WiFo: A Diagnostic Tool for IEEE 802.11 MAC

Hessan Feghhi
Hamilton Institute
National University of Ireland Maynooth
Co. Kildare, Ireland
Email: Hessan.Feghhi@nuim.ie

David Malone
Hamilton Institute
National University of Ireland Maynooth
Co. Kildare, Ireland
Email: David.Malone@nuim.ie

***Abstract*—WLANs are constantly undergoing extensive research and development, and scientists keep coming up with new methods to improve existing protocols and amend standards. Experimental assessment has been an important part of 802.11 research, however measuring the detailed behaviour of the medium and hardware has been challenging. In this paper we design a diagnostic tool, WiFo, for IEEE 802.11-based WLANs. This tool helps developers and researchers monitor and analyze the wireless signals and details such as backoff distribution in a user-friendly environment. Our solution is much cheaper and easier to use than existing tools, and provides more flexibility by allowing users to add functionality. We then use WiFo to study several aspects of some off-the-shelf hardware and their corresponding software drivers, and show some interesting results regarding how they apply standard specifications.**

I. Introduction

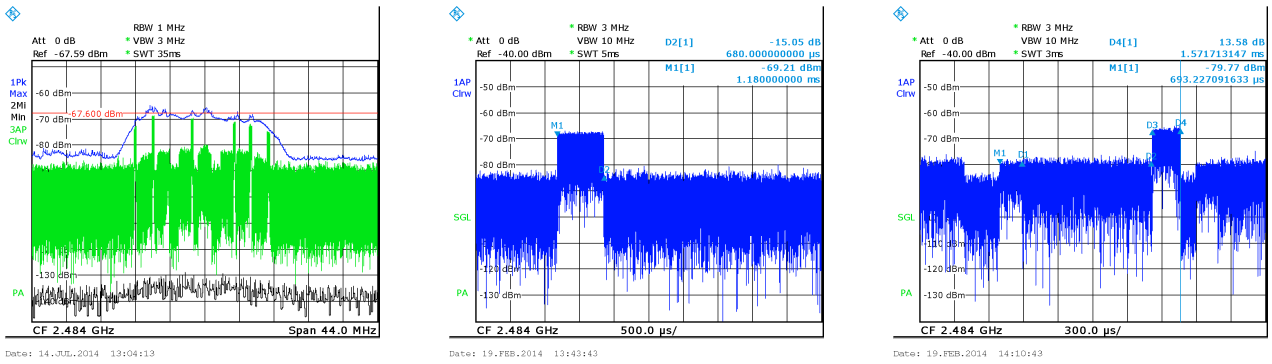
With ever increasing interest in WLANs, researchers have been trying to improve current protocols in terms of performance [1], [2], [3], security [4], [5], and scalability [6]. Mathematical analysis and simulations are common ways of evaluating new methods [7], [8]. The final step in evaluating a new method is putting it into practice on a real network. In order to implement and test on a wireless medium, we also need to understand the behaviour on the medium itself. Although standard specifications are available in detail [9], [10], in many cases what happens in practice can be different to expectations based on standards. Sometimes this is due to the implementation flexibility provided by the standard, and other times there are clear deviations from the standard [11]. Even if we assume all wireless hardware behaves exactly as the standards suggest, other factors, such as interference or competing traffic, may alter expected results of an experiment.

Thus, in order to have controlled experiments, we need to understand the hardware we use and also channel conditions. For example, if a wireless device has an unexpectedly high saturation throughput, we can infer that it is not following the standard. However, without better diagnostics we cannot know which part of the standard is not being followed: it could be an abuse of TXOP, use of a small contention window (CW) or some other issue. For unexpectedly low throughput, other factors, such as hardware failure, protocol errors, channel interference, etc., are possible. In order to identify the underlying problem, we need more information.

One device that is designed for detailed analysis of the wireless medium is a spectrum analyzer. A spectrum analyzer measures the magnitude of an input signal at a particular frequency within a range. By analyzing the spectra of electrical signals, dominant frequency, power, distortion, harmonics, bandwidth, ... details can be observed that are not easily detectable otherwise. Spectrum analyzers are particularly useful for understanding the physical (PHY) layer of a transmitter or receiver, such as power levels, distortion and interference. Some spectrum analyzers even come with add-ons that characterize PHY symbols or packets. We can use a spectrum analyzer to observe the IEEE 802.11 medium by simply adjusting the frequency range to that of the specific channel we would like to study. In Fig. 1a we have done this for WiFi channel 14 while one station transmits saturated traffic to an access point (AP) in this channel. The first plot shows the maximum, minimum and current power observed, and the WiFi channel is clearly visible within the bump in the blue line that shows peak power per frequency. The spectrum analyzer can also give us the changes in power over a period of time using "zero span" mode. By switching to this mode, the x-axis will then show time instead of frequency, with the y-axis still showing power. With zero span, we can actually understand temporal aspects of the channel. Fig. 1b and 1c are snapshots from a spectrum analyzer screen on zero span mode while WiFi traffic is ongoing on channel 14.

While we can observe traffic using a spectrum analyzer, and it provides a number of ways to process the observed data, there are some downsides, including financial cost. A spectrum analyzer is a versatile but relatively expensive device, and thus the expense may not be justified for a group developing WiFi drivers or analyzing some performance anomaly. In addition, because it is a general-purpose device, many 802.11 properties are not recognized by a typical spectrum analyzer, or are only understood by specialist add-on packages. When we study 802.11, we are often interested in things like backoff period, throughput and transmission time. We often need to have numerous samples in order to understand the stochastic behaviour of the WiFi MAC and PHY. Although there are ways to export spectrum data using a spectrum analyzer and process it later on a computer, the whole process is often not easily automated or tailored to those studying WiFi, and so experiments can be difficult without human interaction.

In this paper, we provide a solution that addresses these



(a) Saturated traffic on channel 14 (b) A single beacon (c) Successful frame transmission and ACK

Fig. 1: Spectrum analyzer: 44 MHz frequency span for (a), and zero span for (b) and (c).

problems. While a spectrum analyzer receives, digests and displays electronic signals, a cheap off-the-shelf wireless adapter also receives all those signals, but it then uses them to provide data transmission and reception over the medium. If we can extract the carrier-sense data from the wireless adapter, we may be able to provide and automate some of the features needed to diagnose WiFi performance. Using the flexibility provided by OpenFWWF [12], we will use a Broadcom BCM4318 wireless adapter to create an inexpensive, easily usable, and extensible tool for researchers and developers to study 802.11. What we want to develop is a tool that can monitor WiFi traffic and provide a visual representation of the received data, as well as statistical studies on transmitted frames. The tool will sit just above the PHY layer and focus on the interaction of PHY and MAC layers. Additional functionality can also be introduced to the system through extensions.

