Reliable WiFi Backscatter Communication in WiTAG

by

Manoj Adhikari

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Manoj Adhikari is the sole author of all chapters except Chapter 2. All chapters in this thesis were written under the supervision of Dr. Tim Brecht. Research presented in portions of Chapter 2 is the exception to sole authorship and is based on text from the SIGCOMM paper on WiTAG [5]. The SIGCOMM paper was published by students and professors at the University of Waterloo (Dr. Ali Abedi, Farzan Dehbashi, Mohammad Hossein Mazaheri, Dr. Omid Abari, and Dr. Tim Brecht).

Abstract

WiFi backscatter systems offer the potential to provide low-powered WiFi-compatible communication. This technology is especially promising when coupled with low-power sensors to periodically communicate readings from IoT devices. WiTAG is an extremely attractive approach because it greatly reduces power consumption by avoiding the use of WiFi receivers or signal detectors while ensuring compatibility with existing WiFi infrastructure. WiTAG operates at the MAC layer by corrupting or not corrupting subframes (MPDUs) within a transmitted frame (A-MPDU). For example, corruption of an MPDU signals a 0 and non-corruption signals a 1. Because it eschews the use of receivers and signal detectors WiTAG is unable to sense when frames are being sent by nearby WiFi devices that it relies on for communication.

In this thesis, we describe the significant challenges that arise when formulating, transmitting, and reliably detecting and decoding messages transmitted from WiTAG. We design a message encoding framework to overcome these challenges. We show that although WiTAG relies on probabilities for overlapping a tag's message with an A-MPDU it is possible to increase the odds of an overlap, thus increasing message rates. This permits the transmission of highly reliable messages in a relatively short period of time.

Acknowledgments

I would like to thank my supervisor, Professor Tim Brecht, for his guidance and support throughout my degree. His supervision was instrumental in navigating the hurdles I faced during this project, and his inspiration kept me motivated.

I would like to thank Professor Ali Abedi for providing his technical expertise to solve experimental challenges and draw insights from the experimental results.

I would like to thank my colleague Kamran Nishat for his help with setting up and running experiments. I would also like to thank Charlie Liu for his help in measuring and improving the tag's code efficiency.

I would like to thank Professor Mina Tahmasbi Arashloo and Professor Samer Al-Kiswany for reviewing my thesis and providing feedback.

I want to dedicate this thesis to my father (Tulsi Ram Adhikari), my mother (Bishnu Adhikari), and my sister (Menuka Adhikari), who have always encouraged me to seek knowledge and consistently supported me throughout my educational journey.

Table of Contents

Author's Declaration ii

Statement of Contributions iii

Abstract iv

Acknowledgments v

List of Figures x

List of Tables xii

1 Introduction 1

1.1 Background 1

1.2 Overview of Communication 2

1.3 Motivation and Goals 4

1.4 Contributions 4

1.5 Thesis Outline 5

2 Background 7

2.1 802.11 Frame Aggregation 7

2.2 Channel Estimation and Compensation 8

2.3 Targeting A-MPDUs . . . 8
2.4 Software Retries . . . 9
2.5 WiTAG Implementation . . . 10
2.5.1 Changing the Wireless Channel . . . 10

3 Reliable Communication: Problems and Challenges **12**
3.1 Coarse-Grained Synchronization . . . 12
3.2 Fine-Grained Synchronization . . . 14
3.3 Bit Encoding . . . 15
3.4 Message Encoding . . . 17
3.5 Block ACKs Filtering . . . 18
3.6 Message Decoding . . . 19
3.7 Message Reliability . . . 19

4 Design and Implementation **20**
4.1 Complete A-MPDU Overlap . . . 20
4.1.1 Computing Tag Message Rates . . . 23
4.1.2 Weighted Probability of Overlap for Expected Rates . . . 24
4.1.3 MPDU Duration . . . 25
4.2 MPDU Overlap . . . 26
4.3 Bit Encoding . . . 26
4.3.1 Issues with Overlap Probability . . . 27
4.4 Message Encoding . . . 27
4.5 Block ACK Filtering . . . 27
4.6 Message Decoding . . . 28
4.7 Message Reliability: Using N Messages . . . 29

5 Experimental Setup **30**
5.1 Device Preparation . . . 30
5.1.1 Synchronization for WiTAG Evaluation . . . 31
5.2 Experiment Execution . . . 32
5.3 Data Processing . . . 33

6 Performance Evaluation **34**
6.1 Fine-Grained Synchronization . . . 34
6.1.1 Implementation Issue: Fine-Grained synchronization . . . 34
6.2 Coarse-Grained Synchronization . . . 35
6.2.1 Block ACK Filtering . . . 35
6.2.2 Implementation Issue: Coarse-Grained Synchronization . . . 35
6.2.3 Methodology: Impact of Time Delay . . . 36
6.2.4 Experimental Results: Impact of Time Delay . . . 36
6.2.5 Message Length, Time Delay, and Message Rate Relationships . . . 38
6.2.6 Relationships Between Message, Time delay and Message Rate . . . 40
6.2.7 Choosing a Time Delay Impacts Synchronization . . . 42
6.3 Bit Encoding . . . 44
6.3.1 Methodology: Different Values for Bit Encoding . . . 44
6.3.2 Experimental Results: Different Values for Bit Encoding . . . 45
6.4 Message Decoding . . . 45
6.4.1 Methodology: Levenshtein Distance Algorithm . . . 46
6.4.2 Experimental Results: Levenshtein Distance Algorithm . . . 46
6.5 Message Reliability . . . 48
6.5.1 Methodology: Using Multiple Messages . . . 48
6.5.2 Experimental Results: Using Multiple Messages . . . 48

7 Related Work **51**

7.1 Bit Encoding . . . 52

7.2 Message Encoding . . . 52

7.3 Decoding . . . 54

7.4 Consensus . . . 55

8 Conclusions and Future Work **57**

8.1 Thesis Summary . . . 57

8.2 Future Work . . . 59

8.2.1 Using a Neural Network for Larger Messages . . . 60

8.2.2 Message Encoding to Improve Reliability . . . 60

8.2.3 Antenna Enhancement to Improve Distance . . . 60

8.2.4 Supporting Multiple Tags . . . 61

8.2.5 Quantify Power Savings . . . 61

8.3 Concluding Remarks . . . 62

References **63**

List of Figures

1.1 Communication overview 2

2.1 802.11n/ac/ax A-MPDU structure [5] 8

2.2 Corrupting subframes. Block ACKs of four A-MPDUs (with 16 subframes) when a) the tag is doing nothing, and b) the tag periodically changes the channel between two states [5] 11

3.1 Coarse-grained synchronization 13

3.2 Fine-grained synchronization 15

3.3 Encoding one bit using multiple MPDUs 16

3.4 Example block ACKS, showing the addition of a preamble and postamble (delimiters) to the tag's message. Orange bits denote preamble and postamble, blue and green are used to denote alternating bits of actual data, and purple denote leading and trailing 1's (added to detect if overlap is complete or not) 18

4.1 An A-MPDU and associated overhead. The A-MPDU has overhead before and after its transmission 22

4.2 Probability of complete A-MPDU overlap for different message lengths based on three different MPDU durations (assuming that all A-MPDUs contain 64 MPDUs) 22

4.3 Expected tag message rates for different message lengths using three different time delays (assuming that all A-MPDUs contain 64 MPDUs) 23

6.1 Expected and actual message rates for time delays ranging from 1.1 to 6.2 ms (message length is 32) 37

6.2 Expected and actual message rates for time delays ranging from 1.1 to 5.1 ms, repeated three times (message length is 32) 38
6.3 Expected and actual message rates when YouTube video traffic is present (message length is 32) 39
6.4 Expected and actual message rates with 95% confidence intervals for time delays ranging from 1.1 to 6.2 ms (message length is 16) 40
6.5 95 % confidence intervals for expected and actual message rates for 32 patterns using two different time delays 41
6.6 Good synchronization because of a well-chosen time delay 43
6.7 Bad synchronization because of a poorly chosen time delay 43
6.8 Decoding accuracy using the Levenshtein distance algorithm for 32 different values. Note that the y-axis covers values from 90 to 100 (zoomed in) to allow us to see the differences. The accuracy is above 90% for all values in all experiments. 47

List of Tables

6.1 Experimental results showing the worst-case decoding accuracy for $b = 5$ using different m values at 0.3 m and 1.5 m distances. Message rates are obtained using a 1.8 ms time delay 46

6.2 Improving reliability with multiple messages 49

6.3 Time to send different numbers of messages based on different message rates 49

Chapter 1

Introduction

1.1 Background

Backscatter communication modifies and reflects existing radio frequency signals to encode data, rather than generating new signals. This technique allows wireless nodes (e.g., IoT devices) to communicate without requiring active radio frequency (RF) components on the tag [9]. This approach enables low-power communication because utilizing existing signals generated by other devices doesn't require as much power as generating them. WiTAG is a backscatter communication system that is compatible with existing, modern WiFi technologies (e.g., 802.11n, ac, and ax). To further reduce power consumption, WiTAG eschews power hungry receivers and packet detection circuitry, instead relying on the ability to probabilistically overlap the tag's communication with transmitted WiFi packets [6, 5].

To enable practical systems to be built, recent advances in WiFi backscatter communication are designed to be compatible with existing WiFi networks without requiring hardware or software modifications. In contrast with prior methods that utilize the physical layer for backscatter communication, WiTAG leverages features of the MAC layer to communicate. For example, WiTAG is designed to send data by selectively interfering with subframes (MPDUs) in an aggregated frame (A-MPDU). Data is packaged in the data-link layer in the form of MPDUs (MAC protocol data unit). A feature of the 802.11n and subsequent WiFi standards is that multiple MPDUs can be aggregated into one single frame (an A-MPDU). This aggregation improves network efficiency by minimizing the overhead required when compared with sending each MPDU separately. One key example of such an improvement (for this thesis) is the use of a single acknowledgment (block ACK) to indicate which of the MPDUs in an A-MPDU have and have not been received correctly.

Overheads are significantly reduced because up to 64 MPDUs can be sent at once (in 802.11n and ac networks) and acknowledged with one 64-bit block ACK.

In WiTAG, a WiFi device transmits "query" packets that are transmitted for the purpose of allowing a tag to communicate. The query packets use a PHY rate that is chosen to ensure successful transmission. During regular communication (i.e., when a WiTAG device is not attempting to communicate) all MPDUs would be received and the block ACK would contain all 1's. The tag then uses corruption of an MPDU to signal a 0 and non-corruption to signal a 1. As a result, the block ACK for the query packet will contain the message from the tag. This enables standard compliant communication using modern, open or encrypted 802.11n, 802.11ac and 802.11ax networks.

1.2 Overview of Communication

One of the use cases for WiTAG communication is in IoT devices where there is one-way communication from the IoT device (e.g., sensor readings) to a WiFi device. Figure 1.1 shows an overview of how a querying device attempts to obtain data a tag is communicating. In this example, the tag is attached to a temperature sensor and it is sending the sensor reading of 23. We show an A-MPDU with 7 MPDUs. The steps for communication are as follows:

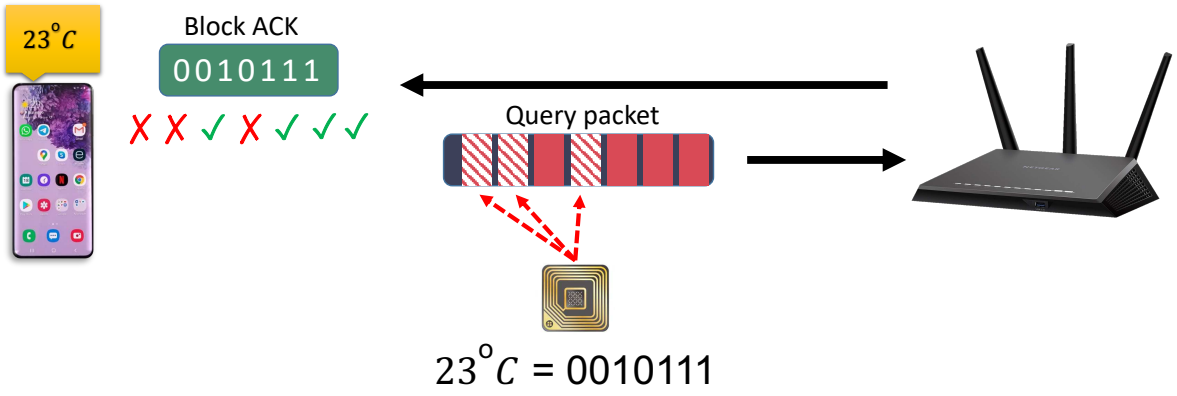


Figure 1.1: Communication overview

1. The querying device (in this example, the smartphone on the left) sends a query packet (A-MPDU) to the WiFi access point.

2. While the query packet is in flight the tag attempts to corrupt specific MPDUs within an A-MPDU. In this example it targets the fourth, sixth and seventh MPDUs, to indicate the 0's in the value being transmitted. The tag cannot distinguish packets from the querying device from other traffic in the environment. As a result, there may be some impact on other devices. However, the degree to which other devices are affected depends on the values being sent by the tag (more 0's means more retransmissions), the frequency at which the tag needs to send its data and the location of the tag relative to the other WiFi devices. Dehbashi et al. provide more details on the potential impact [11].

3. The access point receives the A-MPDU, attempts to decode the MPDUs and, as dictated by the WiFi protocol, generates a block ACK to indicate which MPDUs it was able to decode (denoted with 1's) and which MPDUs it wasn't able to decode (denoted with 0's). It then transmits the block ACK to the querying device.

4. The querying device receives the block ACK and uses the bits in the block ACK to decode the message the tag transmitted.

If the timing of the tag's corruptions correspond with the time that an A-MPDU is in flight and it is able to corrupt the desired MPDUs at precisely the correct times, the querying device will receive the tag's message. Because WiTAG avoids using power-hungry signal detectors, it is not able to sense when an A-MPDU is being sent. This presents several challenges that are examined in this thesis.

To increase the probability of a tag's transmission overlapping with an A-MPDU one would want to transmit A-MPDUs as rapidly as possible. Likewise, if the tag repeatedly transmits its data for some period of time the odds of overlapping increase. However, the tag's transmissions should only overlap the message once in each A-MPDU, because if the same message overlaps twice with the same A-MPDU, decoding the message becomes significantly more difficult. It is also important that the tag attempts to avoid overlapping with A-MPDUs that are the result of a software retry, which we describe in Section 2.4.

If the tag tries to communicate while the querying device is transmitting, that device will receive the tag's data. However, if the tag happens to communicate when another (non-querying) WiFi device is transmitting, that device may have to retransmit MPDUs that were corrupted. The retransmission occurs naturally as part of the 802.11 protocol. The impact on the non-querying devices will be relatively small, especially because IoT devices will typically communicate periodically. Dehbashi et al. [11] examine the impact of tag on neighboring WiFi networks and discover that the throughput drops by less than

30 percent when a tag is always active (and tags are expected to only periodically send data from IoT devices).

1.3 Motivation and Goals

Previous work has focused on the hardware design and demonstrating the viability of WiTAG [6, 5, 11]. The motivation of this thesis is to focus on the software components of the system and in particular, the design and implementation of a protocol for reliable communication.

The combination of not being able to detect WiFi packets and the utilization of corruption of MPDUs presents several challenges and design decisions that need to be studied and resolved to reliably send messages from WiTAG. The goals of this thesis are to:

- Identify and describe the challenges faced when trying to implement reliable WiFi backscatter communication in a system without a receiver or a signal detector.
- Design, implement, and evaluate possible solutions, to better understand pros and cons of various approaches.
- Develop a framework for the empirical evaluation of possible solutions. The framework should enable data collection for a variety of different WiTAG messages without frequent manual intervention.
- Implement and evaluate the identified design choices using our framework and incorporate the findings to enable low-power, reliable communication in WiTAG.

To effectively utilize WiTAG for message transmission, we must address the current challenges. By gaining a deeper understanding into these obstacles and creating a robust communication framework, we can ensure WiTAG's practicality in real-world scenarios.

1.4 Contributions

In this thesis we study: (1) The probability that, without any coordination between WiFi devices and a tag, the tag is able to send a message while a WiFi packet is in flight (coarse-grained synchronization). (2) The issues that arise while trying to selectively corrupt or

not corrupt a specific MPDU (fine-grained synchronization). (3) How to design and encode a tag's message. (4) How to reliably identify and decode a tag's message. (5) The tradeoffs in these design decisions and how reliability can be improved by using multiple messages. We make the following contributions in this thesis:

- We describe several problems that need to be overcome in order to reliably transmit data from WiFi backscatter devices that do not use a receiver or signal detector and describe and evaluate possible solutions.
- We devise a model for determining the probability of overlapping the message from a WiFi backscatter device (a tag) with a transmitted WiFi packet and study the impact that important parameters have on the probability of a complete (successful) overlap.
- Interestingly, we demonstrate that it is possible to increase the probability of an overlap which enables increased message rates.
- We evaluate various parameters for bit encoding and message encoding to determine a mechanism that balances between minimizing message length and obtaining good reliability.
- We evaluate the use of the Levenshtein distance algorithm for decoding messages and find that it provides excellent results.
- We demonstrate how reliability can be improved by sending multiple copies of the same message.

1.5 Thesis Outline

The subsequent chapters of this thesis are arranged in the following manner. Chapter 2 presents background information related to how WiTAG performs backscatter communication. Chapter 3 describes the problems and challenges related to reliable communication in a system in which the tag is not able to sense when a WiFi packet is being transmitted and there is no coordination between the tag and any WiFi devices. In Chapter 4, we design and implement a communication protocol to address these challenges. We develop solutions to the challenges of synchronization, encoding, block ACK filtering, decoding, and message reliability. In Chapter 5, we describe the experimental setup used to evaluate our system. In Chapter 6, we empirically evaluate our implementation. In Chapter 7,

we discuss related work. Finally, Chapter 8 presents our conclusions and ideas for future work.

Chapter 2

Background

As noted previously WiTAG takes advantage of MAC-layer frame aggregation (the combining of several subframes to create larger frames) to communicate data by altering the wireless channel. We now provide a bit of background on frame aggregation and PHY layer channel estimation and correction.

