmit slot can be estimated. Therefore, after sending a packet to the BLE chip, BBC estimates the next transmit clock and calculates the pseudorandom sequence. After radio transmission, the BLE chip sends the clock value of the next transmission slot to the driver. If the actual clock matches the estimation, BBC encrypts the payload by simply performing the XOR on the bitstream.

6.2.13 L2CAP

After encryption, the L2CAP (Logical Link Control and Adaptation Layer Protocol) traffic can be transferred using BBC. From this point on, BBC operates just like a conventional Bluetooth-Classic chip that the upper layer (BlueZ) can send and receive arbitrary L2CAP data.

After notifying that encryption is successful, the upper layer automatically initiates various protocols over the L2CAP layer. These includes SDP (Service Discovery Protocol [142]), AVCTP (Audio/Video Control Transport Protocol [146]) and AVDTP (Audio/Video Distribution Transport Protocol [147]). These protocols must first be connected and negotiated before audio streaming. However, they are already implemented in BlueZ, and hence BBC directly leverages it. After negotiation of these protocols, BlueZ continuously sends audio packets over the L2CAP layer BBC provides.

In Bluetooth, high-quality audio packets are large (~850 bytes) and the HCI layer (which connects BlueZ and BBC) typically fragments an audio packet into multiple HCI packets. Therefore, BBC first defragments HCI packets into an audio packet before fragmenting it into multiple DH3 payloads and queuing them in the FIFO.

6.3 Implementation and Evaluation

We implement BBC on readily-available BLE chips. For evaluation, we conduct microbenchmarks using the industry-standard Bluetooth measurement tool. We then conduct end-to-end evaluation of BBC by directly connecting and streaming music to unmodified, commodity Bluetooth headphones.

We implement BBC on CC2540 BLE chip from Texas Instruments (TI) and use the CC2540EMK-USB development board [148] as the hardware. We use SDCC (Small Device C Compiler) [149] to develop the firmware and CC-DEBUGGER [85] from TI is used to flash the firmware. The raw CMD_BLE_TX and CMD_BLE_RX in TI's SDK are used to transmit and receive raw FSK packets. We also set the frequency deviation to 160kHz using the MDMCTRL0 register. Since modulation of modern BLE chips is achieved by digital circuits and since radio (e.g., FCC) regulations require measurement of the output power of the unmodulated carrier (i.e., zero frequency deviation), the frequency deviation can be configured. We also build a Bluetooth driver (kernel module) to generate and process the raw packets. Our driver registers a new HCI Bluetooth device when the kernel detects a matched VID and PID. After the Bluetooth device is opened, the kernel sends various HCI messages to the driver. BBC also starts paging Bluetooth headphones after initialization. Once the headphones are successfully paged, the BLE chip constantly updates the driver with current clock value and the raw FSK bits received, and the driver sends the frequency hops, the number of slots and the raw FSK bits to be transmitted to the BLE chip.

Since BBC emulates the operation of Bluetooth Classic in the driver, most of the HCI messages are processed and responded to by the driver. For a small subset of HCI messages (most notably name request, encryption requests and the ACL traffic), the driver constructs LMP or DH3 packets accordingly and puts them in the Tx FIFO. In the driver, the key processing steps (Sec. 2.6–2.9 and Sec. 2.11–2.12) are implemented in the USB receive callback function. For authentication (Sec. 2.10), the hash function is called when the driver receives authentication HCI messages, but the LMP exchanges are implemented in the receive callback function.

6.3.1 Evaluation Setup

In the evaluations, we use Ubuntu 20.04 with BlueZ 5.53 and PulseAudio 14.0. The transmit power of the BLE chip is set to 4dBm, which corresponds to Class 2 devices (with an operational range of up to 10m) in the standard [142]. Therefore, in both PHY and system evaluations, we test BBC at 1, 5 and 10m.

For microbenchmarks evaluation, we use Teledyne LeCroy's FTS4BT [43], the *de facto* industry-standard [150] sniffer. This Bluetooth test equipment allows us to directly capture the over-the-air (OTA) Bluetooth traffic and measure the physical-layer performance, such as packet error rate (PER) and the maximum achievable throughputs.

Table 6.1: Evaluation with unmodified Bluetooth headphones

Headphones	Bluetooth Chip Used
Sennheiser CX150	Qualcomm QCC3024
Sony SBH20	CSR CSR8640
Apple Airpods 2	Apple H1

We also conduct system evaluation where BBC creates a direct connection with off-the-shelf Bluetooth headphones (Table 6.1), negotiates the capabilities and streams audio, just like a conventional Bluetooth chip. The system evaluation involves all aspects of Bluetooth, including PHY, packet-retransmission (ARQN/SEQN), and all other components (LMP, authentication/encryption, BlueZ and PulseAudio). On Ubuntu 20.04, the audio codec is implemented in PulseAudio and SBC is used in the system evaluation. (SBC is mandatory in Bluetooth and is the only codec supported by all headphones in Table 6.1. Note that Ubuntu does not ship with AAC due to licensing issues and

Table 6.2: Packet Error Rate (PER) and Throughputs at the Physical Layer

Packet Type	POLL			DM1			DH3		
Distance	1m	5m	10m	1m	5m	10m	1m	5m	10m
Correctly Received Packets	4094	4087	4077	4094	4075	4074	3965	3954	3897
Packet Error Rate (%)	0.05	0.22	0.46	0.05	0.51	0.54	3.20	3.47	4.86
Throughputs (kbps)	No payload			54.37	54.12	54.11	566.87	565.30	557.15

AAC actually requires less throughputs (~265kbps in Apple's guidelines [151]) than SBC at high quality.) Our system evaluation uses the default SBC setting (Bitpool: 53, Block:16, Allocation: Loudness, Subbands:8). This is the "high quality" setting defined in the A2DP standard [137], and is commonly the highest SBC setting of Bluetooth headphones. The setting corresponds to a bitrate of 328kbps.

6.3.2 Microbenchmark

Because the frequency hopping and bit scrambling depend on the Bluetooth clock, Teledyne LeCroy's FTS4BT tool has to be synchronized with BBC first before capturing the packets. The FTS4BT suite provides three possible clock synchronization methods and we use the "Central Inquiry" mode. In this mode, FTS4BT sends an ID packet and BBC should reply with an FHS packet. Since the FHS packet contains BBC's current clock value, FTS4BT can be synchronized with BBC's clock. We add this synchronization message exchange in firmware to satisfy the synchronization requirement of FTS4BT. This exchange is very similar to the second to third packets of the paging process (Sec. 2.4). The key difference is that instead of using Peripheral's access code, the GIAC (General Inquiry Access Code) and the corresponding hopping sequence should be used. Additionally, HEC and CRC calculations have to be initialized with zeros.

After the FHS is sent, interrupts are enabled and the packet transmission starts. We evaluate the physical-layer performance at 1, 5, and 10m, and we test POLL, DM1 and DH3 packets at each distance. For each test case, we send 4096 packets. We send the maximum amount of payload for each payload-bearing packet (17 bytes for DM1 and 183 bytes for DH3). From the FTS4BT capture, we count the number of correctly received packets (with correct HEC and correct CRC), and calculate the PER and effective throughputs.

Table 6.2 shows the results. For POLL and DM1 packets, the PER is very low. At 1m, only 2 of the 4096 packets are received erroneously, corresponding to a PER of 0.05%. The PER steadily increases with distance, but still remains very low at 10m at around 0.5%. Compared to POLL, DM1 has a slightly higher PER because DM1 packets are longer. The PER is noticeably higher for DH3 packets because DH3 packets have 183 bytes of payload (plus payload header and CRC) and are much longer than DM1 and POLL packets. However, BBC still has good performance with a PER of 3–5%.

We also calculate the throughputs achieved. Table 6.2 shows that DH3's throughputs are much higher than DM1's because of its much larger packets and lower overheads. Thus, even though DH3 is optional, supporting DH3 packets is important for audio streaming because DH3 is about 10x faster than DM1. Note that we calculate the throughputs by the number of bytes (DM1: 17, DH3: 183) that can be actually used by the L2CAP layer, and we do not count the payload header and CRC bytes as usable payloads. DH3 achieves a throughput of ~560kbps, which provides ample headrooms for the bitrate required by audio streaming (Sec. 3.1).

