ESP32 WiFi Sentry: Detection and RSSI-Driven Localization of WiFi De-authentication Attacks In 2D Indoor Environments

Wallace Schageman and Santiago Castillo

1 Introduction

802.11 WiFi networks are vulnerable to a common denial-of-service (DoS) attack called de-authentication attacks, violating the confidentiality, integrity, and availability of any 2.4 GHz WiFi network. These attacks prevent clients from being connected to WiFi networks by sending spoofed management frames to a WiFi access point (AP). De-authentication management frames instruct the AP to de-authenticate a client's connection. These attacks can also be carried out en masse, de-authenticating every currently connected device to the AP. This is effectively a complete denial of service, making the WiFi network unusable until the attack is over.

De-authentication attacks usually involve a WiFi-enabled attacker machine that has the capability to clone MAC addresses for clients connected to nearby WiFi access points. First, the attacker machine will clone the MAC address of a currently authenticated device to the WiFi network. Once the attacker has cloned the chosen MAC address, the attacker sends a spoofed de-authentication management frame from their device to the AP using the cloned MAC address as the source address. This instructs the AP to de-authenticate the device with the spoofed MAC address. This process can be automated with open-source tools such as aircrack-ng for Linux operating systems and the ESP32 WiFi Marauder firmware for ESP32 boards.

The overall goal of this project is to develop a system that will detect whether a de-authentication attack is occurring and, using the signal strength data (RSSI), will localize the source of the attack via trilateration and multilateration. This system will be made of a low-cost distributed sensor network that is capable of sniffing and transmitting 802.11 management frames and RSSI data.

2 Related work

There has been significant academic and hobbyist focus in the fields of de-authentication attack detection as well as triangulation using RSSI readings. We hope to combine the existing work and concepts from these areas to develop a complete intrusion detection system for de-authentication attacks.

2.1 De-authentication attack detection

The work in this area involves detecting de-authentication attacks using the ESP8266 and the MQTT IoT protocol [1][2]. Our system will use a similar technique for detection but will be developed on the ESP32 using the ESP-NOW protocol. Machine learning techniques were used in both of these projects to detect de-authentication attacks in real time. While ML techniques are on the cutting edge of this field, they are out of scope for this project but could be a possible extension in the future.

2.2 RSSI-based 802.11 Localization

RSSI readings have been used to detect evil twin access points in 2D indoor scenarios [3], demonstrating the feasibility of using signal strength for localization. The researchers were able to localize the evil twin AP within ≤ 1 m, which is in the range of our desired error. Another group of researchers looked into Kalman filtering for RSSI-based localization systems in wireless sensor networks [4]. These researchers were able to localize devices with around 1 m of error. In our system, we also plan to implement digital filters to account for noise in RSSI readings but with less complexity than a Kalman Filter, as we are assuming a static target.

2.3 Miscellaneous

Novel ways to store and optimize IoT and wireless timestamped data on constrained systems have been shown in research [5]. This will allow for persistent data storage and fast reads and writes. This research includes an IoT data ingestion pipeline which we plan to implement.

3 Approach

3.1 Constraints and Assumptions

Before we explain the core components of the detection system, we must first cover the major constraints on our system.

The first and biggest constraint is that the system must not associate with the same AP where the attack is occurring. If any of the devices in the detection system are connected to a nearby AP, they can potentially also be de-authenticated. This would completely defeat the purpose of the system. The distributed sensor network must also communicate with each other wirelessly. If the system is wireless, repositioning sensors for higher attack detection will be seamless. Also, adding cables and wires into the mix could potentially alert an attacker to a WiFi defense system.

The second constraint is cost. We want to make this system as cost-effective as possible. The biggest decision that we made to preserve the cost-effective constraint is to use ESP32 boards as sensors instead of standalone Raspberry Pi systems. Keeping the cost of the system in line allows for scalability in the future, leading to higher accuracy and efficiency.

The third constraint relates to an assumption that the de-authentication attack is completely stationary and located in a medium-sized two-dimensional environment. Accounting for movement and 3D coordinates while triangulating would add an additional level of complexity, requiring more advanced techniques such as implementing a Kalman Filter. While it is a realistic scenario, we want to first address a stationary target before focusing on a moving one.

