BlueFi: Bluetooth over WiFi

Hsun-Wei Cho and Kang G. Shin
The University of Michigan

ABSTRACT

Bluetooth and WiFi are the two dominant technologies enabling the communication of mobile and IoT devices. Built with specific design goals and principles, they are vastly different, each using its own hardware and software. Thus, they are not interoperable and require different hardware.

One may, therefore, ask a simple, yet seemingly impossible question: "Can we transmit Bluetooth packets on commercial off-the-shelf (COTS) WiFi hardware?" We answer this question positively 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 apps — Bluetooth beacon and audio — to showcase the practical use of `BlueFi`. The former allows ordinary APs to send location beacons; the latter enables WiFi chips to stream Bluetooth audio in real time.

CCS CONCEPTS

• **Networks** → **Wireless access networks**.

KEYWORDS

Cross-Technology Communication, Bluetooth, WiFi

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 [1]. 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 [2].

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 [3]. 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 [4]. 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 paper, 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.

SIGCOMM '21, August 23–28, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8383-7/21/08…$15.00
https://doi.org/10.1145/3452296.3472920

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 [5] 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 SYSTEM DESIGN

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. A list of acronyms is compiled and provided in Appendix A.4.

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 [6]. 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

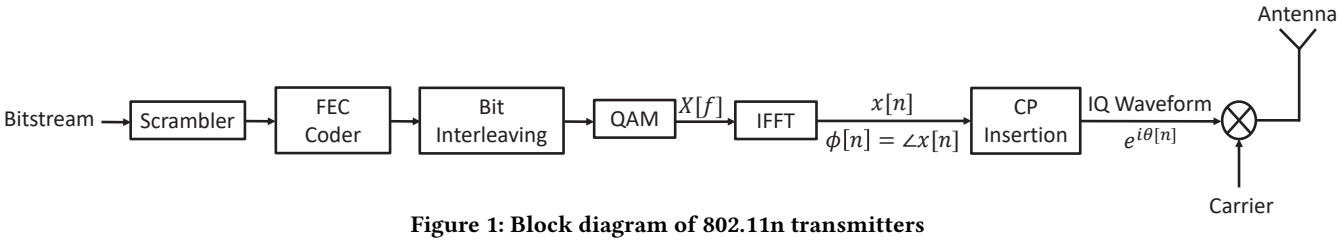


Figure 1: Block diagram of 802.11n transmitters

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.

2.1.2 802.11n. Officially known as High Throughput (HT) PHY in the 802.11 standard [7], 802.11n inherited the same OFDM structure as 802.11a and 802.11g. Fig. 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 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.*

2.3 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 $\theta[n]$.

2.4 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.

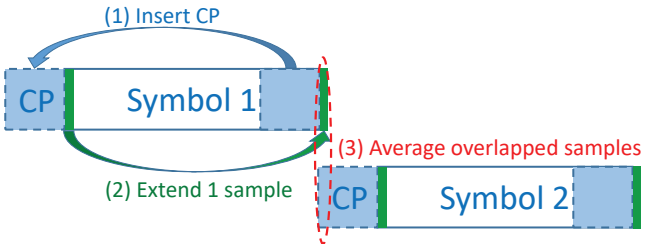


Figure 2: CP insertion and OFDM symbol windowing

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. 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.