6.3.3 System Evaluation

Table 6.3: System Performance with Sennheiser CX150

	1m		5m		10m	
	Ack'd/Sent	PER (%)	Ack'd/Sent	PER (%)	Ack'd/Sent	PER (%)
POLL	8714/8825	1.26	8472/8617	1.68	8276/8549	3.19
DM1	18/18	0.00	18/18	0.00	19/19	0.00
DH3	14709/15157	2.96	14723/15365	4.18	14588/15432	5.47
Total	23441/24000	2.33	23213/24000	3.28	22883/24000	4.65
Total Payload	2490987 Bytes		2493419 Bytes		2470472 Bytes	
Throughput (Avg./Peak)	332.13 kbps/338.90 kbps		332.46kbps/338.54 kbps		329.40 kbps/338.19 kbps	

Table 6.4: System Performance with Sony SBH20

	1m		5m		10m	
	Ack'd/Sent	PER (%)	Ack'd/Sent	PER (%)	Ack'd/Sent	PER (%)
POLL	8797/8873	0.86	8564/8654	1.04	8563/8701	1.59
DM1	17/17	0.00	17/18	5.56	17/17	0.00
DH3	14701/15110	2.71	14701/15328	4.09	14701/15282	3.80
Total	23515/24000	2.02	23282/24000	2.99	23281/24000	3.00
Total Payload	2491546 Bytes		2491546 Bytes		2491546 Bytes	
Throughput (Avg./Peak)	332.21 kbps/336.13 kbps		332.21 kbps/338.54 kbps		332.21 kbps/336.22 kbps	

In the system evaluation, BBC directly connects to Bluetooth headphones and streams audio. We measure the performance for 1 min after BBC successfully pages the headphones. Since Bluetooth

Table 6.5: System Performance with Apple Airpods

	1m		5m		10m	
	Ack'd/Sent	PER (%)	Ack'd/Sent	PER (%)	Ack'd/Sent	PER (%)
POLL	5103/6204	17.75	4813/6019	20.04	4774/5987	20.26
DM1	21/27	22.22	21/28	25.00	21/26	19.23
DH3	13739/17769	22.68	13741/17953	23.46	13740/17987	23.61
Total	18863/24000	21.40	18575/24000	22.60	18535/24000	22.77
Total Payload	2330133 Bytes		2330499 Bytes		2330316 Bytes	
Throughput (Avg./Peak)	310.68 kbps/340.00 kbps		310.73 kbps/338.54 kbps		310.71 kbps/344.39 kbps	

operates strictly based on time slots, 60s corresponds to 96000 slots. As described in Sec. 2.5, transmission is scheduled at a 4-slot interval (which maximizes the throughput of DH3). Therefore, BBC exactly transmits 24000 packets within the 1-min duration. At each transmission, BBC either sends POLL or DM1 (or DH3), depending on the status of the Tx FIFO. We gather the statistics of the number of packets sent and acknowledged. A DM1 or DH3 packet is acknowledged if BBC receives a packet with correct HEC and ARQN. A POLL packet is acknowledged if a reply with a correct HEC is received. (POLL has no effect on the ARQN bit.) We also calculate the number of payload bytes of all acknowledged packets. We then convert this number to average (60s) and peak (1s) throughputs.

Table 6.3 shows the system performance with Sennheiser CX150 headphones. Similarly to the microbenchmarks, the PER increases with distance, but the overall PER is still relatively low (~5%) at 10m. From the individual breakdown, we can see that DH3 has a higher PER because of the longer length. Note that there are very few DM1 packets because we only use DM1 for LMP packets and use DH3 for all other L2CAP traffic. (The Bluetooth standard specifies that DM1 should always be used for LMP.) Finally, because the maximum throughput at the physical layer (~560 kbps) is significantly higher than the application bitrate (~328 kbps), the total payload delivered and throughputs are very similar across distances (and are governed by the application bitrate). By using ARQN/SEQN, audio packets are reliably transmitted to the headphones and the audio does not have glitches or interruption. Similarly, Table 6.4 shows the system performance with Sony SBH20 headphones. The overall PER is only 2–3%. Because Bluetooth uses time slots very strictly and the traffic is entirely scheduled by Central, the traffic pattern is highly predictable at very low PER. Note that the number of acknowledged DM1 and DH3 packets is exactly the same, and thus the total payload and average throughputs are the same. However, the peak throughputs have very small variations due to occasional retransmissions.

6.3.3.1 Deep Dive into Airpods' Bluetooth Connection

Table 6.5 shows the performance with Apple Airpods, and the PER is significantly higher than the other headphones. We take a close look at Airpods' Bluetooth connection. We find that the higher PER comes from the design and implementation on the Airpods' side, and **the result is similar to the PER of using an off-the-shelf Bluetooth chip with Airpods**.

Specifically, as a reference of using a standard Bluetooth chip, we use the Qualcomm WCN6856 card (Qualcomm Atheros FastConnect 6900 [152]) on a Windows 10 laptop to connect to Apple Airpods, and capture the OTA traffic using Teledyne LeCroy's FTS4BT. In the packet trace, we counted the number of packets transmitted by WCN6856 and the number of packets not acknowledged by Airpods. (We manually counted the packets within the duration of 300 packets because this specific condition cannot be easily configured in the FTS4BT suite.) Among the 165 packets transmitted by WCN6856, 30 packets are not acknowledged, which corresponds to a PER of $30/165 = 18.18\%$. BBC's PER (see Table 6.5) is about 3–5% higher than this baseline, which is similar to the results of CX150.

There are several factors that might cause Airpods to only respond to around 80% of the packets. Airpods are known as true wireless headphones where left and right earbuds both have a Bluetooth chip. Since the original Bluetooth standard [144] was designed to communicate with only one Bluetooth chip, the key design principle of true wireless headphones is to make a pair of Bluetooth chips appear as one *logical* chip. The packet capture between WCN6856 and Airpods shows that Airpods use the standard Bluetooth protocol. However, because Airpods actually have two physical Bluetooth chips, Airpods might be temporarily unable to respond to some incoming packets due to the need for coordination between the chips on two earbuds.

Note, however, because the PHY provides significantly higher throughputs, BBC actually streams audio to Airpods without interruptions or glitches. Another difference of Airpods from others is that Airpods start audio streaming after the "connected" sound effect (about 5s after paging is successful). This contributes to the lower number of total payload bytes and average throughputs in Table 6.5. In the steady state (11~60s), the average throughputs are 334.62, 334.68, and 334.67 kbps, which are consistent with prior results.

Another important observation from the packet capture of the Qualcomm WCN6856 with Apple Airpods is that the audio configuration and the authentication/encryption algorithms are exactly the same as BBC. Without any modification, Qualcomm WCN6856 and Apple Airpods automatically use the high-quality A2DP SBC setting (the same setting as Sec. 3.1) and the default authentication (Sec. 2.10) and encryption (Sec. 2.11). Note that WCN6856 is one of Qualcomm's latest flagship solutions for Bluetooth connectivity on laptops and smartphones, and BBC provides identical audio quality and security for typical Bluetooth headphones like Airpods.

6.3.4 Computation Time

We also evaluate the computation time of BBC using an HP 800 desktop with an i5-4570S processor. BBC only requires simple logic and integer operations, and is efficient on modern hardware. Table 6.6 shows the time required for generating packets and for precomputing encryption sequences. Packets are sent every 2500μs (4 slots) and the computation (for DH3 packets with encryption) only takes ~1.6% of the time.

Table 6.6: Computation Time of BBC

	POLL	DM1	DH3	Encryption
Mean (μs)	1.19	2.32	10.03	29.67
Std. Deviation (μs)	0.65	1.03	2.70	7.62

6.4 Alternative Design Choices

We have explored alternative design options in BBC. However, we find that they either make the system incompatible with Bluetooth headphones, or cannot establish a connection at all. In particular, we have tried using the open-source E0 cipher. However, this breaks the compatibility with commodity Bluetooth headphones and cannot stream music at all because the E0 cipher does not produce the correct bit sequence for encryption. Also, in the LMP implementation, we tried streaming music without setting up link authentication and encryption first (since they are technically optional according to the standard). However, we find that Bluetooth headphones will not accept audio packets without authentication and encryption. Finally, for the slotted communication, we have tried longer transmission intervals (instead of 2500μs, which contains 3 Tx slots and 1 Rx slot). However, we find that such a design cannot sustain the required throughput for music streaming, which results in audio glitches.

6.5 Related Work

Few academic or open-source research deals with the internals (between FSK and HCI layers) of Bluetooth, and BBC is the only fully operational Bluetooth system (from connection establishment to audio streaming) to the best of our knowledge. Some prior work, with the bottom-up approach, focuses on sniffing Bluetooth traffic. Ubertooth [62] can follow and passively sniff an active Bluetooth connection, and is used in numerous projects [153, 154, 155]. However, it operates on the raw FSK level (without Bluetooth bit processing or packet processing) and lacks the ability to create (i.e., paging), maintain (ARQN and LMP), or encrypt a connection. BlueEar [156] adds

Bluetooth clock acquisition and adaptive hop selection to Ubertooth for sniffing. Sniffing raw FSK bits using USRP [156, 157] is also proposed. We find FTS4BT, the commercial sniffer used internally by CSR and Qualcomm [158], has the same (and significantly more) sniffing features and is the most reliable.