3.2 Core Components

Our system will consist of five main components:

- **(3x) ESP32 sensors:** Capture 802.11 frames and RSSI data; transmit to receiver via ESP-NOW.
- **(1x) ESP32 gateway:** Receive data from sensors and send to Pi via UART.
- **Raspberry Pi 3B+:** Main compute module for the system.
- **Attacker device:** Flipper Zero equipped with a WiFi dev board flashed with ESP32 WiFi Marauder.

- **WiFi network testbed:** To avoid legal trouble, all testing is carried out on trusted, private, home WiFi networks.

3.3 Data Collection and Attack Detection

Data collection will occur via the 3 ESP32 sensors (three are necessary to localize attacker position) and can scale up if needed to improve the overall accuracy and efficiency of the system. The sensors will operate in promiscuous mode, which will allow them to monitor 802.11 frames. Furthermore, the onboard receiver will also capture raw RSSI readings. In promiscuous mode, the ESP32 sensor will look for '1100' in the frame type/subtype field in order to distinguish de-authentication frames. The sensor checks these bits and begins buffering metadata for any candidate de-authentication frames it observes.

For each candidate source MAC, a sensor collects a short burst of RSSI samples during an aggregation window of length T_w (milliseconds). We denote the per-window RSSI sample set as

$$\mathcal{R} = \{r_1, r_2, \ldots, r_N\}, \qquad r_k \text{ in dBm},$$

where N is the number of received frames from that source during the window. From $\mathcal{R}$ the sensor computes compact summary statistics used for both event detection and later weighting in localization: the sample mean

$$\bar{r} = \frac{1}{N}\sum_{k=1}^{N} r_k,$$

the median $\tilde{r} = \text{median}(\mathcal{R})$ (which is robust to outliers), and the sample variance

$$s_r^2 = \frac{1}{N-1}\sum_{k=1}^{N}(r_k - \bar{r})^2.$$

The sensor also computes the raw frame count $C = N$ and the observed frame rate (frames per second)

$$\text{rate} = \frac{C}{T_w/1000}.$$

These statistics are combined in a lightweight thresholding rule that the ESP runs locally to decide whether the observed traffic likely constitutes a de-authentication attack. Concretely, the sensor declares an *event* when at least one of the following conditions is true:

$$\text{event} \;=\; \big(C \geq C_{\min}\big) \quad \vee \quad \big(\text{rate} \geq R_{\text{th}}\big) \quad \vee \quad \big(\tilde{r} \geq r_{\text{near}}\big).$$

Starter values for these constants in a lab environment are $C_{\min} = 5$, $R_{\text{th}} = 20$ frames/s, and $r_{\text{near}} = -55$ dBm; these should be tuned on-site to account for ambient traffic and radio conditions.

When the rule fires, the sensor packages a compact event summary and transmits it to the aggregator. The event summary contains the sensor identifier, the observed source MAC, the summary RSSI statistics, the frame count and aggregation window, and a local timestamp:

$$\text{summary} = (\text{sensor\_id},\ \text{src\_mac},\ \tilde{r},\ s_r,\ C,\ T_w,\ t_{\text{local}}).$$

Sending median RSSI $\tilde{r}$, sample variance s_r^2, and sample count C is important: the median provides a robust signal-strength estimate, while s_r^2 and C allow the aggregator to weight each sensor's measurement according to its reliability when performing RSSI-to-distance conversion and multilateration.

3.4 Data Transfer

It is important to note that the system cannot be on the WiFi network as it would be de-authenticated when the attack occurs. Because of this, the ESP32s will not be able to send information to and from each other using any TCP/IP protocols since they cannot associate to a wireless network and must remain wireless. To send their buffer contents, the ESP32s will communicate over the ESP-NOW protocol. Similar to BLE and Zigbee, ESP-NOW is a wireless peer-to-peer communication protocol based on the data-link layer, reducing the five layers of the OSI model to only one. This way, the data does not need to be transmitted through the network layer, the transport layer, the session layer, the presentation layer, and the application layer. ESP-NOW is optimized for ESP32 communication and occupies fewer CPU, flash, and power resources than BLE and Zigbee.