The rest of the paper is organized as follows. In Sec. II we briefly review related work. In Sec. III we describe the design and implementation details of WiFo. In Sec. IV we gauge our system against expected results for some scenarios. In Sec. V we use WiFo to study some popular 802.11 adapters and drivers; although these tests use 802.11b for simplicity, the tool itself could well be used for more recent protocols as it is not protocol dependent. Finally, Sec. VI concludes this paper.

II. Related Work

The use of off-the-shelf wireless adapters in research is widespread, partly due to widely-available open-source device drivers. Examples of existing work that detects misbehaving 802.11 devices include [13], where they evaluate previously proposed schemes including DOMINO (see [14]) and SPRT-based schemes (e.g. [15]) by using experimental results from off-the-shelf devices to confirm the correctness of their theoretical analysis. There is a wide range of other experimental work, from [16] and [17], which introduce and implement new rate adaptation algorithms, to [18] where they introduce and experimentally evaluate a new MAC protocol suitable for long-distance multi-hop links.

Tools for testing and debugging have received relatively less attention. In experimental work, such as those just mentioned, and in driver development, intensive testing is required to make sure implementations work as desired. It is widely understood in the research community that debugging an implementation can take many hours. Most of the available tools and approaches e.g. [19], [20] focus on protocol level monitoring, and less on other interactions.

In our approach, we leverage the fact that all wireless adapters have the means of sensing the medium to make a test tool. This feature of the hardware has been used before, for example to implement spectrum sensing for cognitive radio [21] or to detect non-WiFi sources of interference [22].

We use carrier sensing of off-the-shelf wireless hardware in a different way; rather than using data collected from the medium for one particular purpose, we aim to export it to the application layer, where it can be analyzed using high level tools. Works such as [11], where the study of existing wireless hardware or software is intended or required, emphasize the need for the tool we present. We use Broadcom BCM43xx chipsets and open-source firmware that has also been used to allow visual reprograming the wireless layer [23], [24].

III. Design and Architecture

This section covers the implementation details of our diagnostic tool. As we mentioned, we use a commercial off-the-shelf wireless adapter to monitor the medium. We use a custom firmware and driver for the wireless adapter running on a monitor host. To allow greater flexibility in the processing and visualization, we transfer the data from the monitoring host using a small TCP-based server to a front-end host. This separation of monitoring and front-end hosts allows us to install the monitor on a small device, such as a Soekris net4801 [25], while using a higher-powered device for storage and visualization. Our front-end visualizes the data and allows the user to analyze and study the data. Fig. 2 depicts the building blocks of our system. We will discuss the different parts of this diagram in this section. The full source code, along with the modified `b43` driver, as described below, is available in [26].

A. Firmware

The firmware is the software running directly on the NIC chipset and the first layer above the hardware. Signals from

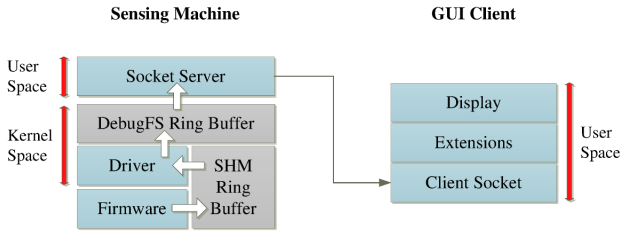


Fig. 2: Architecture diagram of the WiFo diagnostic tool.

Bit	Meaning
15	Flip to 1 when time reserved for receiving PLCP has passed
11	Flip to 1 when RX'ing or TX'ing (same time receiver flips on, and $1\mu s$ after transmitter flips on)
10	Flip to 1 when RX'ing or TX'ing (same time receiver or transmitter flips on)
9	Flip to 1 when receiver has started decoding
8	Flip to 1 when transmitter is working
7	Flip to 1 when backoff is zero
4	Flip to 1 when time reserved for receiving MPDU has passed
3	Flip to 1 when channel is sensed free (phy+nav) for more than two slots
2	Flip to 1 when channel is sensed free (phy+nav) for more than one slot
1	Flip to 1 when channel is sensed free (physically)
0	Flip to 1 when channel is sensed free (virtually through NAV)

TABLE I: Important bits of the "IFS Status" register.

the medium are translated into a digital representation and digested by cores within the chip, and results are provided to the firmware in an "IFS Status" register [27]. The most relevant flags via the IFS Status register are listed in Table I. Using these we can see when the card begins sending or receiving and the state of the medium, and also capture important timeouts.

In order to track the state of the medium, we record the value of the IFS Status register over time and base our analysis on flips of the flags. The firmware has access to 4 KB of shared memory, a large portion of which is unused by its normal workflow. We use the available free memory in the form of a ring buffer, and whenever at least one bit of IFS Status flips, we save its value along with a timestamp from the chip's TSF register (which offers granularity of $1\mu s$). We only save a new record when there is a change as this saves space compared to periodic recording.

Indeed, available memory is the main barrier to implementation. We use the 32 least significant bits of the card's current TSF timer as our timestamp. The IFS Status register itself is 16 bits, giving 48 bits (6 bytes) per record. Even if we could use the chip's whole shared memory, we could only store 667 records. But the shared memory is not completely free and putting aside the memory used for the firmware's normal workflow we are left with space sufficient to accommodate about 250 records. A single frame can trigger multiple bit flips and bits such as 8 and 11 can be flipped $1\mu s$ apart from each other, leading to a large possible number of flips per unit time. In order to increase the number of relevant events captured, we mask out redundant flags and thus reduce the number of triggered events. We combine this with the greater resources available to the driver running on the host to record events.

One might ask where this procedure is actually inserted into the firmware. Event handling in the firmware architecture is not interrupt-based and the firmware continuously polls for events and handles them accordingly. Whenever there are no events, the firmware calls a `nap` instruction to sleep for a short period as a power saving measure, and then continues. This is the best place to insert any repeated code. We simply replace the `nap` instruction with our code. Other parts of the standard OpenFWWF firmware are left unchanged, in order to maintain initialization of the chip, the interface with the driver, etc.