Some other work, using the top-down approach, focuses on leveraging hidden features on conventional Bluetooth chips for low-level Bluetooth experiments. InternalBlue [159, 160, 161] reverse-engineers Broadcom's Bluetooth chips and directly sends LMP messages below the HCI layer. BrakTooth Sniffer [162] reverse-engineers the ESP32's Bluetooth implementation, and allows sniffing or injecting of Bluetooth packets. Their goal is modifying the normal packet exchange on Bluetooth-Classic chips, whereas BBC aims to emulate full Bluetooth Classic with BLE chips. They focus on packet-level (POLL, LMP, etc.) interactions in the connected state, whereas BBC handles bit-level processing, paging, frequency hopping and reliable delivery.

Some researchers investigate the security of Bluetooth. SoK [163] provides an overview of the security flaws in Bluetooth. An open-source (but incorrect) E0 implementation [145] has been attempted. KNOB [164, 165] and BIAS [166, 167] focus on the authentication and key entropy in Bluetooth. Blacktooth [168] implements more attacks using InternalBlue and BIAS. BLUR [169, 170] uncovers the flaws of key derivation across Bluetooth and BLE. BrakTooth [171] discovers abnormal Bluetooth packets that trigger crashes and deadlocks.

BBC is also related to CTC research. BlueFi [72] and WiBeacon [73] enable communication from WiFi to Bluetooth. They are limited to either broadcasts or one-way communication for audio streaming (i.e., without connection establishment). FLEW [78] and Unify [103] convert Bluetooth chips into WiFi chips. Bluetooth-Zigbee communication [21, 22] is also possible, but the research focuses on PHY and not on the full protocol emulation.

6.6 Conclusion

BBC is a complete system that enables BLE chips to directly support Bluetooth Classic. By eliminating the need for Bluetooth-Classic hardware blocks, BBC enables using simpler hardware (BLE chips) in the future while maintaining compatibility in the driver. BBC also enables existing BLE chips to directly communicate with Bluetooth headphones, opening the possibility of new audio applications on current BLE devices. Our extensive evaluations demonstrate BBC's low PER and high throughputs, and its ability to stream audio to conventional headphones with the same audio quality as COTS Bluetooth-Classic chips.

CHAPTER 7

Conclusion

This thesis makes direct communication between WiFi and Bluetooth a reality. Such communications are motivated by the popularity and importance of WiFi and Bluetooth, and the immense opportunities to enable new connectivity to tens of billions of devices around the world. We design five innovative systems, BlueFi, FLEW, Unify, DREW, and BBC. Each system proposes innovative ideas to enable cross-technology communication (CTC) under different settings. Each system serves as a milestone for Bluetooth–WiFi communication and advances the state of the art of wireless communication. We derive theoretical groundings and reasonings that underpin our system designs. We identify unique challenges and overcome them with innovative and effective solutions. We provide several insights and develop important building blocks and techniques in our solutions. Our designs are practical and applicable because off-the-shelf devices and chips are directly used. In order to provide this practicality, we address a number of significant implementational obstacles. We also address the computational challenges so that our systems can be run in real time. Finally, we thoroughly evaluate the systems and show that they achieve high performance in the real world.

7.1 Contributions

In Chapter 2, we show that direct WiFi–Bluetooth communication is possible by leveraging the idea of crafting waveform compatibility and similarity. BlueFi works by finding special 802.11n packets that produce IQ signals of FSK waveforms, which can be demodulated by Bluetooth devices without modifications. We propose the design methodology of tracing back the WiFi modulation process and work from IQ signals to WiFi bits. This design process further involves four major technical challenges, which are CP insertion, QAM modulation, pilots/nulls, and FEC coding. We address each challenge with signal processing theories and simulations. We show the validity of our system by implementing it on widely-adopted WiFi chips. We also evaluate the performance with off-the-shelf Bluetooth chips. BlueFi enables highly practical applications, including broadcasting BLE beacons and streaming Bluetooth audio to headphones. We also address the computational aspects and optimize the signal processing so that BlueFi can be run in real time.

In Chapter 3, we achieve the milestone of bidirectional communication. The key contribution of FLEW is enabling full, bidirectional WiFi communication without modifying WiFi access points. This is made possible with three major components, which are WiFi-to-FSK, FSK-to-WiFi, and MAC-layer designs. For WiFi-to-FSK communication, we make a key signal processing insight where FSK radio, operated with a frequency shift, can be used effectively as a WiFi receiver. We also propose using the FSK pattern matching hardware to efficiently detect WiFi packets. This is possible via the observation that the scrambling seed is always the same for DSSS packets. We address the computational issues so that the demodulation process, particularly the descrambling and CRC calculation, can be run in real time. For FSK-to-WiFi communication, we propose the techniques of injecting the digital baseband waveform and reusing the mixers of FSK chips. For the MAC-layer design, we devise methods for FSK chips to properly support CSMA/CA, the standard medium-access mechanism of WiFi. We propose methods to support the ACK and RTS/CTS timings so that FSK chips can directly coexist with WiFi systems. Finally, we present the finite-state machine that enables FSK chips to properly support WiFi operations. We conduct extensive evaluation of FLEW. In addition to enabling heterogeneous, bidirectional connectivity with unmodified WiFi access points, another significant benefit of FLEW is that as a result of the simple hardware, using FSK chips on the device side allows for very low power consumption. By enabling the direct communication, FLEW combines the advantages of both WiFi and Bluetooth.

In Chapter 4, we demonstrate bidirectional communication can be achieved all in one chip and without modifying the hardware. Unify focuses on newer BLE chips, which come in the form of SoCs. To meet the design goals, we propose three major techniques. The first technique is using the DMA, timers and the DAC registers to effectively transmit WiFi waveforms. Our design leverages the fast interconnection within a single chip so that WiFi transmission can be realized only by firmware modification (as opposed to hardware modification). The second technique builds on the findings of FLEW where WiFi packets can be detected by Bluetooth chips' pattern matching and decoded by the FSK demodulator. To solve the unique challenges of SoCs, however, the second technique is leveraging the SPI hardware in conjunction with the FSK observation signals. We show that, by purposely making the receiver stuck in the receive mode, this technique allows BLE chips to receive long WiFi packets properly. Finally, Unify introduces the technique of using the power override signals to significantly shorten the radio turnaround time on BLE chips, which is crucial for BLE and FSK SoCs to directly support standard WiFi operations. We also show that by carefully designing the packet decoding and by reusing the CRC hardware on BLE chips, the entire reception process can be run in real time using the built-in 8-bit microcontroller on BLE SoCs. We implement our system design on widely popular and adopted BLE chips and evaluate the performance thoroughly. We show that these ubiquitous BLE chips can be directly connected to

WiFi access points equipped with WiFi chips from all major vendors. Because SoCs are single-chip solutions with high integration and a small footprint, Unify is shown to be considerably smaller, cheaper and to consume significantly less power than prior work. On the broader scope, Unify shows that WiFi connectivity can be achieved under the same power budget, device footprint and cost as BLE.

In Chapter 5, we achieve the milestone of doubling the downlink throughputs of bidirectional communication and making it applicable to ultra-low-power Bluetooth chips. Motivated by multimedia applications, including lossless audio streaming with a bitrate of 1.411Mbps, DREW supports QPSK demodulation of standard WiFi packets. DREW also enables Bluetooth–WiFi communication on ultra-low-power BLE chips where the direct modulation architecture is used and transmission mixers are not available. To achieve QPSK demodulation, we propose using the IQ sampling capability on new BLE chips to collect narrow-band samples of WiFi waveforms. One key contribution of DREW is showing the IQ sampling feature, which was only used for localization and angle-of-arrival estimations, can be used for data communication very effectively. Demodulating bits from a large number of IQ samples in real time, however, requires fast processing. To address this computational problem, we propose leveraging SIMD parallel processing to simultaneously handle four bitstreams. We also show that this SIMD method, which provides 4× performance boost, can be implemented with the conventional 32-bit addition and subtraction instructions. By devising two ways of setting the decision boundaries and extracting bits, DREW is capable of demodulating both BPSK and QPSK packets, thus doubling the throughputs. DREW also proposes solutions for efficiently synchronizing and detecting (long and short) preambles from narrow-band IQ samples. To achieve BLE-to-WiFi communication without using mixers, we make key observations of the PSK and DSSS modulation process and engineer the similarity of waveforms transmitted by BLE chips and expected by WiFi receivers. We implement packet transmission by carefully designing assembly codes with precise timings. We further boost throughputs by proposing the zero-wait packet transmission. We also design a MAC layer for coordinating the new transmission and reception designs. We comprehensively evaluate DREW with unmodified WiFi NICs and APs and the highlight of the evaluation is the near-zero PER when receiving BPSK and QPSK packets. We also present practical end-to-end applications that leverage the doubled throughputs and the direct WiFi connectivity using ULP BLE chips. We demonstrate that DREW enables streaming stereo high-fidelity audio directly from WiFi devices and without any audio compression.