Over ESP-NOW, the ESP32 Gateway component will receive all incoming data from all three ESP32 sensor components. The ESP32 Gateway component will send its data to the Raspberry Pi server component over UART. The UART protocol was chosen because of its simplicity and its lack of reliance on timing synchronization. Since both the ESP32 and the Pi use 3.3 V logic levels, no additional hardware besides jumper cables will be necessary. For testing purposes, the baud rate will be set at 115200, which is common for prototypes such as this one. This baud rate could be subject to change once we begin testing.

3.5 Data Processing

The Pi will receive all incoming data from the ESP32 receiver module over UART and deserialize it. Once the data has been de-serialized back into the original alert object that the ESP32 sensors created, the Pi will attempt to correlate alerts from all three ESP32 sensors (most likely using timestamp values), connecting them all to the same de-authentication attack event.

Afterward, the raw RSSI values will be converted to distances using the RSSI-to-distance model (log-distance path loss). Using the calibrated log-distance model, the estimated distance from sensor j is

$$d_j \;=\; 10^{\frac{P_0 - \tilde{r}_j}{10\,n}},$$

where $\tilde{r}_j$ is the sensor's median RSSI (dBm), P_0 is the reference RSSI at 1 m (dBm), and n is the path-loss exponent. These constants will need to be tested and calibrated for each of the 3 ESP32 sensors through a linear regression model. These distances are used in a least-squares model estimate which will be discussed later in the paper.

The Pi will also create the final attack alert including necessary information such as **attacker MAC address**, **timestamp**, and **x, y coordinates of the attacker**. All alerts are stored in a lightweight time-series database on the Pi. The database technology being used for time-series data aggregation is DuckDB because of its lightweight columnar data storage capabilities.

3.6 Least-Squares Trilateration Method

Given sensors at known coordinates (x_i, y_i) and RSSI-derived distance estimates $\hat{d}_i$, trilateration seeks the device position (x, y) that best satisfies the circle constraints

$$(x - x_i)^2 + (y - y_i)^2 = \hat{d}_i^2. \tag{1}$$

Because RSSI measurements are noisy, the circles rarely intersect at a single point. A least-squares formulation is therefore used.

3.6.1 Linearization

Subtracting the circle equation of a reference sensor (sensor 1) from the others eliminates the quadratic terms and yields linear equations of the form

$$A \begin{bmatrix} x \\ y \end{bmatrix} = b, \tag{2}$$

where each row of A contains differences of sensor coordinates, and each entry of b depends on the differences of squared distances and coordinates. With three sensors, the system is 2×2; with more sensors it becomes overdetermined.

3.6.2 Least-Squares Solution

The position estimate $(\hat{x}, \hat{y})$ is obtained by solving

$$\hat{p} = \arg\min_{p} \|Ap - b\|^2, \qquad p = \begin{bmatrix} x \\ y \end{bmatrix}, \tag{3}$$

whose closed-form solution is

$$\hat{p} = (A^\top A)^{-1} A^\top b. \tag{4}$$

3.6.3 RSSI-Based Distances

Distances are estimated using the log-distance path-loss model

$$\hat{d}_i = 10^{\frac{RSSI_0 - RSSI_i}{10n}}, \tag{5}$$

where n is the path-loss exponent. Because RSSI is highly variable, the distance estimates are noisy, and no set of circles intersects perfectly. Least-squares trilateration provides a stable estimate by finding the point that best fits all distance constraints simultaneously.

4 Implementation

4.1 ESP32 Distributed Sensor Network

The distributed sensor network is comprised of three ESP32 sensors and a single ESP32 gateway connected by UART to the Raspberry Pi. The firmware for both the ESP32 Sensors and ESP32 Gateway was written in C. The code for both used the ESP-IDF (ESP Integrated Development Framework) API and the FreeRTOS library.

