# WiSpider

Authors: Anish Singhani, Thomas Horton King, Ethan Oh
Affiliation: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—With the increasing ability to build tiny wireless-connected IoT devices, there is massive potential for hostile actors to infringe on individual, corporate, and governmental privacy using hidden wireless devices such as microphones and cameras. We want to create a device that can detect and localize these hidden wireless devices. We will also exploit wireless protocols to obtain some information about the hidden device, such as manufacturer or device purpose. We will create a visual user-interface to display where any detected devices are located, relative to the user. Given the pervasiveness of WiFi for communications, our prototype/MVP will primarily focus on sniffing WiFi packets.**

***Index Terms*—Privacy, WiFi Localization, Internet-of-Things, WiFi Security, 802.11**

# 1 INTRODUCTION

With the rapid expansion and adoption of wireless IoT (Internet-of-Things) devices, it has become easier for hostile actors are to spy on and infringe the privacy of individuals and corporations. In Surveillance devices can take the form of cameras, microphones, or presence detectors, which are either (i) miniaturized to the point of being unnoticeable to the human eye or (ii) disguised as harmless devices, such as wall plugs or light bulbs. Existing technologies to address this problem either focus on detecting magnetic signatures specifically emitted by cameras, which are unable to detect other kinds of surveillance devices, or are designed as 'wands' that measure the Received Signal Strength (RSS) at a certain bandwidth, which can be fooled by intermittently transmitting devices or low transmit power IoT devices.

In this report, we propose WiSpider, a platform to detect and localize these hidden, hostile wireless devices non-cooperatively. Specifically, we focus on sensing devices operating on the WiFi band, which makes up a large part of the IoT market [3]. To address the shortfalls of previous implementations, we will passively sniff device addresses from the air, then force channels between our product and target devices by exploiting the 802.11 WiFi protocol. We will then extract the channel information over a series of user movements and measurements, and aggregate those measurements into an AR (Augmented Reality) interface which shows the user where we suspect the hidden devices to be. To target the majority of off-the-shelf IoT spyware, we intend our product to be used by both individual and corporate users.

# 2 USE-CASE REQUIREMENTS

## 2.1 Constraints

**Sensing WiFi:** While IoT devices span a diverse range of wireless frequencies and modalities, including and BLE (Bluetooth Low-Energy), we restrict our implementation to specifically sensing WiFi devices in 2.4GHz band for this proof of concept, because of the ease-of-use and wide availability for non-advanced attackers using commercial off-the-shelf devices.
**Indoors:** Our system will be designed and tested solely indoors. This is where the majority of privacy-infringing devices are located, and indoor environments are inherently more challenging for wireless sensing due to obstructions and dense multipath, so the system could be generalized to outdoors.
**No moving devices:** We are focused solely on sensing devices permanently or semi-permanently planted in the environment - not computer, phones, or laptops, which may change position with their user.

## 2.2 Metrics

**Weight and Size:** We require WiSpider to *weigh less than 10 pounds* and *fit within 1 cubic foot*. We envision our device being easy for user to carry, to move between rooms as they are scanning, and to pack and take with them if they are going to a different place. To this end, our device needs to be both lightweight and small.
**Cost:** We require WiSpider to *cost less than $150*. This places us in-line with most currently available commercial solutions and puts WiSpider in the price range of both private and corporate users.
**Detection Rate:** We require a *device detection rate of 90%*, where the detection rate is defined as how often we are able to detect a device that is located within our minimum range. If a device's address is observed or sniffed, it would count as being detected.
**Scan time:** We require WiSpider to *finish scanning within 5 minutes*. This enables the end-user to detect devices in a relatively short period of time.
**Lateral accuracy:** We require WiSpider to *localize a device within 1m from its actual location*. The user can then easily search manually within the detection zone.
**Range:** We require a *minimum detection range* of 10m, meaning that devices up to 10 m away can be detected by WiSpider. This is to allow a user to scan an entire room easily and with little movement (i.e. without needing to manually sweep across every wall).
**Resolution:** WiSpider requires a *spatial resolution of 0.5m*, measuring how well the product is able to distin-

guish between co-located devices. Wireless devices may be very closely located to each-other: in order to provide the user with information about devices that could be hiding near another signature, we need to be able to distinguish closely located devices from each other.

### 2.3 Features

**Detect and locate hidden devices:** The main feature of WiSpider is to detect and locate hidden devices. The device will sniff WiFi packets to detect devices, and exploit Polite Wifi[1] behavior to gather Time-of-Flight data. These data will be used to localize devices.

**Non-cooperative:** It is likely that a user will want to scan for hidden IoT devices in an unknown environment. WiSpider is able to operate in such situations where the user is not connected to the same network that the IoT devices are connected to.

**Visualization:** WiSpider must be able to show the user where any detected devices are located in the scanned space.

We acknowledge there is potential for this technology itself to infringe on the privacy of network users by giving their location to an unpriveledged third party. However, this method has already been developed as a vector for attackers to use in other projetcs [2]. Additionally, we believe that the restriction that WiSpider will only localize immobile devices will make our product much more appealing in the toolkit of a defender than an attacker.

## 3 ARCHITECTURE

### 3.1 Overall System Diagram

Our architecture has 3 major components: a hardware component, using WiFi chips and a microcontroller; a signals component, processing those measurements using gradient descent and an MLE (Maximum-Likelihood Estimation) localization algorithm; and a software component, creating the AR interface and integrating all of WiSpider's devices.

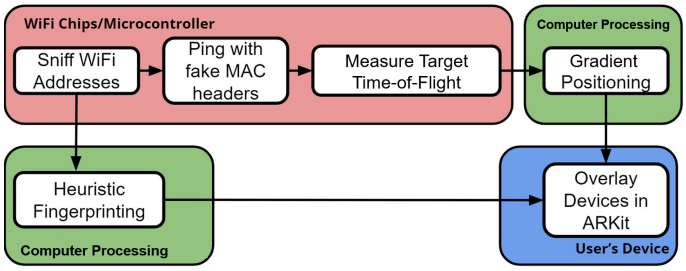


Figure 1: The system diagram of WiSpider, showing the overall pipeline and where each component runs.