In Chapter 6, we extend the idea of engineering waveform compatibility and similarity for heterogeneous Bluetooth devices, and we enable communication between BLE chips and Bluetooth headphones. BBC is motivated by the disparity where more and more Bluetooth devices use BLE but the majority of Bluetooth headphones still rely on Bluetooth Classic. BBC is based on the insight that it is possible to emulate most of the Bluetooth Classic standard in drivers. By proposing and implementing the lean architecture, BBC eliminates the need for the hardware blocks on Bluetooth Classic chips and thus enables BLE chips to support bidirectional communication with Bluetooth Classic devices. The key contribution of BBC is building a fully operational Bluetooth Classic system using BLE-only chips. This design goal requires precise and correct emulations at various layers. We demonstrate using BLE chips to emulate the Bluetooth clock and slotted communication. Furthermore, we show that BLE chips can emulate the 6-packet paging process, which is a critical step for establishing a Bluetooth Classic connection. For transmitting Bluetooth Classic packets, BBC emulates the packet generation in software with correct access codes, headers, payload, CRC, bit scrambling and FEC. For receiving Bluetooth Classic packets, BBC uses the pattern matching hardware on BLE chips and applies processing procedures opposite to those on the transmitter. Furthermore, BBC achieves the reliable packet delivery by emulating the Bluetooth Classic's acknowledgement mechanism in its driver. At the higher layers, BBC emulates the Link Manager as well as the link authentication and encryption functions. All these components are used to ultimately emulate the L2CAP layer and the HCI layer. By carefully designing the emulation in the driver and firmware of BBC, we make BLE chips appear as an off-the-shelf Bluetooth Classic chip, which can be used to establish a connection and stream audio to unmodified Bluetooth headphones. We also measure the performance with Bluetooth test equipment and commodity Bluetooth headphones. BBC achieves low PER and high throughputs for Bluetooth Classic connections. Furthermore, the evaluation shows that BBC matches the throughputs and configuration of off-the-shelf Bluetooth Classic chips. Therefore, BBC provides the same audio quality as standard Bluetooth Classic chips, and is an ideal solution for future or current BLE-only chips to provide compatibility with Bluetooth headphones.

7.2 Notable Experiences and Other Discoveries

In addition to the challenges and contributions that are directly related to the communication between WiFi and Bluetooth, we also gain other insights and lessons that might be valuable to future research of wireless communication.

First, we observe that many WiFi or Bluetooth chips have more capabilities than those described in public datasheets. For example, in BlueFi, we find undocumented registers that allow us to turn off random scrambler generation. In Unify, we find undocumented registers controlling the DAC, the power signals, and those making the receiver stuck in reception. Such information is very helpful for wireless communication research and should ideally be accessible to developers. However, we find that this is usually not the case. Even so, we solve these challenges and make them tangible by extensive hardware experiments and careful analysis.

Second, we find that some common hardware and peripherals on embedded systems are surprisingly useful for WiFi or Bluetooth communication. In Unify, we use the DMA and timer peripherals to accurately transmit WiFi waveforms and the SPI peripheral to receive WiFi bits. In DREW, we extensively leverage the ARM microprocessor to process the IQ samples in parallel, and to precisely control the power amplifier for packet transmission. In BBC, we use the hardware timer for emulating time-slot communication.

Third, even though WiFi and Bluetooth are standardized, their specifications do not guarantee identical behaviors across vendors. We observe significant implementation and performance variations between devices. From the evaluations of FLEW, Unify and DREW, we observe differences in terms of PER and throughput measurements. From the signal and packet capture, we also see that WiFi chipsets use different rate-adaptation algorithms and use different strategies to select the data rate. In the extreme case, we also observe that Realtek chips use non-standard scrambler seeds. Similarly, in BBC, we observe different PER and throughput results from different Bluetooth chips.

Another insight we gained is that the interference between Bluetooth and WiFi has become less of a concern if robust WiFi modulation is used and Bluetooth's adaptive frequency hopping is enabled. In the evaluations of FLEW, Unify and DREW, interference and throughput reductions are minimal with concurrent WiFi and Bluetooth traffic because Bluetooth devices actively steer clear of busy WiFi channels.

Finally, our evaluations show that backward compatibility is still important for WiFi and Bluetooth systems. For example, many WiFi devices still use the original 802.11 waveform for beacon and management frames. We also see that modern Bluetooth devices still rely on Bluetooth Classic, along with its original authentication and encryption algorithms, for streaming music. Even though newer WiFi and Bluetooth specifications have been developed, certain components from the old standards have to be preserved to ensure backward compatibility with existing devices.

7.3 Future Directions

7.3.1 Extending BlueFi

BlueFi realizes Bluetooth's radio and physical layers (radio, "baseband" and link control in Bluetooth's terminology), on which *all* applications and profiles are built, and more. These layers transmit a series of 1's and 0's and are oblivious of the content these bits represent. Therefore, any application or profile can use BlueFi for Bluetooth transmission. As the first step in exploring Bluetooth and WiFi communication, we focused on transmission. Note that transmission alone is still very useful in many cases. For example, signal reception is totally unnecessary for beacons. Also, when using A2DP to stream audio, the uplink traffic is only for sending ACK packets and is not critical to the audio operation. Furthermore, the nature of audio streaming makes ARQ less useful. For example, for very low latency audio, retransmitted packets will miss the deadline and thus become useless. In addition, excessive retransmissions not only increase latency but also decrease usable throughput or goodput. We also note that regulatory certification is not needed for receivers. Therefore, it is possible to use BlueFi in conjunction with a dedicated reception chip to realize full Bluetooth function without the need for regulatory certification. The reception function for BlueFi is part of our future work.

Some Bluetooth chips are capable of supporting optional modulation modes other than GFSK, and thus increase throughput by up to 3×. It is also possible to use 40MHz WiFi channels to support 2× the number of Bluetooth channels and increase throughput. We leave these two directions as future work.

7.3.2 Extending BBC

In BBC, our primary design goal is using BLE to emulate a fully-functional Bluetooth Classic system. The main innovation of BBC, in the CTC context, is making BLE-only chips from not supporting Bluetooth Classic connections at all to supporting proper bidirectional Bluetooth Classic communication. In future, one may consider extensions and optional Bluetooth features as an extension to BBC. BBC currently supports one active Bluetooth connection. Note that Windows (or Linux, etc.) streams audio through only one selected output when multiple Bluetooth devices are connected. Therefore, in practice, supporting one active connection for audio streaming in BBC does not limit the usability. If users wish to switch between audio outputs, BBC can simply page other headphones and start audio streaming after paging. If multiple simultaneous connections are still desired, it should be straightforward to implement the feature because BBC provides the capability of bidirectional Bluetooth–Classic packet communication. More specifically, this feature can be realized by using a different AM_ADDR and by time-multiplexing the communication. Such an endeavor, however, focuses more on the implementation of protocols and less on advancing the state-of-the-art of CTC. Another future direction is merging different firmware to simultaneously support Bluetooth Classic and BLE on the same chip. Currently, supporting both requires switching firmware. Finally, adaptive frequency hopping (AFH) is an optional feature in Bluetooth that allows avoidance of certain bad RF channels dynamically, and can be added to BBC in the future.

7.3.3 Exploring the Security Implication

In the systems presented in this thesis, a communication link can directly leverage the security protocol of WiFi or Bluetooth. For example, WiFi encryption (WPA2-CCMP) is enabled for all evaluations of FLEW, Unify and DREW, and thus they are as secure as conventional WiFi devices.

FLEW, Unify and DREW provide a better security solution for IoT devices because they can use WiFi encryption, which is less vulnerable than Bluetooth. For instance, WiFi's 4-way handshake is provably secure whereas Bluetooth's authentication is susceptible to known attacks.

On the other hand, CTC creates a new mode of communication. The ability to communicate across different wireless protocols has security and privacy implications. Most existing security protocols implicitly assume that all nodes, even malicious ones, run the same wireless technology. Because we show that inter-protocol communication is possible, it implies that inter-protocol attacks may become a new attack surface. For example, WiFi networks may become vulnerable to compromised Bluetooth networks. Such an attack will be especially relevant as the number of IoT devices grows.