4.1.1 Sensors

The ESP32 sensors handle the distributed de-authentication attack detection portion of this system. The sensor code defines two threads of execution. The first thread is the packet sniffing and de-auth detection thread. The second thread is the ESP-NOW sending thread. This is done so that the ESP-NOW sending process does not block packet sniffing and event detection functionality. The two threads utilize a shared FreeRTOS queue of Event objects where the first thread produces Events (queue push), and the second thread consumes Events from the queue (queue pop). The Event data structure is described in detail below.

The sensors always remain in promiscuous mode. Promiscuous mode is a passive packet sniffer mode for ESP32s that detects all packets on a specified 802.11 2.4 GHz channel. Promiscuous mode is also able to switch between channels, but for our use case, it was set to a single channel. After setting the ESP32 to use promiscuous mode, we also set the sensors to ignore all packets that did not follow the de-authentication

frame header format. Namely, a de-authentication frame must include 0 as the type field (management frame) and 0xC as the subtype field (de-authentication).

Instead of relying on a single de-authentication packet to throw an alert, the sensors use a sliding window average mechanism instead. First the sensors define an Event data structure. The Event data structure includes the MAC address of the attacker, the MAC address of the sensor, a mean RSSI value, an RSSI variance value, a frame count value, and a timestamp. The sensor uses a sliding window attack detection algorithm to populate instances of this Event data structure as seen in Algorithm 1.

Algorithm 1: Sliding Window Deauthentication Detection

Input: Packet stream P, time window Δt, threshold Θ
Output: Deauthentication attack alert
Initialize sliding window W of recent deauth timestamps;
Initialize circular buffer R of recent RSSI values;
foreach *frame* $f \in P$ **do**
  if f *is not a deauthentication frame* **then**
    continue
  end
  record $f.rssi$ in R;
  insert current time t into W;
  remove all timestamps $< t - \Delta t$ from W;
  if $|W| \geq \Theta$ **then**
    compute RSSI mean μ and variance σ^2 over R;
    create Event with attacker MAC, sensor MAC, μ, σ^2, and $|W|$;
    clear W and R;
  end
end

Once an Event has been created, it is pushed to the shared memory queue between the producer and consumer threads. The consumer thread constantly checks the shared Event queue for new Events. Once it detects that an Event has entered the queue, the thread awakens and sends the event over ESP-NOW to a hard-coded MAC address for our ESP32 Gateway device.

4.1.2 Gateway

The ESP32 gateway is responsible for receiving Events from all ESP32 sensors over ESP-NOW, buffering the Events, and then sending them over a serial UART connection to the Raspberry Pi for processing. The Gateway ESP32 has the same Event data structure defined in its code that the ESP32 Sensors have. First, the Gateway copies the data received over ESP-NOW into a newly created Event object. It does this by defining a callback function for ESP-NOW that is called every time data is received over the protocol. In the same callback function, the Gateway sends this newly created Event object over UART to the Raspberry Pi. On both the Gateway and the Raspberry Pi, we configured the baud rate for UART to be 115200. We never changed this value, and it ended up being perfect for what we needed.

4.2 Data Processing and Computation

All data processing, computation, and alerting was done on a Raspberry Pi 3B+ and written in C++. Similar to the code written for the ESP32 Sensors, the C++ code for the Raspberry Pi is split into 3 distinct threads of execution. First is the producer thread, which handles UART de-serialization and data ingestion. Second is the consumer thread, which handles Event ordering and database insertions. Finally, the main thread focuses on localization computations and alerting. Splitting the computation into producer and consumer threads has the desired effect of preventing the UART ingestion and database inserts from blocking each

other or the localization computations. This program also includes the same exact Event data structure as before for ease of type casting and modularity.

4.2.1 The Producer

The producer and consumer threads work very similarly to the producer and consumer threads for the ESP32 Sensors. Both threads share a FIFO queue of events with a lock to prevent race conditions. This thread reads the UART data stream to the exact number of bytes that are in an Event object. It casts this stream of bytes to a newly created Event object. After type casting, the newly created event is pushed to the shared queue.

