Lightweight GPS-Spoofing Detection Mechanism in UAV SWARMS

Dr. Anand Baswade, Ujjayant Prakash, Rishabh Gupta

10th March, 2024

A quick overview of how the Spoofing attack works:

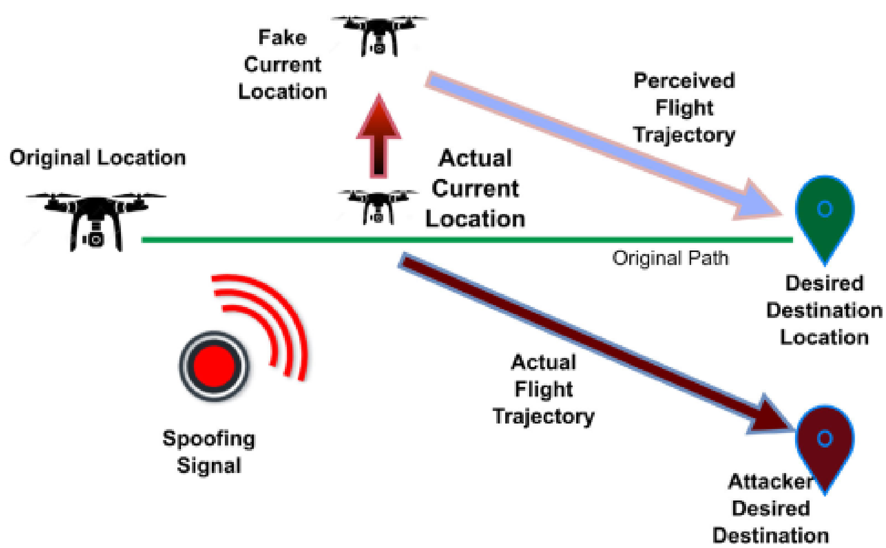

Illustration of GPS spoofing attack in UAV.

GPS spoofing attacks are typically executed through two primary methods.

Firstly, the attacker initiates the attack by locking onto the GPS receiver of the target. This is achieved by computing a pseudo-distance initially. Subsequently, the attacker interferes with the received signal by jamming it, introducing delays, and then forwarding it.

Secondly, the attacker may opt to generate counterfeit signals by meticulously analyzing the characteristics of satellite signals. These counterfeit signals are then broadcasted within the target area. Both of these methods aim to deceive the target by providing erroneous locational information.

The ramifications of GPS spoofing are extensive. Not only does it disrupt the operation of the entire network, but it also undermines the integrity of the network itself.

To illustrate a scenario of a GPS spoofing attack [1], consider the figure above. In this scenario, a UAV initially situated at the 'Original Location' is intended to travel to the 'Desired Destination Location' following the 'Original Path', depicted in green. However, the attacker aims to redirect the UAV towards the 'Attacker Desired Destination'. To achieve this, the spoofer transmits false locational information to the UAV. Consequently, the UAV perceives itself to be navigating along an incorrect trajectory. Although the UAV is positioned at the 'Actual Current Location', the spoofer manipulates it to believe it is at the 'Fake Current Location'. Consequently, the UAV follows the 'Perceived Flight Trajectory', as indicated by the blue arrow in the attached figure. This deceptive movement ultimately guides the UAV along the 'Actual Flight Trajectory', illustrated by the dark red arrow, leading it to the 'Attacker Desired Destination'.

A brief description of the work done till now:

So essentially, we are trying to adjust the speed of the sound [6] [7] [2][3]for a more accurate acoustic sensor distance reading based on the environmental conditions which can be recorded at GCS either externally or if reported by the swarms and post that we are also accounting for the fact that for small distances especially less than 5-10 meters [4] Haversine formula works better for calculating distance from given GPS co-ordinates and for distances greater than that spherical law of cosines works much better but with a slight trade-off that Haversine computation is slightly more intensive but still the difference is not very significant and we get better accuracy with Haversine for small distances.