7.3.4 Diverse Applications of Ultra-low-power Internet Connectivity

Direct communication between WiFi and Bluetooth bridges two of the most common wireless technologies, and hence it is worth exploring a broad set of new applications based on this novel connection paradigm. In DREW, we use uncompressed HiFi audio streaming as an example. Since personal audio streaming plays an important role for listening to music, for conducting virtual meetings, and even for using hearing aids, it is worth exploring different end-to-end applications and designing next-generation wearable devices that will benefit the public. By directly using WiFi, such systems can overcome the inherent limitation imposed by the Bluetooth standard. Better, wired-equivalent audio quality and lower audio latency are possible at the same time. More broadly, we can also rethink other wearable and sensing applications with this direct WiFi and Internet connectivity. For example, one impactful application may be healthcare monitoring where WiFi infrastructures are used to collect sensor readings and emergency messages. Another direction is exploring direct communication between other heterogeneous wireless standards.

7.3.5 Designing CTC with AI Methods

In recent years, AI and machine learning (ML) have become increasingly powerful. AI methods, either as a processing block or as a design tool, will likely have significantly more presence on wireless systems. One future direction might be using ML for designing CTC. ML approaches can be data-centric, which is significantly different from prior methods where communication theories under well-defined conditions are heavily used. ML-based methods can potentially find solutions for problems that are otherwise intractable.

BIBLIOGRAPHY

[1] NXP. MKW41Z/31Z/21Z Data Sheet. `https://www.nxp.com/docs/en/data-sheet/MKW41Z512.pdf`, Mar 2018.

[2] Texas Instruments. CC2650 SimpleLink™ Multistandard Wireless MCU. `https://www.ti.com/lit/ds/symlink/cc2650.pdf`, Feb 2015.

[3] P. Gawlowicz, A. Zubow, and A. Wolisz. Enabling Cross-technology Communication between LTE Unlicensed and WiFi. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 144–152, 2018.

[4] Z. Chi, Z. Huang, Y. Yao, T. Xie, H. Sun, and T. Zhu. EMF: Embedding multiple flows of information in existing traffic for concurrent communication among heterogeneous IoT devices. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[5] Z. Yin, W. Jiang, S. M. Kim, and T. He. C-Morse: Cross-technology communication with transparent Morse coding. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[6] Song Min Kim and Tian He. FreeBee: Cross-Technology Communication via Free Side-Channel. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, page 317–330, New York, NY, USA, 2015. Association for Computing Machinery.

[7] Kameswari Chebrolu and Ashutosh Dhekne. Esense: Communication through Energy Sensing. In *Proceedings of the 15th Annual International Conference on Mobile Computing and Networking*, MobiCom '09, pages 85–96, New York, NY, USA, 2009. Association for Computing Machinery.

[8] Zicheng Chi, Yan Li, Hongyu Sun, Yao Yao, Zheng Lu, and Ting Zhu. B2W2: N-Way Concurrent Communication for IoT Devices. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, pages 245–258, New York, NY, USA, 2016. Association for Computing Machinery.

[9] Zhimeng Yin, Wenchao Jiang, Song Min Kim, and Tian He. C-Morse: Cross-technology communication with transparent Morse coding. pages 1–9, 2017.

[10] Xiuzhen Guo, Xiaolong Zheng, and Yuan He. WiZig: Cross-technology energy communication over a noisy channel. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[11] Zicheng Chi, Yan Li, Zhichuan Huang, Hongyu Sun, and Ting Zhu. Simultaneous Bi-directional Communications and Data Forwarding using a Single ZigBee Data Stream. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 577–585, 2019.

[12] Zicheng Chi, Zhichuan Huang, Yao Yao, Tiantian Xie, Hongyu Sun, and Ting Zhu. EMF: Embedding multiple flows of information in existing traffic for concurrent communication among heterogeneous IoT devices. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[13] Wenchao Jiang, Zhimeng Yin, Song Min Kim, and Tian He. Side Channel Communication over Wireless Traffic: A CTC Design: Poster Abstract. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, pages 346–347, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Piotr Gaw lowicz, Anatolij Zubow, Suzan Bayhan, and Adam Wolisz. OfdmFi: Enabling Cross-Technology Communication Between LTE-U/LAA and WiFi, 2019.

[15] Piotr Gawlowicz, Anatolij Zubow, Suzan Bayhan, and Adam Wolisz. Punched Cards over the Air: Cross-Technology Communication Between LTE-U/LAA and WiFi. In *2020 IEEE 21st International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, pages 297–306, 2020.

[16] Piotr Gaw lowicz, Anatolij Zubow, and Suzan Bayhan. Demo Abstract: Cross-Technology Communication between LTE-U/LAA and WiFi. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1272–1273, 2020.

[17] Zhijun Li and Tian He. WEBee: Physical-Layer Cross-Technology Communication via Emulation. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, page 2–14, New York, NY, USA, 2017. Association for Computing Machinery.

[18] Zhijun Li and Tian He. LongBee: Enabling Long-Range Cross-Technology Communication. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 162–170, 2018.

[19] Yongrui Chen, Shuai Wang, Zhijun Li, and Tian He. Reliable Physical-Layer Cross-Technology Communication With Emulation Error Correction. *IEEE/ACM Transactions on Networking*, 28(2):612–624, 2020.

[20] Xiuzhen Guo, Yuan He, Jia Zhang, and Haotian Jiang. WIDE: Physical-level CTC via Digital Emulation. In *2019 18th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 49–60, 2019.

[21] Wenchao Jiang, Zhimeng Yin, Ruofeng Liu, Zhijun Li, Song Min Kim, and Tian He. BlueBee: A 10,000x Faster Cross-Technology Communication via PHY Emulation. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, New York, NY, USA, 2017. Association for Computing Machinery.

[22] Wenchao Jiang, Song Min Kim, Zhijun Li, and Tian He. Achieving Receiver-Side Cross-Technology Communication with Cross-Decoding. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 639–652, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Ruofeng Liu, Zhimeng Yin, Wenchao Jiang, and Tian He. LTE2B: Time-Domain Cross-Technology Emulation under LTE Constraints. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, page 179–191, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Bluetooth SIG. Bluetooth Market Update 2019. https://www.bluetooth.com/wp-content/uploads/2018/04/2019-Bluetooth-Market-Update.pdf, 2019.

[25] Bluetooth SIG. Bluetooth Market Update 2020. https://www.bluetooth.com/wp-content/uploads/2020/03/2020_Market_Update-EN.pdf, 2020.

[26] Wi-Fi Alliance. Wi-Fi® in 2019. https://www.wi-fi.org/news-events/newsroom/wi-fi-in-2019, Feb 2019.

[27] Cisco. Cisco Annual Internet Report (2018–2023). https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf, Mar 2020.

[28] Cisco. Cisco Virtual Beacon Solution. https://content.etilize.com/Manufacturer-Brochure/1044848777.pdf, Dec 2017.

[29] Texas Instruments. Bluetooth Low Energy Scanning and Advertising. https://dev.ti.com/tirex/explore/node?node=AKvX4BPHvI6W3ea9a0OTxA__pTTHBmu__LATEST, 2020.

[30] IEEE Standard for Information technology—Telecommunications and information exchange between systems Local and metropolitan area networks—Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pages 1–3534, 2016.

[31] A. Viterbi. Convolutional Codes and Their Performance in Communication Systems. *IEEE Transactions on Communication Technology*, 19(5):751–772, 1971.

[32] G. D. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[33] OpenWrt. OpenWrt. https://openwrt.org, May 2020.

[34] OpenWrt. ath79. https://openwrt.org/docs/techref/targets/ath79, May 2020.

[35] SciPy. SciPy. https://www.scipy.org/, Jan 2021.

[36] Tien Dang Vo-Huu, Triet Dang Vo-Huu, and Guevara Noubir. Fingerprinting Wi-Fi Devices Using Software Defined Radios. In *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '16, page 3–14, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Mathy Vanhoef, Célestin Matte, Mathieu Cunche, Leonardo S. Cardoso, and Frank Piessens. Why MAC Address Randomization is Not Enough: An Analysis of Wi-Fi Network Discovery Mechanisms. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 413–424, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Nordic Semiconductor. nRF Connect for Mobile. https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Connect-for-mobile, 2020.

[39] Nicolas Bridoux. Beacon Scanner. https://play.google.com/store/apps/details?id=com.bridou_n.beaconscanner, 2020.

[40] Vincent Hiribarren. Beacon Simulator. https://play.google.com/store/apps/details?id=net.alea.beaconsimulator, 2020.

[41] iPerf. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/, 2020.

[42] OpenWrt. The high-resolution timer API. https://lwn.net/Articles/167897/, Jan 2006.

[43] Inc Teledyne LeCroy. FTS4BT Bluetooth Protocol Analyzer and Packet Sniffer. https://www.fte.com/products/fts4bt.aspx, 2021.