The operating steps of WiSpider are as follows: (i) Sniff WiFi packets and addresses passively from the surrounding area (ii) Ping those addresses to non-cooperatively create a channel between WiSpider and the target devices using the "Polite WiFi" exploit described in [1]. (iii) Convert measurements into Time-of-Flight ranging and aggregate across user movements to localize where the devices are. (iv) Visualize to the user where the devices are located in the space, and any information we can glean from their unencrypted packet headers (i.e. manufacturer, data rate). This overall pipeline is shown in Figure 1.

### 3.2 Hardware Diagram

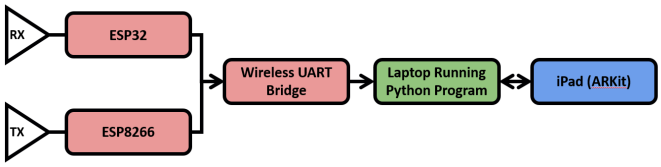


Figure 2: WiSpider operates 2 different WiFi chips simultaneously to extract different measurements of the channel.

Due to the limitations of individual WiFi chips, which will be further described in Section 6, we elect to use two WiFi chips to sense the channel, as shown in Figure 2. At a high level, one will be used solely to ping target devices with null packets, and the other device will be used to measure the time between outgoing packets and incoming responding acknowledgment packets.

### 3.3 Localization Diagram

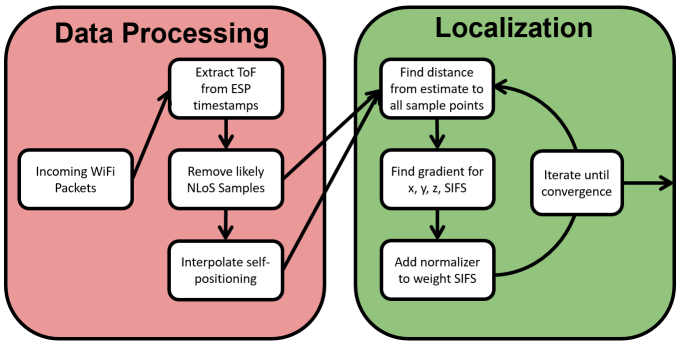


Figure 3: WiSpider extracts information from the WiFi channel to localize the target in 3 dimensions using an iterative gradient descent optimization.

Figure 3 shows the outline of how WiSpider handles the measurements it receives. In short, WiSpider takes a sample starting position and calculates the possible total error when a small step is taken in each dimension from that

position. It combines those measurements into a gradient and steps to the new lower-error position. WiSpider then iterates this process until the position converges into a final measurement.

# 4 DESIGN REQUIREMENTS

**Physical:** As stated in the use-case requirement, WiSpider needs to be both lightweight and small, and our design should weigh less than 10 pounds and fit within 1 cubic foot.
**Sniffing rate:** WiSpider should sniff and identify access points for 1 minute, and should use fake beacon packets to discover devices connected to each access point for another minute.
**Injection rate:** 100 fake packets per second should be sent.
**ToF measurement rate:** Since half of the fake packets should be fake beacon packets to prevent the device from going to sleep, there should be 50 ToF measurements per second.
**ToF accuracy:** It is hard to test the ToF accuracy in real setting. Therefore, in a test setting with clear line of sight, we expect that the distance calculated from the measured ToF is off from the actual distance of the device by no more than 2 meters. The testing method is further discussed in section 7.
**Distinguish devices:** WiSpider should distinguish different devices by obtaining each device's unique identifiers, such as its MAC address, bandwidth, or transmission rate. This should work even if the devices are physically close together.
**Locate infrequently transmitting devices:** Even if a device transmits infrequently, if the device is detected at least once, WiSpider should be able to locate it.
**Self-localization accuracy:** Our self-localization error should be within 0.3 meters such that tracking devices can be done to an accuracy of 1 meter across a large number of measurements.
**AR accuracy:** In order to make it very clear for a user where a device is located, the AR visualization error should be within 0.3 meters of the actual calculated location. The AR visualization should also be smooth (even if there is some error or offset, there should be minimal jitter) so that the user has a comfortable experience looking at it.

# 5 DESIGN TRADE STUDIES

In reaching our current design, we explored a variety of dimensions of the design space, both in terms of design specifications and implementation details.

## 5.1 End-User Interface

Since our use case is primarily based on allowing users to easily locate hidden devices, we explored different options on how to show users where the detected devices are located. Most existing solutions [4] are in the form of a device (a wand or handheld antenna) that will beep or show a display when brought near strong RF (Radio-Frequency) emitters. We note that these implementations have major downsides, including needing to sweep the entire room at a very high granularity, potential interference from other devices, and the inability to distinguish multiple devices in the same area.

We also explored doing a hybrid solution, where the user would be able to see directional and signal strength information, as well as decoded information distinguishing multiple devices by MAC address. This would make it easier for the user to distinguish devices without requiring the system to have a self-localization stack (since the user would be able to mentally determine where the devices are, based on the results they see). While this reduces complexity and cost, it also places a significant cognitive load on the user, especially if there are many wireless devices (most of which might be benign) in the room.

We finally settled on using an augmented reality visualization to show the user exactly where the devices are. This catches the best of all worlds - the user can walk around for a while before looking at the data; it can distinguish devices from each other (and filter out trusted devices); and give the user the ability to see where the hidden devices are located in their environment.

## 5.2 Wireless Detection: SDR vs Dedicated Receiver

We initially investigated the use of a Software-Defined Radio (SDR) for detecting wireless devices. There are WiFi baseband implementations [7] that can run on an SDR. These implementations can provide us raw IQ-level data (such as phase-difference between antennas, and extremely-precise Time-of-Flight) along with the decoded 802.11 packets. This would give us the highest quality of data, but an SDR with such capabilities would cost $2-4K minimum, and require an external power source in order to function (along with a lot of specialized software). While this may be appropriate for certain use-cases (including military and RF spectrum enforcement), our end-users are mostly individual users and businesses, for whom a several-thousand-dollar piece of hardware would be unreasonable.