B. Driver

The amount of memory available to the driver is typically much larger than the firmware, as it runs on the host machine which has more resources. Another useful feature of the b43 driver used with OpenFWWF is that it can do periodic work at relatively small intervals. We use these to our advantage. In our implementation, we allocate a large ring buffer in the driver, which can hold 10 times as many records as the firmware. We then read the shared memory periodically and add new records to said ring buffer. The period is chosen so that every record on the shared memory ring buffer can be read before it is overwritten. In our experiments with saturated traffic, this proves to be 25ms; anything less than this period will frequently read redundant information, and longer periods might lead to loss of information.

Exporting the information to user space is our next step. For this, we use the Linux DebugFS module[1], which allows the kernel modules, such as drivers, to export data via a filesystem. The filesystem can be set up using a simple Linux command[2]. Files in this filesystem correspond to pieces of kernel memory, and are used as interfaces between kernel code and user space. We use DebugFS to create a DebugFS "blob" file that points to the driver's ring buffer array. This file can then be periodically read from a normal Linux application.

C. Socket Server

The next building block of our tool is a socket server. While we collect state information from the wireless adapter, we want to let a potentially external client access this information. To collect state information from the driver, we use the DebugFS file. On the application side, we read the whole array from this file at an interval slightly less than it takes the driver to fill the ring buffer and begin overwriting. For instance, we use 200ms for the driver we described before, as it normally fills the DebugFS blob in 10 iterations (i.e. 250ms). This is to avoid missing data due to processing delays. Old items in the array can be identified via each record's TSF timestamp, allowing us to remove items that are read twice.

When read at this rate, the driver's array is large enough to keep up with the overwriting speed, however it is not large enough to keep a full history. To maintain a full history, the data is exported from the application using a TCP socket. In fact, the server application does not keep newly read items, instead it listens on a TCP socket for incoming connections.

[1] Available in Linux kernel version 2.6.10-rc3 and higher.
[2] mount -t debugfs none /sys/kernel/debug

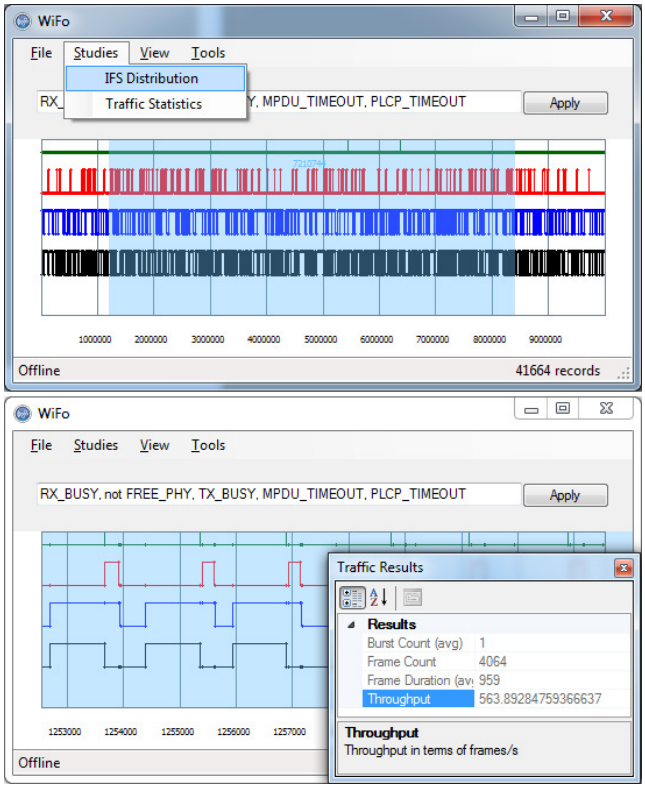


Fig. 3: Screenshots from the diagnostic tool's front-end (WiFo).

When a client connects, the server flushes the data to the client after each DebugFS read. Note that the system we described up to here, including the firmware, driver and server, can run on a low-spec Linux box.

D. Front-end

1) Main Graphical Interface: In the previous subsection we discussed how a small server relays a wireless card's internal state information to an external client. The client that we use to receive this data is a graphical client named WiFo [28], developed using the .NET framework. It connects to the server and displays the data received from it. Fig. 3 shows screenshots from this application.

The most basic feature of the UI allows you to graph arbitrary *Boolean expressions* of bits of the IFS Status register, where you may also benefit from predefined macros instead of remembering bit usages. This fascilitates easier understanding of the flags. The following expression is used in all screenshots in this paper (e.g. Fig. 8)

```
RX_BUSY, not FREE_PHY, TX_BUSY,
  MPDU_TIMEOUT, PLCP_TIMEOUT
```

which declares the five plots you see in the figures (plot definitions are separated by comma). Each plot can have a complicated Boolean expression of bits, or a simple bit query like above example. Live plots corresponding to selected expression are displayed on a chart called the *timeline*. By

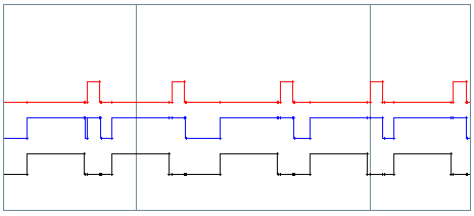


Fig. 4: Transmission of packets captured on WiFo. The bottom, middle and top curves show busy status for the RX engine, the medium (PHY) and the TX engine respectively.

choosing a suitable combination of bits we can identify frames, ACKs and other events on the channel. Fig. 4 shows an example timeline. The bits displayed are bit 9 (RX engine busy status, bottom, black), the complement of bit 1 (which corresponds to PHY busy status, middle, blue) and bit 8 (TX engine busy status, top, red). Therefore bumps on the bottom line represent frame transmissions. In this example, the monitoring code is running on the AP, and we see bit 8 is flipped on ACK or beacon transmissions. We see that after each black period we have a short period where the red line rises. The duration of this flip is the duration of an ACK, which confirms an ACK transmission.

Another feature of the diagnostic tool's front-end is the measurement of the time between two events (see top image of Fig. 3), which allows manual investigation of the duration of those events. It also allows the selection of a timeframe to perform automated studies. The application has access to the full history of the data from the time it connects to the server. Combined with zooming and panning functionality, these automated studies allow various statistics to be calculated over particular time periods. The amount of data processed is only limited by the memory on the front-end host. Other features include data export to persistent storage for later use. This data can later be reloaded and analyzed either by WiFo or using an external tool such as MATLAB.