[44] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[45] Intel Open Source Technology Center. PowerTOP. https://01.org/powertop, 2021.

[46] Matthias Schulz, Jakob Link, Francesco Gringoli, and Matthias Hollick. Shadow Wi-Fi: Teaching smartphones to transmit raw signals and to extract channel state information to implement practical covert channels over wi-fi. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, page 256–268, New York, NY, USA, 2018. Association for Computing Machinery.

[47] Eugene Chai, Karthik Sundaresan, Mohammad A. Khojastepour, and Sampath Rangarajan. LTE in Unlicensed Spectrum: Are We There Yet? In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, page 135–148, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Vikram Iyer, Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua Smith. Inter-Technology Backscatter: Towards Internet Connectivity for Implanted Devices. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 356–369, New York, NY, USA, 2016. Association for Computing Machinery.

[49] Zhijun Li and Yongrui Chen. BlueFi: Physical-layer Cross-Technology Communication from Bluetooth to WiFi. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 399–409, 2020.

[50] Zhenjiang Li, Yaxiong Xie, Mo Li, and Kyle Jamieson. Recitation: Rehearsing Wireless Packet Reception in Software. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, page 291–303, New York, NY, USA, 2015. Association for Computing Machinery.

[51] B. Bloessl, C. Sommer, F. Dressier, and D. Eckhoff. The scrambler attack: A robust physical layer attack on location privacy in vehicular networks. In *2015 International Conference on Computing, Networking and Communications (ICNC)*, pages 395–400, Feb 2015.

[52] Silicon Labs. WF200 Data Sheet: Wi-Fi Network Co-Processor. https://www.silabs.com/documents/public/data-sheets/wf200-datasheet.pdf, Sep 2020.

[53] Texas Instruments. CC2500 Low-Cost Low-Power 2.4 GHz RF Transceiver. https://www.ti.com/lit/ds/swrs040c/swrs040c.pdf, May 2008.

[54] Thomas Zachariah, Noah Klugman, Bradford Campbell, Joshua Adkins, Neal Jackson, and Prabal Dutta. The Internet of Things Has a Gateway Problem. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile '15, page 27–32, New York, NY, USA, 2015. Association for Computing Machinery.

[55] Cisco. Cisco Wireless Mesh Access Points, Design and Deployment Guide, Release 7.4. https://www.cisco.com/c/en/us/td/docs/wireless/technology/mesh/7-4/design/guide/mesh74/mesh74_chapter_011.html, Jun 2020.

[56] Cisco. Cisco Aironet 2700 Series Access Points Data Sheet. https://www.cisco.com/c/en/us/products/collateral/wireless/aironet-2700-series-access-point/datasheet-c78-730593.html, Jul 2020.

[57] Texas Instruments. CC2620 SimpleLink ZigBee RF4CE Wireless MCU. https://www.ti.com/lit/ds/symlink/cc2620.pdf, Dec 2020.

[58] Atmel. AT86RF233. http://ww1.microchip.com/downloads/en/devicedoc/atmel-8351-mcu_wireless-at86rf233_datasheet.pdf, Jul 2014.

[59] Texas Instruments. CC2652R SimpleLink Multiprotocol 2.4 GHz Wireless MCU. https://www.ti.com/lit/ds/symlink/cc2652r.pdf, Mar 2021.

[60] Qualcomm. CSR8811. https://www.qualcomm.com/products/csr8811, 2021.

[61] Cypress. CYW43012. https://www.cypress.com/file/497511/download, Jun 2020.

[62] Michael Ossmann. Ubertooth One. https://greatscottgadgets.com/ubertoothone/, 2021.

[63] Texas Instruments. CC2400 2.4 GHz Low-Power RF Transceiver. https://www.ti.com/lit/ds/symlink/cc2400.pdf, Mar 2006.

[64] NXP. Scalable Mainstream 32-bit Microcontroller (MCU) based on Arm Cortex-M3 Core. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc1700-cortex-m3/scalable-mainstream-32-bit-microcontroller-mcu-based-on-arm-cortex-m3-core:LPC1756FBD80 , 2021.

[65] Duy Nguyen, J. J. Garcia-Luna-Aceves, and Cedric Westphal. Throughput enabled rate adaptation in wireless networks. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 1173–1178, 2013.

[66] Electronic Products. Ralink Premiers Industry's First 450 Mbps 802.11n Router Solution with Beamforming Technology at CES 2009. https://www.electronicproducts.com/ralink-premiers-industrys-first-450-mbps-802-11n-router-solution-with-beamforming-technology-at-ces-2009/ , Jan 2009.

[67] Rémy Grünblatt, Isabelle Guérin-Lassous, and Olivier Simonin. Simulation and Performance Evaluation of the Intel Rate Adaptation Algorithm. In *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWIM '19, page 27–34, New York, NY, USA, 2019. Association for Computing Machinery.

[68] Connectivity Standards Alliance. Zigbee FAQ. https://zigbeealliance.org/zigbee-faq/, 2021.

[69] Cisco. Cisco Aironet 1570 Series Outdoor Access Point Data Sheet. https://www.cisco.com/c/en/us/products/collateral/wireless/aironet-1570-series/datasheet-c78-732348.html, Mar 2021.

[70] Texas Instruments. CC2591 2.4-GHz RF Front End. https://www.ti.com/lit/ds/swrs070b/swrs070b.pdf, Sep 2014.

[71] James E. Gilley. Digital Phase Modulation. https://www.efjohnson.com/resources/dyn/files/75832z342fce97/_fn/Digital_Phase_Modulation.pdf, Aug 2003.

[72] Hsun-Wei Cho and Kang G. Shin. BlueFi: Bluetooth over WiFi. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 475–487, New York, NY, USA, 2021. Association for Computing Machinery.

[73] Ruofeng Liu, Zhimeng Yin, Wenchao Jiang, and Tian He. WiBeacon: Expanding BLE Location-Based Services via Wifi. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, page 83–96, New York, NY, USA, 2021. Association for Computing Machinery.

[74] Lingang Li, Yongrui Chen, and Zhijun Li. Poster Abstract: Physical-layer Cross-Technology Communication with Narrow-Band Decoding. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–2, 2019.

[75] Mohammed Abdullah Zubair, Ajay Kumar Nain, Jagadish Bandaru, P. Rajalakshmi, and U.B. Desai. Reconfigurable dual mode ieee 802.15.4 digital baseband receiver for diverse iot applications. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 389–394, 2016.

[76] Ruirong Chen and Wei Gao. TransFi: Emulating Custom Wireless Physical Layer from Commodity Wifi. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, pages 357–370, New York, NY, USA, 2022. Association for Computing Machinery.

[77] Lingang Li, Yongrui Chen, and Zhijun Li. WiBle: Physical-Layer Cross-Technology Communication with Symbol Transition Mapping. In *2021 18th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 1–9, 2021.

[78] Hsun-Wei Cho and Kang G. Shin. FLEW: Fully Emulated Wifi. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, MobiCom '22, page 29–41, New York, NY, USA, 2022. Association for Computing Machinery.

[79] Microwave Journal. TI delivers industry's only OTA download capabilities for seamless software updates of Bluetooth Smart products. https://www.microwavejournal.com/articles/19352-ti-delivers-industrys-only-ota-download-capabilities-for-seamless-software-updates-of-bluetooth-smart-products, Mar 2013.

[80] Texas Instruments. CC2544: System-on-Chip for 2.4-GHz USB Applications. https://www.ti.com/lit/ds/symlink/cc2544.pdf, 2012.

[81] Texas Instruments. CC2544 2.4 GHz RF Value Line SoC with 32kB flash, USB, SPI and UART. https://www.ti.com/product/CC2544, 2022.

[82] Intel Corporation. MCS51 Microcontroller Family User's Manual, Feb 1994.

[83] T. Aardenne-Ehrenfest, van and N.G. Bruijn, de. ”Circuits and trees in oriented linear graphs”. *Simon Stevin : Wis- en Natuurkundig Tijdschrift*, 28:203–217, 1951.

[84] IAR Systems. IAR Embedded Workbench for 8051. https://www.iar.com/products/architectures/iar-embedded-workbench-for-8051/, 2021.

[85] Texas Instruments. CC-DEBUGGER. https://www.ti.com/tool/CC-DEBUGGER, 2014.

[86] Ioannis Charalampidis. CCLib: An arduino library that implements the CC.Debugger protocol of TI. https://github.com/wavesoft/CCLib, 2017.

[87] Texas Instruments. New TI Bluetooth low energy software stack delivers over-the-air downloads and support for multiple stacks on one SoC. https://www.prnewswire.com/news-releases/new-ti-bluetooth-low-energy-software-stack-delivers-over-the-air-downloads-and-support-for-multiple-stacks-on-one-soc-184270451.html, Dec 2012.