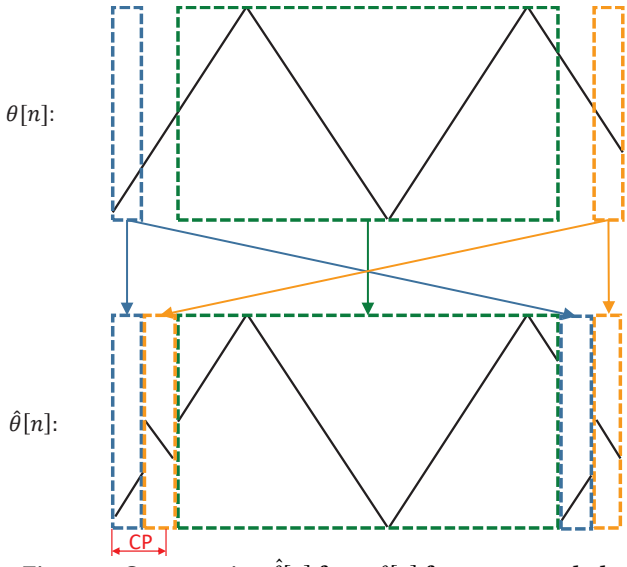


Figure 3: Constructing $\hat{\theta}[n]$ from $\theta[n]$ for every symbol

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. 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 $\hat{\theta}[n]$.

2.5 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, $\hat{X}[f] \in \{a + bi \mid a \in \{\pm1, \pm3, \pm5, \pm7\}, b \in \{\pm1, \pm3, \pm5, \pm7\}\}$.) 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. 4.

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.

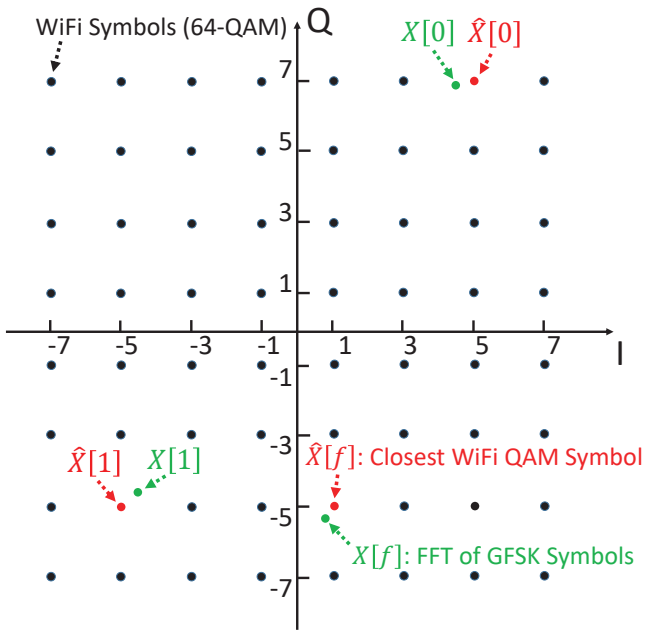


Figure 4: Selecting $\hat{X}[f]$ from $X[f]$. $X[f] = FFT(x[n])$. A scale factor A is applied ($x[n] = A \cdot e^{i \cdot \phi[n]}$) so that $X[f]$ is appropriately scaled w.r.t. origin.

2.7 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 [8, 9]. 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 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 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		⋮	

2.8 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.

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 [10]. The AR9331 belongs to Atheros's widely-adopted ath79 product family. OpenWrt supports at least 272 routers with ath79 chips [11]. `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. 2. The Scipy library [12] 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, [13] 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 [14, 15]. 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.

4 EVALUATION

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 [16] on the iPhone and the Beacon Scanner [17] 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. 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.

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. 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 have different sensitivity. We observe the same behavior even when dedicated Bluetooth hardware is used (Sec. 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. 5c shows the results of using RTL8811AU under the same condition. Compared to Fig. 5b, there are some variations in terms of RSSI, but devices can still steadily receive Bluetooth packets using `BlueFi`.

4.3 Performance vs. WiFi Tx Power

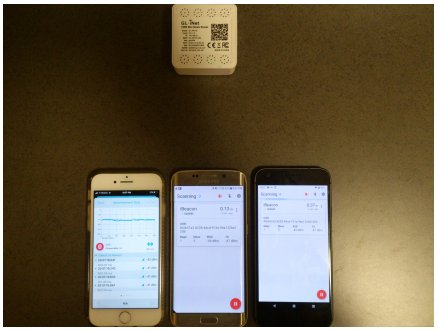
OpenWrt provides a convenient way to control the transmit power, and hence we also measure the received signal strength with respect to different transmit power levels. We placed the phones 1.5m away from the WiFi router. Fig. 6 shows the results. The RSSI is very high on the Pixel and gradually decreases 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.