4.2.2 The Consumer

The consumer thread "consumes" events from the shared queue. Once an Event is pushed to the queue, the consumer thread wakes up. The consumer thread is constantly checking this shared queue for new Events. Incoming events are sorted based on timestamp and in-flight record ordering to ensure the strict ordering of data for fast and efficient database reads and writes, with the timestamp representing row indices [5]. Once the data within a configurable time window has been sorted based on timestamp, the data is batch inserted into an in-memory DuckDB database. DuckDB is a columnar record store, which allows it to efficiently compress columns based on their data type and similarity rather than compressing rows separately. This lets the entire database reside in memory during execution, allowing for very low database latency.

4.2.3 Localization and Alerting (Main Thread)

The main thread performs all localization computations using the data inserted by the consumer thread. It periodically (currently every 2s) queries the in-memory database for events that fall within the most recent time window (currently 2s also) and aggregates them by sensor. This produces one averaged RSSI reading, one RSSI variance reading, and a total frame count per sensor for that window. Then, the main thread converts the averaged RSSI readings into distance estimates and pairs them with the fixed coordinates assigned to each sensor. With these distances and coordinates, the system runs either the closed-form trilateration function or the least-squares multilateration functions implemented in the code to compute an (x,y) estimate of the attacker's location. See Sections 3.5 and 3.6 for more details.

After the position has been computed, the main thread checks itself to ensure a valid result. This includes confirming that all required sensors produced readings for the window and that the solver did not return an undefined coordinate. Finally, the thread constructs a final attack alert object that includes the attacker MAC address, a timestamp for the detection, and the computed attacker coordinates.

5 Experimental Results

To assess the performance of the intrusion detection and localization system, we evaluate it across four main categories: classification accuracy, timing and latency, RSSI signal stability, and localization error. Each category includes quantitative metrics and success criteria.

5.1 Classification Metrics

The classification subsystem determines whether a detected frame corresponds to a de-authentication attack. *Success Criteria:* A successful system should achieve at least 90% accuracy and an F1-score above 0.85 during controlled de-authentication simulations, ensuring that legitimate traffic is not falsely flagged.

5.2 Timing and Latency Metrics

Latency is critical for real-time detection and triangulation.

- **Processing delay:**

$$L_{\text{proc}} = t_{\text{detect}} - t_{\text{rx,Pi}}$$

Success Criteria: A total latency below 100 ms is considered acceptable for near real-time detection, while values exceeding 500 ms may result in delayed alerts and inaccurate localization.

5.3 RSSI Variance

The RSSI measurements from each ESP32 node determine localization accuracy. To evaluate signal quality, we compute the mean and variance of the RSSI values during detection windows:

$$\mu_{\text{RSSI}} = \frac{1}{N} \sum_{i=1}^{N} RSSI_i$$

$$\sigma^2_{\text{RSSI}} = \frac{1}{N-1} \sum_{i=1}^{N} (RSSI_i - \mu_{\text{RSSI}})^2$$

Success Criteria: A low RSSI variance ($\sigma^2_{\text{RSSI}} < 5$ dB^2) indicates stable readings suitable for triangulation. Higher variance suggests multipath interference or hardware instability.

5.4 Localization Error

Localization accuracy measures how close the estimated attacker position $(\hat{x}, \hat{y})$ is to the true position (x, y):

$$E_{\text{loc}} = \sqrt{(\hat{x} - x)^2 + (\hat{y} - y)^2}$$

Success Criteria: An average localization error $E_{\text{loc}} < 2$ m is considered good performance in a small indoor environment (e.g., lab setup). Errors above 5 m indicate inadequate signal correlation or geometric configuration of sensors.

5.5 Overall Evaluation

The system is considered effective if:

- Detection accuracy $> 90\%$
- Detection latency < 100 ms
- RSSI variance < 5 dB^2
- Mean localization error $= 0.3796$ m

These metrics collectively assess the correctness, responsiveness, signal reliability, and spatial precision of the proposed intrusion detection and triangulation framework.

6 Results

6.1 Localization Error by Sensor Configuration

Table 1 summarizes the localization error for the three sensor configurations (Equilateral, Right Triangle, Isosceles Triangle) under both path-loss exponents $n = 3$ and $n = 4$.