As an alternative, we decided to use hardware containing dedicated WiFi receiver chips. This meets our end-user requirements more effectively, since it is much lower in cost (<$50) and complexity (WiFi cards can be slotted into any standard laptop, and there are even standalone WiFi microcontrollers such as the ESP32). These will give us a lower precision of data - we do not get any raw IQ data, only Time-of-Flight based on the internal clock. We do get some limited multi-antenna phase data through the CSI (Channel State Information) which is normally used for MIMO (multi-in-multi-out) and beamforming, but it is not quite as powerful as we would get with an SDR. However, based on a review of similar work [2][11], we believe that it

is still possible to do sufficiently-accurate localization with only off-the-shelf WiFi hardware.

## 5.3 WiFi Interface Hardware

We explored several different options for WiFi hardware/software stacks. Because we want as much raw data as possible (Time-of-Flight, signal strength across multiple antennas, phase shift) we are limited to stacks which expose such data to a user-level application (as writing modded firmware for a WiFi card is out of scope for our work).

### 5.3.1 AX200 + PicoScenes

The Intel AX200 series of WiFi cards has a relatively good firmware-level support for extracting raw CSI and timing information, thanks to its support for modern MIMO and Time-of-Flight protocols. It is also supported by PicoScenes, which allows us to easily configure the card and programatically extract most of the raw CSI data that we care about. The AX200 supports two antennas, allowing us to extract phase-difference across antennas a half-wavelength apart, and/or see the variation in power across antennas. The major downsides are (1) this requires a user to install the card inside their laptop and run an antenna cable out the back of the laptop and (2) the free version of PicoScenes doesn't support high-precision timestamping, so we cannot do Time-of-Flight analysis.

### 5.3.2 IWL5300 + Linux CSI Tool

Arguably the most popular stack for WiFi research (used by thousands of papers), especially before PicoScenes; the IWL5300 supports 802.11n WiFi and three distinct antennas. The latter feature is particularly useful for working with directional antennas, as we could use one omnidirectional and two directional antennas to get a very strong idea of the orientation towards a device. The major downside with the Linux CSI Tool is that there is no support for 802.11a, which is required for the Polite WiFi attack that we rely on for Time-of-Flight ranging.

### 5.3.3 QCA9500 + Atheros CSI Tool

The Atheros CSI tool has (with the Atheros QCA card) a very similar feature set but slightly higher resolution. It however suffers from the same other issues as the Intel CSI tool, including the lack of 802.11a support.

### 5.3.4 ESP32

The ESP32 is a commonly-used microcontroller which has a very low overhead (hence we can do clock-cycle-accurate timestamping in software), well-documented open-source libraries, and allows us fairly raw access to the 802.11 WiFi stack. Since it can run custom code, we can use the serial port to relay results back to a computer over a USB UART connection. It has been used in past ToF-based research such as [2]. The major downside with the ESP is that it only supports a single antenna, so we cannot do phase or CSI measurements. It also doesn't support high-precision measurement of frame injection times, so we must use a second WiFi device to inject frame and then measure the reception time of the ping, and the response frames from the ESP.

We eventually settled on using one ESP8266 for pinging and one ESP32 for receiving the ACK for measuring Time-of-Flight, interfaced to the localization software (running on a nearby laptop) using a wireless telemetry radio link. This allows our project to be both cost-effective and perform Time-of-Flight measurement, which is a more robust metric than RSS or angle-of-arrival.

## 5.4 Self-Localization

We explored a few different options for self-localization (tracking the antenna itself, within the room). We looked at various LiDAR and depth camera options, including RPLIDAR and Intel Realsense. However, these would very quickly break our cost requirements - a depth camera like a RealSense D455 alone costs >$400 and a T265 (their flagship odometry camera) costs around $300. Additionally, the long-term accuracy of the T265 is not very high, it can have drift of several meters after a few minutes of movement. It is generally meant to be used when fused with other data sources such as a 3D scan (obtained with a LiDAR or depth camera). This fusion would add additional cost and software complexity to our system.

Instead, we settled on using a mobile device as our camera. Because of augmented-reality and 3D scanning applications, high-end mobile phones come with very well-calibrated cameras (sometimes even depth cameras) and IMUs; and augmented reality stacks such as ARKit will handle 3D scanning and loop closure under-the-hood. This means that if we develop an augmented reality app using one of these pre-existing tools, we will also receive accurate, low-drift user localization. This also has the additional benefit of allowing us to use it for visualization - instead of having two separate origin points (one for our antenna reference frame and one for our visualization reference frame), both are relative to the AR scene so there are no added error in the visualization. To create an underlying coordinate system, we simply add a visual marker [8] at an arbitrary location in the environment.

# 6 SYSTEM IMPLEMENTATION

The overall system works in 4 steps.

## 6.1 Device Detection

We use Scapy[10] to passively sniff packets and discover devices, the method of which is inspired by Wi-Peep.[2] During this phase, WiSpider sniffs and identifies access points and their SSIDs (Service Set Identifiers) - for this, we invoke the iw command.[6] For each access point identified,

we inject a beacon frame that appears to be coming from the access point, with the TIM (Traffic Indication Map) bitmap set to all 1's. This wakes up and makes the devices on that network send a response packet to the access point; we can sniff these packets to detect all the devices connected to that access point. We limit the sniffing time per access point to 10 seconds, to meet the use-case requirement of a scan time of less than 5 minutes. The user is given this list of devices and MAC addresses, from which the user can filter out known-trusted devices (like their own phones and laptops).

## 6.2 Device Pinging

Once we have a list of devices, we use the Polite WiFi mechanism to ping the devices. Specifically, we create a fake packet with null data frame, and send them to the devices; the devices respond with an ACK, even though the packet itself has an invalid source address and no data. This is because the maximum time to send an acknowledgment, the SIFS (Short-Interframe Space) time, required by the 802.11 protocol is too short for the device to actually validate the address before responding. To prevent the devices from going to sleep, we alternate these null packets with a beacon frame with TIM bitmap set to 1 which will force all devices connected to each AP to respond. Using the device responses, we measure the ToF, which is done by counting the clock cycles between receiving the outgoing packet and the acknowledge packet, using the ESP32 microcontrollers.

