Location Verification of Crowd-Sourced Sensors

Christopher Kitras*, Carter Pollan*, Kyle Myers*, Camille Wirthlin Tischner†, Philip Lundrigan*
*Department of Electrical and Computer Engineering, Brigham Young University, Provo, UT, USA
†Department of Electrical and Computer Engineering, Utah State University, Logan, UT, USA
Email: {chkitras, cjp82, kyle2004, lundrigan}@byu.edu and camille.wirthlin@gmail.com

***Abstract*—Community-driven sensor networks have been instrumental in providing easy access to affordable, large-scale measurement recording, facilitated by the accessibility of inexpensive sensor hardware. The simplicity of this hardware makes it challenging to retrieve trustworthy location data without added hardware such as GPS. We introduce the LaMDA framework, a software-based solution run solely in a web browser to determine the location of a device with the aid of a registering device. We monitor the device for any location changes by analyzing its `traceroute` data from a central server. Our solution allows minimal firmware changes to be made to fleets of devices without recall or changes to hardware.**

***Index Terms*—crowdsourcing, citizen science, sensor networks, air quality, IoT**

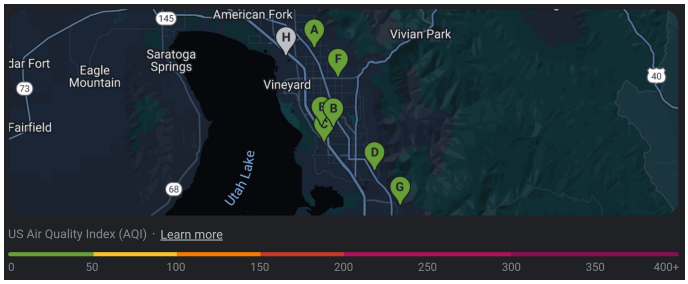


Fig. 1: A Google search for air quality shows data from both a government source *and* PurpleAir sensors

I. Introduction

The recent boom of community-driven sensor networks is a fundamental step in the progress of affordable, distributed health/environmental monitoring. This is especially helpful for those engaged in citizen science initiatives, where volunteers from the community deploy their devices to provide measurements to a central service. A popular field that is flourishing with the help of citizen science engagement is the recording of environmental data, specifically air quality (AQ) data where tens of thousands of nodes are deployed. Like many other environmental monitoring research, air quality research benefits from large quantities of spatially diverse sensor readings. Crowd-sourced sensing device deployments lend themselves well to this type of research.

PurpleAir [1] is one such popular network of community-driven low-cost internet-enabled air quality sensors. These sensors are deployed by individuals and the data is uploaded to a central repository and map where people can view the data. The sensors are not strictly calibrated, maintained, or quality assured like the government-run stations. However, studies have shown that their accuracy is similar to government AQ stations with a correction factor added to the data [2]. These community-driven AQ device networks have reached enough critical mass that *government websites* now display the low-cost devices' readings alongside government stations [3]. Recently, Google integrated the data stream from community-driven AQ devices into Google Maps (Figure 1). It is evident that the integration of this community-collected data into popular, trusted platforms shows the increased trust in citizen science platforms.

However, there is a hidden risk associated with using this data. Since the sensors are low-cost, there is no GPS module provided to determine their location. If there are no positioning capabilities on the devices, the citizen scientists become the source of location. **With this lack of location verification, anyone from absent-minded, well-meaning users to malicious actors intent on ruining the integrity of the system's data could falsely place a sensing device anywhere on the map.** For example, a motivated adversary could store these devices in a clean location while registering them to regions with poor AQ, thus artificially improving the AQ in the area from the perception of an AQ map.

Many people rely on this public data, especially those with elevated sensitivities to pollen or smoke. Though the government and Google show the AQ readings to help people make decisions on health, no one has verified that the device is actually in the location it is registered to. Now that these devices' data are displayed on popular and trusted websites, this problem needs to be addressed. **People are making important health decisions on data that can not be trusted.** It is irresponsible for services to display the information without any location verification. We need a way to provide location verification, thus improving the trust of the devices without a costly solution. The central question of this research becomes, *how can we prove that a sensing device is installed in its registered location without extra hardware?* While we focus on air quality, this problem is generalizable to all community-driven sensor deployments.