Table 1: Localization Error by Sensor Configuration

Configuration	**n**	**Error (m)**
Equilateral	3	0.0887
Equilateral	4	0.1029
Right Triangle 1	3	0.7329
Right Triangle 2	3	0.3983
Right Triangle 1	4	0.8284
Right Triangle 2	4	0.4385
Isosceles 1	3	1.6890
Isosceles 2	3	0.8440
Isosceles 1	4	0.2330
Isosceles 2	4	0.2952

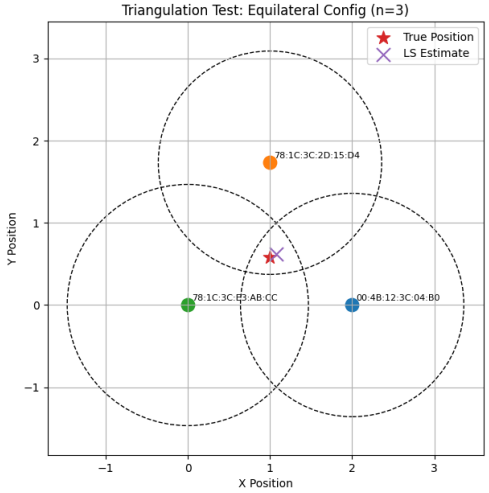
6.2 Effect of Path-Loss Exponent

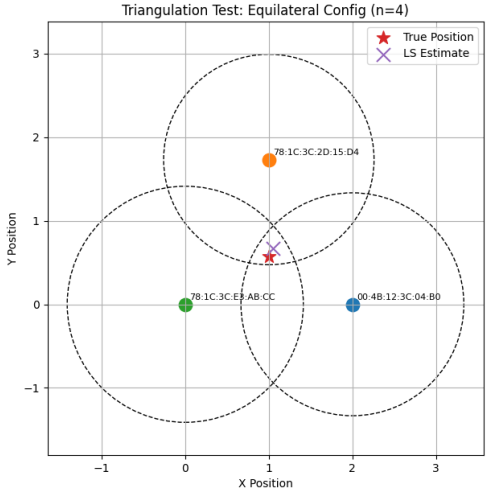
Table 2 compares performance between $n = 3$ and $n = 4$.

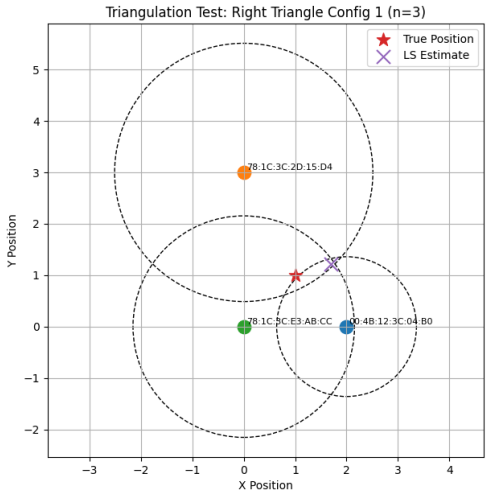
Table 2: Localization Error vs. Path-Loss Exponent

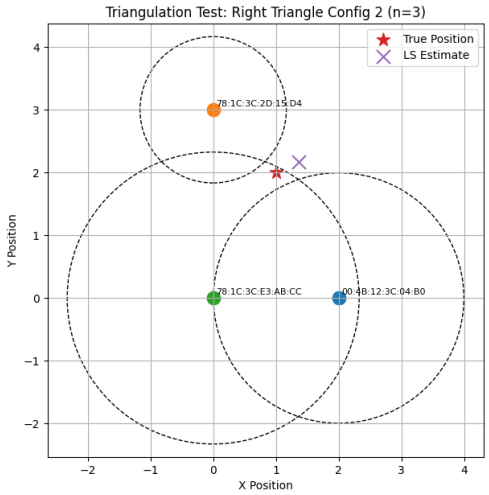
Exponent	**Mean (m)**	**Median (m)**
$n = 3$	0.7506	0.7329
$n = 4$	0.3796	0.2952