2.1 802.11 Frame Aggregation

The IEEE 802.11n, 802.11ac and 802.11ax standards provide a frame aggregation mechanism to improve the efficiency of the MAC layer [15, 16]. To avoid overheads such as performing channel sensing and transmitting an acknowledgment per frame, multiple *MAC Protocol DATA Units* (MPDUs) are combined into a larger aggregated frame (A-MPDU), as illustrated in Figure 2.1. By aggregating multiple subframes, the overheads are amortized over more data and therefore the efficiency of the MAC layer improves significantly. The receiver of an A-MPDU transmits a *block ACK* back to the sender to report the fate of the individual subframes inside the A-MPDU. A block ACK is similar to the legacy 802.11 acknowledgment, however rather than acknowledging the successful reception of one frame, it reports the fate of each MPDU using an appropriately sized bitmap (e.g., 64-bits for 802.11n and 802.11ac and up to 256-bits for 802.11ax). WiTAG leverages this frame aggregation scheme and block ACKs to allow a tag to transmit data to a WiFi device.

Figure 2.1: 802.11n/ac/ax A-MPDU structure [5]

2.2 Channel Estimation and Compensation

In all 802.11 standards including n, ac and ax, the PHY header starts with multiple known training symbols. The receiver utilizes these symbols to perform channel estimation [15, 16]. The wireless channel determines how a signal changes as it propagates from transmitter to receiver. Since the PHY header includes known symbols for each subcarrier, the receiver can estimate the phase and amplitude of the signal per subcarrier. This is known as Channel State Information (CSI). The receiver uses the estimated CSI to remove the effect of the channel from the received signal.

In WiTAG a WiFi device transmits packets (called query packets) using a PHY rate that will be able to successfully transmit packets to a targeted receiving WiFi device. Query packets are transmitted solely for the purpose of providing backscatter devices with an opportunity to communicate. Because the PHY rate is chosen so that packets can be successfully decoded by the receiver, block ACKs from the receiver would normally contain all 1's (indicating the successful reception of all packets). A tag communicates its data (e.g., a 0) by altering the wireless channel for a short period of time during the transmission of an MPDU, ideally resulting in a subframe that can not be decoded. Using this technique a tag can embed its data in a block ACK by corrupting or not corrupting MPDUs to send zeros and ones respectively. Details of how MPDUs can be corrupted can be found in [6].

2.3 Targeting A-MPDUs

One use case for low-power WiFi backscatter communication systems like WiTAG is to permit IoT devices to periodically communicate data from sensors. In such a scenario the querying WiFi device and an IoT device would each be initialized with a communication schedule. For example, every 15 minutes (starting on the hour). This permits the IoT device to spend most of its time sleeping and conserving power. It wakes up periodically (every 15 minutes) to spend a short amount of time reading and transmitting data from a sensor embedded in the IoT device. The querying device sends query packets during the time the IoT device is expected to be transmitting.

Some WiFi backscatter systems [20, 35], utilize a receiver or signal detector in the tag to determine when WiFi packets are being transmitted. This allows the tag to precisely coordinate the timing of the transmission of its data with WiFi packets that are in flight. A critical feature of the design of WiTAG, is that it consumes considerably less power because it does not use a receiver or signal detector.

Without precise information about whether or not WiFi packets are in flight, a WiTAG device, upon waking up, repeatedly transmits its data (for some period of time) in the hope that its data will overlap with one or more in-flight WiFi query packets. This is a non-trivial problem because even though a querying device might try to transmit packets in quick succession to provide more targets for the tag, the timing of those packets will vary due to delays caused by channel sensing and software overheads as the packets work their way through the application and network stack. The subjects of study in this thesis are the challenges and possible solutions to the problem of building a system that permits reliable communication from a tag without being able to detect when WiFi packets are in flight.

2.4 Software Retries

Block ACKs are part of the 802.11 protocol and are designed to accommodate cases where an A-MPDU or MPDU are not successfully decoded. Upon reception of a block ACK that contains one or more 0's the querying device is required to retransmit the MPDUs that were not received (this retransmission of corrupted MPDUs is part of the standard protocols).

For A-MPDUs that are not software retries the bits in the block ACK have a one-to-one correspondence with the position of the MPDUs within the A-MPDU. This is because if we have 64 MPDUs and a block ACK of length 64, each MPDU position is equal to its bit position in a block ACK. However, for software retries this is not necessarily the case. The position of the retried MPDUs depends on the size of the block ACK window [15] and the sequence numbers of the MPDUs that require retransmission. For example, if only the first and the third MPDUs in the previous A-MPDU were corrupted, those two MPDUs are included in a software retry. Since the first MPDU was corrupted, The block ACK window doesn't advance. As a result, the software retry cannot include any more MPDUs. Even though the position of the MPDUs in the software retry are 1 and 2, their corresponding bit in the block ACK will be in position 1 and 3 based on their original sequence numbers. As a result, the tag also attempts (probabilistically) to avoid overlapping with A-MPDUs that result from a software retry. This is done by inserting a time delay, that approximates the time required for a software retry, between consecutive tag transmissions. Because the

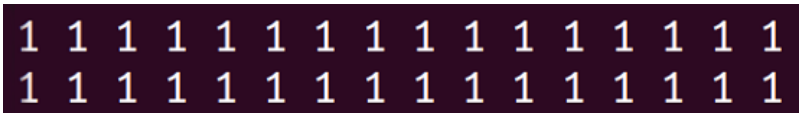
tag knows the PHY rate being used for query transmissions and such transmissions are completely defined by the 802.11 protocol we can compute a reasonable approximation for time required for a software retry. Note that using a longer delay increases the likelihood of avoiding a software retry but at the expense of reducing the rate at which the tag can communicate data. Interestingly, as will be seen in 6.2.3, the time used for this delay impacts the probability of overlaps and therefore the tag's message rates in unexpected ways.

2.5 WiTAG Implementation

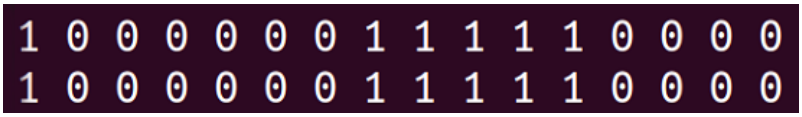
As explained in Section 2.1, an A-MPDU contains a single PHY header followed by n subframes. The header is used to estimate the channel. This estimation is used to correct the channel for subsequent subframes. In a WiFi network, the duration of an A-MPDU transmission is only a few milliseconds and wireless channels typically do not change during this time [37, 38]. Therefore, the channel estimation done at the beginning of the A-MPDU is sufficient to correct subsequent subframes. WiTAG leverages this knowledge by modifying the wireless channel during the transmission of subframes. The channel estimation done at the beginning of the A-MPDU is no longer representative of the channel state when the tag later modifies the wireless channel and the subframe subjected to WiTAG's channel modifications will not be received successfully. As a result, WiTAG can selectively corrupt desired subframes by changing the wireless channel during their transmission. WiTAG uses knowledge of the PHY rate and size of the MPDUs payload to determine the MPDU length (in time) and can therefore selectively modify or not modify the wireless channel to transmit 0's or 1's, respectively.

2.5.1 Changing the Wireless Channel

A wireless channel consists of a direct and multiple indirect paths created by signal reflections within an environment. If the phase or amplitude of a reflected signal changes, the wireless channel will change. WiTAG uses an antenna which can be switched between two modes: reflective and non-reflective. An antenna is reflective when short circuited and non-reflective when it is an open circuit [29]. This allows WiTAG to quickly change the wireless channel by switching its antenna between these two modes. The tag can be non-reflective during channel estimation (i.e., the beginning of an A-MPDU), and then become reflective during the transmission of subframes. This will corrupt the subframe since the channel estimation is no longer valid for later subframes. In reality, WiTAG is constantly



(a) Tag is in sleep mode



(b) Tag changes the channel

Figure 2.2: Corrupting subframes. Block ACKs of four A-MPDUs (with 16 subframes) when a) the tag is doing nothing, and b) the tag periodically changes the channel between two states [5]

altering the channel by different degrees for 0's and 1's to maximize the changes it imparts on the channel (see Abedi et al. [6] for details).

Figure 2.2 shows the block ACKs for four A-MPDUs (each consisting of 16 subframes) when a) the tag is not active, and b) the tag is active and changes the channel state. When the tag is in sleep mode, (as seen in Figure 2.2a) it has no impact on the channel and all subframes are received. Figure 2.2b demonstrates what happens when a WiTAG device periodically changes the channel. In this case the bits in the block ACKs are 0 during the time that the tag changes the channel and 1 otherwise. This shows that WiTAG can corrupt subframes by changing the channel during their transmission.

Chapter 3

Reliable Communication: Problems and Challenges

We now outline the problems and challenges that need to be resolved in order to enable reliable WiFi backscatter communication. We begin with a detailed description of the challenges in overlapping a tag's data transmission with an A-MPDU and with corrupting specific, individual MPDUs within an A-MPDU. Because WiFi channel conditions are constantly changing we also need to design a system that will take into account the fact that sometimes when a tag is attempting to corrupt an MPDU it may not be able to and sometimes when a tag is not corrupting an MPDU, it may still not be able to be decoded by the receiver. Both of these conditions could result in a misinterpretation of the tag's intended data and are also a source of challenges that will be described throughout this chapter. Our goal is to understand and describe the considerations for the design of a tag packet (we will refer to this as a "message" and use "packet" to refer to units of transmission by WiFi devices).

3.1 Coarse-Grained Synchronization

Coarse-grained synchronization refers to the timing that is required so that a tag's transmission occurs not only when a WiFi packet is in flight, but so the transmission overlaps precisely and completely with the portion of the WiFi packet that contains the A-MPDU.

Figure 3.1a shows an example of a complete overlap, where the transmission of the tag's message begins and ends within an A-MPDU. In this case, all of tag's message will

Figure 3.1: Coarse-grained synchronization

be embedded into the A-MPDU. However, if there is no overlap or only a partial overlap the tag's data will not be able to be completely transmitted (i.e., the tag will not be able to form a message). For example, in Figure 3.1b, when the tag is transmitting, there is no A-MPDU and therefore no overlap. Figure 3.1c shows partial overlap where the tag's transmission starts before the transmission of the A-MPDU has started and it ends part way into the A-MPDU. When part of the PHY header is corrupted, the receiver may not be able to decode any MPDUs and returns a block ACK containing all 0's. Figure 3.1d also shows a partial overlap, where the tag's transmission begins during an A-MPDU but the A-MPDU ends before the tag completes its transmission.

As can be seen in these examples, only a complete overlap of the tag's transmission with an A-MPDU will support the tag's data transmission. It is worth noting that even if a querying device is transmitting packets as fast as it possibly can, there will be a considerable amount of time between consecutive WiFi packets (due to time spent waiting for block ACKs, performing channel sensing, and other parts of the 802.11 protocol that limit the packet rate). As a result, a tag's communication attempt is not likely to coincide with any part of a WiFi packet (as depicted in Figure 3.1b). Therefore, the querying device and the tag will both continuously transmit for a period of time, to increase the

probability that the tag's message will completely overlap with an A-MPDU (as shown in Figure 3.1a).

In Section 4.1, we will show that despite the lack of coordination between the querying device and the tag, the probability that the tag's message coincides with the transmitting WiFi devices's A-MPDU is sufficiently high that reliable communication can be supported.

3.2 Fine-Grained Synchronization

A tag communicates bits by corrupting or not corrupting an MPDU. The ability to precisely corrupt or not corrupt particular MPDUs is fundamental to WiTAG's ability to transmit bits. Corrupting an MPDU accurately is challenging because it depends on whether or not the tag is capable of targeting specific MPDUs without any form of synchronization between the querying device and the tag.

To target an individual MPDU, the tag needs to be able to corrupt for a particular duration of time. The precise duration depends on the duration of an MPDU which can be computed using the amount of data in the MPDU and the PHY rate being used for transmission. The software used to send query packets controls the PHY rate and the amount of data transmitted in an MPDU. These values are both known to the tag which can compute the length of time it will use to attempt to corrupt an MPDU.

If the start of a tag's corruption coincides exactly with the start of an MPDU (i.e., they just happen to be synchronized) it will corrupt one and only one MPDU. Figure 3.2a, shows an example when the i^{th} MPDU coincides with the duration of the tag's corruption. The figure indicates that it has been corrupted by showing the i^{th} bit in the block ACK (b_i) which has been set to a 0. All other bits in the block ACK are 1 because the tag does not corrupt them.

If the tag's attempt at corruption does not precisely coincide with the start of the targeted MPDU, the duration of the corruption will instead coincide with portions of two adjacent MPDUs. If a tag's corruption is shorter than the full duration of an MPDU, the receiver may or may not be able to decode that MPDU (i.e., we cannot predict whether a partially corrupted MPDU can be decoded or not). Figures 3.2b and 3.2c show examples where, although the tag is corrupting for the duration of one MPDU, because of when that corruption starts and ends, the tag's corruption overlaps portions of two consecutive MPDUs. These two figures show that the $i-1^{th}$ MPDU is not corrupted (because there is no corruption during that MPDU). They also show that fate of the i^{th} and $i+1^{th}$ MPDUs are not clear because the duration of corruption does not coincide with the entire duration

of the MPDU. This is indicated in Figures 3.2b and 3.2c by showing that the values of bits b_i and b_{i+1} in the block ACK are not known (denoted with ?).

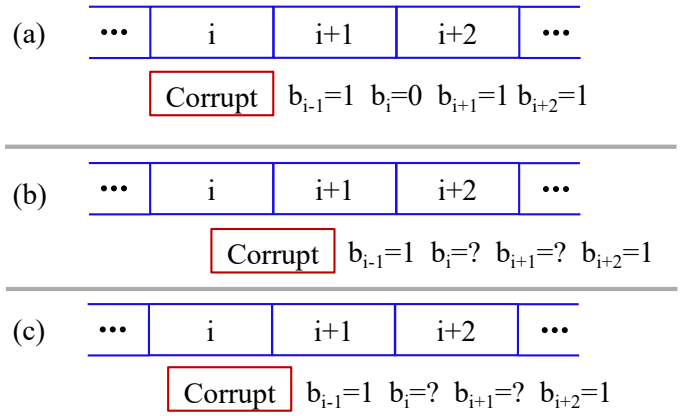


Figure 3.2: Fine-grained synchronization

To reliably transmit a message from the tag, our design and implementation needs to account for the fact that a receiver's ability to decode each MPDU may depend on how much of a tag's corruption coincides with an MPDUs duration.

3.3 Bit Encoding

We now examine issues raised because of the potential and likelihood that the duration of the tag's corruption might coincide with two consecutive MPDUs. Since it is unlikely that the beginning of the tag's corruption precisely coincides with the exact start of a targeted MPDU, it is highly likely that two consecutive MPDUs may be affected.

As shown in Figures 3.2b and 3.2c, when trying to corrupt a single MPDU, it is unclear whether one, both, or none of the MPDUs will be successfully decoded. Therefore, trying to corrupt a single MPDU will not be sufficient to reliably encode a single bit of data. We next consider how many MPDUs would need to be corrupted to reliably encode a single bit of data.

Figure 3.3 shows some different scenarios that could occur when trying to encode a 0 bit with two or three MPDUs (similar issues occur when trying to encode a 1). Figure 3.3a shows what could happen if the start of corruption perfectly coincides with the beginning of MPDU i and also covers all of MPDU $i + 1$. In this case, bits b_i and b_{i+1} of the block

ACK should contain 0's while the other bits will be 1. However, Figure 3.3b shows what could happen if the start of corruption does not perfectly coincide with the beginning of MPDU i and instead covers part of MPDU i, all of MPDU $i+1$ and part of MPDU $i+2$. In this case, bit b_{i+1} of the block ACK should be 0, bits b_i and b_{i+2} would be uncertain (denoted with ?) while the other bits should be 1. Because the abliity to successfully corrupt an MPDU will depend on ever changing channel conditions it might be the case that although one of the MPDUs in Figure 3.3b should be corrupted, one needs to consider what happens if the corruption of that MPDU is unsuccessful.

As a result, one might consider trying to encode a 0 bit using three (or more) consecutive MPDUs. Figure 3.3c shows an example of what might occur when trying to corrupt three consecutive MPDUs. In this case, bit b_{i+1} and b_{i+2} of the block ACK should be 0, while bits b_i and b_{i+3} would be uncertain (denoted with ?) while the other bits should be 1. This increases the probability that the receiver will not be able to decode at least one (and likely two) MPDUs and could improve reliability through redundancy (repetition).

The challenge is to encode a bit using a sufficient number of MPDUs to ensure a reasonable level of reliability while trying to not require too many MPDUs (which reduces the number of bits that can be transmitted in a single block ACK). In Section 6.3, we conduct experiments to examine the pros and cons of using different numbers of MPDUs to encode a bit.

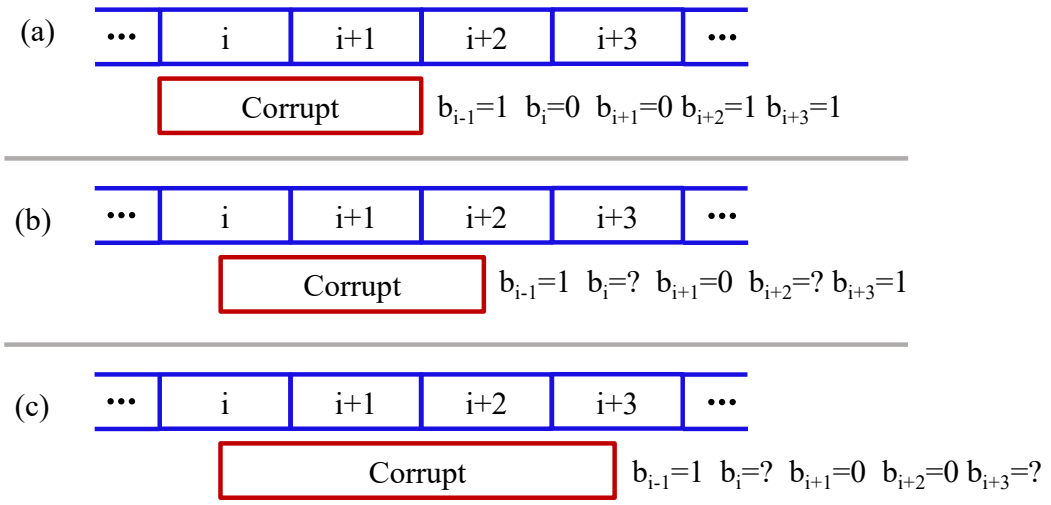


Figure 3.3: Encoding one bit using multiple MPDUs

3.4 Message Encoding

As discussed in Section 3.1, the majority of WiFi packets are not expected to coincide with the transmission of a message and all bits in those block ACKs are expected to contain ones. Consequently, if the tag wants to transmit a message where all of the bits in the message are ones, a block ACK that contains such a message will be indistinguishable from a block ACK that did not coincide with a tag's transmission.