4.4 Comparison with Bluetooth Hardware

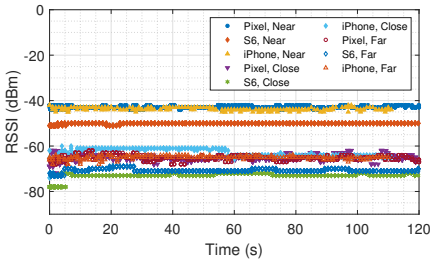
To compare `BlueFi` with dedicated Bluetooth hardware, we also measure the performance of using conventional Bluetooth transmitters. Beacon packets are sent using the Beacon Simulator app [18] on Android. We set the Bluetooth Tx power to high and set the broadcasting frequency to 10Hz. All other conditions are exactly the same as those in Sec. 4.3.

The results are plotted in Fig. 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. 4.3. We can also see that the RSSI is lower on S6 than on iPhone under the same condition.

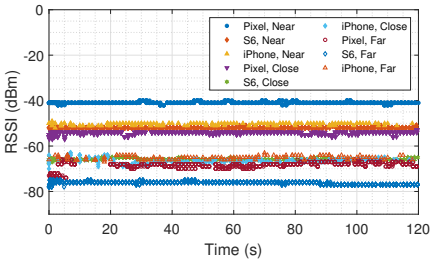
Since conditions are exactly the same as those in Sec. 4.3, we can directly compare Figs. 6 and 7a. At the Tx power of 8dBm, the performance of `BlueFi` is found comparable to those of using a



(a) Experimental setup

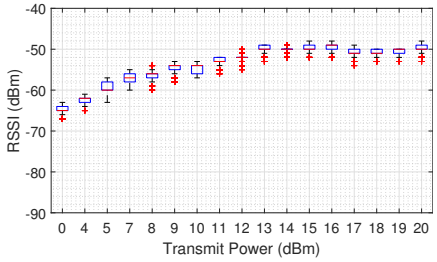


(b) AR9331

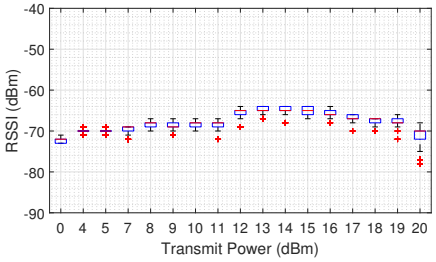


(c) RTL8811AU

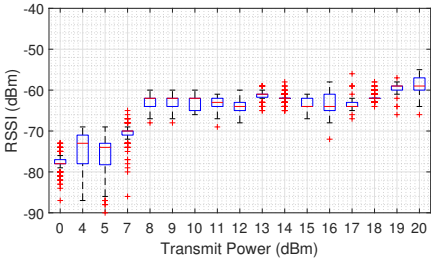
Figure 5: Evaluation of `BlueFi`



(a) Pixel

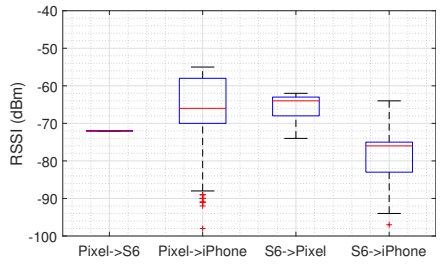


(b) S6

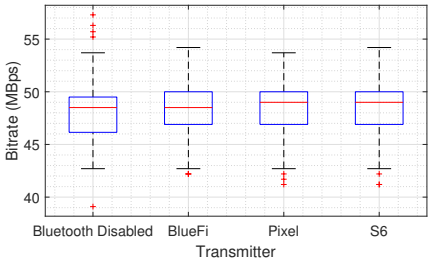


(c) iPhone

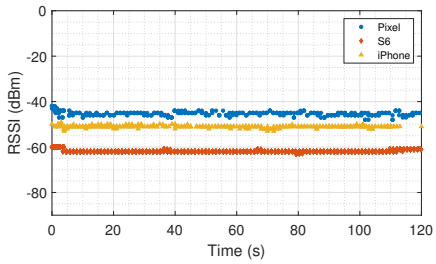
Figure 6: Performance vs. Transmit power