From Fig. 4 we can observe a limitation of our current implementation. Looking at the middle line, we see a short idle SIFS period at the end of the first packet transmission. This is only seen for the first frame, and is missing for other frames. As we discussed in Section III-A, the firmware on the wireless chipset also performs its normal 802.11 operation; monitoring is an additional task. The card's normal operation requires it sometimes to halt until a certain even occurs, and in these cases the monitoring code may miss an event. Scanning the firmware code, we can identify relevant sections of code, and replace them with copies of the monitoring code. We believe this would alleviate the problem of short-duration missing events, however we could still miss events that are closer together than two reads of the IFS status register. For the studies we conduct in this paper, we have found that the simple modification of replacing the `nap` instruction is sufficient.

2) Additional Functionality: Additional functionality can be added to WiFo through a plugin system with APIs for both .NET Framework and Python. Placing a .NET class library or a Python script in WiFo's extensions directory will automatically activate it on start-up. Extension developers do

not need to worry about collection of data, as it is passed in an accessible data structure to an extension; this is a special-purpose enumerable list in .NET, and a list in Python. Each extension can have its own settings and output formats, which can be integrated into the user interface through the API.

There are currently two types of extensions. *Studies* are extensions that take a subset of the data identified by a time range and produce results. These results can be of any type and the API provides a flexible format to display results. The bottom image in Fig. 3 shows an example of results produced by a study. Results in the form of a plot can easily be displayed through the API. *Timeline view* extensions are replacements for the original timeline view. They have access to the graphics canvas of the timeline and the state information, and they can offer a different representation of the existing data, or combine it with external data sources to give more insight. For performance reasons the API for timeline views is currently only available for .NET Framework.

WiFo provides some default extensions, including packet recognition and an inter-frame space (IFS) distribution calculator. The former simply uses the state information to count packets in a given timeframe and provides statistics (e.g., bottom image in Fig. 3). The latter generates a bar plot for the distribution of the inter-frame space (e.g. Fig. 9). We will discuss these plots in detail in Section V. For example, recognizing successfully-received packets is performed by scanning through all records and looking for the following pattern:

1) RX engine becomes busy for longer than $192\mu s$
2) RX engine becomes idle
3) TX engine becomes busy after $10\mu s$
4) TX engine becomes idle (after an ACK duration)

This pattern represents the transmission and acknowledgement of a single frame if WiFo's back-end runs on the receiving access point. For other receive cases, the final two steps examine the RX bit. In practice, WiFo checks both cases. Note also that these two steps will also be absent if the frame is not acknowledged. The inter-frame space can be calculated as the time difference from one match of this pattern to the next. Starting from the first two, the time differences between the above steps correspond to frame duration, SIFS and ACK duration respectively.

E. Linking to PCAP Data

A useful complement to the PHY/MAC layer data provided by our monitoring system is the data provided by tools such as *tcpdump*. These tools monitor the medium through a network adapter and capture frames as they are observed. The type and a range of bytes from every frame are recorded and can be filtered and studied. Additional information made available by a NIC, such as PHY rate and power, can also be recorded via the *Radiotap*[29] extensions.

PCAP information and WiFo's raw data are complementary to each other. For example, our core system also falls short in studying packets as it backend only sees state changes. While this information is enough to determine the timing of packets, it cannot tell us the content of a packet, including source and destination addresses. *tcpdump* gives us this extra information. On the other hand, *tcpdump* only captures frames that are either transmitted successfully or at least key parts of which can be decoded. Our system does not have this limitation; it captures every activity on the channel, whether or not it is a successful frame. Even the noise from a microwave oven can be seen using our tool (see Section IV-C).

To benefit from both, we can integrate data from our monitoring system and *tcpdump* by reading data in the common PCAP format. Using PCAP API for .NET Framework[3], we develop a PCAP extension for the system. This extension displays PCAP information on the timeline, by aligning TSF and timestamp values. [4] This helps verify frame transmissions and have extra information to perform further analysis. Note that PCAP data used in our tool can come from any source and so a second wireless adapter, potentially with different capabilities, can be used. This could also be on the the monitoring host or on some other device.

IV. Validation

In this section we present a number of tests to demonstrate that our monitoring system and analysis tools are robust. In these basic tests we have one transmitting station, a Soekris net4801 with Broadcom BCM4318 NIC with the OpenFWWF firmware. The receiving station is the AP, a PC equipped with a BCM4318 adapter running the monitoring system described in Section III-A. The WiFo front-end runs on a separate PC. We send saturated traffic using MGEN [30] in all the following tests.

A. TX Duration

One of the fundamental aspects of the system is the timing of flag changes, the correctness of which is crucial to any application of the diagnostic tool. To this end, we verify the effect of varying frame sizes on the observed duration of their transmission. We run a series of tests with different payload sizes, from 100 bytes to 1400 bytes (with granularity of 100 bytes). The diagnostic tool measures the duration of frames by using the pattern described in Section III-D and measuring the time the RX engine remains busy for that frame.

We use saturated UDP traffic at $11Mb/s$ for all tests, and for each payload size we average transmission duration over 2500 frames. We use long preamble in these tests ($192\mu s$ PLCP). We also calculate the expected duration $D(l)$ for each payload size l as

$$D(l) = d_{PLCP} + d_f(l). \quad (1)$$

where $d_{PLCP} = 192\mu s$ is the duration of the preamble and the PLCP header and $d_f(l)$ is the time required to transmit the payload and protocol headers associated with different layers, which is calculated as

$$d_f(l) = \frac{8(l + l_{H,LLC} + l_{H,IP} + l_{H,UDP} + l_{H,802.11} + l_{FCS})}{r}.$$

[3] SharpPCAP/PacketDotNet

[4] BCM43xx chipset internally uses a TSF timer that is never synchronized with the network. Instead, a register keeps the difference between the internal and network TSF, the value of which is updated each time a beacon is received. We use this value to align signal records, which hold the internal TSF, with TSF values from PCAP data.