Figure 3.4a shows an example of a block ACK that could be a result of a message that contains all ones or an A-MPDU that did not coincide with a tag's transmission. A natural solution to this problem would be to use a predefined pattern of bits to denote the start and end of a tag's message. We call the bit pattern to denote the start of the message the preamble and the pattern that denotes the end of the message the postamble. Since not corrupting MPDUs is used to represent a 1, one could naturally use a 0 as the preamble and as the postamble (i.e., enclose the tag's message data with a 0 bit).

Momentarily ignoring the fact that one MPDU will not work well to represent one bit, Figure 3.4b shows an example of a tag that uses one MPDU for each bit of data. This message is sending 10 bits of data (all ones) with one 0 used to denote the start and end of the contents of the tag's message. Note that because we need to distinguish cases of partial overlaps with an A-MPDU, this example uses a 1 before the preamble and a 1 after the postamble (we call these fenceposts). Without the leading and trailing ones we can not reliably distinguish partial overlaps that start and/or end with the first or last MPDU in the A-MPDU. We use p to denote the number of MPDUs used in each of the preamble and postamble. The number of bits in the message is denoted with b and m denotes the number of MPDUs used to represent each bit. The length of a message, ml will therefore be $(2 \times p) + (b \times m) + 2$.

As noted previously, one MPDU is not sufficient to reliably denote a single bit. For the same reasons, one can not reliably use a single 0 bit to denote the start and the end of the message. Figure 3.4c shows an example of a tag's message that uses $p = 2$ and $m = 2$. In this example, assuming the same number of MPDUs are used for the message, that means the number of bits one could transmit would now be four ($b = 4$). Similarly, Figure 3.4d shows an example using $p = 3$ and $m = 3$ which would support the transmission of two bits of data ($b = 2$) using the same number of MPDUs.

Note that the preamble and postamble are also used to determine if a tag's message completely overlaps with the A-MPDU. The querying device and the tag would both be initialized to know the values of p, m and b. This permits us to implement a message recognition algorithm (an algorithm that filters block ACKs that do not appear to contain

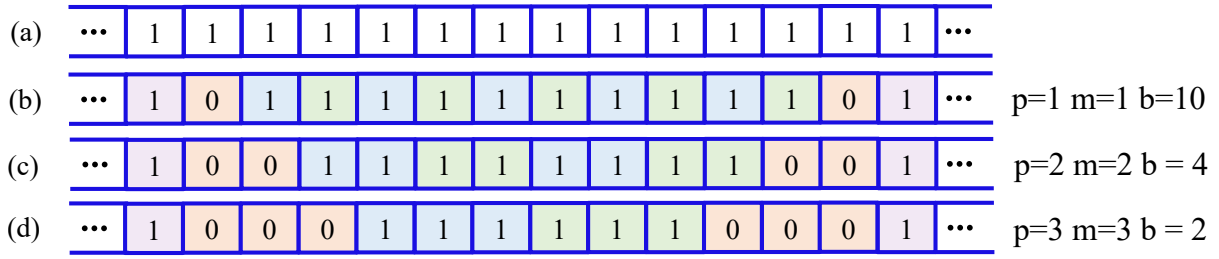


Figure 3.4: Example block ACKS, showing the addition of a preamble and postamble (delimiters) to the tag's message. Orange bits denote preamble and postamble, blue and green are used to denote alternating bits of actual data, and purple denote leading and trailing 1's (added to detect if overlap is complete or not)

a message from the tag).

Intuitively, one can see that shorter message lengths have a higher probability of completely overlapping with the A-MPDU than longer messages (because longer messages have a higher probably of overlapping with the header or overlapping with only the last part of the A-MPDU (as shown in Figure 3.1c and d)). In Chapter 4 we examine how different values of m, p and b impact the message length (ml) and the probability of completely overlapping with an A-MPDU and the influence they have on the ability to decode the tag's message.

3.5 Block ACKs Filtering

As noted previously, a significant number of transmitted query packets will not coincide with any tag messages. As a result, some form of filtering algorithm is useful to identify block ACKs that do not contain a tag's message. Otherwise, the system will try to decode block ACKs that do not contain messages and incorrect data will be delivered.

Note that due to environmental factors that can corrupt MPDUs (e.g., changing channel conditions, interference from other WiFi devices, or other devices like microwave ovens), block ACK filtering is a bit more complex than simply discarding ACKs that do not contain preamble and postamble. An additional complication occurs because one can't know a priori if the number of MPDUs used for a corrupted bit will be m or $m+1$ (i.e., it will be m if the MPDU corruption starts precisely at the beginning of an MPDU and will be $m+1$ otherwise). In Section 4.5 we describe the design and implementation of our block ACK filtering algorithm.

3.6 Message Decoding

After block ACKs that are believed not to contain a tag message are filtered, the message needs to be decoded. Although decoding is driven by the encoding technique, environmental factors may create ambiguous messages which may be difficult to decipher. As noted previously, the challenge is that at different points in time and/or depending on how much of the MPDU or which parts of the MPDU are subjected to the tag's attempt to corrupt, the receiver may or may not be able to successfully decode the MPDU. Similarly, although the PHY rate is chosen to try to ensure that all WiFi transmissions will be successfully decoded by the receiver, changing channel conditions means that this can not be guaranteed. As a result, the method used to decode the tag's message needs to account for possible ambiguities in the message. In Section 4.6 we describe how we use Levenshtein's edit distance algorithm to decode messages.

3.7 Message Reliability

Since there are several issues that can arise at various points during communication of a message, we need a mechanism to try to ensure that a tag's message is received reliably.

One natural approach would be to use error-detection or error-correction techniques. Unfortunately, they typically require the use of additional bits of data and increase the complexity of encoding and decoding. Recall that our goal is to build a low-power communication mechanism.

The entire tag's message is restricted to fitting within a block ACK (e.g., 64 bits in 802.11n and ac networks). As noted previously and will be shown in detail in Section 4.1 longer tag messages decrease the probability of a complete A-MPDU overlap (decreasing the number of messages that can be transmitted within a period of time). Also, as described previously, the message length (ml) depends on the values of p, m and b. For example, if $p = 4$, $m = 3$ and $b = 10$, the message will require the use of 40 MPDUs. Adding more bits for error detection or correction requires m MPDUs for each bit added which can reduce the probability of an overlap.

We increase reliability by having the tag repeat its message multiple times. This approach fits naturally within the system because the tag must repeatedly send its message in order to probabilistically overlap with one or more A-MPDUs. The challenge is to determine the number of times the tag should repeat its message to achieve the desired level of reliability (e.g, 99.999%). In Section 4.7 we discuss ways of predicting message reliability based on the number of messages.

Chapter 4

Design and Implementation

In this chapter, we provide some theoretical foundations for the probability that a tag's message will coincide with a query packet and how to reliably transfer data from the tag. In doing so, we then explore some of the design space with the goal of being able to reliably receive and decode a tag's messages in a small amount of time. The overarching goal of this chapter is to explore possible solutions to the challenges of synchronization, encoding, filtering, decoding, and improving reliability that are presented in Chapter 3 to inform our implementation of WiTAG.

4.1 Complete A-MPDU Overlap

WiFi packet transmission varies due to factors like RTS/CTS, random backoffs, and channel access wait times during third-party device transmissions. Therefore, we assume query packet transmissions are independent of each other.

The probability that the transmission of the tag's message and the A-MPDU of a query packet perfectly coincide is a critical factor in WiTAG's ability to transfer data. For a complete overlap between a tag's message of length ml and an A-MPDU of length al, the tag's corruption must happen in the window between the start of the first MPDU and the start of the $(al - ml)^{th}$ MPDU (as shown in Figure 3.1a). Otherwise, there may be a partial A-MPDU overlap (e.g., as shown in Figure 3.1b or 3.1c) or no overlap (as shown in Figure 3.1d) and the tag's message won't be fully transmitted.

The probability of a tag's message completely overlapping an A-MPDU (P_{ol}) can be derived as:

$$P_{ol} = \begin{cases} 0, & \text{for } al < ml \\ (Dur_{al} - Dur_{ml})/(Dur_{al} + Dur_{oh}), & \text{for } al >= ml \end{cases} \tag{4.1}$$

In Equation 4.1, Dur_{al} is the duration of time spent transmitting an A-MPDU of length al. Dur_{oh} is the duration of time the querying device is not transmitting data but is required for the transmission of the A-MPDU due to protocol overheads. If the number of MPDUs in an A-MPDU (al) is smaller than the message length (ml), there cannot be a complete overlap because a message won't fit in an A-MPDU. Hence, the value of P_{ol} will be zero for al less than ml.

If the number of MPDUs in the A-MPDU is large enough to contain a full message (i.e., $al \geq ml$), the probability of a message and the A-MDPU completely overlapping (P_{ol}) is shown in the second row of Equation 4.1. In this case, the probability is determined by the chance that the message is transmitted early enough while the A-MPDU is in flight so that it will be completed before the end of the A-MPDU (i.e., in a window of time determined by $Dur_{al} - Dur_{ml}$). The probability of that complete overlap is determined by considering the duration of that time window relative to the time required to transmit a WiFi packet (i.e., the time between two consecutive packets). The time to complete the transmission of a packet is denoted as $(Dur_{al} + Dur_{oh})$.

Figure 4.1 provides a visual representation of A-MPDUs and associated overheads. The overhead before an A-MPDU is 229.5 µs and it accounts for Distributed Interframe Space (DIFS), Backoff, Request To Send (RTS), Short Interframe Space (SIFS), Clear To Send (CTS), and the Physical Layer Convergence Protocol (PLCP) header. Timings of these overheads are known (they are specified in the 802.11 standard.) The overhead after an A-MPDU is for SIFS and the block ACK of 48 µs, which makes the total overhead a constant value of 277.5 µs. For an A-MPDU containing 64 MPDUs with each MPDU's payload size being 324 bytes (i.e., a suitable 18 µs MPDU duration as we will show in Section 4.1.3), the A-MPDU duration is (64×18) 1152 µs. This duration will be different for MPDUs with different payload sizes or for an A-MPDU with a different number of MPDUs. For a full A-MPDU consisting of 64 MPDUs and of payload size 324 bytes, the total duration of an A-MPDU and overhead is 1429.5 µs.

Figure 4.2 shows the probability of a complete overlap between an A-MPDU of length 64 and a tag message. Note that the tag's message length is denoted on the x-axis as the number of MPDUs needed to send a message. The probability of overlap changes based on MPDU durations (determined by payload size). Larger MPDUs have a higher probability of overlap compared to smaller MPDUs. This is because increasing MPDU duration also increases the A-MPDU duration while the overhead duration remains constant. This makes

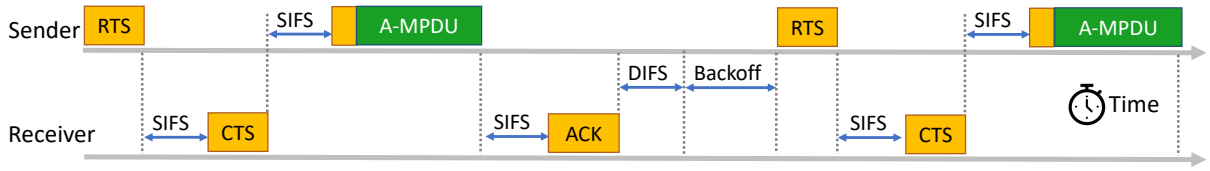


Figure 4.1: An A-MPDU and associated overhead. The A-MPDU has overhead before and after its transmission

the duration of the overhead a smaller fraction of the total duration (overhead duration + A-MPDU duration) for larger MPDUs. The figure shows that for the shown MPDU sizes, the probability of overlap is sufficient to support communication for most message lengths and that longer tag messages decrease the probability of a complete overlap.

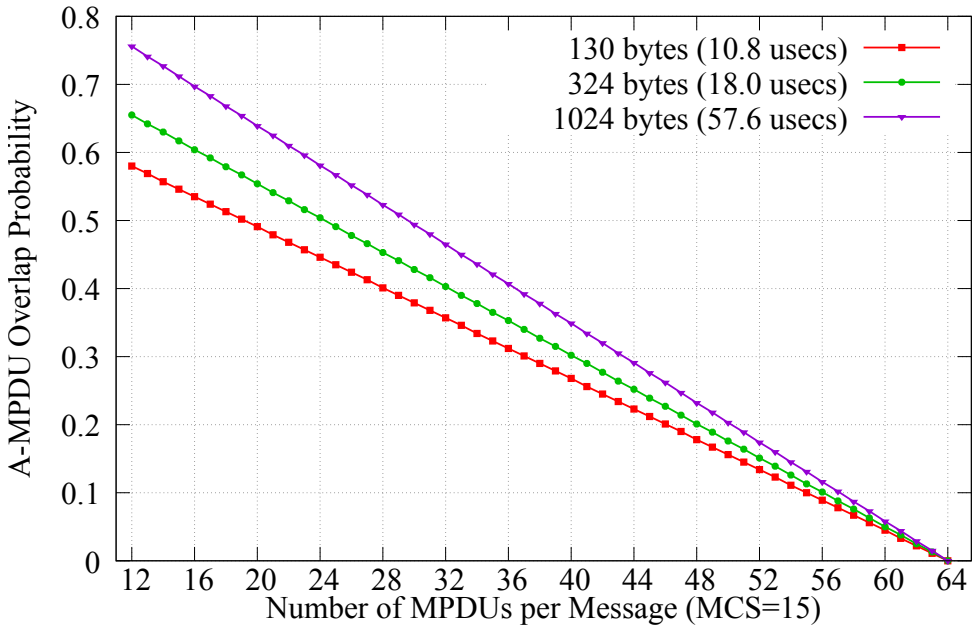


Figure 4.2: Probability of complete A-MPDU overlap for different message lengths based on three different MPDU durations (assuming that all A-MPDUs contain 64 MPDUs)

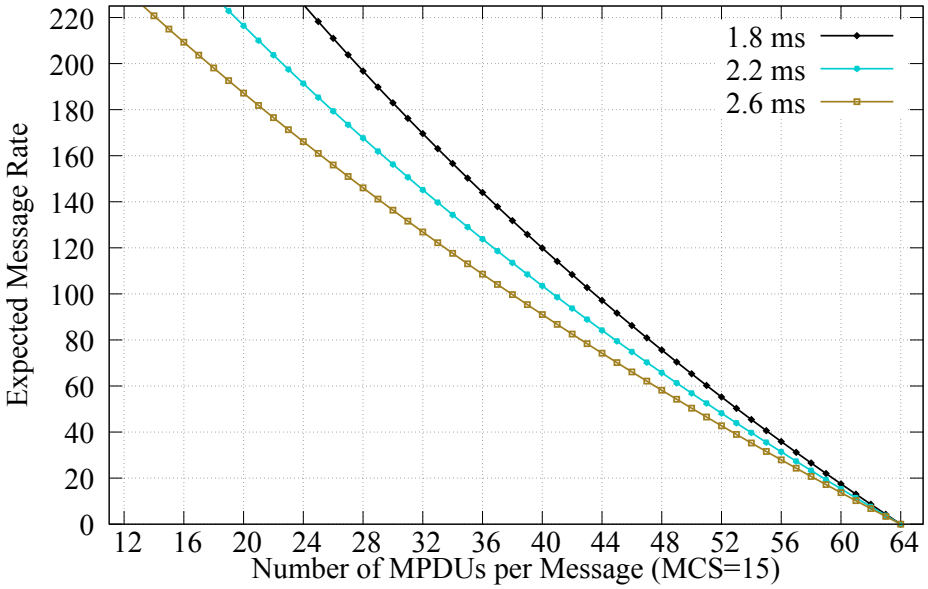


Figure 4.3: Expected tag message rates for different message lengths using three different time delays (assuming that all A-MPDUs contain 64 MPDUs)

4.1.1 Computing Tag Message Rates

Because we intend to repeat messages to improve reliable delivery, we now show how to determine the expected message rate given different message lengths. As will be seen in Section 6.5, the higher the message rate, the more reliability we can provide in a limited window of time. First, the number of times a tag attempts to transmit a message in 1 second is:

$$MsgsPerSec = 1s/(Dur_{ml} + T_d) \tag{4.2}$$

where T_d is the time delay used to try to avoid software retries.

However, not all of the tag's attempts will completely overlap with an A-MPDU. Therefore, we compute the expected message rate ($MsgRate$) by multiplying the number of messages the tag is attempting to send per second ($MsgsPerSec$) by the probability of overlap (P_{ol}) shown in Equation 4.1.

$$MsgRate = MsgsPerSec \times P_{ol} \tag{4.3}$$

Figure 4.3 shows the message rate ($MsgRate$) for MPDU size of 18 µs and three different time delays (T_d) of 1.8, 2.2, and 2.6 ms. The figure demonstrates how the message rate

varies based on the number of MPDUs per message and the time delay. Messages that contain fewer MPDUs and lower time delays give the best message rates. This figure shows that for reasonable message lengths the message rate is expected to be quite high (perhaps surprisingly so).

4.1.2 Weighted Probability of Overlap for Expected Rates

In reality, A-MPDUs will vary in length. Frame aggregation typically takes place in the WiFi chipset firmware and depends on the rate and size of the data being sent and the particular algorithm being used. Additionally, due to a tag's corruption of MPDUs (or environmental factors), MPDUs that are not able to be decoded will be retransmitted by the querying device (i.e., as part of the standard protocol the device would transmit a software retry). Assuming the first subframe with sequence number Seq in an A-MPDU needs to be re-transmitted, then only the frames whose sequence numbers that are lower than $Seq + L$ (maximum A-MPDU length i.e., 64) can be aggregated in the next retransmitted A-MPDU [28]. A-MPDUs with different lengths (Dur_{al}) will have different probabilities (P_{ol}) of coinciding with a tag's transmission.