For detection as of now we consider single jamming signal and single node affected so it compares it's distances to all the other nodes in the swarms and if for more than half of them the difference is more than threshold distance, we return as true else false.

How the algorithm aims to handle the single node affected scenario:

1. The entire process is done by each node and it uses the ranging results from all of it's other sibling nodes.

2. The algorithm chooses between Haversine and spherical law of cosines based on the distance between two points so for less than or equal to 5 meters it goes with Haversine else spherical law of cosines.

3. It calculates the threshold distance by comparing absolute differences between initial acoustic distances and GPS distances for 1 second, and then taking their average. It's an asynchronous function.

4. It finally compares distances returned by GPS-based and acoustic sensor-based distance calculation functions. If the difference between each acoustic distance and the GPS distance exceeds the given threshold distance for more than half of the measurements, it returns True; otherwise, it returns False.

Some points left to consider:

1. How do we decide the d_threshold, we can base it on general errors in instrument measurement readings or we can sense for sometime the initial readings and take that to be the d_threshold assuming the attack does not start instantly as the swarm sets in so as of the current implementation we have used the latter for calculating the threshold value.

2. How do we deal with multi node and multi transmitter scenarios, can we take inspiration from the existing work [5] or should we think of some other method.

3. Finally, measuring the time complexity of the algorithm.

Some general points for future developments:

1. Once a spoofing attack has been detected, finding out how many nodes in the swarm have been affected and what region has been affected overall.

2. Although, the algorithm is significantly light-weight it still carries some overhead so scaling the number of times the algorithm is run based on the alert rates and if a node suffers then checking for it's subsequent neighbours as they would have a high probability of being spoofed too and further on, growing in that particular direction can help optimize the overall performance and resilience of the system.

3. As of now we have assumed Line-Of-Sight (LOS) values for the ultrasonic sensor readings but in reality, it might so happen that the waves bounce off obstacles and then that can introduce some errors or deviations in results which have to be accounted for but as of current implementation, LOS sensing is assumed.

4. Carrying forward from the previous point, for dealing with obstacles etc. IR-UWB (Impulse Radio-Ultra Wide Band) ranging values can also be used to complement the mechanism and they too would not introduce any significant overhead as they are generally employed for ranging and especially for collision avoidance so GCS (Ground Control Station) gets their ranging data normally as well.