The laptop will display some information about the device (MAC address, manufacturer if known, channel, etc.) on a separate panel. This information is extracted from the unprivileged information in the response packets of each device.

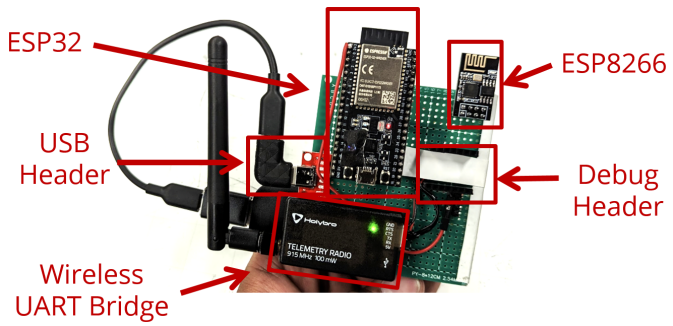


Figure 4: A labeled diagram of the physical component of WiSpider, including the two ESPs used for pinging and measuring ToF.

## 6.3 Device Location Mapping

### 6.3.1 Data Processing

To perform timestamping, our ESP32 counts the clock cycles between when a transmitted packet is detected and the first valid received packet is detected. If the result arrives more than 50% before or after a 10 $\mu s$ average SIFS time, we reject the measurement.

Once all ToF measurements have been collected, we perform operations on the entire data set. To reject likely NLoS (Non-Line-of-Sight) samples, we compare the mean ToF with those at nearby locations, and any locations with an abrupt increase are removed. Additionally, any measurements that would fall outside 3 standard deviations of the mean ToF are also removed.

Next, because we transmit 10 self-localization positions per second and only receive one batch of approximately 500 ToF measurements per second, we need to interpolate the timestamps and positioning for our data. We subtract a linear estimate based on the sample number from each timestamp. Additionally, we perform linear interpolation on the self-positioning to fit the data to the ToF timestamps: we choose linear interpolation to prevent any large peaks from occurring and to simplify calculations.

To standardize each target's distribution, we additionally zero-mean the ToF data. This should not change the localization results since a constant change is accounted for by the SIFS parameter in the gradient optimization.

### 6.3.2 Localization

In order to speed up processing, we found that by using gradient descent, our localization program was able to run much faster than our initially proposed grid-search methodology. Gradient descent functions by iterating a test position, calculating the gradient based on the error at the test position, then stepping the test position in the direction of the lowest error. First, let us define how we extract a location error estimate from the measurement ToF.

$$E_1 = \sum_{u=1}^{u=U} \Big| \sqrt{(x_i - x_u)^2 + (y_i - y_u)^2 + (z_i - z_u)^2} + c \cdot (SIFS_i - \delta_u) \Big| \quad (1)$$

where $E_1$ is the total error, $u$ is the sample index over $U$ total samples, $c$ is the speed of light, $(x_u, y_u, z_u)$ is our estimate of the user's location, $(x_i, y_i, z_i, SIFS_i)$ are our estimates of the target device's parameters, and $\delta_u$ is the measured ToF. While we include $z_i$ in our estimate, due to the low amount of variation we expect in the vertical dimension, we do not include this value in our visualizations.

Next, in the current formulation of the error, we observed that if there was low variation in ToF along one dimension, the SIFS could blow up to unreasonable levels along with the low-variability dimension. to prevent this, we add a regularizer term, $\lambda$, to minimize SIFS and ensure small gains from increasing the SIFS do not cause exploding gradients.

$$E_2 = E_1 + \frac{\lambda}{U} \cdot |SIFS_i| \quad (2)$$

where $E2$ is the final error we use for creating gradients.

Once the norm of our gradient is below a threshold value, we stop iterating and report the final localization result.

### 6.4 Self-Localization + AR Visualization

Our system needs to know its own location (the location of its antennas) at all times, in order to implement the aforementioned device mapping algorithms. It is also very sensitive to drift, since otherwise the accuracy of our measurements will deteriorate over time (i.e. the measurements taken at the start of a 5-minute scan vs at the end of the 5 minutes). We use a mobile phone's augmented reality stack (ARKit) on top of which we will develop our front-end application. The ARKit stack includes a fairly well-calibrated camera model as well as the ability to do loop-closure by matching against earlier scans, so it is unlikely to drift significantly while walking around a room. Since we can get the device's location and orientation from the AR app using the toolkit from [9], we will stream this data back to our server to be fused together (by matching timestamps) with the measurements taken using the WiFi front-end.

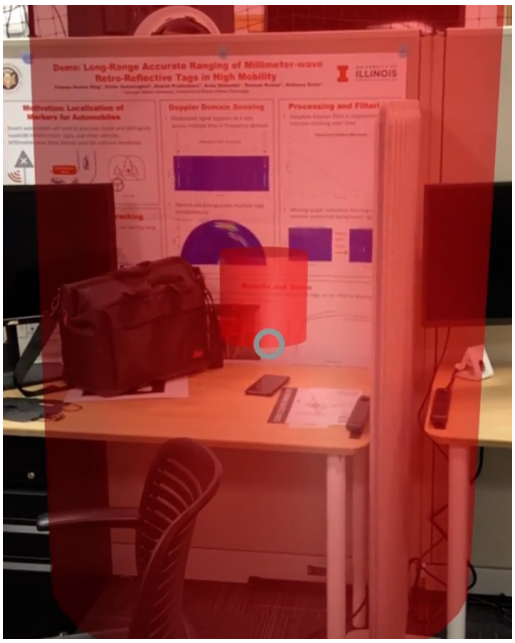

Figure 5: The AR interface of WiSpider, overlaying a phone that was being targeted by our localization system.

Once the scan is taken and we have a confident measurement as to the location of a given WiFi device, we visualize it in the same AR interface. This is done by showing an overlay on the user's phone, in the form of a grid at the floor level. For locations where devices are detected, we show a 3D cylinder of 1-meter radius around the device (Figure 5).