If the distribution of A-MPDU lengths is known (each being between 1 and the maximum A-MPDU length, L), then a weighted probability of overlap (P_o) can be derived.

First, we multiply the probability of complete overlap for an A-MPDU of length al (P_{ol}) with the frequency with which al length A-MPDUs occur in the data being examined (F_l). This gives an expected number of messages from A-MPDUs of length al.

Summing the number of messages from all A-MPDUs of all lengths gives the total number of expected messages. Then, the weighted probability of overlap (P_o) is the total number of messages divided by the total number of A-MPDUs.

$$P_o = \frac{\sum_{al=1}^{L} P_{ol} F_l}{\sum_{al=1}^{L} F_l} \tag{4.4}$$

The weighted probability of overlap (P_o) accounts for the actual distributions of A-MPDU lengths for a particular experiment. We utilize this probability to compute the expected message rates for WiTAG experiments. We will use this calculation later in Section 6.2 when we compute the expected message rates for different data sets.

4.1.3 MPDU Duration

The duration of an MPDU (Dur_{MPDU}) is determined by the number of bytes transmitted per symbol ($MCSBytes$). A symbol refers to a unit of data transmission representing a specific modulation state. It represents a unique combination of amplitude, frequency, or phase variations that carry multiple bits of information [12]. Based on a modulation and coding scheme such as (MCS-15 or MCS-21), a symbol duration ($GuardSymbolDur$) of a short or a long guard interval which impacts the transmission rate, and the payload size ($payload$), MPDU duration is:

$$Dur_{MPDU} = ceil(\frac{payload + payload\,\%\,4}{MCSBytes}) \times GuardSymbolDur \tag{4.5}$$

We divide the payload by the symbol size ($MCSBytes$) because that determines the number of symbols required for the payload. Then we multiply the result by guard duration ($GuardSymbolDur$) to calculate how long it takes to transmit the payload.

In our experiments, the sender uses MCS-15, the highest coding scheme, with two antennas (this is a PHY rate of 144.4 mpbs). For our sending device higher PHY rates increase the chance of signal corruption because they are less robust. We also use a short guard interval to allow a higher A-MPDU rate. By using a short guard interval, more A-MPDUs can be sent per second. The short guard symbol duration ($GuardSymbolDur$) is 3.6 µs. Each symbol contains 520 bits (65 bytes) when using MCS-15.

In addition, while taking into account the size of the payload, including 4-byte padding and 2-byte delimiters, we aim to pick a payload that maximizes the message rate ($MsgsPerSecs$) while keeping the payload size small. The payload size should ideally be a multiple of 4 for efficient transmission. Any remaining bytes are padded to meet this requirement. This is denoted by addition of $payload$ % 4 in the equation. We also want the resulting MPDU duration to be a whole number to avoid sub-microsecond corruption (since the Arduino Duo we use provides microsecond timer resolution). To meet this goal, we use 324 bytes as the payload size, which results in a high message rate and an MPDU duration that is a multiple of 1 µs.

Substituting the values for $MCSBytes$ (65 bytes transmitted per symbol when using MCS-15), $GuardSymbolDur$ (3.6 µs), and $payload$ (324 bytes) in Equation 4.5 gives:

$$Dur_{MPDU} = ceil(\frac{324 + 324\,\%\,4}{65}) \times 3.6\,\mu s = 18\,\mu s$$

4.2 MPDU Overlap

WiTAG doesn't use a synchronization mechanism to ensure a complete overlap or for the corruption of sub-frames. As a result, it is likely that two adjacent MPDUs are affected for each bit a tag transfers. The probability of affecting each MPDU is in the range [0,1]. By transferring the same bit value multiple times, a tag can increase the probability of encoding a bit in such a way that it can be successfully decoded.

This idea is shown in Figure 3.3a and 3.3b where the block ACK value for position $i+1$ is guaranteed to be a 0 if a tag attempts to corrupt for two MPDUs starting in the i^{th} MPDU. As shown in Figure 3.3c, the block ACK values for positions $i+1$ and $i+2$ are guaranteed to be 0's if a tag attempts to corrupt three MPDUs starting in the i^{th} MPDU.

As discussed in Sections 3.1 and 3.2, the lack of synchronization between the querying device and the tag, and the tag's inability to detect packets prevent a tag's message from being transferred as expected. Therefore we have to encode each bit (Section 4.3) and the whole message (Section 4.4) such that a message can be transmitted and decoded correctly.

4.3 Bit Encoding

We now examine techniques that could be used to encode a single bit of data. As described in Section 3.2, targeting only one MPDU to encode a bit will not be reliable because when the timing of the tag's corruption does not completely align with only one MPDU, two adjacent MPDUs may be affected. As a result, each of the two MPDUs may or may not be corrupted. If we use two adjacent MPDUs to encode one bit, if the timing is perfect two MPDUs would likely encode the desired value and if not at least one out of the three affected MPDU has the expected value that the tag is attempting to transmit.

In general, this approach could be extended to using more MPDUs to encode a bit in order to obtain higher decoding accuracy. This is a common form of encoding; repeating bits to provide redundancy. We will refer to the number of MPDUs a tag attempts to use to encode a bit as m. As long as the majority of MPDUs are correctly encoded one should be able to decode the bit correctly. In other words, using more MPDUs to encode a bit (adding redundancy) provides more robustness to bit errors. Therefore, we now examine the question of what is a good number of MPDUs to use to encode a single bit of tag data (i.e., how to choose a value for m).

4.3.1 Issues with Overlap Probability

At some point a tradeoff will need to be made because as m increases, the message length increases, which decreases the probability of the tag's data overlapping with an A-MPDU (P_{ol}). For the time being, we separate these two issues and study the degree to which decoding accuracy increases as m increases. In Section 6.3, we will evaluate different values of m for encoding a bit.

4.4 Message Encoding

When the query packet and the message do not coincide (i.e. there is no overlap), the block ACKs corresponding to all MPDUs should be 1. To distinguish cases with no overlap from a message that the tag transmits, we add a 0 bit in the beginning (preamble) and the end (postamble) of a message. We repeat the 0 bit (p times) in order to enhance the encoding's robustness. A suitable value for p can be based on the number of MPDUs to encode enclosed bits (i.e., the m value) since all of the same issues arise for encoding a single bit. This will be determined through our evaluation of different m values in Section 6.3.

4.5 Block ACK Filtering

As discussed in Section 3.5, a block ACK may arrive for an A-MPDU that did not overlap with a message. We want to ignore block ACKs that are unlikely to be decodable tag messages. This includes block ACKs where there is not a complete overlap and block ACKs that have been corrupted due to changing channel conditions, interference or other environmental factors. As a result, we use the following rule to filter block ACKs.

1. Too few zeros: The first filter checks if there are too few zeros in a block ACK. For example, if there is no overlap between the WiFi packet and tag message, the block ACK values will be all 1's. A complete overlap will have at least p zeros in the preamble and p zeros in the postamble. Therefore, we filter out block ACKs that contain less than $2p$ zeros.

2. Too many zeros: Likewise, if a block ACK contains too many zeros, it needs to be filtered. If the number we are trying to send is 0, we will have at most $(2 \times p) +$

$(b \times m) + 1$ MPDUs that should be corrupted. If a block ACK contains more zeros, it indicates corruption due to environmental factors and we can eliminate that block ACK.

3. Unexpected message length: Finally, a block ACK that contains a message of unexpected length will be filtered. As discussed in Section 3.4, since we know the m, p and b values, we can determine that our message will be of length ml or $ml + 1$ depending on the timing of the attempted corruption of MPDUs. To check for a potential message in a block ACK, we determine its message length by checking how far apart the preamble (first p zeros) and the postamble (last p zeros) are. If the message length that we determine for a potential message is not ml or $ml + 1$, we filter the block ACK. Only checking the first and last occurrence of p zeros to compute message length is sufficient because if the length is greater than $ml + 1$, it suggests that the A-MPDU was corrupted by external sources and if the length is less than ml, it suggests insufficient corruption caused by partial overlap or potentially unsuccessful corruptions.

4.6 Message Decoding

After filtering block ACKs that do not contain a message or are unlikely to be decoded correctly, we then attempt to determine the value in the message that was sent. Because we are aware of the 2^b possible values that a message might contain, we employ an edit distance algorithm to identify the most likely transmitted value. An edit distance algorithm modifies bits (through deletion, insertion, and swapping) to determine the minimum number of changes from a potential message to a known value, in this case any of the 2^b values.

The particular algorithm we use to compute edit distance is the Levenshtein distance algorithm [21, 31, 32]. We first strip the fence posts, preamble and postamble and then compute the edit distance from the received value to the binary representation of all 2^b possible values. We choose the message that corresponds to the value with the smallest edit distance. There are several possibilities for breaking ties. Currently, we simply use the lowest value with the smallest edit distance. We evaluate the Levenshtein distance algorithm in Section 6.4 and show that it obtains highly accurate results.

4.7 Message Reliability: Using N Messages

Recall that we repeatedly send A-MPDUs because we are relying on the probability of an A-MPDU and a tag message coinciding in time and completely overlapping. This approach inherently requires repeated message transmissions from the tag. Since message repetition is fundamental to how WiTAG operates, we use multiple messages to improve reliability.

When using N messages, a common approach for reaching consensus [26] is a plurality majority voting. If this technique is used to detect a value, the probability that the number of correct determinations will be greater than the number of incorrect determinations can be represented as:

$$\mathrm{P}(|c| > |i_j|, \forall j) = \sum_{k=1}^{N} \mathrm{P}(|c| = k)\mathrm{P}(|c| > |i_j|, \forall j \mid |c| = k) \tag{4.6}$$

Out of the 2^b possible values for a message, there is only one correct decoding. We denote the list of messages that are decoded correctly as c (for correct) and the list of messages that are decoded incorrectly as i_j's (the message was incorrectly decoded to the value j). The range of j is from 1 to $2^b - 1$ and covers each of the possible incorrect values (one of the values is correct and $2^b - 1$ are incorrect).

In Equation 4.6, $\mathrm{P}(|c| > |i_j|, \forall j)$ is the probability that the number of messages in the correct list ($|c|$) is greater than the number of messages in each incorrect list ($|i_j|$'s). In the summation, k ranges from 1 to N to cover the N messages that are being decoded.

Two conditions need to be met to determine a value correctly. First, $\mathrm{P}(|c| = k)$ is the probability that the number of messages in the correct list $|c|$ is equal to k. Second, $\mathrm{P}(|c| > |i_j|, \forall j \mid |c| = k)$ is the conditional probability that the number of messages in the correct list $|c|$ is greater than the length of each incorrect message list $|i_j|$ for all j. This condition needs to be met given $|c|$ is equal to k.

The formula in Equation 4.6 can be used to compute the probability of correctly determining the expected value. This computation requires us to know how the correct and the incorrect values are distributed. However, we don't know the distribution of values for our experiments beforehand. In Section 6.5 we run real experiments to show how increasing N increases reliability.

Chapter 5

Experimental Setup

In this section we describe the experimental setup used to evaluate WiTAG's performance when transmitting data. The evaluation involves transmitting specific values from a WiTAG device and analyzing the corresponding block ACKs to determine the success of transmissions.

5.1 Device Preparation

The WiTAG prototype device used in this thesis is built using an Arduino Duo microcontroller. We use a breadboard to connect two HMC536MS8G RF switches (manufactured by Analog Devices Inc.) to the Arduino so we can control when the switches reflect signals. As discussed in Section 2.5.1, the switches can be non-reflective when channel estimation is performed and become reflective during the transmission of a subframe in order to corrupt the subframe. When the switch changes the state, the channel estimation is no longer valid and the receiver will ideally be unable to decode the corrupted MPDUs. The switches are connected to two TECHTOO 2X 9 dBi omnidirectional WiFi antennas to improve the ability to corrupt signals from the sending access point.

We use a D-Link AC1200 DIR-822 WiFi Router that runs OpenWrt software version 22.03.2 as a sending access point (i.e., the querying device) and a TP-Link AC 1750 Wireless Dual Band Gigabit Router as a receiving access point. We also use a laptop to load WiTAG code onto the Arduino, to configure the WiFi devices, and control our experiments.

We connect the WiTAG device to the laptop with a USB-to-micro-USB cable. We use the Arduino IDE on the laptop to modify WiTAG's code and parameters such as m, b,

and p, the time delay and the experiment's duration. Our code uses the time delay as the period of time to wait after each message transmission for software retries to complete. We compile the code on the laptop and update the WiTAG code that is run on Arduino board.

We plug a memory stick into the sending access point to store block ACK data. This allows us to run longer experiments (generating large amounts of data), collect all the data (e.g., block ACKs) and analyze them later.

We want to evaluate how well WiTAG transmits multiple (2^b) values. One approach would be to run multiple experiments where WiTAG sends one value at a time and the sending access point captures block ACKs for that value in its own file. However, this technique is error prone and consumes a lot of time in performing the manual work of stopping and starting the experiments.

Instead, we program the WiTAG device to send all 2^b different values in one experiment and the sender stores block ACKs for all of the values in one file. This is done by sending each value for a predetermined length of time with no other transmissions in between. Afterwards, we utilize the timestamps of when each new value starts to be transmitted from WiTAG and the timestamps of when block ACKs are recorded by the sender to split the block ACKs for each value into separate files. This requires that the time on the laptop and the sending access point are roughly synchronized so that their timestamps can be compared. Loose synchronization is sufficient because we only need to know the approximate time at which the value being sent changes (gaps are left between different values by not sending data from the tag).

We connect the sending access point to the laptop with an ethernet cable and log into the access point using OpenWrt's web interface to synchronize the clocks on the two devices. This ensures the time on both devices are close (within 1 second). As noted, to account for minor time discrepancies between the devices, we add a 10 seconds delay after each value transmission is complete. During this delay, the sender continues to send A-MPDUs but the tag doesn't corrupt them, resulting in block ACKs that don't correspond to any value that we transmit. This ensures that even if the times in the sending access point and the laptop are not exactly the same, block ACKs corresponding to different values can be split into separate files for each value.

5.1.1 Synchronization for WiTAG Evaluation

Since multiple values can be transmitted within the same experiment, we differentiate which block ACKs correspond to which value by comparing the time of transmission from

a tag with the time of the block ACK's arrival on the sender. When collecting data for evaluation, the tag code includes *Serial.print* statements to display the timing of the value being transmitted from WiTAG in the serial window of Arduino IDE on the laptop. We copy and store the timing information into a separate file so that we can compare it with the block ACKs' timestamps. The timing of each block ACK arrival is recorded by the sender (i.e., the querying device). The timing synchronization is only used to test the accuracy of WiTAG for specific value transmissions. When WiTAG is deployed in a real-world application, synchronization is not used or necessary.

5.2 Experiment Execution

Using the laptop we log into the sending access point to begin sending WiFi packets. We use ifconfig to set the Maximum Transmission Unit (MTU) which sets the largest packet size that can be transmitted on the network interface to 256 bytes. We set the MTU size to 256 bytes so that the network layer fragments large packets into smaller chunks that can fit within MPDUs. The total MPDU payload size will be larger than the MTU size because it includes additional overheads such as a MAC header, Frame Check Sequence, and padding. We have carefully chosen the MTU size to ensure that the size of the MPDU will be 324 bytes (18 µs in duration as discussed in Section 4.1.3.)

We use iperf to send packets to the receiving access point using the following parameters:

- -c 192.168.2.1: This specifies the IP address for the receiving access point.
- -u: This option instructs iperf to use UDP for the test. Using UDP means that we don't need the receiver to run iperf allowing any compatible WiFi device to be used as a receiver.
- -b 150M: This sets the target bandwidth to 150 Mbps because we want to send packets as fast as possible in a short time for data collection. It exceeds the 144.4 Mbps PHY rate being used (MCS-15 with 20 MHz bandwidth and short guard intervals). During experimentation we observed that this rate is sufficient to fill the hardware queue.
- -l 20700: This option sets the UDP datagram size to 20,700 bytes. Since we set the MTU to 256 and each MPDU will be 324 bytes which is less than the UDP datagram size, packets will be fragmented into $ceil(20700/324) = 64$ MPDUs for transmission. This helps to create A-MPDUs with 64 MPDUs and improves the probability of an A-MPDU completely overlapping with a tag message.

- -P 3: This creates multiple parallel UDP streams using 3 processes so that packets are sent concurrently as fast as possible.
- -t 100000: This option specifies that iperf will run for 100,000 seconds (approximately 2 hours 46 minutes). We use a large value to ensure that we send packets for the entire duration of the test.

We also run tcpdump to record block ACKs coming from the receiving access point. We provide the following parameters to tcpdump:

- -XX: This enables a very verbose output that includes hexadecimal data representation along with additional information such as checksums and timestamps. We convert the hexadecimal values to binary later during data processing.
- -n: This option instructs tcpdump to avoid name resolution (converting IP addresses to hostnames). This improves performance.
- -i mon: This specifies that the network interface to listen on is the monitor interface which is used to capture traffic.
- type ctl: This is the capture filter, which defines which packets tcpdump will capture based on specific criteria. Here, type ctl instructs tcpdump to capture only control packets. WiFi control packets are used for network management and communication between devices. Most importantly, block ACKs are considered control packets.

We typically collect 5 minutes worth of data per value being transmitted and use that data for analysis.

5.3 Data Processing

After the experiment is complete, we parse the file that stores block ACKs and split it into separate files for each value based on the timestamps recorded from the tag. For example, if the tag reports it starts transmitting a value v_1 at time t_1 and ends transmitting a value v_1 at time t_2, we copy all block ACKs with time greater than t_1 and less than t_2 into a separate block ACK file for v_1. Then we perform analysis on each value (file) to study the accuracy of our encoding and decoding methods.