GPS-Spoofing Detection Algorithm

```
import math
import asyncio

def CalcSpeedOfSound(T, Rh=50, P=101.325):
    """Compute the speed of sound
        T: temperature in degees C
        Rh: relative humidity in % (default 50)
        P: pressure in kPa (default 101.325)
    adapted from:
    http://resource.npl.co.uk/acoustics/techguides/speedair/
    "This calculation shows the speed of sound in humid
        air according to Owen Cramer, "JASA, 93, p. 2510,
        1993", with saturation
    vapor pressure taken from Richard S. Davis,
        "Metrologia, 29, p. 67, 1992", and a mole
        fraction of carbon dioxide of 0.0004.
    The calculator is valid over the temperature range 0
        to 30' C (273.15 - 303.15 K) and over the
        pressure range 75 to 102 kPa.
    In the region between the air pressures 95.000 und
        104.000 kPa there is no noticeable changing of
        the speed of sound c.
    The standard airpressure is 101325 Pa = 101.325 kPa
        or 1013.25 hectopascal."
    """
    # constants (for convience and clarity)
    Kelvin = 273.15 # For converting to Kelvin
    e = math.e
    pi = math.pi
    # ensure all the inputs are floats
    T_kel = float(T) + Kelvin  # ambient temperature in
        Kelvin
    Rh = float(Rh)
    P = float(P) * 1000.0  # convert pressure from kPa
        to Pa
    # Molecular concentration of water vapour (ENH)
        calculated from Rh
    # using Giacomos method by Davis (1991) as
        implemented in DTU report 11b-1997
```

```
    ENH = pi*10**(-8)*P + 1.00062 + T**2*5.6*10**(-7)
    PSV1 =
        T_kel**2*1.2378847*10**(-5)-1.9121316*10**(-2)*T_kel
    PSV2 = 33.93711047-6.3431645*10**(3)/T_kel
    PSV = e**PSV1*e**PSV2
    H = Rh*ENH*PSV/P  # molecular concentration of water
        vapour
    Xw = H/100.0  # Mole fraction of water vapour
    Xc = 400.0*10**(-6)  # Mole fraction of carbon
        dioxide
    # Speed calculated using the method of Cramer from
    # JASA vol 93 pg 2510
    C1 = 0.603055*T + 331.5024 - T**2*5.28*10**(-4) +
        (0.1495874*T + 51.471935 -T**2*7.82*10**(-4))*Xw
    C2 =
        (-1.82*10**(-7)+3.73*10**(-8)*T-T**2*2.93*10**(-10))*P+(-85.20931-0.22852
    C3 = Xw**2*2.835149 + P**2*2.15*10**(-13) -
        Xc**2*29.179762 - 4.86*10**(-4)*Xw*P*Xc
    C = C1 + C2 - C3  # speed
    return C

def haversine(lat1, lon1, lat2, lon2):
    """
    Calculate the great circle distance between two
        points
    on the earth (specified in decimal degrees)
    """
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1,
        lon1, lat2, lon2])
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = math.sin(dlat/2)**2 + math.cos(lat1) *
        math.cos(lat2) * math.sin(dlon/2)**2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
    distance = 6371000 * c  # Earth radius in meters
    return distance

def spherical_law_of_cosines(lat1, lon1, lat2, lon2):
    """
    Calculate the distance between two points on the
        earth
    using the spherical law of cosines formula
    """
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1,
        lon1, lat2, lon2])
    distance = math.acos(math.sin(lat1) * math.sin(lat2)
```

```
        + math.cos(lat1) * math.cos(lat2) * math.cos(lon2
        - lon1)) * 6371000
    return distance

def calculate_distance(lat1, lon1, lat2, lon2):
    """
    Calculate distance between two points using
        Haversine formula
    if the distance is less than or equal to 5 m, else
        use the spherical
    law of cosines formula.
    """
    distance = haversine(lat1, lon1, lat2, lon2)
    if distance <= 5:
        return distance
    else:
        return spherical_law_of_cosines(lat1, lon1,
            lat2, lon2)

async def
    calculate_threshold_distance_async(initial_target,
    initial_gps, initial_tdoa, humidity, pressure,
    temperature):
    """
    Calculate the threshold distance by comparing the
        absolute differences
    between acoustic distances and GPS distances, and
        taking their average.
    """
    total_difference = 0
    for i in range(len(initial_gps)):
        acoustic_distance =
            calculate_distance_acoustic_sensor([initial_tdoa[i]],
            humidity, pressure, temperature)[0]
        gps_distance =
            calculate_distance(initial_target[0],
            initial_target[1], initial_gps[i][0],
            initial_gps[i][1])
        total_difference += abs(acoustic_distance -
            gps_distance)

    return total_difference / len(initial_gps)

def calculate_distance_acoustic_sensor(TDOA_values,
    humidity, pressure, temperature):
    """
```

```
    Calculate the distance between two drones using
        multiple acoustic sensor readings
    while considering environmental conditions such as
        humidity, pressure, and temperature.
    """
    speed_of_sound_adjusted =
        CalcSpeedOfSound(temperature,humidity,pressure)
    distances = [TDOA * speed_of_sound_adjusted for TDOA
        in TDOA_values]

    return distances

def compare_distances(subsequent_target, subsequent_gps,
    subsequent_tdoa, threshold_distance, humidity,
    pressure, temperature):
    """
    Compare distances returned by GPS-based distance
        calculation function
    and multiple acoustic sensor-based distance
        calculation function.
    If the number of times the difference between each
        acoustic distance and the GPS distance
    exceeds the given threshold distance is greater than
        half of the total number of measurements,
    return True; otherwise, return False.
    """
    count_exceed_threshold = 0
    for i in range(len(subsequent_gps)):
        acoustic_distance =
            calculate_distance_acoustic_sensor([subsequent_tdoa[i]],
            humidity, pressure, temperature)[0]
        gps_distance =
            calculate_distance(subsequent_target[0],
            subsequent_target[1], subsequent_gps[i][0],
            subsequent_gps[i][1])
        if abs(acoustic_distance - gps_distance) >
            threshold_distance:
             count_exceed_threshold += 1

    if count_exceed_threshold > len(subsequent_gps) / 2:
        return True
    else:
        return False

async def main():
    initial_target = (40.7128, -74.0060)
```

```
    initial_gps = [(40.7128, -74.0060), (34.0522,
        -118.2437), (51.5074, -0.1278)]
    initial_tdoa = [0.05, 0.07, 0.08]
    subsequent_target = (34.0522, -118.2437)
    subsequent_gps = [(34.0522, -118.2437), (51.5074,
        -0.1278), (48.8566, 2.3522)]
    subsequent_tdoa = [0.06, 0.09, 0.1]
    humidity = 60  # Relative humidity in percentage
    pressure = 101.325  # Atmospheric pressure in kPa
    temperature = 25  # Temperature in degrees Celsius

    threshold_distance_task =
        asyncio.create_task(calculate_threshold_distance_async(initial_target,
        initial_gps, initial_tdoa, humidity, pressure,
        temperature))

    await asyncio.sleep(1)

    threshold_distance = await threshold_distance_task

    if compare_distances(subsequent_target,
        subsequent_gps, subsequent_tdoa,
        threshold_distance, humidity, pressure,
        temperature):
         print("Difference between distances exceeds
             threshold for more than half of the
             measurements and GPS Spoofing attack is
             detected.")
    else:
         print("Difference between distances is within
             threshold for at least half of the
             measurements and the system is safe.")

if __name__ == "__main__":
    asyncio.run(main())
```