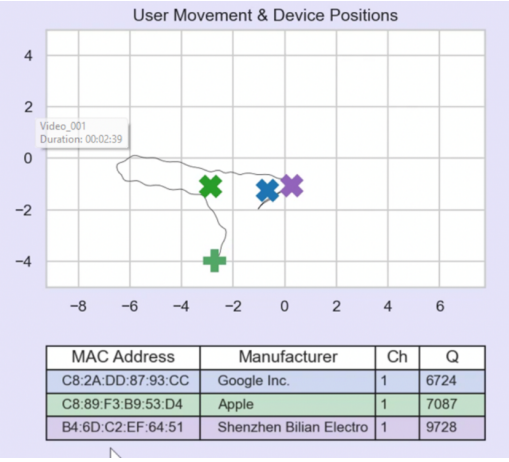


Figure 6: The AR interface of WiSpider, overlaying a phone that was being targeted by our localization system.

We also include a top-down GUI view on the computer to give additional information to the user that may be hard to see in the AR interface (Figure 6).

## 7 TEST & VALIDATION

We conducted a unit test on individual sub-systems to check if they meet the design requirements, followed by an integration test of the use-case requirements. We uses several IoT devices specified in Table 4, along with our personal devices to test these requirements. Specifically, the tests were done for non-moving devices that use 2.4 GHz, and we checked if they meet use-case requirements and design requirements specified in Sections 2 and 4.

### 7.1 Unit-Tests for Device Sniffing

The first test for this part was done with a single known device (i.e. a phone), and detecting whether our device can detect that device with the mechanism explained in section 6.1. Then, we put several devices - phones, tablets, smart cameras, etc. - in the same network and determined if WiSpider can detect all of them.

### 7.2 Unit-Tests for Device Pinging

First unit test consisted of pinging a single known device, and seeing whether we are successfully able to get a response using the Polite WiFi mechanism. Then, we connected it to a known access point, and used the method explained in section 6.2 to consistently get response from pings for 2 minutes without the device going to sleep. Then, we tested it on 5 devices on a single access point, and observed if the devices continuously responded to the pings. Lastly, we tested the devices on two different access points on whether they consistently responded to the pings.

### 7.3 Unit-Tests for ToF

Testing for ToF was important, as small inaccuracies in ToF measurement can lead to large error in localization. We did this by collecting ToF against one device at known location. While SIFS is different across devices and therefore is unknown, we figured it out by having the device and WiSpider right next to each other, with distance of 0m. With the SIFS figured out, we moved the known device with a constant velocity, and compared the measured time against the expected time. Once we verified that the measurements were accurate, we tested it with multiple devices.

### 7.4 Unit-Tests for Localization & AR

To unit-test our localization performance, we set up a test AR app which logs the user's position. We then followed a known track (measured using a measuring tape) and compared the user's located points against the known shape of the track, to figure out how much positional error we have. We also did the opposite (drawing AR points on top of known locations) to verify that our AR visualization is close enough to the real location.

### 7.5 Integration Test

After testing and verifying individual components through unit tests, we integrated all the components and conducted an integration test. For this, we set up a testbed using devices listed in Table 4. We then set a point of origin, from which we measured the ground-truth locations of individual devices. Then, we ran WiSpider through a certain path, and verified the results against the ground truth location, checked if all of our known devices were detected, and visually tested that the AR locations matched the physical coordinates. We repeated this through different devices at different locations through different traces.

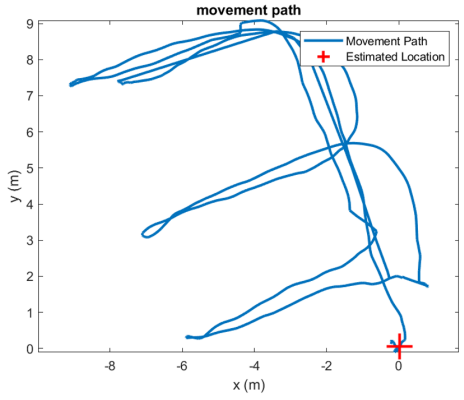


Figure 7: A birds-eye view graph showing the path the user walked along in blue along with the final measured location as a red marker. The actual target location was at (0,0).

### 7.6 Evaluation

The results of the unit tests are as listed in Table 1, in terms of the design requirements described in Section 4. The results of the integration test are as listed in Table 2, in terms of the design requirements described in Section 2. Most results are straightforward and do not need additional explanation. ToF accuracy for individual packets were noisy; however, we measure ToF on a large number of packets, which when aggregated results in about 1 meter of accuracy. For device detection, we saw that 10 seconds per access point was occasionally not enough for some devices.

## 8 PROJECT MANAGEMENT

### 8.1 Schedule

The schedule (with light updates based on how our project progress evolved over time) is shown in a Gantt chart in Fig. 8. Early on, we experimented with a lot of different design parameters and prototyped different components of the project individually, before finalizing our design and working heavily in the second half to integrate everything together. Allocating a lot of slack time in our initial plan worked very well, as we were able to take advantage of this time (despite a setback when we switched from AX200 to ESP32) to complete our full system integration and a substantial amount of testing before the final presentation. This allowed us to confidently assert during our final presentation that our system works and meets our metrics. After the final presentation, we worked on finishing up the remaining evaluation testing, and refining our demo to make it easy to show to the public in the expo-style demo session.

Our weekly progress reports can also be found on our WordPress blog.

### 8.2 Team Member Responsibilities

Thomas primarily worked on the device-localization subsystem; specifically, he researched and developed all of the tracking and filtering algorithms, multipath and noise handling routines, and device localization using the ToF data from the measurement subsystem. He also worked on optimizing the implementation to run in real-time and integrate easily with the measurement software. He also worked on refining the AR visualizations to be intuitive for an end-user to use for finding hidden devices.

Anish worked on developing the measurement subsystem, including the physical electronics package, the WiFi pinging firmware for the two ESP microcontrollers, and the software on the laptop which used the telemetry radio to talk to the two microcontrollers and record the ToF measurements in a format that could be fed to the device-localization subsystem. He also worked on the frontend UI to show the localization results.