Chapter 6

Performance Evaluation

We aim to evaluate our design and implementation theories regarding synchronizations, encoding and decoding techniques, the Levenshtein distance algorithm, and message reliability. We first focus on ensuring the accuracy of our fine-grained synchronization mechanism before evaluating the coarse-grained synchronization. This approach helps to improve the accuracy of coarse-grained synchronization, which relies on the effective implementation of fine-grained synchronization. After evaluating the synchronization mechanisms, we determine the appropriate m value for encoding a bit using the Levenshtein distance algorithm. We then evaluate the effectiveness of the Levenshtein distance algorithm for decoding different values. Finally, we show how repeating messages can be leveraged to determine a value with a desired level of accuracy.

6.1 Fine-Grained Synchronization

6.1.1 Implementation Issue: Fine-Grained synchronization

Recall that in Section 4.2, we corrupt a group of MPDUs rather than just one (e.g., we corrupt for 18×3 duration to corrupt 3 MPDUs for transmitting 1 bit). The duration of 54 µs was chosen for a group of three MPDUs, where each MPDU duration is 18 µs. This value is important to keep in mind when considering the issue of tag corruption that we encountered during the evaluation of fine-grained synchronization. While measuring the corruption duration using an oscilloscope, we discovered that the tag was corrupting for about 57 µs instead of the intended 54 µs.

Upon further code review, we realized that the root cause of the issue was the way we were computing the duration for each bit we wanted to set, which delayed the change required for performing corruption or non-corruption. Specifically, before each bit transmission, we performed a multiplication operation to compute the corruption or non-corruption duration by multiplying the m value with the MPDU duration.

To shorten the 3 µs corruption difference, we pre-compute the duration calculation which is the same value for each bit we want to send. This approach avoided multiplication operations in between the tag's operation and made duration of a bit within 1 µs of the expected duration. We discovered that setting a high or a low pin on the Arduino may take time which contributes to 1 µs difference in actual versus expected corruption duration. The corruption duration was slightly under 54 µs and the non corruption duration was slightly above 54 µs. Despite these slight discrepancies, there was no noticeable effect on the number of corrupted MPDUs for each bit.

6.2 Coarse-Grained Synchronization

6.2.1 Block ACK Filtering

Our block ACK filtering mechanism is discussed in Section 4.5. We don't evaluate the filtering approach separately because it can be tied to our evaluation of coarse-grained synchronization. If we can match the expected and the actual message rates, we can infer that our block ACK filtering approach is acceptable. If we're filtering too many packets, the actual message rate will be lower than the expected rate and vice versa.

6.2.2 Implementation Issue: Coarse-Grained Synchronization

Initially, we used a 3 ms time delay (more than 2 times the A-MPDU and its overhead duration which totals 1429.5 µs), assuming that it would be sufficient to avoid corrupting A-MPDUs due to software retries. However, when sending messages of length of 23 ($b = 5$, $m = 3$, and $p = 3$) using this delay, we were not able to match the expected and the actual message rates.

When we checked the difference in the expected and the actual message rates using a different time delay (2.7 ms), we expected to get higher message rates using the smaller time delay. Surprisingly, we observed lower message rates than when using the larger delay

of 3 ms. These unexpected results led us to gain an insight that the choice of time delay may have a much different impact on message rates than we initially thought.

6.2.3 Methodology: Impact of Time Delay

Our goal is to understand the impact of time delay on the difference between the expected and the actual message rates. We transmit a 32-bit message using b, m and p values of 1, 24 and 3 respectively providing a message length of 32. The tag was placed 0.3 meters away from the receiver, between the sender and the receiver, to ensure good tag performance. We varied the time delay from 1.1 to 6.2 ms, in increments of 0.1 ms. This provides a good range of 51 different delays to gain insights into the corresponding change in message rates. Note that the graphs in Figures 6.1 and 6.2 have data points in between the x-axis markers. Each experiment runs for 2 minutes, allowing enough time to collect a sufficient number of packets. Using the experimental data, we compute the weighted probability of overlap (as explained in Section 4.1.2) and utilize that to obtain the expected message rates (as explained in Section 4.1.1). We then compared the expected message rates with the actual message rates observed in order to evaluate the difference between two rates.

6.2.4 Experimental Results: Impact of Time Delay

Figure 6.1 shows 95% confidence intervals for the expected and the actual message rates for time delays between 1.1 and 6.2 ms. In general, the expected message rate decreases for larger time delays. However, as can be seen in the figure, the actual message rates may differ significantly from the expected rates for smaller time delays. For example, for a time delay of 1.8 ms, the expected message rate is 76 messages per second, but the actual message rate is 163 messages per second, more than 2 times than the expected message rate. In contrast, for a delay of 2.6 ms, the expected message rate is 72 messages per second and the actual message rate is 44 messages per second. In this case, the actual message rate is much lower than expected. Our theory for computing the expected message rates assumed that the query packets are independent of each other because random backoffs and channel sensing are required before sending each packet. However, in reality it turns out that the query packets aren't necessarily independent. In Section 6.2.7, we will explain how the choice of a time delay can result in actual message rates that are much higher or lower than expected.

Additionally, the peaks and the valleys seen for actual rates, get closer to the expected rates as the delay increases. Even though the expected and the actual message rates get

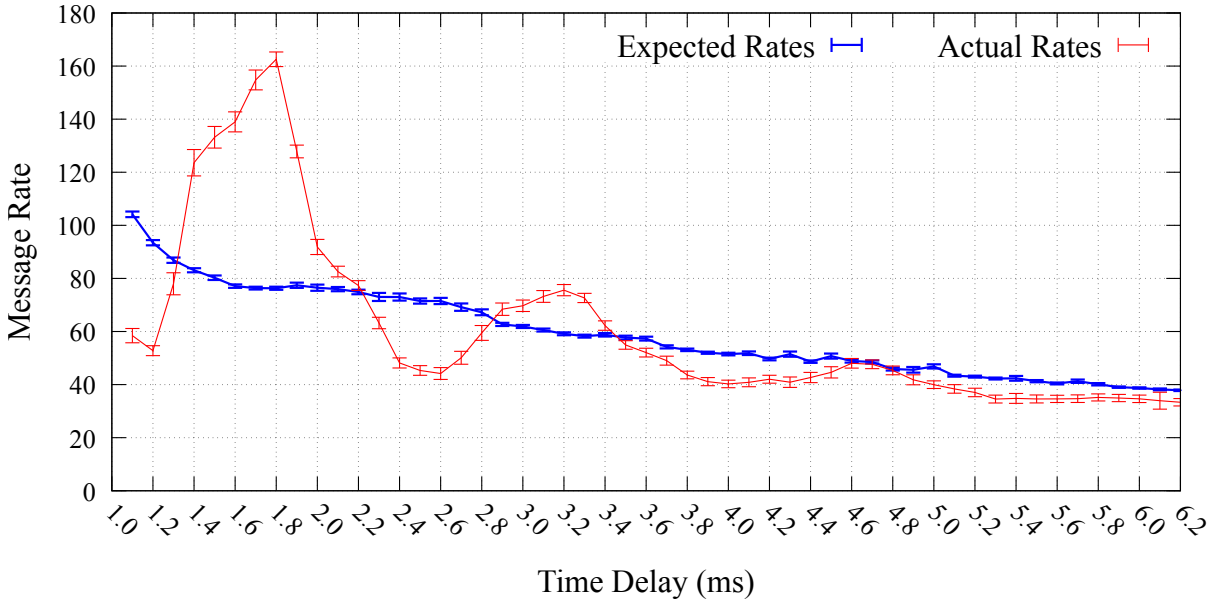


Figure 6.1: Expected and actual message rates for time delays ranging from 1.1 to 6.2 ms (message length is 32)

closer with larger delays, a large time delay is not desirable because both rates are low.

If these experiments are repeatable, it suggests that it may be possible to achieve higher than expected message rates, if we choose the right time delay. Figure 6.2 shows results from conducting similar experiments on three different days. As can be seen, the relationship between delay and message rates is fairly consistent and repeatable.

Further, similar phenomena can be observed even when there is other traffic on the network. Figure 6.3 shows the expected and actual message rates when streaming one, two, and three YouTube videos simultaneously. In general, as expected, message rates drop when streaming more videos. Additionally as the number of video streams increases, access to the network for the querying device becomes more random and actual rates are closer to expected rates. Despite external traffic, by using an appropriate time delay we can get higher than expected message rates. As shown in the figure, the actual message rates are higher than the expected message rates when using good time delays of 1.6 and 1.8 ms and when there are one or two video streams of traffic. With three querying devices, traffic is random enough that there are no statistically significant differences between the expected and actual rates.

Next, we want to know if picking a good time delay can give high message rates even

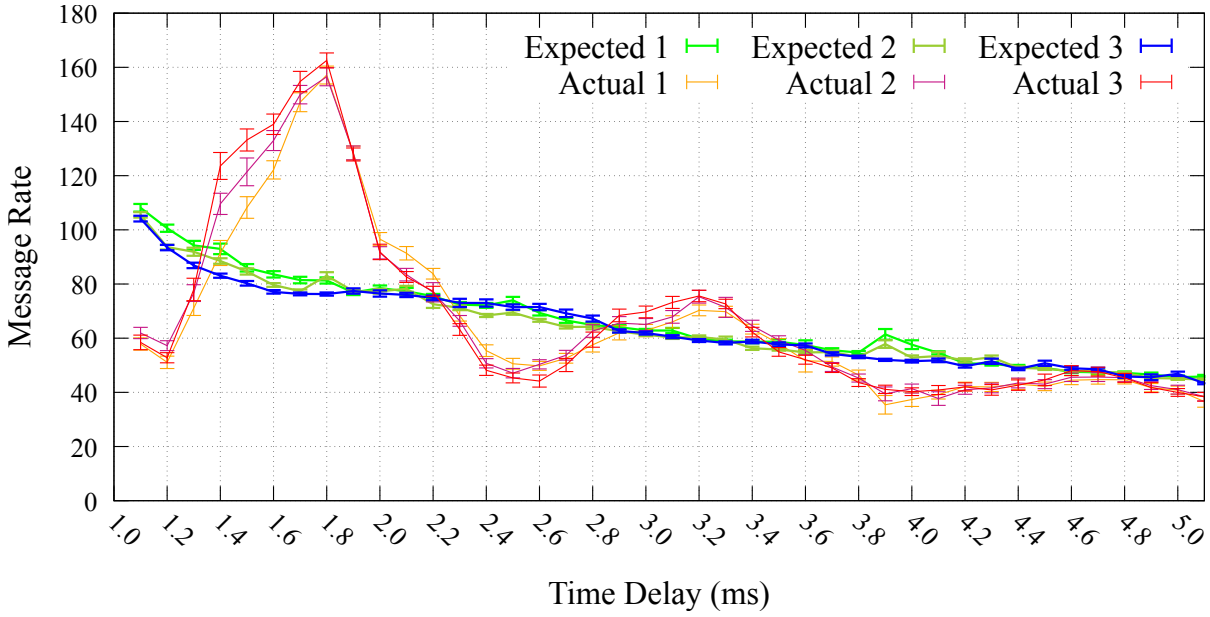
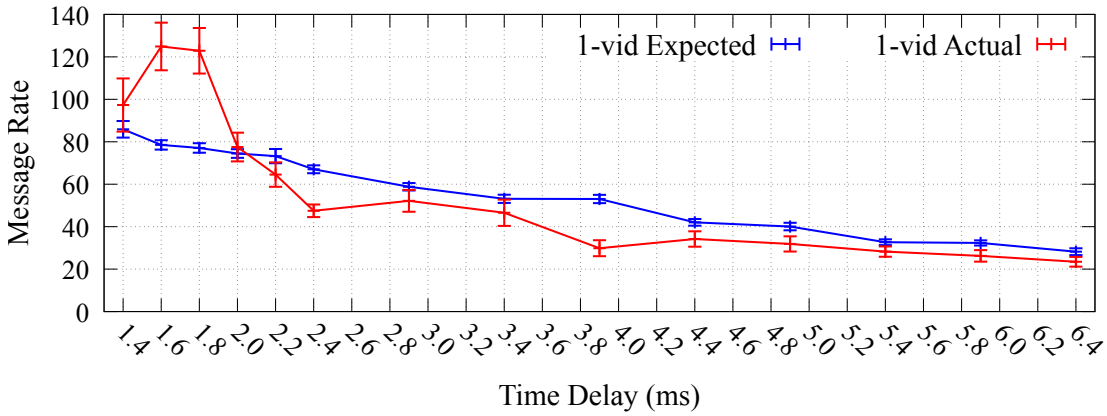


Figure 6.2: Expected and actual message rates for time delays ranging from 1.1 to 5.1 ms, repeated three times (message length is 32)

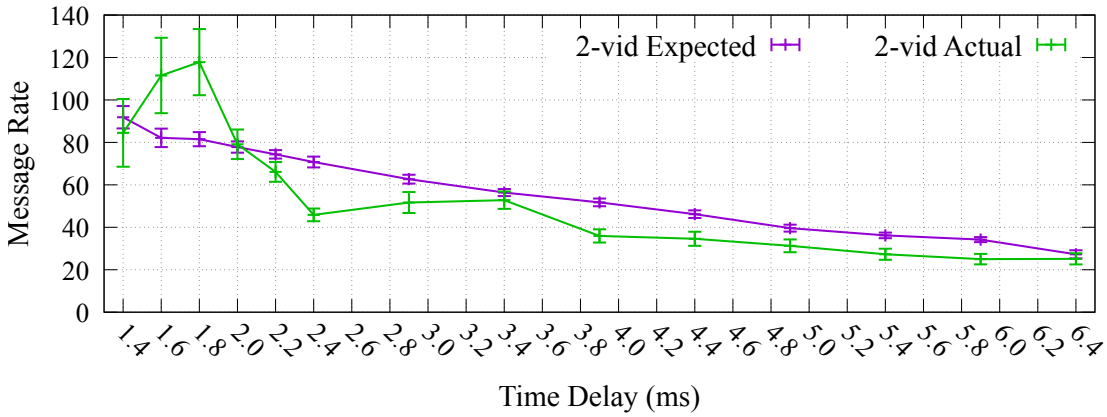
with a change in the message length because the expected probability of overlap changes with the message length, which changes the message rate. We want to see how much a change in message length impacts actual message rates and if we can get higher-than-expected rates for different message lengths.

6.2.5 Message Length, Time Delay, and Message Rate Relationships

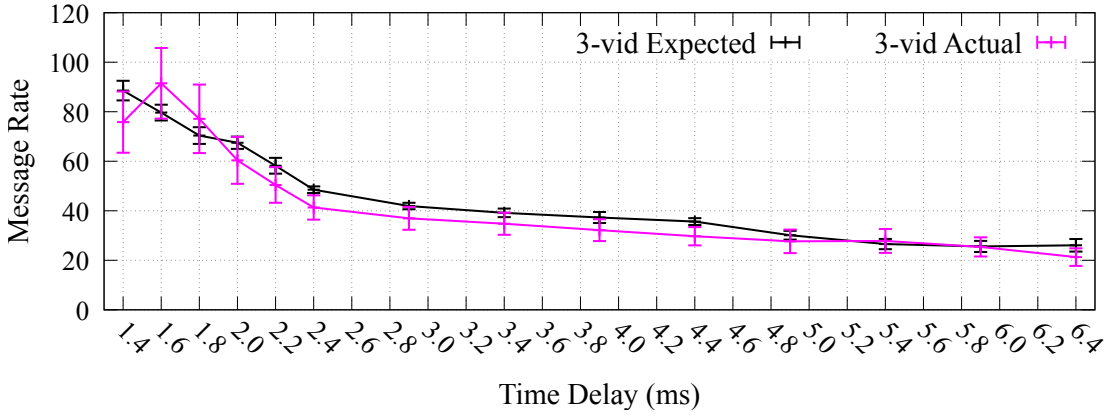
We now investigate how a change in message length may impact the actual message rate for different delays. Using a methodology similar to Section 6.2.3 but with a message length of 16 ($m = 8$, $p = 3$, and $b = 1$), we see in Figure 6.4 that while the message length has some effect on how time delays impact actual message rates, the locations of peaks and valleys (i.e., delays used) remain similar to those observed with a message length of 32 (Figure 6.1). This indicates that it may be possible to select a delay that may yield higher-than-expected message rates, regardless of message length. In future work, we would conduct a more comprehensive analysis of the impact of message lengths on the most suitable delay for different message lengths.