[88] kernel.org. ip-sysctl.txt. https://kernel.org/doc/Documentation/networking/ip-sysctl.txt, 2022.

[89] Ludovic Jacquin, Vincent Roca, and Jean-Louis Roch. Too big or too small? the ptb-pts icmp-based attack against ipsec gateways. In *2014 IEEE Global Communications Conference*, pages 530–536, 2014.

[90] Nokia. Nokia SR Linux: Network-instances. https://infocenter.nokia.com/public/SRLINUX200R6A/index.jsp?topic=/com.srlinux.configbasics/html/configb-network_instances.html, 2020.

[91] Matthew Luckie and Ben Stasiewicz. Measuring Path MTU Discovery Behaviour. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 102–108, New York, NY, USA, 2010. Association for Computing Machinery.

[92] Arun Batra and James R. Zeidler. Narrowband interference mitigation in OFDM systems. In *MILCOM 2008 - 2008 IEEE Military Communications Conference*, pages 1–7, 2008.

[93] Mohamed Marey and Heidi Steendam. Analysis of the Narrowband Interference Effect on OFDM Timing Synchronization. *IEEE Transactions on Signal Processing*, 55(9):4558–4566, 2007.

[94] Jialiang Yan, Siyao Cheng, Zhijun Li, and Jie Liu. PCTC: Parallel Cross Technology Communication in Heterogeneous wireless systems. In *2022 21st ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 67–78, 2022.

[95] Shuai Wang, Woojae Jeong, Jinhwan Jung, and Song Min Kim. X-MIMO: Cross-Technology Multi-User MIMO. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, pages 218–231, New York, NY, USA, 2020. Association for Computing Machinery.

[96] Dan Xia, Xiaolong Zheng, Fu Yu, Liang Liu, and Huadong Ma. WiRa: Enabling Cross-Technology Communication from WiFi to LoRa with IEEE 802.11ax. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 430–439, 2022.

[97] Piotr Gaw lowicz, Anatolij Zubow, and Falko Dressler. Wi-Lo: Emulation of LoRa using Commodity 802.11b WiFi Devices. In *IEEE International Conference on Communications (ICC 2022)*, Seoul, South Korea, 5 2022. IEEE.

[98] Yuanhe Shu, Jingwei Wang, Linghe Kong, Jiadi Yu, Guisong Yang, Yueping Cai, Zhen Wang, and Muhammad Khurram Khan. WiBWi: Encoding-based Bidirectional Physical-Layer Cross-Technology Communication between BLE and WiFi. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 356–363, 2021.

[99] Silicon Labs. Silicon Labs Launches Blue Gecko Bluetooth Smart Solutions. https://news.silabs.com/2015-02-23-Silicon-Labs-Launches-Blue-Gecko-Bluetooth-Smart-Solutions, Feb 2015.

[100] Silicon Labs. Six Hidden Costs in a 99 Cent Wireless SoC. https://www.rs-online.com/designspark/rel-assets/ds-assets/uploads/knowledge-items/application-notes-for-the-internet-of-things/Six-hidden-costs-of-a-99-cent-soc.pdf, 2015.

[101] Texas Instruments. Low-Power Carbon Monoxide Detector With BLE and 10-Year Coin Cell Battery Life Reference Design. https://www.tij.co.jp/lit/ug/tidubx8c/tidubx8c.pdf, Nov 2017.

[102] Thomas Zachariah, Neal Jackson, and Prabal Dutta. The Internet of Things Still Has a Gateway Problem. In *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications*, HotMobile '22, page 109–115, New York, NY, USA, 2022. Association for Computing Machinery.

[103] Hsun-Wei Cho and Kang G. Shin. Unify: Turning BLE/FSK SoC into WiFi SoC. In *Proceedings of the 29th Annual International Conference on Mobile Computing And Networking*, MobiCom '23, New York, NY, USA, 2023. Association for Computing Machinery.

[104] Hadi Givehchian, Nishant Bhaskar, Eliana Rodriguez Herrera, Héctor Rodrigo López Soto, Christian Dameff, Dinesh Bharadia, and Aaron Schulman. Evaluating Physical-Layer BLE Location Tracking Attacks on Mobile Devices. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1690–1704, 2022.

[105] Texas Instruments. 2.4-GHz Bluetooth low energy and Proprietary System-on-Chip. https://www.ti.com/lit/ds/symlink/cc2541.pdf, Jan 2012.

[106] Bluetooth SIG. Core specification v5.1. , Jan 2019.

[107] STMicroelectronics. Arm Cortex-M0 in a nutshell. https://www.st.com/content/st_com/en/arm-32-bit-microcontrollers/arm-cortex-m0.html.

[108] ARM. Neon. https://developer.arm.com/Architectures/Neon, 2022.

[109] NXP. MKW41Z/31Z/21Z Reference Manual. https://www.nxp.com/files-static/32bit/doc/ref_manual/MKW41Z512RM.pdf, Oct 2016.

[110] ARM. Arm Cortex-M0+ Processor Datasheet. https://documentation-service.arm.com/static/620545c494e7af28dd7c9cbc, 2020.

[111] NXP. Bluetooth Low Energy/IEEE 802.15.4 Packet Sniffer/USB Dongle. https://www.nxp.com/design/development-boards/freedom-development-boards/wireless-connectivy/bluetooth-low-energy-ieee-802-15-4-packet-sniffer-usb-dongle:USB-KW41Z, Mar 2018.

[112] Panasonic Industry Europe GmbH. PAN4620 (KW41Z). https://industry.panasonic.eu/products/devices/wireless-connectivity/ieee-802154-modules/pan4620-kw41z, Feb 2020.

[113] ublox. R41Z Stand-alone Bluetooth 4.2 low energy and IEEE 802.15.4 module. https://content.u-blox.com/sites/default/files/R41Z_DataSheet_UBX-19033355.pdf, Jun 2022.

[114] NXP. MCUXpresso Integrated Development Environment (IDE). https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDE, Jan 2023.

[115] ARM. GNU Arm Embedded Toolchain Downloads. https://developer.arm.com/downloads/-/gnu-rm, Oct 2021.

[116] NXP. OpenSDA Serial and Debug Adapter. https://www.nxp.com/design/software/development-software/sensor-toolbox-sensor-development-ecosystem/opensda-serial-and-debug-adapter:OPENSDA , 2023.

[117] Freescale Semiconductor. M68HC08 to HCS08 Transition. https://www.nxp.com/docs/en/application-note/AN2717.pdf, Aug 2006.

[118] Tom Kistner. Scream - Virtual network sound card for Microsoft Windows. https://github.com/duncanthrax/scream, Sep 2022.

[119] SparkLAN Communications. WNFQ-158ACN. https://www.sparklan.com/product/wnfq-158acnbt-qca9377-5-m-2-industrial-module/, 2023.

[120] Ali Abedi, Omid Abari, and Tim Brecht. Wi-LE: Can WiFi Replace Bluetooth? HotNets '19, page 117–124, New York, NY, USA, 2019. Association for Computing Machinery.

[121] Espressif Systems. ESP32 Series Datasheet. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf, Jul 2023.

[122] Xiuzhen Guo, Yuan He, Xiaolong Zheng, Zihao Yu, and Yunhao Liu. LEGO-Fi: Transmitter-Transparent CTC with Cross-Demapping. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 2125–2133, 2019.

[123] Ruofeng Liu, Zhimeng Yin, Wenchao Jiang, and Tian He. XFi: Cross-technology IoT Data Collection via Commodity WiFi. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–11, 2020.

[124] Zicheng Chi, Yan Li, Xin Liu, Yao Yao, Yanchao Zhang, and Ting Zhu. Parallel Inclusive Communication for Connecting Heterogeneous IoT Devices at the Edge. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, page 205–218, New York, NY, USA, 2019. Association for Computing Machinery.

[125] Zicheng Chi, Yan Li, Yao Yao, and Ting Zhu. PMC: Parallel multi-protocol communication to heterogeneous IoT radios within a single WiFi channel. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.

[126] Sihan Yu, Xiaonan Zhang, Pei Huang, and Linke Guo. Physical-Level Parallel Inclusive Communication for Heterogeneous IoT Devices. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 380–389, 2022.

[127] Xinyou Qiu, Bowen Wang, Jian Wang, and Yuan Shen. AOA-Based BLE Localization with Carrier Frequency Offset Mitigation. In *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–5, 2020.

[128] Cheng Huang, Yuan Zhuang, Hao Liu, Jianyu Li, and Wei Wang. A Performance Evaluation Framework for Direction Finding Using BLE AoA/AoD Receivers. *IEEE Internet of Things Journal*, 8(5):3331–3345, 2021.