Ethan worked on the device-detection subsystem and network-security aspect of the project. He researched the

Table 1: Unit Test Results

| Design Requirement | Metric | Result |
|---|---|---|
| Physical | $<$ 10 lbs, 1 ft$^3$ | 1.2 lbs, 0.1 ft$^3$ (including user's phone) |
| Sniffing rate | $<$ 1 min | 15 seconds |
| Injection rate | 100 pkt/s | $\sim$400 pkt/s |
| ToF measurement rate | 50 pkt/s | $\sim$200-400 pkt/s depending on target device |
| ToF accuracy | $<$ 2m | individually noisy, $\sim$1m when aggregated |
| Distinguish devices | Yes | Yes, by MAC address; even across channels |
| Infrequent device | Yes | Yes, using wake-up packets |
| Self-localization and AR | $<$ 0.3m | verified $<$ 0.3m with occasional re-localization |

Table 2: Integration Test Results

| Use-case Requirements | Metric | Result |
|---|---|---|
| Cost | $<$ \$150 | \$146.42 |
| Size/Weight | $<$ 10 lbs, 1 ft$^3$ | 1.2 lbs, 0.1 ft$^3$ (including user's phone) |
| Detection Rate | $>$ 90% | 90% |
| Scan Time | $<$ 5 min | 3$\sim$5 minutes |
| Lateral Accuracy | $<$ 1m | $\sim$0.8m |
| Detection Range | $>$ 10m | verified at $>$40m range |
| Detection Resolution | $<$ 0.5m | can distinguish devices $\sim$0.2m apart |

different protocols and implemented a pipeline to sniff packets (including real-time channel hopping), detect and identify devices, ensure devices don't go to sleep, and feed the collected data to the measurement system. He also implemented and debugged the initial proof-of-concept of the pinging attack, which Anish subsequently adapted to use for real-time ToF measurements.

## 8.3 Bill of Materials and Budget

Our Bill of Materials for WiSpider is as listed in Table 3. This includes all of the components required to build the physical WiSpider device. We expect the end-user would already own a mobile phone and a laptop, which they can run our software on (hence we do not include these in the system cost). The total cost for WiSpider is \$146.42, which is within our \$150 target cost.

We also have a second bill-of-materials for an IoT testbed, listed in Table 4. Our testbed includes a router and several devices (both cameras and benign IoT devices such as plugs and lights) which we will use to test our system's detection and localization capabilities. We also included some mobile devices and computers which we had easily available, in order to increase the number of devices we used for testing.

Our total expenditures were \$357.45. To avoid wasting money on hardware that would only be used for one-time integration testing, we used borrowed equipment for the IoT testbed wherever possible, and only ordered equipment which we couldn't easily get access to. However, the components for the WiSpider hardware itself (along with some spares) were all purchased using our project budget (other than wire, perfboard, and headers, which were readily available in the Makerspace where we did our hardware assembly).

## 8.4 Risk Management

We managed risk throughout our project development by individually unit-testing the different components as we developed them. This began very early in the project, when we did basic testing of our detection and pinging methodology by typing Scapy commands into a Python console, looking at the packets in a hex editor, and verifying responses by looking at them in Wireshark [12] on a different computer. While this wasn't efficient, and certainly not scalable, it allowed us to have a high degree of confidence that our methodology was correct before spending a lot of time implementing it on the embedded platform. This meant that when we ran into issues further down the line, we knew it was an implementation bug rather than a fundamental flaw in our methodology. In general, manually inspecting packets/frames (both those received and transmitted by our system) was very helpful in iteratively testing different functionalities before we fully integrated them.

Similarly, we developed and tested our localization algorithms using the open-source Intel RTT Time-of-Flight dataset [5]. The data wasn't entirely representative of our use-case (it was using cooperative localization between phones and access-points, as opposed to the non-cooperative localization that we're using). However, it was similar enough that we were able to use it to prototype and test the localization algorithms in parallel with building the measurement subsystem; and then we just had to make some small adaptations to make it work with our real data.

We also allocated a substantial amount of slack time in our schedule, which was beneficial since we lost some time

Table 3: Bill of materials for WiSpider

| Description | Model # | Manufacturer | Quantity | Cost @ | Total |
|---|---|---|---|---|---|
| ESP32 Microcontroller | ESP32-S2 Saola 1R | Espressif | 1 | $14.50 | $14.50 |
| ESP8266 Microcontroller | ESP-01S | Espressif | 1 | $6.99 | $6.99 |
| 915MHz Telemetry Radio | SiK Radio V3 | Holybro | 1 | $89.95 | $89.95 |
| USB Battery Pack | Ultra Slim 6000mAh | Miisso | 1 | $29.98 | $29.98 |
| Protoboard, wire, pin-headers | (from makerspace) | – | – | ∼$5.00 | $5.00 |
| | | | | | $146.42 |

Table 4: Bill of materials for IoT testbed

| Description | Model # | Manufacturer | Quantity | Cost @ | Total |
|---|---|---|---|---|---|
| WiFi Access Point | N300 | TP-Link | 1 | $29.99 | $29.99 |
| Hidden Spy Camera | iQCharger | Alpha Tech | 1 | $51.99 | $51.99 |
| Mini PTZ Camera | Wall Plug-in Camera | ARMIDO | 1 | $36.99 | $36.99 |
| Name-Brand Security Camera | Wyze Cam v3 | Wyze | 1 | $35.98 | $35.98 |
| Name-Brand Security Camera | Tapo C100 | TP-Link | 1 | $19.99 | $19.99 |
| Mobile Phone | iPhone 12 mini | Apple | 1 | – | – |
| Mobile Phone | Pixel 7 | Google | 1 | – | – |
| Laptop | Macbook Pro 14¨ | Apple | 1 | – | – |
| Tablet | iPad Air | Apple | 1 | – | – |
| | | | | | $174.94 |

due to our switch from AX200 to ESP32 but were still able to finish our integration and testing in time for the final presentation.

# 9 ETHICAL CONSIDERATIONS

Our product is intended to solve the problem of hidden wireless devices (such as cameras and microphones) being used to spy on people in places like hotels, Airbnbs, and even in their own homes. It does this by allowing the user to build a map of where each WiFi device is located, as well as extracting some information about the device (to distinguish trusted devices from unknown ones).