(a) Using dedicated Bluetooth hardware



(b) WiFi throughput measurements



(c) RSSI with background WiFi traffic

Figure 7: Comparison with dedicated hardware and effect of background WiFi traffic

dedicated Bluetooth chip. Therefore, with WiFi chips that nominally come with a default Tx power of 18dBm, one could expect better performance with `BlueFi`.

4.5 Effect on Concurrent WiFi Traffic

We also evaluate the effect of `BlueFi` on concurrent WiFi traffic. For this, we use iPerf3 [19], 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. 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. 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.

4.6 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. 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.

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 [20] 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 [21] 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.

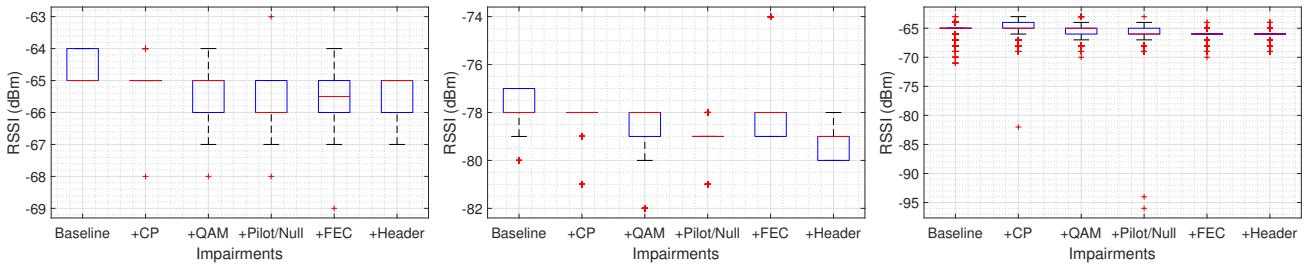
Due to nulls and pilots, the performance of transmission on each Bluetooth channel is different within a single WiFi channel. For example, Fig. 9 shows the packet error rate (PER) reported by FTS4BT of BlueFi transmitting single-slot packets on 10 different channels. PER is shown to be as low as 1.9% 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 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. 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



(a) Pixel (b) S6 (c) iPhone

Figure 8: Effect of each impairment

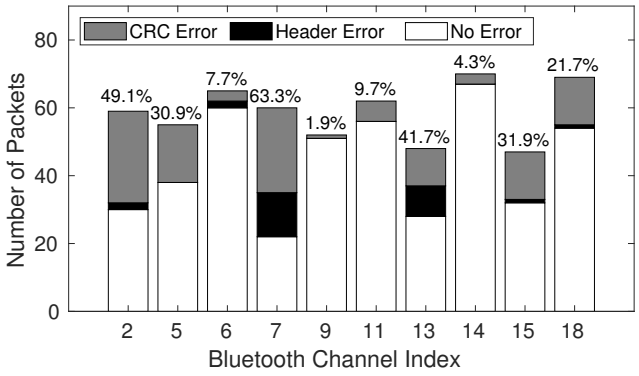


Figure 9: PER with single-slot packets

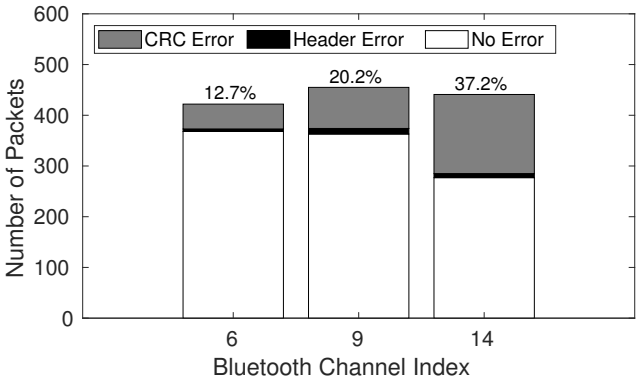


Figure 10: PER with 5-slot packets (audio)

1's and 0's and upper layers are oblivious to how the radio and PHY are actually implemented.