[129] Marco Cominelli, Paul Patras, and Francesco Gringoli. Dead on Arrival: An Empirical Study of The Bluetooth 5.1 Positioning System. In *Proceedings of the 13th International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization*, WiNTECH '19, page 13–20, New York, NY, USA, 2019. Association for Computing Machinery.

[130] Pooneh Mohaghegh, Alexis Boegli, and Yves Perriard. Bluetooth Low Energy Direction Finding Principle. In *2021 24th International Conference on Electrical Machines and Systems (ICEMS)*, pages 830–834, 2021.

[131] Da Sun, Yinong Zhang, Weiwei Xia, Zhiyuan Geng, Feng Yan, Lianfeng Shen, and Yingbin Gao. A BLE Indoor Positioning Algorithm based on Weighted Fingerprint Feature Matching Using AOA and RSSI. In *2021 13th International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6, 2021.

[132] Wei He, Chenglin Huang, Zengshan Tian, Kaikai Liu, and Ze Li. Design and Implementation of Bluetooth Low Energy AoA Estimation System. In *2022 IEEE International Symposium on Antennas and Propagation and USNC-URSI Radio Science Meeting (AP-S/URSI)*, pages 311–312, 2022.

[133] Shuai He, Hang Long, and Wei Zhang. Multi-antenna Array-based AoA Estimation Using Bluetooth Low Energy for Indoor Positioning. In *2021 7th International Conference on Computer and Communications (ICCC)*, pages 2160–2164, 2021.

[134] Gaurav Kumar, Vrinda Gupta, and Rahul Tank. Phase-based Angle estimation approach in Indoor Localization system using Bluetooth Low Energy. In *2020 International Conference on Smart Electronics and Communication (ICOSEC)*, pages 904–912, 2020.

[135] Usman Raza, Aftab Khan, Roget Kou, Tim Farnham, Thajanee Premalal, Aleksandar Stanoev, and William Thompson. Dataset: Indoor Localization with Narrow-Band, Ultra-Wideband, and Motion Capture Systems. In *Proceedings of the 2nd Workshop on Data Acquisition To Analysis*, DATA'19, page 34–36, New York, NY, USA, 2019. Association for Computing Machinery.

[136] Bluetooth SIG. 2023 Bluetooth Market Update. https://www.bluetooth.com/2023-market-update/, 2023.

[137] Bluetooth SIG. Advanced Audio Distribution Profile 1.4. https://www.bluetooth.com/specifications/specs/advanced-audio-distribution-profile-1-4/, 2022.

[138] MathWorks. Bluetooth Protocol Stack. https://www.mathworks.com/help/bluetooth/ug/bluetooth-protocol-stack.html, 2024.

[139] Kevin Townsend, Carles Cuf, Akiba, and Robert Davidson. *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking*. O'Reilly Media, Inc., 1st edition, 2014.

[140] Google. Low Complexity Communication Codec (LC3). https://github.com/google/liblc3, 2022.

[141] Personal blind comparison of the Bluetooth codecs, AAC vs LC3, re-encoding. https://hydrogenaud.io/index.php/topic,122575.0.html, 2022.

[142] Bluetooth SIG. Bluetooth Core Specification v5.3. https://www.bluetooth.com/specifications/specs/core-specification-5-3/, 2021.

[143] Adoroma. What is Bluetooth Range? What You Need to Know. https://www.adorama.com/alc/bluetooth-range/#:~:text=Class%201%20is%20the%20longest,not%20used%20for%20consumer%20products., 2022.

[144] Bluetooth SIG. Specification of the Bluetooth System, 1999.

[145] Arnaud Delmas. A C implementation of the Bluetooth stream cipher E0. https://github.com/adelmas/e0, 2015.

[146] Bluetooth SIG. A/V Control Transport Protocol 1.4. https://www.bluetooth.com/specifications/specs/a-v-control-transport-protocol-1-4/, 2012.

[147] Bluetooth SIG. A/V Distribution Transport Protocol 1.3. https://www.bluetooth.com/specifications/specs/a-v-distribution-transport-protocol-1-3/, 2012.

[148] Texas Instruments. CC2540EMK-USB. https://www.ti.com/tool/CC2540EMK-USB, 2023.

[149] Sandeep Dutta. SDCC - Small Device C Compiler. https://sdcc.sourceforge.net/, 2024.

[150] Frontline Test Equipment. Frontline Introduces World's Only Bluetooth v3.0 + HS Protocol Analyzer. https://fte.com/docs/PressReleases/FTS4BT-HS-press-release.pdf, 2009.

[151] Apple Inc. Accessory Design Guidelines for Apple Devices. https://developer.apple.com/accessories/Accessory-Design-Guidelines.pdf, 2023.

[152] Qualcomm. Qualcomm FastConnect 6900 System. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/qualcomm-fastconnect-6900-product-brief_finalv7.pdf, 2020.

[153] Maxim Chernyshev, Craig Valli, and Michael Johnstone. Revisiting Urban War Nibbling: Mobile Passive Discovery of Classic Bluetooth Devices Using Ubertooth One. *Trans. Info. For. Sec.*, 12(7):1625–1636, jul 2017.

[154] Aaron Kinfe, Chijung Jung, Kai Lin, Marshall Clyburn, and Fnu Suya. HackWrt: Network Traffic-Based Eavesdropping of Handwriting. In *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*, CPS-IoT Week '23, page 55–60, New York, NY, USA, 2023. Association for Computing Machinery.

[155] Thomas Willingham, Cody Henderson, Blair Kiel, Md Shariful Haque, and Travis Atkison. Testing vulnerabilities in bluetooth low energy. In *Proceedings of the ACMSE 2018 Conference*, ACMSE '18, New York, NY, USA, 2018. Association for Computing Machinery.

[156] Wahhab Albazrqaoe, Jun Huang, and Guoliang Xing. Practical Bluetooth Traffic Sniffing: Systems and Privacy Implications. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 333–345, New York, NY, USA, 2016. Association for Computing Machinery.

[157] Marco Cominelli, Francesco Gringoli, Paul Patras, Margus Lind, and Guevara Noubir. Even Black Cats Cannot Stay Hidden in the Dark: Full-band De-anonymization of Bluetooth Classic Devices. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 534–548, 2020.

[158] Frontline Test Equipment. FTS4BT Wireless Sniffer for Bluetooth Technology. http://www.er-soft.com/files/media/files/Frontline--Bluetooth--v2-1--EDR-Analyzer--High-Speed--UART--FTS4BT.Pdf, 2007.

[159] Jiska Classen and Matthias Hollick. Inside job: diagnosing bluetooth lower layers using off-the-shelf devices. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '19, page 186–191, New York, NY, USA, 2019. Association for Computing Machinery.

[160] The InternalBlue Team. Bluetooth experimentation framework for Broadcom and Cypress chips. https://github.com/seemoo-lab/internalblue, 2021.

[161] Dennis Mantz. InternalBlue—A Bluetooth Experimentation Framework Based on Mobile Device Reverse Engineering. Master's thesis. Technische Universität Darmstadt. http://tubiblio.ulb.tu-darmstadt.de/107125/, 2018.

[162] Matheus Eduardo. BrakTooth ESP32 BR/EDR Active Sniffer/Injector. https://github.com/Matheus-Garbelini/esp32_bluetooth_classic_sniffer, 2023.

[163] J. Wu, R. Wu, D. Xu, D. Tian, and A. Bianchi. SoK: The Long Journey of Exploiting and Defending the Legacy of King Harald Bluetooth. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 23–23, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[164] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, August 2019.

[165] NIST. CVE-2019-9506. https://nvd.nist.gov/vuln/detail/CVE-2019-9506, 2019.

[166] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. BIAS: Bluetooth Impersonation AttackS. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2020.

[167] NIST. CVE-2020-10135. https://nvd.nist.gov/vuln/detail/CVE-2020-10135, 2020.

[168] Mingrui Ai, Kaiping Xue, Bo Luo, Lutong Chen, Nenghai Yu, Qibin Sun, and Feng Wu. Blacktooth: Breaking through the Defense of Bluetooth in Silence. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 55–68, New York, NY, USA, 2022. Association for Computing Machinery.

[169] Daniele Antonioli, Nils Ole Tippenhauer, Kasper Rasmussen, and Mathias Payer. BLURtooth: Exploiting Cross-Transport Key Derivation in Bluetooth Classic and Bluetooth Low Energy. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 196–207, New York, NY, USA, 2022. Association for Computing Machinery.

[170] NIST. CVE-2020-15802. https://nvd.nist.gov/vuln/detail/CVE-2020-15802, 2020.

[171] Matheus E. Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. BrakTooth: Causing havoc on bluetooth link manager via directed fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1025–1042, Boston, MA, August 2022. USENIX Association.