This is generally very beneficial to public welfare, as it allows people to ensure their privacy and protect themselves against hidden surveillance devices. When used appropriately, our product generally improves people's lives.

However, there are some failure modes and misapplications that may cause the product to have negative impacts on society. We detail the most significant ones below:

- Misapplication: Spying on People - One potential misapplication is a government using our system to locate specific people and monitor the population. Given the ubiquity of WiFi-connected devices, we could imagine people fleeing from a government that is trying to persecute them, but then being located using our technology (especially in dense areas like apartment buildings and public spaces). This risk is partially reduced by the fact that our implementation only works on non-moving target devices (and people tend to move around quite a bit), but is still a concern.

- Misapplication: Theft & Crime - Cameras and other surveillance equipment are generally used to provide legitimate protection to homes and public spaces. A malicious actor could use our system to find all of the cameras in a space and either disarm them, or abuse their blind-spots to engage in unscrupulous activities (such as theft) without being caught by the cameras. One could also imagine this system being used to locate possible objects to steal (phones, computers) inside a building during a break-in.

- False Positive: If the system had too many false positives, it could cause the user significant lost time and sanity trying to hunt down a "hidden" device that the system detects, but isn't actually there.

- False Negative: If a user were too confident in the accuracy of the system and it ever missed a device, they may inadvertently have unfounded confidence in the privacy of their space (i.e. having sensitive conversations in a room, and then it turns out there was a hidden wireless microphone there all along).

After thoroughly considering the worst-case scenarios, we still believe that our system has an overall positive benefit to society as it can help individual people regain their privacy.

While there is considerable potential for harm with misapplication, such large threat-actors (especially governments) likely have other effective ways to obtain similar results.

# 10 RELATED WORK

Polite WiFi[1] is a behavior in which a WiFi device will respond to any frame with an ACK, if the destination address matches its own MAC address. This can create many opportunities as well as threats; for instance, while it can be used to make WiFi sensing more convenient, one can also use this behavior to identify and localize many devices that they should not be able to, or drain a target device's battery by forcing it to continuously acknowledge the pings.

Wi-Peep[2] is one such example of how this behavior might be used maliciously. Using off-the-shelf WiFi modules and a cheap drone, they were able to detect devices on a network they're not connected to, measure ToF using Polite WiFi behavior, and localize devices from outside a building.

Lumos[11] is a similar work aimed at identifying and locating hidden devices with their phone. Lumos differs in the approach they take, in that they fingerprint the devices with a machine learning approach, and use RSSI and VIO for localization.

PicoScenes[7] is an OSS framework for WiFi CSI and metadata collection. It supports CSI measurement from commercial off-the-shelf NICs and SDRs, while also allowing for packet injection. It also is possible to use MATLAB and python with its CSI format, as well as creating plugins.

The Intel Open WiFi RTT Dataset[5] is an open-source dataset of Time-of-Flight WiFi measurements between various client devices and access-points in an office environment. This dataset has been used to develop localization algorithms based on WiFi measurements, however we do note that the dataset was captured using an API for cooperative WiFi localization (the access points support a dedicated method of getting Time-of-Flight measurements, as opposed to the non-cooperative approach we used in WiSpider).

# 11 SUMMARY

WiSpider is a platform that detects and localizes hidden wireless devices in a non-cooperative environment. Using the device, the user is able to easily locate where hidden devices are, by looking at the AR interface. Additionally, the device is easy to carry around and low-cost.

We learned the challenges in creating a full-stack system, including trying to integrate disparate components, prioritizing areas needed to create an MVP for testing, and making sure programs created by different people can communicate with each other. We all also got experience dealing with WiFi protocols, including CSI, ToF measurements, and packet generation.

One possible expansion for WiSpider is expanding the target of devices to a larger set, such as devices in 5GHz WiFi band or devices using Bluetooth. Additionally, it would be interesting to see if it is possible to collect RSS/CSI data and use those to improve the accuracy of the localization. WiSpider also currently requires the user's laptop and phone to work; an improvement would be to abstract out the laptop and do processing on the phone or on the cloud.

# 12 ACKNOWLEDGEMENTS

We would like to thank Professor Hyong Kim and Omkar Savkur for their feedback and guidance throughout this entire semester.

We would also like to thank Quinn Hagerty and the other course staff for assisting in hardware acquisition and purchasing (especially when some of the devices for our IoT testbed had to be from not-so-reputable brands).

Finally, we would like to thank Zhiping Jiang for donating a PicoScenes evaluation license. While our final implementation ended up not relying on PicoScenes due to design constraints, having access to the tool's datalogging capabilities was invaluable in our experimentation and testing.

# Glossary of Acronyms

**AR** Augmented Reality. 1, 2, 3

**BLE** Bluetooth Low-Energy. 1

**CSI** Channel State Information. 3, 4

**IMU** Inertial Measurement Unit. 4

**IoT** Internet of Things. 1, 2

**IQ** In-phase Quadrature. 3

**LiDAR** Light Detection And Ranging. 4

**MAC** Media Access Control. 3, 5

**MIMO** Multiple-Input Multiple-Output. 3, 4

**MLE** Maximum-Likelihood Estimation. 2

**MVP** Minimum Viable Product. 1

**NLoS** Non-Line-of-Sight. 5

**RF** Radio Frequency. 3

**RSS** Received Signal Strength. 1, 4

**SDR** Software-Defined Radio. 3

**SIFS** Short Interframe Space. 5

**SSID** Service Set Identifier. 4

**TIM** Traffic Indication Map. 5

**ToF** Time-of-Flight. 3, 4, 5

**UART** Universal Asynchronous Receiver / Transmitter. 4

**USB** Universal Serial Bus. 4

# References


[1] Ali Abedi and Omid Abari. "WiFi Says "Hi!" Back to Strangers!" In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. HotNets '20. Virtual Event, USA: Association for Computing Machinery, 2020, 132–138. ISBN: 9781450381451. DOI: 10.1145/3422604.3425951. URL: https://doi.org/10.1145/3422604.3425951.