(a) Streaming one video



(b) Streaming two videos



(c) Streaming three videos

Figure 6.3: Expected and actual message rates when YouTube video traffic is present (message length is 32)

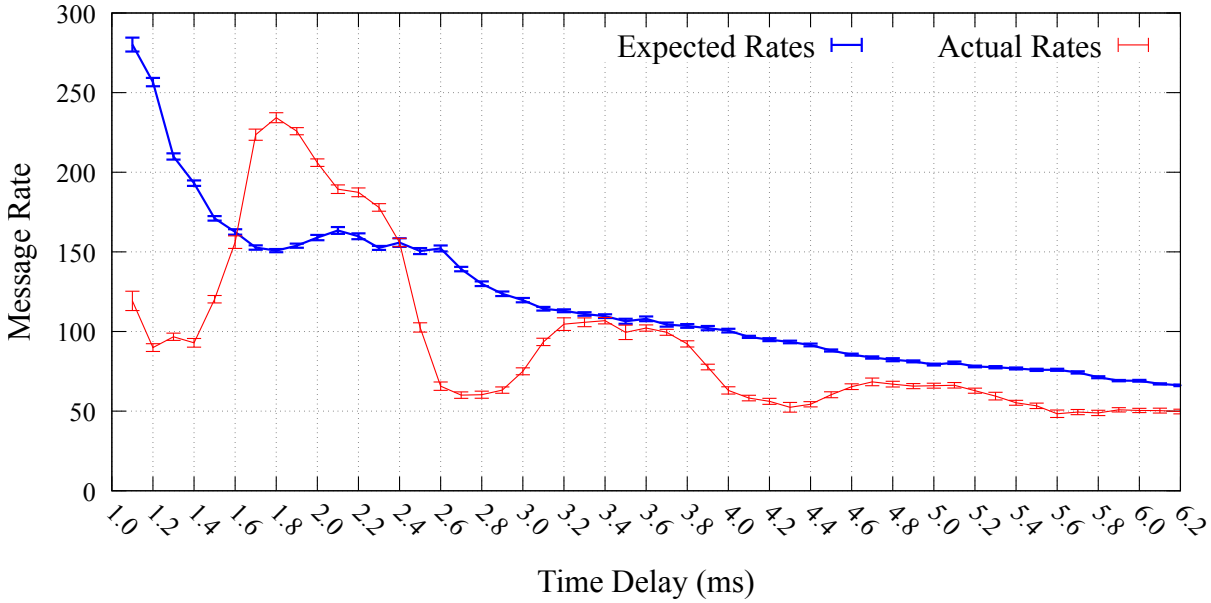
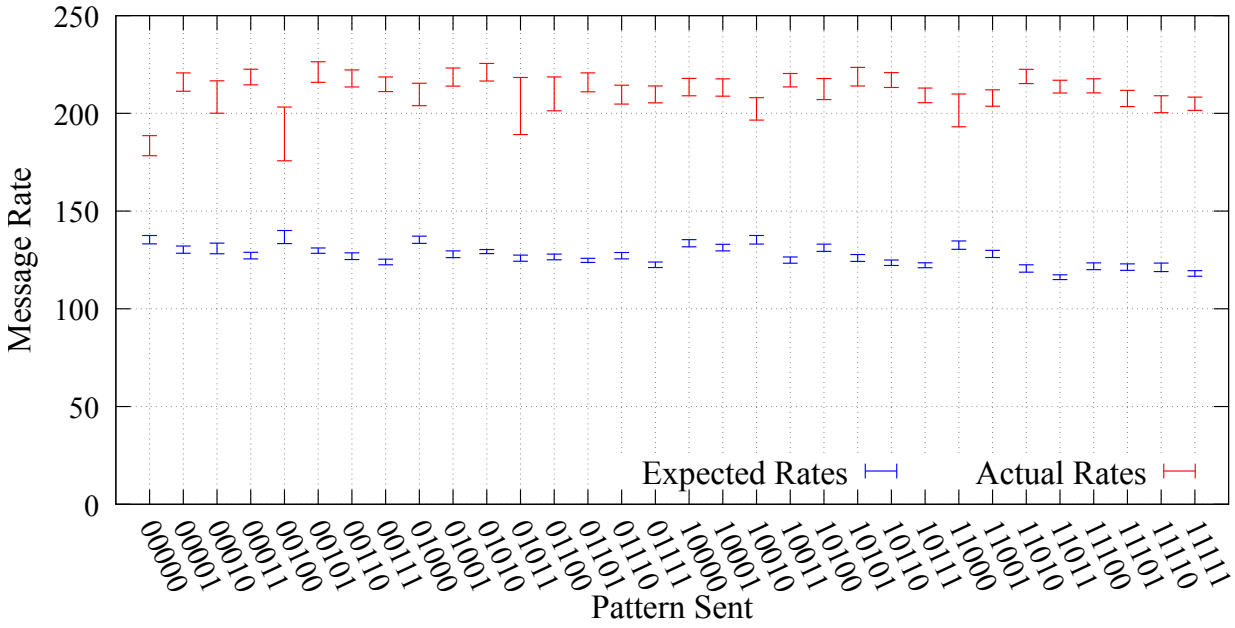


Figure 6.4: Expected and actual message rates with 95% confidence intervals for time delays ranging from 1.1 to 6.2 ms (message length is 16)

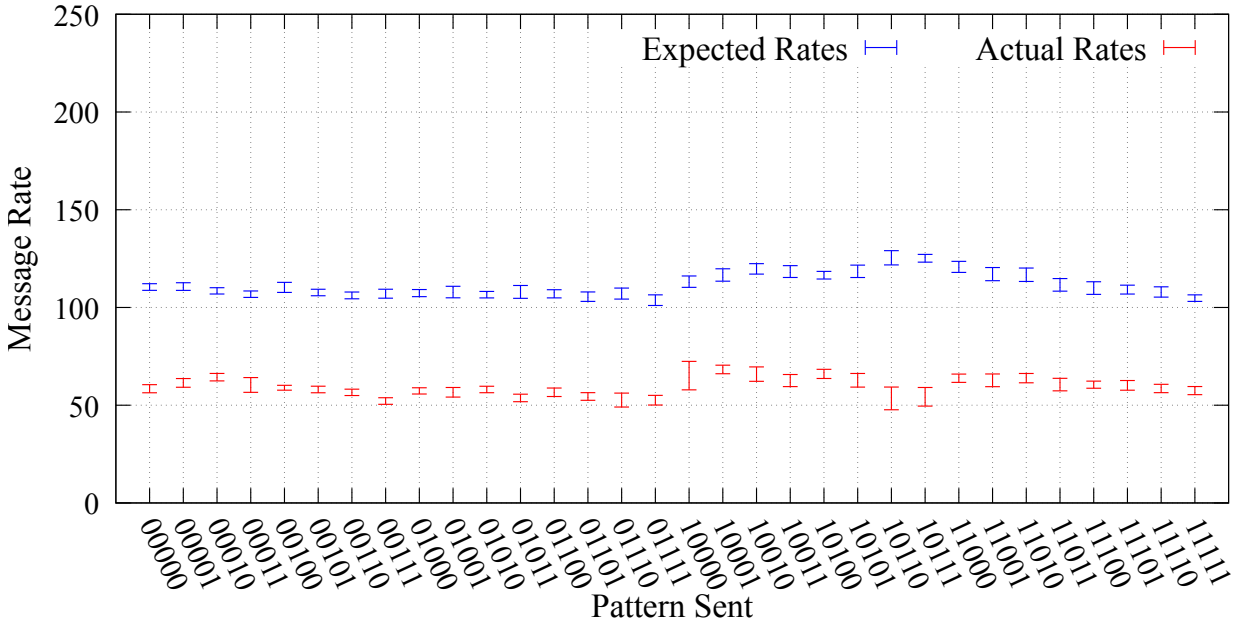
6.2.6 Relationships Between Message, Time delay and Message Rate

We noticed that there can be some variation in message rates for different values. Therefore, we examine the impact that bit patterns for different values might have on message rates. Our main goal here is to determine if we can consistently get higher-than-expected rates despite variations in message rates for different values. Different bit-patterns require the tag to corrupt different number of MPDUs which can impact the length of software retries. We want to ensure that if we pick a good time delay, it will consistently give better than expected rates for different bit-patterns that the tag transmits. To investigate the relationship between bit pattern, time delay, and the actual message rate, we use a similar experimental setup as in Section 6.2.5, but with different parameters. We set the b value to 5, the m value to 3 and the p value to 3 which makes the message length (ml) 23. With b equal to 5, we can send 32 different patterns and determine if the pattern being sent impacts the actual message rate.

Using the peaks and valleys from Figure 6.1 and Figure 6.4, we conduct experiments with time delays of 1.8 ms and 2.7 ms and measure message rates for each of the 32



(a) 1.8 ms Delay



(b) 2.7 ms Delay

Figure 6.5: 95 % confidence intervals for expected and actual message rates for 32 patterns using two different time delays

different values. Figure 6.5 shows the 95 percent confidence intervals for the message rates using two different time delays and 32 different patterns. In Figure 6.5a, we observe that for the 1.8 ms delay, actual message rates for different bit patterns vary, but all of them are consistently and significantly higher than expected rates. In this experiment, although some patterns such as 00000 and 00100 have lower actual rates compared to other patterns, their actual rates are still much higher than their expected rates. Conversely, in Figure 6.5b, for the 2.7 ms delay, actual message rates for all patterns are consistently lower than expected.

Our study demonstrates that selecting an appropriate time delay may allow one to obtain higher than expected message rates and that this likely holds for different bit patterns.

6.2.7 Choosing a Time Delay Impacts Synchronization

As depicted in Figure 6.2, our expectation was for the message rate to decrease with an increase in time delay. However, our experiments indicate that in reality there is no linear correlation between time delay and message rate. We now examine this relationship.

Figure 6.6a shows the interaction between three WiFi packets and two tag messages (labeled Msg and shown in blue). A WiFi packet is comprised of an A-MPDU with overhead (OH). An A-MPDU consists of one or more MPDUs which are shown in green boxes. The yellow box to the left of the MPDUs represents A-MPDU overheads (including the PLCP header) and the yellow box to the right of the MPDUs represents SIFS and block ACK overhead (see Figure 4.1 for a detailed view). The first tag message partially overlaps with the first WiFi packet at the j^{th} MPDU. This results in a software retry (the second WiFi packet) consisting of $j + \delta$ MPDUs. Because the first j MPDUs are received successfully (and the block ACK window advances by j MPDUs) and δ MPDUs that were corrupted need to be retransmitted, the software retry can contain $j + \delta$ MPDUs.

In Figure 6.6b, the first tag message completely overlaps with the first WiFi packet at the k^{th} MPDU. This results in a software retry consisting of $k + \delta$ MPDUs. Since k is smaller than j, the software retry can include new MPDUs up to a smaller sequence number. Therefore in Figure 6.6b, the number of MPDUs in the software retry will be smaller than Figure 6.6a and the subsequent full A-MPDU is sent a little earlier.

To avoid overlapping with software retries, the tag uses a time delay (d_1) between subsequent tag messages. Figure 6.6 demonstrates that a well-chosen time delay (d_1) causes the tag to skip a software retry and completely overlaps its next message with a full A-MPDU. This happens whether there is a partial or a complete overlap for the first A-MPDU. We call this a good synchronization. Figure 6.7 differs from Figure 6.6 as it

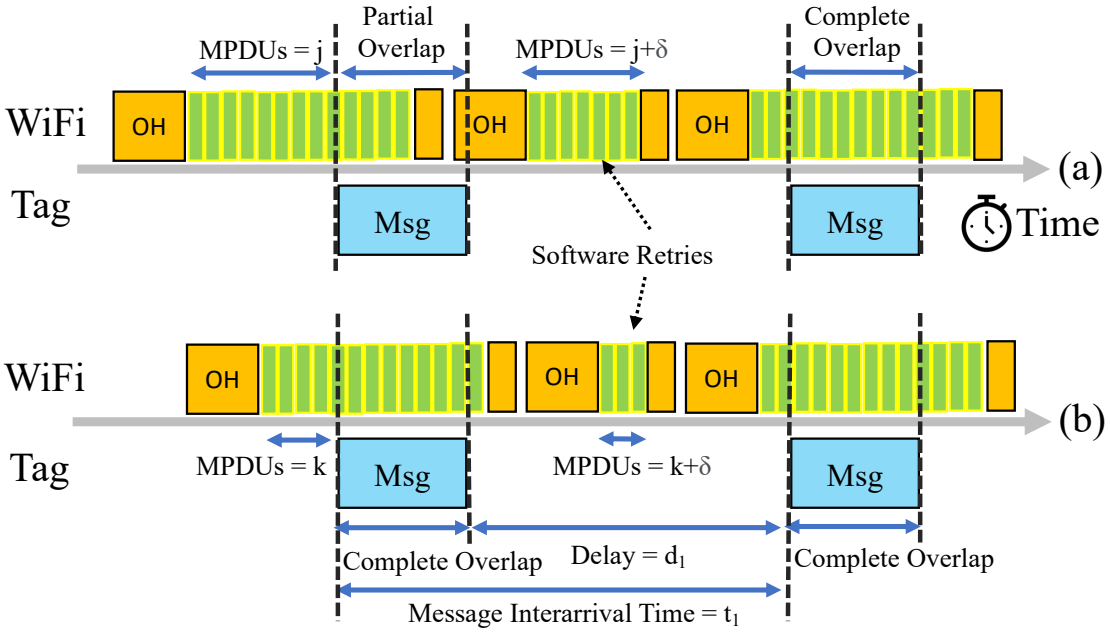


Figure 6.6: Good synchronization because of a well-chosen time delay

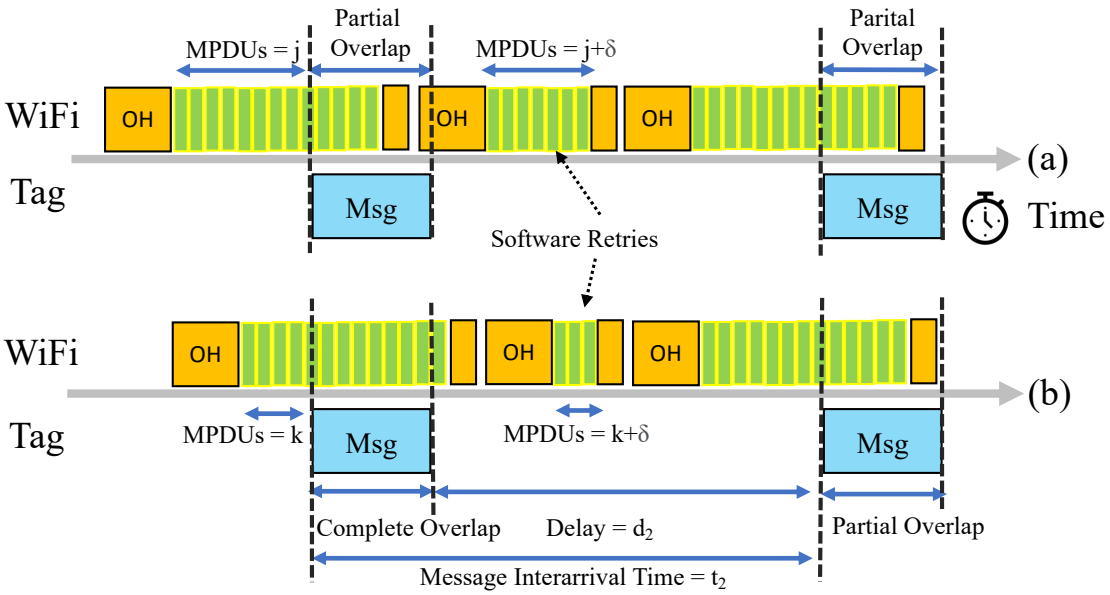


Figure 6.7: Bad synchronization because of a poorly chosen time delay

uses a different time delay (and therefore a different message inter arrival time) resulting in a bad synchronization. The time delay is denoted with d_1 in Figure 6.6 and d_2 in Figure 6.7. Note that d_1 is not equal to d_2. The time between the start of two consecutive tag messages is the message inter arrival time. It is denoted as t_1 in Figure 6.6 and t_2 in Figure 6.7. Note that t_1 and t_2 are also not equal. Figure 6.7 demonstrates that a poorly chosen time delay (d_2) causes the tag message to partially overlap with the next full A-MPDU. This happens regardless of how the tag message overlaps with the first A-MPDU. We call this a bad synchronization.

By choosing a suitable time delay, tag messages can achieve good synchronization, leading to more frequent complete overlaps with WiFi packets. Conversely, a poorly chosen delay results in bad synchronization, decreasing the likelihood of complete overlaps and decreasing the message rate.

6.3 Bit Encoding

6.3.1 Methodology: Different Values for Bit Encoding

We conduct a series of experiments to examine how varying the number of MPDUs to encode a bit (i.e., value of m) affects the reliability of a transferred message. We set the preamble and postamble (p) values to 3 for all experiments, as this is sufficient for filtering messages.

We set the number of bits (b) to 5, to be able to measure the impact of m on 32 different patterns. We evaluate the decoding accuracy of each of the 5-bit value patterns transmitted for values of m (2, 3, 4 and 5). The tag transmits values starting with 0 for a duration of t time units, followed by values 1, 2, ..., $2^b - 1$, each transmitted for t time units. This approach means we know the intended value transmitted by the tag and we can compare it with the decoded value.

To ensure the consistency of our results, we conduct experiments on different days and at varying distances. We run experiments with the sender and the receiver placed 6 meters apart and a tag placed at distances of 0.3 meters and 1.5 meters from the receiver.

We utilize the Levenshtein distance algorithm to evaluate decoding accuracy, as it is simple and suitable for measuring the similarity between a transmitted value and the intended value. We select 2,000 consecutive packets from each experiment as a sufficient number to accurately measure the decoding accuracy.

6.3.2 Experimental Results: Different Values for Bit Encoding

The worst-case decoding accuracy is determined as the minimum percentage of messages correctly decoded among all 32 5-bit values. Table 6.1 presents the worst-case decoding accuracy corresponding to each m value. The first column is the number of MPDUs used to encode a bit (m). The second column is the message length (ml) for five bits of data ($b = 5$) using three bits for preamble and postamble ($p = 3$), m bits to encode each bit and 2 bits for fenceposts.

An m value of 2 provides a message length of 18. When the distance between the tag and the receiver is 0.3 meters, it gives the worst-case decoding accuracy of 3.050%. Similarly, using an m value of 5 gives a message length of 33. When the distance between the tag and the receiver is 1.5 meters, it obtains the worst-case decoding accuracy of 96.400%.

For $m = 2$, the accuracy is very low, indicating that this is not a viable choice. The accuracy for $m = 2$ is low because in the most common case where 3 MPDUs are subject to corruption (as shown in Figure 3.3b), the first and the last MPDU are partially subjected to corruption and may not represent the desired value. For example in Figure 3.3b, there is a possibility that block ACK values for MPDU i and $i + 2$ may be 1's when sending a value of 0. As a result, the value may be incorrectly decoded to a 1 because two out of the three bits in the block ACK are set to 1. For $m \geq 3$, there should be 2 or more consecutive MPDUs with the desired value, reducing a potential source of ambiguity. Our results suggest that quite good accuracy can be achieved at different distances using m values of 3, 4 or 5.

The selection of the appropriate value of m depends not only on the desired decoding accuracy but also on the message rate, which is affected by the message length. Thus, analyzing the message rate for different m values can help to determine the suitable m value for encoding a bit. As shown in the table, increasing an m value increases the message length, decreases the message rate, and does not improve accuracy. Therefore, an m value of 3 is suitable for encoding a bit.