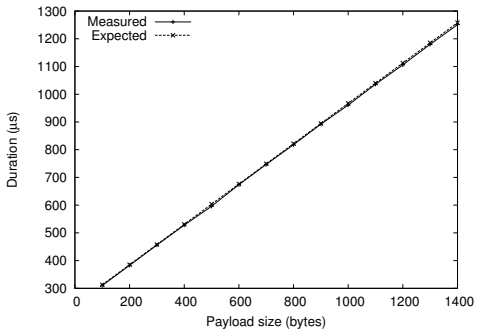


Fig. 5: Duration with increasing payload size.

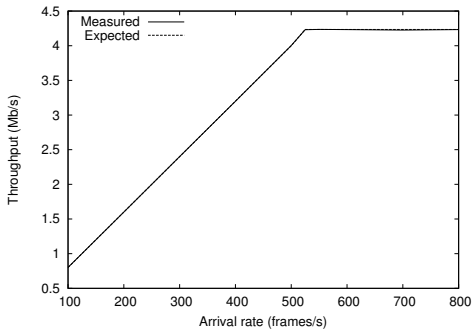


Fig. 6: Throughput with increasing packet-arrival rate.

Here $l_{H,x}$ is the length of the header associated with layer x, l_{FCS} is the length of the FCS, all in bytes, and r is the data rate used to transmit the frame. For example, for a payload size of 1000 and with our experiment settings, $d_f(1000) = \frac{8(1000+4+20+8+30+4)}{11\times10^6} \approx 7.75 \times 10^{-4}s$. Further, this translates to the following for $D(l)$ in microseconds:

$$D(l) = 240 + 0.727l. \quad (2)$$

Fig. 5 shows average durations as observed by the AP as well as the calculated expected duration for each payload size. In fact, the measured packet lengths are tightly clustered around the mean, with variations of only a single microsecond. As shown in the figure, the expected and observed values closely match. Fitting a straight line through measured data gives us the following for $D(l)$:

$$238.055 + 0.725l.$$

which is very close to (2). Slight changes are expected given firmware delays. This verifies the system's pattern matching capabilities and the timeliness of flag changes.

B. Throughput

Throughput is an important metric for 802.11 networks, as it can be an indicator of different network aspects such as performance and fairness. For this reason, it is important for our diagnostic tool to identify all transmitted frames and to measure network throughput correctly.

To validate the diagnostic tool's throughput calculation, we run a fresh test on the same network as previously described, but this time we do not use saturated traffic. Instead, we run a single UDP flow with PHY data rate of $11Mb/s$ with payload size of 1400 bytes for each frame for 30 seconds, and have the tool calculate the throughput. We use a arrival rate of 100 packets/s for the first test, and for each subsequent test we increase the arrival rate, up to 800 packets/s.

The expected throughput is calculated simply by multiplying the transmission rate by the payload size. However, rate should not exceed the saturation throughput. We calculate saturation throughput S as

$$S = \frac{l}{D(l) + D_{SIFS} + D_{ACK} + D_{DIFS} + D_{bo}} = \frac{l}{D_t}, \quad (3)$$

where $D(l)$ comes from (1), D_{ACK} is the duration of the ACK, D_{bo} is the average backoff duration, and D_{DIFS} is the duration of DIFS ($50\mu s$). D_t is used to denote the total duration of a successful frame transmission, including its ACK and average backoff. We will be using this notation later. Also, note that in our experiments we use a fixed packet size l and hence $D(l)$ is a constant.

Fig. 6 depicts the observed and expected throughput values. As you can see, the two values match closely; and the small difference after saturation may be explained by the backoff behavior of the chipset as we will discuss in Section V. The reason this difference does not exist before saturation is that packet inter-arrival times are usually larger than the maximum backoff, eliminating the effect of the backoff mechanism.

C. Microwave Interference

In this section we show that we can see microwave interference using our tool. To demonstrate this, we test two scenarios on a channel where only the AP transmits beacons and there are no other stations or any interference from other WiFi networks. In our first test we use a microwave oven to introduce interference, and for the second test we turn it off. Note, the interference generated by the oven is bursty. The top image in Fig. 7 shows a single microwave oven burst captured by a spectrum analyzer. The duration of this pattern is approximately $8ms$ and it is repeated at intervals of about $20ms$ (see bottom Fig. 7), which is related to the mains frequency of 50Hz.

Fig. 8 shows what we observe using WiFo for both tests. The bottom plot is of a free channel with only beacons, which show as periodic spikes. Note that as the monitoring runs on AP the TX engine remains busy during the transmission of a beacon. The top plot is taken with running microwave oven. There are periodic spikes on the RX engine's activity, the duration of which is around $140\mu s$. We believe this is the time required by the decoder to distinguish noise from WiFi signal. The RX spikes are separated by two slightly different distances which alternate. The shorter distance is $8ms$, which is the burst size. This suggests that the beginning and the end of each burst triggers the chipset's decoder, which soon identifies it as noise and the decoder is deactivated. More generally, any interference on the channel triggers the decoder temporarily.

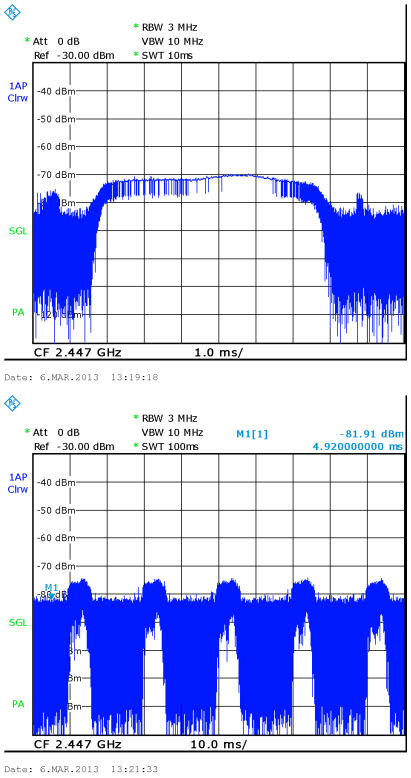


Fig. 7: Waveform of a single microwave oven burst (top), and a sequence of bursts (bottom).