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 [22] 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 [23], 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.

5 DISCUSSION

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.

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.

5.3 Remarks/Observations

BlueFi realizes Bluetooth's radio and physical layers (radio, "baseband" and link control in Bluetooth's terminology), on which *all* apps 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 app or profile can use BlueFi for Bluetooth transmission. As our first work 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. 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 receiving chip to realize full Bluetooth function without the need for regulatory certification. We leave the reception function (WiFi over Bluetooth) as future work.

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

6 RELATED WORK

Shadow Wi-Fi [24] 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 [25–28] 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 [13, 29, 30] 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 [31] emulates WiFi CTS frames using LTE-U waveforms. Interscatter [32] uses WiFi to transmit amplitude-modulated waveforms for RFID communication. WEBee [33] enables WiFi-to-Zigbee communication. As described in Sec. 3.1.3 in [33], 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 [34] focuses on LTE to Zigbee communication. LongBee [35] extends the range of WEBee. TwinBee [36] applies additional channel coding on top of WEBee to improve reliability. WIDE [37] is also similar to WEBee but uses a different pulse-shaping waveform and uses USRP as the transmitter. A recent CTC work [38] 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. 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 [39] examines implementations of WiFi to predict bit-prone locations. Several 802.11 security studies [14, 15, 40] 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).

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.

This work does not raise any ethical issues.

ACKNOWLEDGMENTS

We would like to thank our shepherd and anonymous reviewers for their insightful comments. This work was supported in part by NSF, Grant No. CNS-1646130 and ARO, Grant No. W911NF-21-1-0057.

REFERENCES

[1] Bluetooth SIG. Bluetooth market update 2019. https://www.bluetooth.com/wp-content/uploads/2018/04/2019-Bluetooth-Market-Update.pdf, 2019.
[2] Bluetooth SIG. Bluetooth market update 2020. https://www.bluetooth.com/wp-content/uploads/2020/03/2020_Market_Update-EN.pdf, 2020.
[3] Wi-Fi Alliance. Wi-fi® in 2019. https://www.wi-fi.org/news-events/newsroom/wi-fi-in-2019, Feb 2019.
[4] 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.
[5] Cisco. Cisco virtual beacon solution. https://content.etilize.com/Manufacturer-Brochure/1044848777.pdf, Dec 2017.
[6] Texas Instruments. Bluetooth low energy scanning and advertising. https://dev.ti.com/tirex/explore/node?node=AKvX4BPHvI6W3ea9a0OTxA__pTTHBmu__LATEST, 2020.
[7] 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.
[8] A. Viterbi. Convolutional codes and their performance in communication systems. *IEEE Transactions on Communication Technology*, 19(5):751–772, 1971.
[9] G. D. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.
[10] OpenWrt. Openwrt. https://openwrt.org, May 2020.
[11] OpenWrt. ath79. https://openwrt.org/docs/techref/targets/ath79, May 2020.
[12] SciPy. Scipy. https://www.scipy.org/, Jan 2021.
[13] Piotr Gawłowicz, Anatolij Zubow, Suzan Bayhan, and Adam Wolisz. Ofdmfi: Enabling cross-technology communication between lte-u/laa and wifi, 2019.
[14] 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.
[15] 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.
[16] Nordic Semiconductor. nrf connect for mobile. https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Connect-for-mobile, 2020.
[17] Nicolas Bridoux. Beacon scanner. https://play.google.com/store/apps/details?id=com.bridou_n.beaconscanner, 2020.
[18] Vincent Hiribarren. Beacon simulator. https://play.google.com/store/apps/details?id=net.alea.beaconsimulator, 2020.
[19] iPerf. iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/, 2020.
[20] OpenWrt. The high-resolution timer api. https://lwn.net/Articles/167897/, Jan 2006.
[21] Inc Teledyne LeCroy. Fts4bt bluetooth protocol analyzer and packet sniffer. https://www.fte.com/products/fts4bt.aspx, 2021.
[22] 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".
[23] Intel Open Source Technology Center. Powertop. https://01.org/powertop, 2021.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] Piotr Gawłowicz, 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.
[31] 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.
[32] 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.
[33] 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.
[34] 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.
[35] 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.