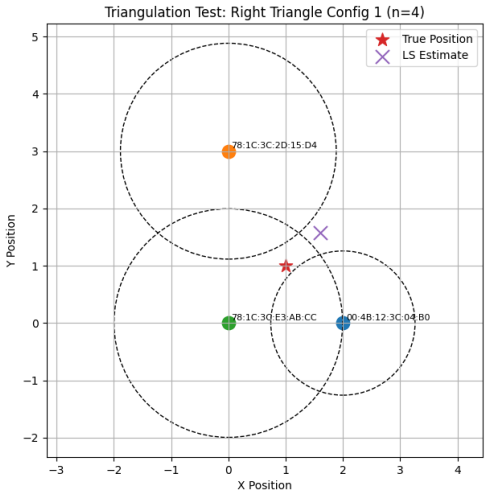
6.3 Results Visualization

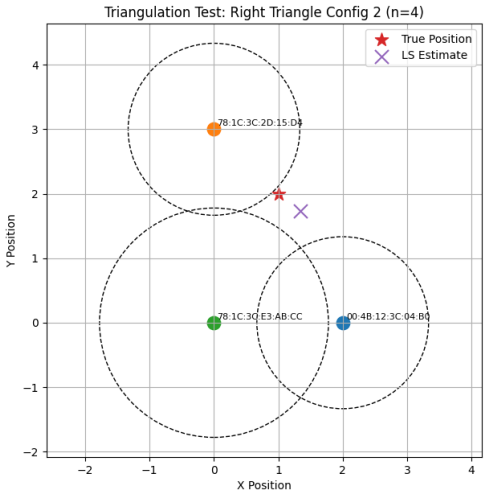


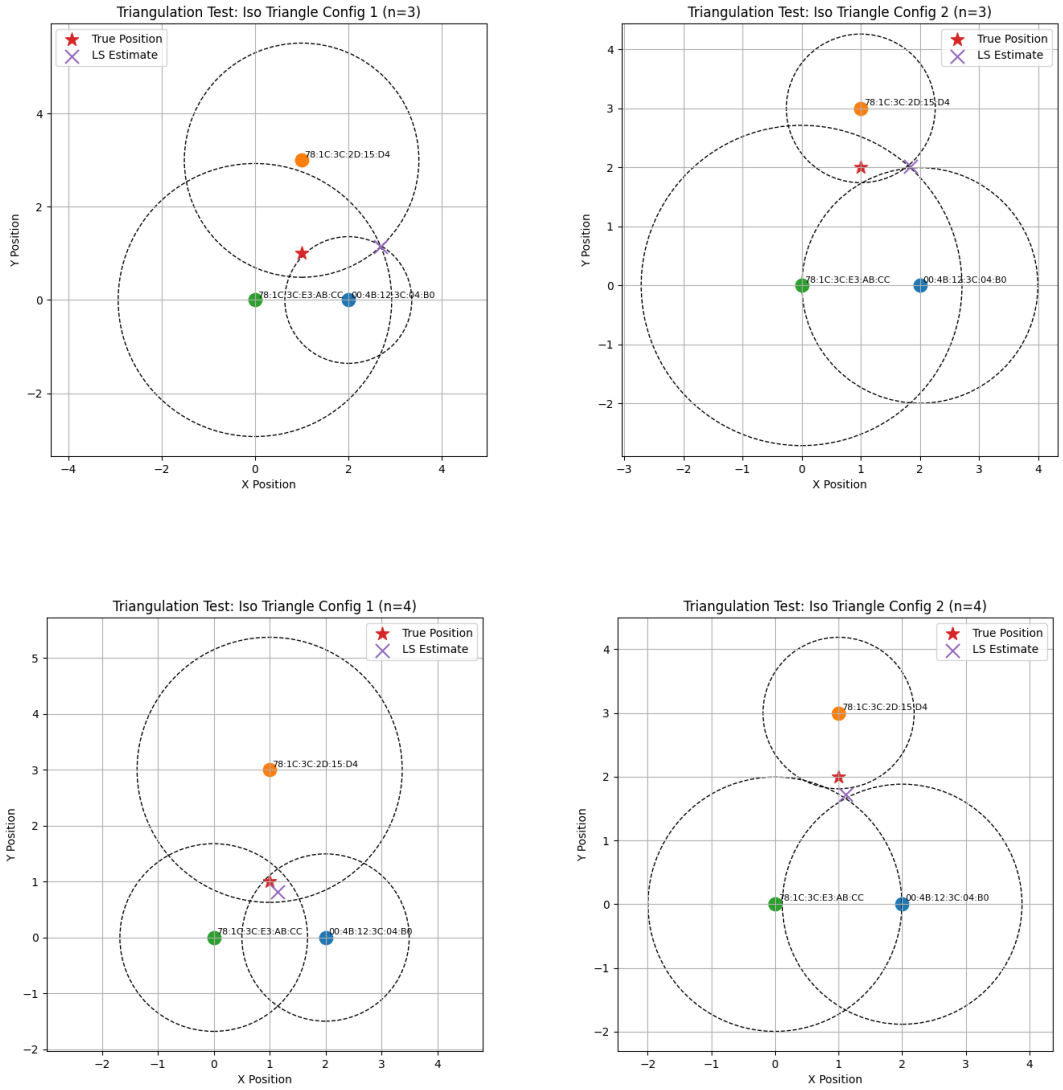












6.4 Discussion

The results show clear performance differences between the three sensor geometries tested. The equilateral configuration consistently produced the lowest localization error for both $n = 3$ and $n = 4$, confirming that geometrically symmetric layouts improve trilateration stability.

Right-triangle configurations performed moderately, with errors strongly dependent on the specific shape and relative sensor spacing. The largest errors occurred in the isosceles configurations when using $n = 3$, where poor geometric conditioning amplified the effect of RSSI noise.

Across all configurations, the path-loss exponent of $n = 4$ yielded substantially lower error than $n = 3$. This suggests that the actual environment exhibits a sharper decay in RSSI over distance than the free-space path-loss model. Calibrating an appropriate value of n was therefore critical for achieving better predictions.

Overall, the equilateral configuration with $n = 4$ yielded the best accuracy, with localization errors below 11 cm.

7 Conclusion

This project successfully demonstrated that RSSI-based localization using ESP32 receivers can achieve sub-meter accuracy when both sensor geometry and path-loss parameters are properly calibrated. Our experiments showed that geometric layout plays a critical role in determining localization stability, with the equilateral configuration