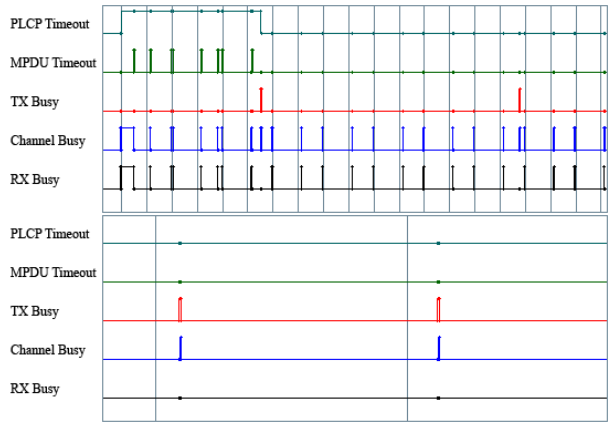


Fig. 8: Channel activity observed by AP running WiFo's backend when a microwave oven is working (top) and when it is not (bottom). Vertical lines are $10ms$ apart in both images.

CW_{min}	χ^2	p-value
32	26.3006	0.70680
16	16.2286	0.36702
8	5.6899	0.57639

TABLE II: Chi-squared test for results in Fig. 9. We use $CW_{min} - 1$ degrees of freedom to obtain the p-value.

V. Debugging Examples

In this section we introduce examples showing how our diagnostic tool can help debug wireless hardware and drivers. A motivation for the development of this tool was to help examine wireless networks closely and to identify reasons for unexpected behavior. This is especially useful for driver and firmware developers, as it could help them verify the behaviour of the wireless cards and debug their implementations. We believe this can also be a practical research tool allowing visualization of information that is otherwise hard to obtain.

A. Contention Window

The Distributed Coordination Function (DCF) is one of the most important parts of the 802.11 standard, as it ensures equal opportunities for all the stations to use the channel. The contention window, used for backoff in DCF, is sometimes misconfigured by wireless card vendors [11], and may also be manually adjusted by users to gain an advantage. This makes the ability to detect possible misconfiguration important. In this section we manually adjust the contention windows of the wireless adapters to standard and non-standard values and then use our system to observe the differences. Using the IFS distribution study, described in Section III-D, we may generate a distribution graph and detect misconfigured stations.

In our next experiments, we use the same network setup that we used in Section IV and we send saturated UDP traffic using packets with 1000 bytes of payload, and we alter the contention window for each experiment. The duration of each experiment is again 30 seconds. Fig. 9 shows the resulting plots for three different values of the minimum contention window, namely 8, 16 and 32. We can see the number of times each backoff value is selected and the range of values in use.

Note that, as we use 802.11 channel 14 for our experiments and we have almost no contention or interference, the station almost never moves on to the second backoff stage. Noting that observed values for the same backoff value are spread by $1\mu s$ apart as a result of firmware delays, so we bin the results into $20\mu s$ bins to get backoff values. Plotting the raw backoff times shows isolated spikes, rather than values spread throughout the backoff interval.

Using these plots we can easily distinguish the CW_{min} value in use. They can also help us see how evenly the backoff is chosen. As this value should be chosen completely at random, we expect, on average, a flat distribution graph. Table II shows the chi-squared test values for these results, comparing them to uniform distribution (the null hypothesis is that the results are uniform). As the table shows, p-values are too large, so it seems unlikely that the backoffs are truly uniform. This may be due to the way the random number generator works on the device.

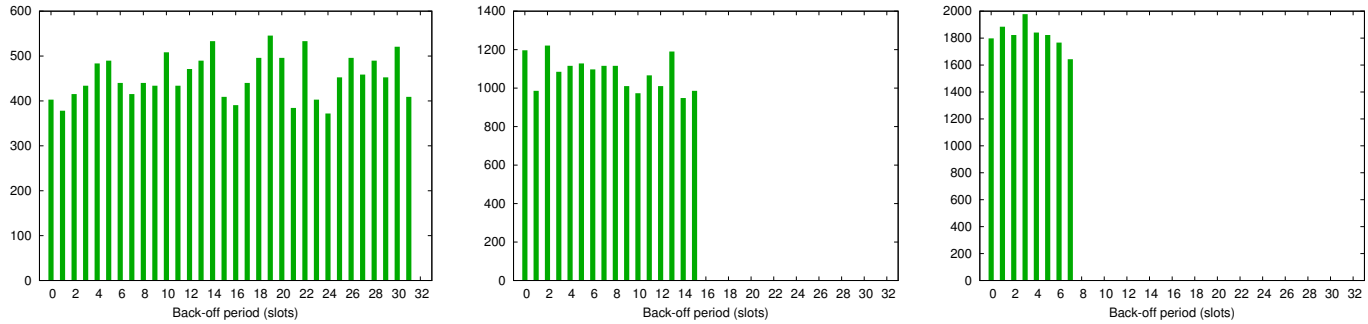


Fig. 9: Backoff distribution for (left) $CW_{min} = 32$, (center) $CW_{min} = 16$, and (right) $CW_{min} = 8$ (BCM4318, b43).

Standard-compliance is not solely the role of hardware and firmware, and differences could exist at the driver level. In the next study, we use an Atheros AR5001X+ adapter for our station and compare the backoff behavior when using different drivers. One of these drivers is ath5k[5], which is a reliable driver for Atheros cards, and the other one is MadWiFi[6].

Both drivers provide a similar average throughput, from which one might guess they both present a similar back-off distribution. However, our observations prove otherwise. Fig. 11 shows the results obtained using the diagnostic tool. As you can see in the figure, the ath5k driver hops between the two ends of the contention window rather than a uniform distribution over the whole window. According to [32], this does not give the station any advantage in the long run, as the average backoff is unchanged. Nevertheless, it is an obvious deviation from the standard and it may affect certain experiments by changing the collision probability, especially when more than one station behaves this way.

B. TXOP Burst

TXOP is the time period during which a station may send as many frames as fit in the TXOP duration [10]. Although the duration can be advertised by the AP, there is no enforcement of the value used. Once a station wins a contention, it could send frames indefinitely, resulting in poor performance of other contending stations. In this section we increase the TXOP period used by a station and use WiFo to count the number of packets that come in a burst. As before, the network has one station connected to an AP equipped running the monitor code. The station is equipped with an Atheros card with the MadWiFi driver, and it transmits saturated traffic using frames with payload 500 bytes for 5 seconds in each test.