6.4 Message Decoding

Recall that in Section 4.6, we described how the Levenshtein distance algorithm could be used to decode a message. We now evaluate our encoding and decoding schemes using this algorithm.

m	Msg Length	Msgs/Second	0.3 m (%)	1.5 m (%)
2	18	280	3.050	10.850
3	23	232	97.100	98.100
4	28	195	95.000	97.550
5	33	162	97.600	96.400

Table 6.1: Experimental results showing the worst-case decoding accuracy for $b = 5$ using different m values at 0.3 m and 1.5 m distances. Message rates are obtained using a 1.8 ms time delay

6.4.1 Methodology: Levenshtein Distance Algorithm

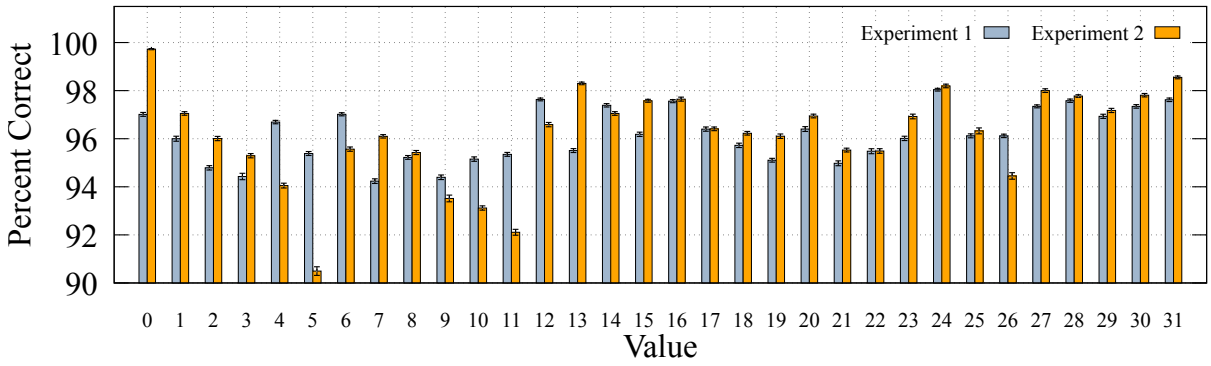
We perform four different experiments. Two experiments were performed with a WiTAG device positioned at 0.3 meters from the receiver and two at 1.5 meters. For all experiments, the sender and the receiver are 6 meters apart and the tag is placed in between them. Each distance was replicated on different days to account for environmental variations. We use m and p values of 3 for efficient bit encoding and a b value of 5 to study 2^b (32) patterns. We evaluate different patterns to study if some patterns are easier to encode or decode. Using 32 different values is a sufficient number for testing while remaining small enough to conduct experiments at two distances.

After collecting 2,100 messages per pattern, we employ a sliding window technique to ensure that all potential message groupings are considered. This technique is used to create 2,000 subgroups of 100 messages each. 100 is a large enough window to compute an average and 2000 is sufficiently large to produce confidence intervals. Using the decoding accuracy of each subgroup, we compute the 95 percent confidence intervals for each of the 32 values. This helps us achieve a comprehensive evaluation of the encoding and decoding accuracy.

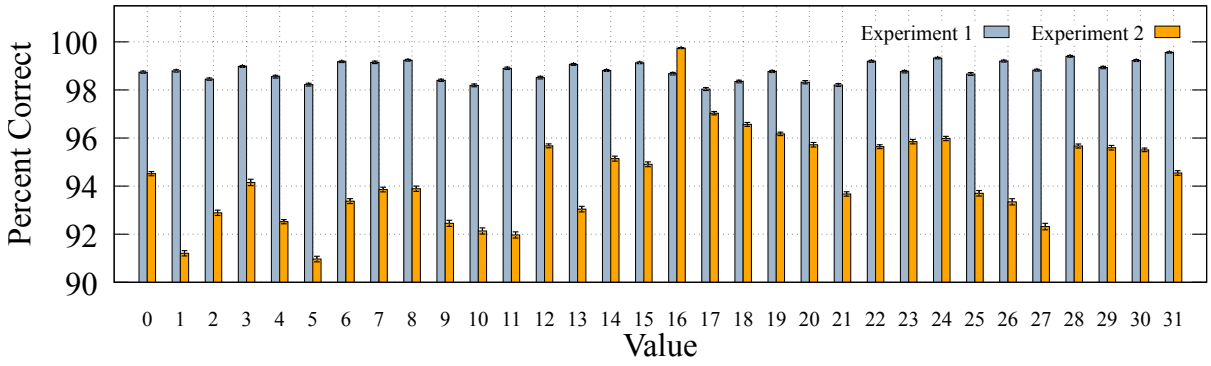
6.4.2 Experimental Results: Levenshtein Distance Algorithm

Figure 6.8a and Figure 6.8b show the decoding accuracy using the Levenshtein distance algorithm for each of the values from 0 to 31. Note that the confidence intervals may be difficult to see for some values because the lower and upper bounds are quite tight.

In Figure 6.8a, the decoding accuracy varies across patterns in both experiments. For example, Experiment 2 has higher decoding accuracy for values zero, one, two, and three, but Experiment 1 has higher decoding accuracy for nine, ten, eleven, and twelve. Five has the lowest accuracy in Experiment 2 whereas seven has the lowest accuracy in Experiment 1.



(a) 0.3 m distance



(b) 1.5 m distance

Figure 6.8: Decoding accuracy using the Levenshtein distance algorithm for 32 different values. Note that the y-axis covers values from 90 to 100 (zoomed in) to allow us to see the differences. The accuracy is above 90% for all values in all experiments.

In Figure 6.8b, Experiment 1's accuracy doesn't vary much for different values but the accuracy varies significantly in Experiment 2. Experiment 1 is more reliable than Experiment 2 for all values except sixteen. It is also more reliable than both experiments in Figure 6.8a which are conducted at 0.3 m distance.

Overall, no values are more or less reliable. Despite variations in accuracy across experiments, all values attain over 90% success regardless of the distance. This makes the Levenshtein distance algorithm a reasonable choice for decoding values sent by WiTAG. For large b values, our approach of using the Levenshtein distance algorithm to compute 2^b edit distances will be inefficient as we will discuss in Section 8.2.1. Next, we show that we can increase reliability by using multiple messages.

6.5 Message Reliability

In Section 4.7, we described how multiple messages could in theory be used to increase reliability. We now assess how the number of messages used affects reliability.

6.5.1 Methodology: Using Multiple Messages

We evaluate message reliability using the same experimental data used in Section 6.4. After running the experiments to transmit a value, we store the block ACKs for that value in a file. Then, we traverse the file, adding the first N messages to a queue. Next, we utilize a plurality majority voting algorithm to determine the most likely value for those N messages and check for accuracy.

Applying the sliding window approach, we add the $N + 1^{th}$ message to the queue and remove the first added message from the queue. The sliding window mimics the reality that the window of N messages used could start from any message. We repeat this approach of updating the window and determining a value 2000 times. Finally, we analyze the percentage of the value determinations that are correct.

We perform the entire process described above for all 2^b values and report on the value with the lowest accuracy. This is because the system will only be as reliable as the least reliable value.

6.5.2 Experimental Results: Using Multiple Messages

The results obtained from our experiments are summarized in Table 6.2. The column labeled N represents the number of consecutive messages used when decoding values. The remaining columns display the percentage of time N values result in a correct decoding. For example, using $N = 3$ for the 0.3 m distance, out of 2000 groups of 3 messages, 98.850% were decoded correctly for Experiment 1 and 97.500 % were decoded correctly for Experiment 2.

As seen in Table 6.2, increasing the number of messages (N) increases accuracy significantly. The maximum group size (N) needed to attain 100% accuracy is 15 for the second experiments at both distances. Recall that Table 6.2 presents the minimum accuracy from all 2^b values. Other values will be much better, some reaching 100% accuracy after only 3 messages. In this case, 100% accuracy means that in our experiments all 2000 groups were correct in the dataset we evaluated. Experiments across various days and distance faced

	Distance = 0.3 m		Distance = 1.5 m	
N	**Expt 1 (%)**	**Expt 2 (%)**	**Expt 1 (%)**	**Expt 2 (%)**
1	94.300	90.500	98.100	91.000
3	98.850	97.500	99.650	97.800
5	99.600	98.850	99.950	99.250
7	99.900	99.500	100.000	99.700
9	100.000	99.850	100.000	99.750
11	100.000	99.950	100.000	99.850
13	100.000	99.950	100.000	99.900
15	100.000	100.000	100.000	100.000

Table 6.2: Improving reliability with multiple messages

fluctuating channel conditions, interference from devices, and human movement. Despite these obstacles, WiTAG can determine a value with remarkable accuracy using surprisingly few messages.

	Time (ms) to send N messages						
Msg Rates	**1**	**5**	**10**	**15**	**20**	**25**	**30**
60	17	83	167	250	333	417	500
80	13	63	125	188	250	313	375
100	10	50	100	150	200	250	300
120	8	42	83	125	167	208	250
140	7	36	71	107	143	179	214
160	6	31	63	94	125	156	188
180	6	28	56	83	111	139	167

Table 6.3: Time to send different numbers of messages based on different message rates

Table 6.3 shows the time (in milliseconds) required to transmit the corresponding number of messages (N) at a specific message rate (msgs/s). For example, with a message rate of 120 messages per second, it will take 125 ms to get 15 messages. For a fixed number of messages (N), higher message rates lead to shorter transmission times.

As seen in Figure 6.5, using a good time delay of 1.8 ms, we can get a rate of at least 175 messages per second for a 23-bit message (i.e., $b = 5$). This message rate only requires 86 ms to get 15 messages. Even for a longer message containing 32 bits (i.e., $b = 8$), Figure 6.2 shows that with a time delay of 1.8 ms, we can get a rate of about 160 messages per

second. For comparison, this message rate requires 94 ms to get 15 messages. With three videos running as seen in Figure 6.3, we can get a rate of 80 messages per second for a 32-bit message. This rate requires 188 ms to get 15 messages. This means a high degree of accuracy can be achieved with few messages in a relatively short period of time, making it possible to utilize multiple tags in the same environment.

Chapter 7

Related Work

Because our work is specific to WiTAG, we now briefly describe other existing WiFi backscatter systems to provide some context for WiTAG and describe why it is an interesting system to work with. As discussed in the SIGCOMM paper on WiTAG [5], WiFi backscatter [23] is the first backscatter system that enables communication with commodity WiFi devices. Unfortunately, due to the self-interference between WiFi transmission and backscatter signals the range of that system is very limited. BackFi [7] and Passive WiFi [24] are two systems that try to increase the range of communication, however they require specialized hardware which hinders the widespread deployment of these systems.

Dehbashi et al. [11] compare WiTAG with three additional backscatter systems. They explain that HitchHike [39], FreeRider [40], and Moxcatter [41] send a backscatter signal to a different access point in order to avoid the self-interference problem [4]. All of these systems require software modifications on the second access point to receive the backscattered packets. Additionally, they modify physical-layer symbols and will not work if the network uses encryption [5] (and encryption is currently used in almost all WiFi networks).

Overall, WiTAG provides better range than WiFi backscatter, works with commodity WiFi devices as opposed to BackFi and Passive WiFi and works with encryption as opposed to HitchHike, FreeRider, and Moxcatter. We now describe and compare various approaches to encoding, decoding, and reaching consensus. Future exploration of alternatives may improve message reliability.

7.1 Bit Encoding

Using the minimum number of MPDUs to encode a bit reduces message length and enables the use of more bits for the actual data. Shorter messages also increase the probability of complete A-MPDU overlap and help to achieve higher message rates. While WiTAG's fine-grained synchronization requires at least two MPDUs per bit, as discussed in Section 6.3, our experiments show low accuracy with $m = 2$. Using a larger m value achieves higher accuracy.

We did not notice a significant increase in accuracy beyond $m = 3$, so encoding a bit using more than 3 MPDUs does not appear to be necessary. Adding a parity bit for each bit encoded will increase the message length by twice because we need to use 3 MPDUs to also encode each of the parity bits. This will significantly reduce the probability of a complete A-MPDU overlap and the message rate. Therefore, exploring alternative bit encoding approaches appears unwarranted.

7.2 Message Encoding

WiFi devices that use 802.11n and 802.11ac standards allow A-MPDUs lengths up to 64 MPDUs. Adding extra encoding bits to encode a message requires more MPDUs and can reduce the complete A-MPDU overlap probability and the resulting message rate. Similar to bit encoding, we want to encode a message using a small number of MPDUs in order to ensure a high probability of a complete A-MPDU overlap. Various approaches for error detection and correction can be utilized to encode a message.

Error detection techniques can be useful to discard messages with errors so that we don't make an incorrect determination by decoding them. We could incorporate a parity bit to discard messages with mismatching parity [8]. Adding an extra parity bit is simple and only requires m MPDUs, however, it can only detect single-bit errors. It is not clear that interference would only impact MPDUs that encode data for 1 bit so a parity bit alone may not add much benefit to WiTAG. Alternatively, checksums calculated by the sender and verified by the receiver could be used. A checksum is more complex to implement and requires more bits compared to parity bits, but it can detect multiple-bit errors. Cyclic Redundancy Check (CRC) polynomials are mathematical expressions that can be used to generate complex checksums. If the CRC's don't match, the receiver knows that some MPDUs are altered in the message that it receives and can discard the message. CRC 32 generates a 32-bit checksum [1]. Adding 32 bits with $m = 3$ requires 96 MPDUs,

more than the number of MPDUs that are available in the entire A-MPDU. Fletcher's 16-bit checksum is more lightweight as it only adds 16 bits [30], however, using $m = 3$ it increases the message length by 48 MPDUs and significantly reduces the probability of an A-MPDU overlap. If the checksum length exceeds the value of b, padding bits are appended to facilitate the checksum algorithm to operate with the necessary number of bits. For b values smaller than eight, an excessive number of padding bits would be necessary to produce 16 or 32-bit checksums. Therefore, Fletcher's 8-bit checksum [2] is preferable for b values less than 8 since it demands fewer padding bits. Using $m = 3$, this algorithm requires 24 MPDUs (still a very large number of MPDUs) to generate checksums. Rather than using checksums that require many bits for a reliable multi-bit error detection, WiTAG might benefit from error-correcting codes.

Instead of discarding messages with errors, error-correcting codes can correct bit-patterns with some errors to determine the transmitted message value. Hamming (7,4) [18] and Reed-Solomon (RS) codes [36] can provide both error detection and correction capabilities. For occasional bit flips, a Hamming (7,4) encoding offers single-bit error correction by expanding a 4-bit message to 7 bits. While effective, it fails when there are multiple bits flipped and can introduce complexity if the total number of bits isn't evenly divisible by four [18]. To encode 5 bits, a Hamming code adds 3 check bits to the data bits, making the total number of bits 8, which is a multiple of 4. Those eight bits are then encoded into a 14-bit codeword which requires 42 MPDUs with $m = 3$. This approach adds many MPDUs, significantly reduces the probability of an A-MPDU overlap, and can only correct any one bit error out of the four bits that are encoded.

RS codes excel against burst errors common in communication channels but suffer from significant overhead and computational complexity [36], making them not very suitable for our needs. RS codes add redundant information, called parity symbols, to a message. For RS codes to offer higher error correction, they require more parity symbols. One can use RS(n,b) to encode b bits into a codeword of length n and correct $(n-b)/2$ errors [25]. For minimal error correction, we can encode 5 data bits ($b = 5$) into a codeword of length 7 ($n = 7$). This code can correct up to $(7-5)/2 = 1$ errors. Using $m = 3$, 7 bits require 21 MPDUs. Similarly, if we encode $b = 5$ into a codeword of length 9 ($n = 9$), we can correct up to $(9-5)/2 = 2$ errors. For maximum error correction, we can encode 5 data bits ($b = 5$) into a codeword of length 15 ($n = 15$). This code can correct up to $(15-5)/2 = 5$ errors. Using $m = 3$, 15 bits require 45 MPDUs. Depending on how noisy we expect the environment to be, we can add a different number of parity symbols to correct 1 through b bit errors. We want to use enough bits to improve the message reliability but not too many because they add more MPDUs, reducing the probability of a complete A-MPDU overlap and lowering the message rate.

In the future, we could study combinations of different approaches to using a preamble and postamble as well as parity bits, checksums, Hamming (7,4) and Reed-Solomon Codes to improve message recognition. The techniques described in this section may become useful on WiFi access points using the 802.11ax standard because their A-MPDUs will support up to 256 MPDUs. Due to the large number of MPDUs in an A-MPDU, even after adding extra bits for message encoding, the probability of an A-MPDU overlapping with a tag message will likely still be sufficient in comparison to 802.11n and 802.11ac standards with only 64 MPDUs.

7.3 Decoding

We want to use a decoding approach that effectively determines the transmitted value using a sequence of 0's and 1's. Algorithms that compute edit distance are a natural fit for finding differences between two sequences. We compute the edit distance between the received message data and all possible values from zero to $2^b - 1$.

The Hamming distance algorithm calculates the number of positions at which the corresponding symbols are different in two equal-length strings [19]. Due to the challenge of fine-grained synchronization, we do not know if ml or $ml + 1$ MPDUs will be required to encode ml bits. This means the length of the transmitted message may be one more than the length of sequence corresponding to the expected values. Since the patterns we are comparing could be of different lengths, we can't use the Hamming distance algorithm.

The Levenshtein distance algorithm can measure the similarity between two sequences of different lengths by calculating the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into the other [21, 31, 32]. This algorithm addresses our challenge of determining the correct value from comparing a received message of $ml + 1$ length to each of the 2^b possible value patterns of ml length.

The Damerau-Levenshtein distance algorithm is an extension of the Levenshtein distance algorithm that also allows for transpositions (adjacent character swaps) [10]. This algorithm is useful when transpositions are common in the type of data being compared. Transpositions may be possible in a potential message because of varying channel conditions. Sometimes when we want to corrupt, the tag may not and sometimes when we don't want to corrupt MPDUs may be corrupted by interference. However, we expect that message transposition occurs rarely and the Damerau-Levenshtein distance algorithm is not essential for decoding values from WiTAG.

The Levenshtein distance algorithm is quite suitable for our decoding process. It effectively handles potential length variations and works well for accurate message interpretation. The issue with comparing sequences using edit distance algorithms such as the Levenshtein distance algorithm is that they are inefficient for large b values. The time to compute distance from a sequence message to each of the 2^b patterns increases exponentially with b. In the future, a neural network algorithm could work as an alternative to overcoming the time inefficiency that is caused by computing 2^b edit distances.