[36] 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.
[37] 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.
[38] 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.
[39] 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.
[40] 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.
[41] Keysight Technologies. Guard interval (802.11a/g/j/p ofdm). http://rfmw.em.keysight.com/wireless/helpfiles/89600B/WebHelp/Subsystems/wlan-ofdm/content/dlg_ofdm_fmt_guardintrvlfrac.htm, 2020.

A APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A.1 Processing Considerations

Although it is possible to implement `BlueFi` entirely on the microprocessors in WiFi APs or NICs, we don't see any reason for doing so if it is implemented in software. Since WiFi devices are typically connected to much more powerful hosts or cloud servers, we can leverage the computing power on those devices to run `BlueFi` so that WiFi hardware only needs to send the final bit-stream.

A.2 Recommendations for WiFi Vendors

In the process of designing `BlueFi`, we gained interesting insights into how WiFi chip makers can integrate this functionality or further improve the signal quality from the devices' point of view.

The CP insertion block can completely disrupt the reception, and therefore the signal quality will improve if it can be bypassed. Such a feature is already available on WiFi testing equipment [41] but not yet on commercial chips. Since the operation of IFFT complicates the optimization of time-domain phases, this problem will be much easier if it can be bypassed. Our system is also easier to be used by chips that can control the scrambler seed and are fully compliant with the PSDU length specification in the 802.11n standard.

Note that these recommendations mainly come from a signal quality's perspective. In practice, it might be more beneficial to run `BlueFi` "as is" since some of the modifications may require recertification and `BlueFi` is perfectly operational without these features.

A.3 Finding an Optimal Solution

The principle of `BlueFi` is finding near optimal inverse operations of WiFi transmitters block-by-block. We used this approach mainly because it is intractable to derive a global optimal solution. A global optimal solution must not only guarantee optimality but also consider the interactions between blocks.

Even finding an optimal inversion of some of the blocks is difficult. As we discussed in Sec. 2.5, finding an optimal combination of the QAM symbols can be formulated as an IP problem and the complexity is practically impossible for 64-QAM. Note that the complexity quickly becomes astronomical for 256- and 1024-QAM.

A.4 Glossary

A2DP: Advanced Audio Distribution Profile
AFH: Adaptive Frequency Hopping
ARQ: Automatic Repeat Request
BPSK: Binary Phase Shift Keying
CFO: Carrier Frequency Offset
CP: Cyclic Prefix
CRC: Cyclic Redundancy Check
FEC: Forward Error Correction
FFT : Fast Fourier Transform
(G)FSK: (Gaussian) Frequency-Shift Keying
IFFT: Inverse Fast Fourier Transform
IP: Integer Programming
IQ: In-phase and Quadrature
L2CAP: Logical Link Control and Adaptation Protocol
LDPC: Low-Density Parity-Check
MAC: Medium Access Control
MTU: Maximum Transmission Unit
OFDM: Orthogonal Frequency-Division Multiplexing
PER: Packet Error Rate
QAM: Quadrature Amplitude Modulation
QPSK: Quadrature Phase Shift Keying
RSSI: Received Signal Strength Indicator
SGI: Short Guard Interval
SBC: Sub-Band Coding