Fig. 10 shows that, as long as the TXOP period is smaller than the duration of a frame, the burst contains only one frame. Each time a new frame can fit in the given period, the burst size increases. In other words, we can calculate burst size as

$$n = \left\lceil \frac{t_{TXOP}}{D_t} \right\rceil,$$

where t_{TXOP} is the TXOP time, and D_t comes from (3). The duration of a single frame transmission and ACK is the distance between two steps in the graph, which we measure as $863\mu s$. The expected duration is calculated as $D(l)+D_{SIFS}+D_{ACK}$. As our AP uses the basic rate of 2Mb/s, this adds up to $862\mu s$, closely matching our measured value.

C. ACK Skipping

Acknowledgements are normally used as a success signal for the transmitter. However, deliberately skipping ACKs can sometimes be desired, e.g. [33], [34], [35], [36]. In this section we implement a simple scheme at the AP: we skip every other ACK for received frames, forcing stations to always make two attempts for each frame. We use WiFo to sanity check our implementation. For the experiment, we use one station connected to the AP, and send saturated traffic at 11Mb/s for 30 seconds using MGEN. Both the station and the AP use Broadcom BCM4318 wireless adapters with OpenFWWF. The station uses minimum CW of 32. By dropping the first ACK, we force it to double this value, and use 64. Fig. 12 shows the resulting backoff distribution graph calculated by WiFo. For values in the range $[0, 32)$, the numbers are almost twice as much as $[32, 64)$, which is exactly what we expect; remember that, on the second backoff stage, the station uniformly selects a backoff time within the range of $[0, 64)$ slots. This not only proves that the implementation works, but also demonstrates another aspect of WiFo's usefulness.

VI. Conclusion

In this paper we have designed and implemented an extensible diagnostic tool for 802.11 wireless cards which can be used to test aspects of the protocol and standards compliance. The purpose of this tool is to give programmers and researchers enough flexibility to test and debug wireless cards and drivers. The diagnostic tool is made using only commercial off-the-shelf devices and offers an alternative to more expensive tools. With an API to add new features to the application, the diagnostic tool can be programmed to do sophisticated analyzes on the data. We gave PCAP integration as an example of additional features that can be added as plugins. We presented results to validate the system, and some experimental results that highlighted use cases of this tool.

[5]Our ath5k driver is a slightly modified version of the one included in compat-wireless[31] 2.6, to enable us to change contention parameters. However, we do not use this feature for these tests.

[6]The MadWiFi version we use is 0.9.4-r4173, and the only modification made to the driver is disabling QoS.

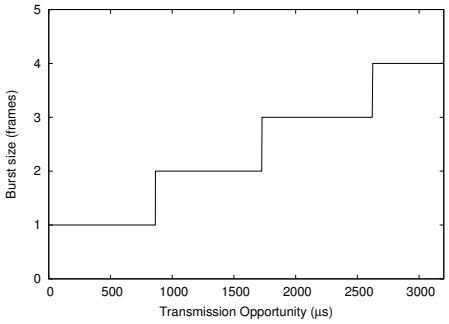
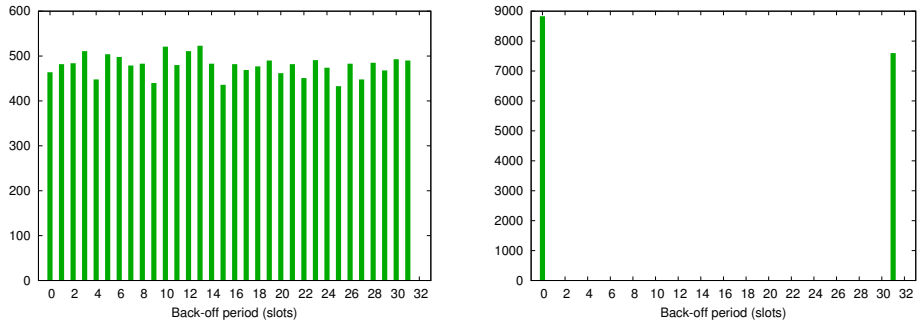


Fig. 10: The number of frames in a burst with increasing TXOP (ath5k).



(a) MadWiFi (b) ath5k

Fig. 11: Backoff distribution for Atheros chipset.

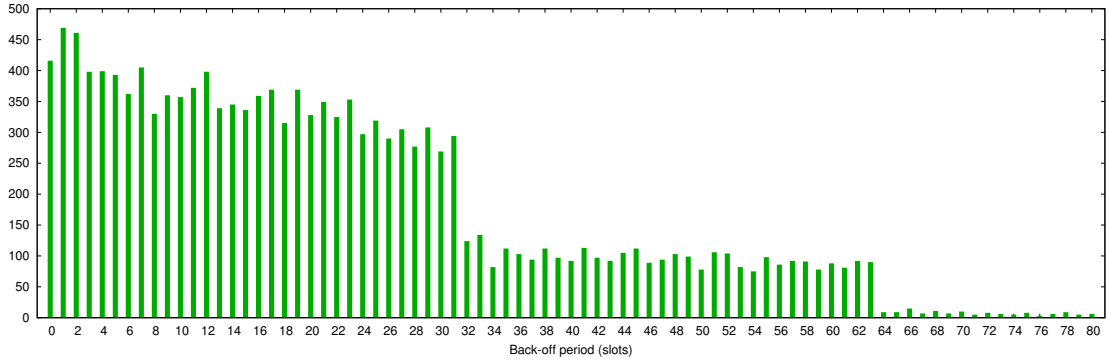


Fig. 12: Backoff distribution when the AP drops every other ACK.

ACKNOWLEDGMENTS

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 13/RC/2077.

REFERENCES

[1] M. Fang, D. Malone, K. Duffy, and D. Leith, "Decentralised Learning MACs for Collision-free Access in WLANs," *Wireless networks*, vol. 19, no. 1, pp. 83–98, 2013.

[2] J. Barceló, B. Bellalta, C. Cano, and M. Oliver, "Learning-BEB: Avoiding Collisions in WLAN," in *Eunice Summer School*, 2008.

[3] L. Sanabria-Russo, J. Barceló, and B. Bellalta, "Prototyping Distributed Collision-Free MAC Protocols for WLANs in Real Hardware," in *MACOM* (M. Jonsson, A. V. Vinel, B. Bellalta, N. Marina, D. Dimitrova, and D. Fiems, eds.), vol. 8310 of *Lecture Notes in Computer Science*, pp. 82–87, Springer, 2013.

[4] Y. Erten, "A layered security architecture for corporate 802.11 wireless networks," in *Wireless Telecommunications Symposium, 2004*, pp. 123–128, May 2004.