7.4 Consensus

To improve reliability, we have implemented the use of multiple messages, each with their own edit distance. As a result, we now describe techniques for reaching a consensus, a general agreement from a group [26]. Different consensus mechanisms can be used to determine a value using multiple messages. Given the potential for WiTAG to operate in diverse environments, some with significant noise, we consider plurality majority voting as the most suitable consensus mechanism. This algorithm appears to be able to determine the correct value even when there are several incorrect messages due to noisy conditions.

Unanimity voting requires all participants to agree on a single option often leading to a more robust decision [13, 33]. This mechanism would be useful if the number of incorrect decodings in our experiments was very low. While this approach provides a reliable value determination, even a single incorrect message would require us to discard all N messages. Since WiTAG messages are susceptible to environmental factors, we anticipate (and have seen in our experiments) a frequent occurrence of errors, rendering unanimous voting impractical.

In absolute majority voting, the chosen value requires more than 50 percent of the votes [14, 27, 26]. This approach can handle errors as long as more than half of the votes are correct. However, this voting mechanism won't work well in noisy environments where 50 percent or more of the messages are decoded incorrectly. If no value has 50 percent of the votes, we would have to discard all N messages requiring extra message transmissions from WiTAG.

Plurality majority voting only requires a chosen value to have more votes than any other options [14]. Even when there are a majority of incorrect messages due to a noisy environment, we may be able to determine a value correctly. We don't have to throw out all N messages because there are a few incorrect messages.

Weighted voting assigns different weights to individual votes based on factors such as

expertise or importance. The option with the highest weighted sum is chosen [34, 17]. In the future, a weighted voting mechanism that uses the edit distance between a message and each of the 2^b values as weights could be implemented. For multiple messages, we would sum the edit distances between the message and each of the 2^b possible values and choose the value that would yield the smallest sum.

Chapter 8

Conclusions and Future Work

8.1 Thesis Summary

WiTAG is a backscatter communication system for low-power IoT devices that avoids power-hungry receivers and relies on a burst of query packets that are sent periodically using an existing WiFi device. It operates at the MAC layer by selectively corrupting subframes within WiFi packets. The corruption indicates a 0 and a non-corruption indicates a 1 and the resulting block ACK from the WiFi device contains the tag's message. Since WiTAG cannot directly sense nearby WiFi transmissions and it needs to corrupt MPDUs for data transmission, it faces several challenges in transmitting and decoding messages.

The timing of WiFi transmission is unknown to WiTAG as it doesn't use a signal detector. This presents a coarse-grained synchronization challenge where part or all of WiTAG's corruption may occur when there is no WiFi packet. It also presents a fine-grained synchronization problem causing two MPDUs to often be partially corrupted when trying to only corrupt one.

Since partial corruption of an MPDU may or may not succeed, it is not sufficient to use a single MPDU to encode a bit. We face the challenge of determining how many MPDUs to use to encode a bit with the need to balance between reliability in encoding and using a minimum number of MPDUs to maximize the number of bits we can transmit. Because many WiFi packets don't completely coincide with a tag's corruption, we need to encode a WiTAG's message data to distinguish a block ACK that may contain a WiTAG message from other block ACKs.

After the tag encodes a message using MPDUs, the sender receives the corresponding block ACK from a receiver. We need to filter block ACKs that do not contain a tag's

message so that we don't incorrectly decode those block ACKs. Once a message is identified, we need to decode the message data reliably despite some ambiguities that may be caused by environmental factors. Since there are several challenges that are inherent during different stages of message transmission, we need to ensure message data can be decoded reliably.

We focus on overcoming these challenges to establish reliable data transmission from a WiTAG device. We design and implement several techniques to understand, and resolve existing challenges. We develop theories for computing the probability of an A-MPDU completely overlapping with a tag message (coarse-grained synchronization) and expected tag message rates using important parameters such as the duration of an A-MPDU, its overhead, message length, and the time delay between messages. We determine that we can overcome the challenge of fine-grained synchronization by corrupting/not corrupting for a duration of at least two MPDUs per bit. Using this approach, we determine that 3 MPDUs works quite well. We denote the number of bits that we need to transmit as b and the number of MPDUs that are used to encode each bit as m.

Since a default (uncorrupted) block ACK contains all 1's, to encode a WiTAG message, we extend the message by adding a 0 bit at the beginning (preamble) and the end (postamble). Since encoding a 0 bit is similar to any other bit, we use m MPDUs to encode the preamble and postamble. We denote these values using (p) and also add two fenceposts, a 1 before the preamble and a 1 after the postamble. These fenceposts help to identify partial overlaps where a tag's corruption may start before the first MPDU or end after the last MPDU. With our design, the message length is the sum of preamble and postamble ($2 \times p$), the number of bits in the message data (b) times the number of MPDUs used to encode each bit (m), and the 2 fencepost bits (i.e., $(2 \times p) + (b \times m) + 2$).

Given the message length we can determine the range of minimum and maximum number of zeros a message could contain. For example, sending the value zero would require at least $(2 \times p) + (b \times m)$ zeros and sending the value $2^b - 1$ would require at least $(2 \times p)$ zeros. To eliminate block ACKs that did not completely overlap with a message and may result in incorrect decoding, we filter the block ACKs with too few zeros, too many zeros, and unexpected message lengths. Once a block ACK that may contain a message is identified, we extract the message data by removing the preamble, postamble and fenceposts. We compute the edit distance from the received message data to all 2^b possible values and determine the correct value as the one with minimum edit distance. Due to the fine-grained synchronization challenge, the message data will often have a length of one more than a bit pattern corresponding to a value. We employ the Levenshtein distance algorithm which can compute edit distance by handling potential length variations in sequences being compared.

Periodically, the querying WiFi device repeatedly sends A-MPDUs to provide a tag with target to use for messages. To increase the message reliability, we utilize message repetition. To determine a value from N messages, we use a plurality majority voting consensus mechanism. This technique chooses the value with the most votes and may determine a value correctly even when there are incorrect individual decodings. Since WiTAG may need to operate in environments with high levels of interference, we want to use a mechanism that can handle errors and doesn't require us to discard a whole group of messages due to a few errors.

We conduct several experiments to evaluate the performance of WiTAG. Initially, to provide enough time for software retries to complete, we conduct experiments with a time delay added between messages. Due to unexpected message rates, we experiment using different time delays and find a novel technique that allows us to increase the probability of overlap and achieve higher than expected message rates. We discover that choosing a good time delay increases the probability of overlap and the message rate. We conduct experiments over multiple days under different environmental conditions.

We obtain above 90 percent decoding accuracy using the Levenshtein distance algorithm for four different experiments run using different distances and on different days with varying environments. This demonstrates that our encoding and decoding techniques work quite well. Finally, we show that it is possible to significantly improve message reliability by using multiple messages which can be transmitted in a relatively short time frame. Even when streaming three YouTube videos simultaneously on the network WiTAG uses, we find that it is possible to get excellent message rates and high reliability.

We were surprised by the nonlinear impact of time delay on message rates, how a single message's reliability doesn't improve when encoding a bit with more than 3 MPDUs, and how quickly we can get higher message reliability by using more messages. We have now studied the impact of time delay between message transmissions, number of MPDUs to encode a bit, identifiers for distinguishing a message, and an algorithm for encoding and decoding. Overall we are able to determine combinations of parameters that allow us to reliably decode WiTAG's messages in a relatively short window of time.

8.2 Future Work

In this section, we present ideas for future work that may improve WiTAG's message transmission process. Implementing these ideas may enhance message rates, encoding and decoding accuracy, decoding speed, the distance at which WiTAG devices can communicate and their usability in the real world.

8.2.1 Using a Neural Network for Larger Messages

Recall that the Levenshtein distance algorithm compares a potential message with all 2^b values and computes the edit distance. For smaller b values such as $b = 5$, this process of comparison is relatively quick. However, as b increases, the complexity for examining all 2^b values for each message increases exponentially. This will become computationally inefficient for large b values.

A neural network could be utilized to improve runtime performance by training a model with large sets of data beforehand. The challenge with using a neural network is that it requires a lot of data and time to train the model. Because experiments need to be run to collect data for all 2^b values, this can take a considerable amount of time. Using neural networks enables us to spend time training the model so that the value determination becomes efficient. In one environment we have conducted a small experiment and the results were encouraging because once trained the neural network worked well in a variety of environments.

8.2.2 Message Encoding to Improve Reliability

The 802.11n and 802.11ac standards limit the size of an A-MPDU to 64 MPDUs. Introducing a new encoding scheme would require additional MPDUs, significantly reducing the chance of an A-MPDU overlap and a tag's message rate. However, the 802.11ax standard uses up to 256 MPDUs per A-MPDU. With more MPDUs other encoding schemes that add extra bits such as Reed-Solomon codes might be more attractive because adding some extra bits may still provide sufficient overlap probabilities and message rates. A different encoding technique may reduce or eliminate the need to rely on multiple messages for reliable value determination.

8.2.3 Antenna Enhancement to Improve Distance

We expect that improved and backscatter specific antenna design may help to improve the communication range for WiTAG. Generally, larger antennas offer better performance but are not feasible for WiTAG due to size constraints. Directional antennas help to focus signals in a specific direction and are ideal for long-range communication between fixed points. Additionally, higher gain antennas amplify the signal strength for improved range and reduced power consumption. Ultimately, since WiTAG can be used in various types

of applications, we may wish to design custom antennas for different use cases and/or environments which may improve both gain and directionality.

In addition, the range of WiTAG devices may be improved further by solving the self-interference problem. The problem is that since the query signal is much stronger than a tag's reflected signal it can dominate the signal at the receiver and limit WiTAG's range. This problem may be mitigated by using the nulling capability in WiFi devices. Nulling is achieved by using multiple antennas for transmission and adjusting the phase of signals transmitted from each antenna. The phases are adjusted in a way that transmitted signals are added destructively where the receiver is located [4].

8.2.4 Supporting Multiple Tags

We have been able to successfully use WiTAG in a real world setting when combined with a light sensor and have created a demo video using a single WiTAG device [3]. Future work would be to study the use of multiple WiTAG devices and the density of tags that could be supported within the same environment.

We envision multiple tags operating near each other in an environment such as a house or a retail store. One of the challenges of multi-tag operation is tag identification. One solution would be to use some bits at the beginning of each message for a tag identifier. In such a scenario, the more bits we use as a tag identifier the more tags we can place in an environment. Therefore, more tags would require longer messages, decreasing message rates. As a result, the density of tags that could be supported would depend on the frequency with which they would communicate and the number of bits required for their messages.

8.2.5 Quantify Power Savings

Adding a signal detector to WiTAG would increase power consumption by two to three orders of magnitude. The benefits of such an addition is that the tag would not send messages when there are no packets in flight and once a packet is detected it might be easier to determine the exact timing for each MPDU (possibly simplifying bit encoding). However, signal detectors do not eliminate the need for tags to repeat messages. For example, if a non-querying device sends a packet, the signal detector would prompt the tag to encode its message on that packet, resulting in the querying device not receiving the tag's message. Since signal detectors cannot distinguish the source of packets, repeated transmissions are still necessary.

Additionally, even if the packet detected by a signal detector is for the querying device, due to changing channel conditions a tag's corruption may or may not work. This means the tag will still have to repeat messages to ensure reliability. We believe that there is a significant advantage to avoiding signal detectors using WiTAG but a detailed comparison in energy usage with and without a signal detector could be performed in the future.

8.3 Concluding Remarks

There is significant potential to utilize WiFi compatible backscatter communication system to develop efficient communication using low-power IoT devices. In this thesis we have proposed an initial framework to enhance WiTAG's communication by addressing basic challenges of synchronization, encoding, decoding and message reliability. We believe our work has improved WiTAG's suitability for use as a WiFi compatible backscatter communication system that supports the use of multiple tags within the same environment.

References

[1] Cyclic redundancy check - Wikipedia. https://en.wikipedia.org/wiki/Cyclic_redundancy_check.

[2] Fletcher's checksum algorithm - Wikipedia. https://en.wikipedia.org/wiki/Fletcher%27s_checksum.

[3] WiTAG light sensor demo video. https://cs.uwaterloo.ca/~brecht/witag/.

[4] Ali Abedi and Omid Abari. Can WiFi backscatter achieve the range of RFID? nulling to the rescue. In *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*, HotNets '21, page 171–177, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3484266.3487372.

[5] Ali Abedi, Farzan Dehbashi, Mohammad Hossein Mazaheri, Omid Abari, and Tim Brecht. WiTAG: Seamless WiFi backscatter communication. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 240–252, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3405866.

[6] Ali Abedi, Mohammad Hossein Mazaheri, Omid Abari, and Tim Brecht. WiTAG: Rethinking backscatter communication for WiFi networks. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 148–154, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3286062.3286084.

[7] Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. Backfi: High throughput WiFi backscatter. *SIGCOMM Comput. Commun. Rev.*, 45(4):283–296, aug 2015. doi:10.1145/2829988.2787490.

[8] William Bolton. *Programmable Logic Controllers*. Newnes, 2015.

[9] Colby Boyer and Sumit Roy. Invited paper — backscatter communication and RFID: coding, energy, and MIMO analysis. *IEEE Transactions on Communications*, 62(3):770–785, 2014. doi:10.1109/TCOMM.2013.120713.130417.

[10] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, Mar 1964. doi:10.1145/363958.363994.

[11] Farzan Dehbashi, Ali Abedi, Tim Brecht, and Omid Abari. Verification: can WiFi backscatter replace RFID? In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, page 97–107, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3447993.3448622.

[12] K.L. Du and M.N.S. Swamy. *Wireless Communication Systems: From RF Subsystems to 4G Enabling Technologies*. Cambridge University Press, 2010. URL: https://books.google.com/books?id=5dGjKLawsTkC.

[13] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr 1985. doi:10.1145/3149.214121.

[14] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, Oct 1985. doi:10.1145/4221.4223.

[15] Mattew S. Gast. *802.11n: A Survival Guide*. O'Reilly, 2012.

[16] Mattew S. Gast. *802.11ac: A Survival Guide*. O'Reilly, 2013.

[17] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, page 150–162, New York, NY, USA, 1979. Association for Computing Machinery. doi:10.1145/800215.806583.

[18] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950. doi:10.1002/j.1538-7305.1950.tb00463.x.

[19] Richard W Hamming. *Coding and Information Theory*. Prentice-Hall, Inc., 1986.

[20] 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. doi:10.1145/2934872.2934894.

[21] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, USA, 1st edition, 2000.

[22] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R. Smith, and David Wetherall. Wi-fi backscatter: internet connectivity for rf-powered devices. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 607–618, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2619239.2626319.

[23] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R. Smith, and David Wetherall. WiFi backscatter: internet connectivity for RF-powered devices. *SIGCOMM Comput. Commun. Rev.*, 44(4):607–618, aug 2014. doi:10.1145/2740070.2626319.

[24] Bryce Kellogg, Vamsi Talla, Joshua R. Smith, and Shyamnath Gollakot. Passive WIFI: Bringing low power to WiFi transmissions. *GetMobile: Mobile Comp. and Comm.*, 20(3):38–41, Jan 2017. doi:10.1145/3036699.3036711.

[25] Ralf Koetter. *Error Correction Coding: Algebraic Codes*. Springer Science and Business Media, 2004.

[26] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, Dec 2001. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[27] Leslie Lamport, Robert Shostak, and Marshall Pease. *The Byzantine Generals Problem*, page 203–226. Association for Computing Machinery, New York, NY, USA, 2019. URL: https://doi.org/10.1145/3335772.3335936.

[28] Jilong Liu, Mingwu Yao, and Zhiliang Qiu. Adaptive A-MPDU retransmission scheme with two-level frame aggregation compensation for IEEE 802.11n/ac/ad WLANs - wireless networks, Jul 2016. URL: https://doi.org/10.1007/s11276-016-1330-z.

[29] Yunfei Ma, Nicholas Selby, and Fadel Adib. Minding the billions: Ultra-wideband localization for deployed RFID tags. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, page 248–260, New

York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3117811.3117833.

[30] Theresa C. Maxino and Philip J. Koopman. The effectiveness of checksums for embedded control networks. *IEEE Transactions on Dependable and Secure Computing*, 6(1):59–72, 2009. doi:10.1109/TDSC.2007.70216.

[31] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, Mar 2001. doi:10.1145/375360.375365.

[32] Saul B Needleman and Christian D Wunsch. General method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970. URL: https://doi.org/10.1016/0022-2836(70)90057-4.

[33] Hannu Nurmi. *Comparing Voting Systems*, volume 3. Springer Science & Business Media, 2012.

[34] B. Parhami. Voting algorithms. *IEEE Transactions on Reliability*, 43(4):617–629, 1994. doi:10.1109/24.370218.

[35] Swadhin Pradhan, Eugene Chai, Karthikeyan Sundaresan, Lili Qiu, Mohammad A. Khojastepour, and Sampath Rangarajan. RIO: A pervasive RFID-based touch gesture interface. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, page 261–274, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3117811.3117818.

[36] Sudhakar Radhakrishnan and Muhammad Sarfraz. *Coding Theory*. IntechOpen, Rijeka, Mar 2020. doi:10.5772/intechopen.77427.

[37] B. Sadeghi, V. Kanodia, A. Sabharwal, and E. Knightly. Opportunistic media access for multirate ad hoc networks. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, MobiCom '02, page 24–35, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/570645.570650.

[38] Wei-Liang Shen, Yu-Chih Tung, Kuang-Che Lee, Kate Ching-Ju Lin, Shyamnath Gollakota, Dina Katabi, and Ming-Syan Chen. Rate adaptation for 802.11 multiuser MIMO networks. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, page 29–40, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2348543.2348551.

[39] Pengyu Zhang, Dinesh Bharadia, Kiran Joshi, and Sachin Katti. Hitchhike: Practical backscatter using commodity WiFi. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, page 259–271, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2994551.2994565.

[40] Pengyu Zhang, Colleen Josephson, Dinesh Bharadia, and Sachin Katti. Freerider: Backscatter communication using commodity radios. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 389–401, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3143361.3143374.

[41] Jia Zhao, Wei Gong, and Jiangchuan Liu. Spatial stream backscatter using commodity WiFi. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, page 191–203, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3210240.3210329.