consistently outperforming right-triangle and isosceles arrangements. Additionally, we found that a path-loss exponent of $n = 4$ provided significantly more accurate distance estimates than $n = 3$, highlighting the importance of environment-specific calibration. While RSSI measurements remain inherently noisy, the use of least-squares trilateration combined with stable sensor placement allowed the system to achieve localization errors as low as 0.09 m. These results indicate that low-cost receivers, when configured correctly, can provide reliable spatial awareness for indoor tracking and intrusion detection applications.

References

[1] S. K. Gebresilassie, J. Rafferty, L. Chen, Z. Cui, and M. Abu-Tair, "Transfer and CNN-based de-authentication (disassociation) DoS attack detection in IoT Wi-Fi Networks," *Electronics* (MDPI), (accessed Oct. 15, 2025).

[2] M. H. Moharam, K. Ashraf, H. Alaa, M. Ahmed, and H. A. El-Hakim, "Real-time detection of Wi-Fi attacks using hybrid deep learning models on NodeMCU," *Scientific Reports* (Nature), (accessed Oct. 15, 2025).

[3] R. Y. Korolkov and S. V. Kutsak, "Received-signal-strength-based approach for detection and 2D indoor localization of evil twin access points in 802.11," IETA / IJSSE.

[4] M. M. George and K. Vadivukkarasi, "Kalman filtering for RSSI-based localization systems in wireless sensor networks," PDF (online), (accessed Oct. 16, 2025).

[5] D. G. Waddington and C. Lin, "A fast lightweight time-series store for IoT Data," *arXiv*, (accessed Oct. 15, 2025).