[5] P. Patras, H. Qi, and D. Malone, "Mitigating collisions through power-hopping to improve 802.11 performance," *Pervasive and Mobile Computing*, 2013.

[6] Z. Dong and T. Lai, "A scalable and adaptive clock synchronization protocol for IEEE 802.11-based multihop ad hoc networks," in *Mobile Adhoc and Sensor Systems Conference, 2005. IEEE International Conference on*, pp. 8 pp.–558, Nov 2005.

[7] J. Tang and Y. Cheng and W. Zhuang, "An analytical approach to real-time misbehavior detection in IEEE 802.11 based wireless networks," in *INFOCOM, 2011 Proceedings IEEE*, pp. 1638–1646, 2011.

[8] D. Shukla, L. Chandran-Wadia, and S. Iyer, "Mitigating the exposed node problem in IEEE 802.11 ad hoc networks," in *Computer Communications and Networks, 2003. ICCCN 2003. Proceedings. The 12th International Conference on*, pp. 157–162, 2003.

[9] *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.

[10] *IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirement. Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Amendment 2: Higher-Speed Physical Layer (PHY) Extension in the 2.4 GHz Band - Corrigendum 1*, 2001.

[11] G. Bianchi, A. D. Stefano, C. Giaconia, L. Scalia, G. Terrazzino, and I. Tinnirello, "Experimental assessment of the backoff behavior of commercial IEEE 802.11b network cards.," in *INFOCOM*, pp. 1181–1189, IEEE, 2007.

[12] F. Gringoli and L. Nava, "Open Firmware for WiFi Networks." http://www.ing.unibs.it/~openfwwf/.

[13] A. Cárdenas, S. Radosavac, and J. Baras, "Evaluation of Detection Algorithms for MAC Layer Misbehavior: Theory and Experiments," 2008.

[14] M. Raya, I. Aad, J.-P. Hubaux, and A. E. Fawal, "DOMINO: Detecting MAC Layer Greedy Behavior in IEEE 802.11 Hotspots," *Mobile Computing, IEEE Transactions on*, vol. 5, no. 12, pp. 1691–1705, 2006.

[15] A. Cárdenas, S. Radosavac, and J. Baras, "Detection and prevention of MAC layer misbehavior in ad hoc networks," in *Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, SASN '04, (New York, NY, USA), pp. 17–22, ACM, 2004.

[16] K. Huang, K. Duffy, and D. Malone, "H-RCA: 802.11 Collision-Aware Rate Control," *Networking, IEEE/ACM Transactions on*, vol. PP, no. 99, pp. 1–1, 2012.

[17] M. Lacage, M. Manshaei, and T. Turletti, "IEEE 802.11 rate adaptation: a practical approach," in *Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, MSWiM '04, (New York, NY, USA), pp. 126–134, ACM, 2004.

[18] B. Raman and K. Chebrolu, "Design and evaluation of a new MAC protocol for long-distance 802.11 mesh networks," in *Proceedings of the 11th Annual International Conference on Mobile Computing and Networking*, MobiCom '05, (New York, NY, USA), pp. 156–169, ACM, 2005.

[19] J. Yeo, M. Youssef, and A. Agrawala, "A framework for wireless LAN monitoring and its applications," in *Proceedings of the 3rd ACM Workshop on Wireless Security*, WiSe'04, (New York, NY, USA), pp. 70–79, ACM, 2004.

[20] J.-Y. Yoo, T. Heuhn, and J. Kim, "Active capture of wireless traces: Overcome the lack in protocol analysis," in *Proceedings of the Third ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, WiNTECH '08, (New York, NY, USA), pp. 41–48, ACM, 2008.

[21] H. Kim, C. Cordeiro, K. Challapali, and K. Shin, "An Experimental Approach to Spectrum Sensing in Cognitive Radio Networks with Off-the-Shelf IEEE 802.11 Devices," in *IEEE Workshop on Cognitive Radio Networks, in conjunction with IEEE CCNC*, 2007.

[22] S. Rayanchu, A. Patro, and S. Banerjee, "Airshark: detecting non-WiFi RF devices using commodity WiFi hardware," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pp. 137–154, ACM, 2011.

[23] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, and F. Gringoli, "Wireless MAC processors: Programming MAC protocols on commodity Hardware.," in *INFOCOM* (A. Greenberg and K. Sohraby, eds.), pp. 1269–1277, IEEE, 2012.

[24] G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, and I. Tinnirello, "MAClets: active MAC protocols over hard-coded devices," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, (New York, NY, USA), pp. 229–240, ACM, 2012.

[25] Soekris Engineering. http://www.soekris.com/.

[26] "WiFo back-end source code." https://github.com/hessan/wifoserver.

[27] "Broadcom BCM43xx Specification." http://bcm-v4.sipsolutions.net/Specification.

[28] "WiFo front-end source code." https://github.com/hessan/wifo.

[29] "Radiotap." http://www.radiotap.org/.

[30] "Multi-Generator (MGEN)." http://www.nrl.navy.mil/itd/ncs/products/mgen.

[31] "Existing Linux Wireless drivers." http://wireless.kernel.org/en/users/Drivers.

[32] A. Kumar, E. Altman, D. Miorandi, and M. Goyal, "New Insights From a Fixed-Point Analysis of Single Cell IEEE 802.11 WLANs," *Networking, IEEE/ACM Transactions on*, vol. 15, pp. 588–601, June 2007.

[33] L. Vollero and G. Iannello, "Frame dropping: A QoS mechanism for multimedia communications in WiFi hot spots," in *Parallel Processing Workshops, 2004. ICPP 2004 Workshops. Proceedings. 2004 International Conference on*, pp. 54–59, 2004.

[34] L. Vollero, A. Banchs, and G. Iannello, "ACKS: a technique to reduce the impact of legacy stations in 802.11e EDCA WLANs," *Communications Letters, IEEE*, vol. 9, no. 4, pp. 346–348, 2005.

[35] H. Feghhi, P. Patras, and D. Malone, "Practical node policing in 802.11WLANs," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE 14th International Symposium and Workshops on a*, pp. 1–3, June 2013.

[36] P. Patras, H. Feghhi, D. Malone, and D. Leith, "Policing 802.11 MAC misbehaviour," *arXiv preprint arXiv:1311.5014*, 2013.