[2] Ali Abedi and Deepak Vasisht. "Non-Cooperative Wi-Fi Localization amp; Its Privacy Implications". In: *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*. MobiCom '22. Sydney, NSW, Australia: Association for Computing Machinery, 2022, 570–582. ISBN: 9781450391818. DOI: 10.1145/3495243.3560530. URL: https://doi.org/10.1145/3495243.3560530.

[3] Jacob Arellano. "Bluetooth vs. Wi-Fi for IoT: Which is Better?" In: (July 9, 2019). URL: https://www.verytechnology.com/iot-insights/bluetooth-vs-wi-fi-for-iot-which-is-better.

[4] Brick House Security. "Concealable wand detects hidden RF and wireless signals". In: (2019 [Online]). URL: https://www.brickhousesecurity.com/counter-surveillance/rf-wand/.

[5] Nir Dvorecki et al. *Intel Open Wi-Fi RTT Dataset*. 2020. DOI: 10.21227/h5c2-5439. URL: https://dx.doi.org/10.21227/h5c2-5439.

[6] *iw*. 2015. URL: https://wireless.wiki.kernel.org/en/users/documentation/iw.

[7] Zhiping Jiang et al. "Eliminating the Barriers: Demystify Wi-Fi Baseband Design And Introduce PicoScenes Wi-Fi Sensing Platform". In: *CoRR* abs/2010.10233 (2020). arXiv: 2010.10233. URL: https://arxiv.org/abs/2010.10233.

[8] Edwin Olson. "AprilTag: A robust and flexible visual fiducial system". In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 3400–3407. DOI: 10.1109/ICRA.2011.5979561.

[9] Nuno Pereira et al. "ARENA: The Augmented Reality Edge Networking Architecture". In: *2021 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. 2021, pp. 479–488. DOI: 10.1109/ISMAR52148.2021.00065.

[10] Philippe Biondi. "Scapy". In: (2008). URL: https://scapy.net/.

[11] Rahul Anand Sharma et al. "Lumos: Identifying and Localizing Diverse Hidden IoT Devices in an Unfamiliar Environment". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1095–1112. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/sharma-rahul.

[12] *Wireshark open-source packet analysis*. 1998. URL: https://www.wireshark.org/.

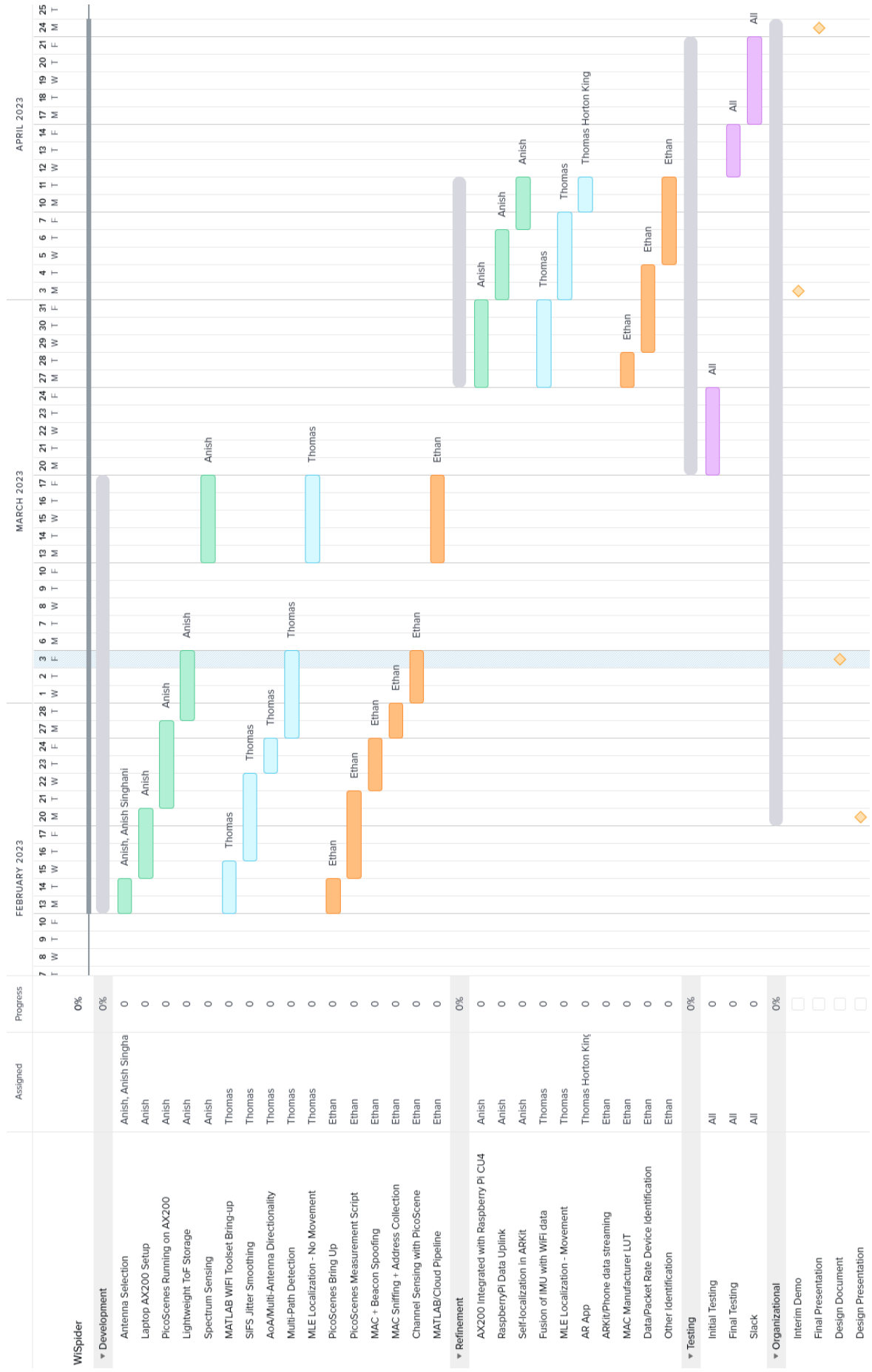


Figure 8: Gantt Chart