References

[1] Ala Altaweel, Hena Mukkath, and Ibrahim Kamel. Gps spoofing attacks in fanets: A systematic literature review. *IEEE Access*, 11:55233–55280, 2023.

[2] Owen Cramer. The variation of the specific heat ratio and the speed of sound in air with temperature, pressure, humidity, and CO2 concentration. *The Journal of the Acoustical Society of America*, 93(5):2510–2516, 05 1993.

[3] Richard Davis. Equation for the determination of the density of moist air (1981/91). *Metrologia*, 29:67–70, 04 1992.

[4] gis stackexchange. Why is law of cosines more preferable than haversine when calculating distance? `https://gis.stackexchange.com/questions/4906/why-is-law-of-cosines-more-preferable-than-haversine-when-calculating-distance-b`, 2024. [Online; accessed 10-March-2024].

[5] Pavlo Mykytyn, Marcin Brzozowski, Zoya Dyka, and Peter Langendoerfer. Gps-spoofing attack detection mechanism for uav swarms. In *2023 12th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–8, 2023.

[6] phys libretexts. `https://phys.libretexts.org/Bookshelves/University_Physics/University_Physics_(OpenStax)/Book%3A_University_Physics_I_-_Mechanics_Sound_Oscillations_and_Waves_(OpenStax)/17%3A_Sound/17.03%3A_Speed_of_Sound#:~:text=The%20equation%20for%20the%20speed%20of%20sound%20in%20air%20v,s%E2%88%9AT273K.&text=v%3Df%CE%BB.`, 2024. [Online; accessed 10-March-2024].

[7] travc. `https://github.com/travc/locbatch/blob/master/locbatch_code/loc_misc.py`, 2024. [Online; accessed 10-March-2024].