We aimed to create a solution that (1) verifies a device's location without extra hardware, (2) detects any changes in the device's location, and (3) scales to be deployed on any system without requiring a platform-specific application. These design goals prevent the need for recalling and retrofitting devices with localization hardware, prevent device relocation after verification, and ensure accessibility to users with unsupported smartphone models.

To meet these goals, we create the Location and Movement Detection on the Application layer (LaMDA) framework that solves the problem of localizing a device *without any additional hardware*. We leverage the huge popularity of the smartphone by using its geolocating services to act as the proxy location of a device. We ensure that the smartphone and device are geographically close together by analyzing network characteristics. We design a system that monitors the device's location to ensure it has not moved after its location is established. To make our system easily adoptable, we design it to not require any special system privileges or network features. This is no small task because this requires our framework to comply with strict security norms required by a web browser to provide secure communication between multiple hosts. The result is a framework that does not require a platform-specific application but instead can be run from a *standard web browser*.

We open source our code so that others can build on it and integrate it into their systems. The code is available on GitHub [4]. Our hope is that maintainers of these crowd-sourced sensing networks will adopt this protocol so that a device's location may be established, thus allowing us to trust its data. Until then, the data reported by these networks can not be trusted.

II. Related Work

This paper focuses on determining the geolocation of a device and, once it is established, determine if it has moved or not. The most obvious solution to this problem is to use GPS. GPS does not fit within our design criteria because it requires extra hardware to work, increasing the cost and complexity and invalidating already deployed devices. Also, GPS does not work in all situations, especially in urban environments and indoors, where there is no clear visibility to the sky [5]. Another possible solution to provide geolocation by triangulating with cell towers. While this is a possible solution, our framework focuses on devices deployed with a WiFi connection since the most popular community-driven sensors are WiFi.

Databases like IP2Location's Geolocate [6] geolocate people based on their IP address. However, research has shown that these systems have are not very good when dealing with distances less than 6 miles (10 km). They also are dependent on your Internet Service Provider (ISP) and population density [7] [8]. For our geolocation approach, we utilize the services provided by smartphone manufacturers that use a combination of cell towers, WiFi, and GPS [9]. Our innovation is not in the methodology of geolocation, but the use of a second smart device as a proxy for the location of the sensing device.

III. Typical Sensing Device Registration

To understand the contributions of this paper, we provide a walk-through of a typical low-cost AQ device registration process [10] as illustrated in Figure 2a, which is also typical of many IoT devices. These devices connect to a central server over WiFi and require network credentials to connect to an available access point. The devices go into AP mode to enable a user to connect to the device and provide network credentials through a self-hosted web page. Once the device receives the WiFi credentials, it switches to client mode and connects to the home's WiFi AP. The user is then forwarded to a registration page containing a questionnaire where they enter specifics about the device deployment, including its location. Current registration processes do not require any confirmation of the location, making them unreliable as the user base grows. This paper presents a design that is easily integrated into existing sensor frameworks and eliminates the need for location verification by the user.

IV. System Design and Architecture

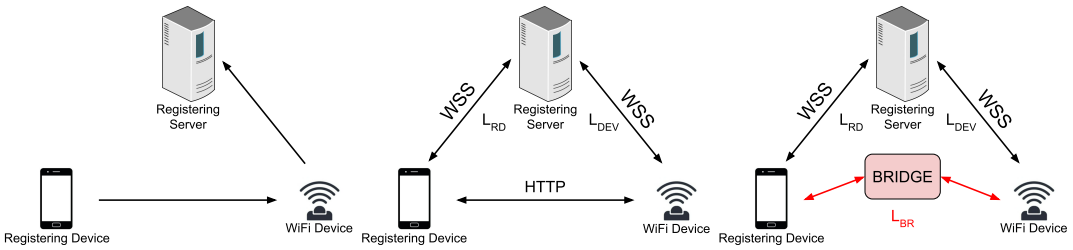
Our system aims to determine device location without user input to eliminate human error and location falsification by bad actors. After location determination, we monitor for any changes, requiring device re-registration to re-establish location trust and verify data uploads. The system consists of two components: device registration and change of location detection, detailed in Section IV-B and Section IV-C. We first outline our expected adversary and their capabilities.

A. Adversary Model

In the design of our system, we assume the following adversary: a malicious actor who has complete control over their local network (e.g., a home network) and can manipulate where traffic and data flow. They have the ability to sniff packets and manipulate them within their network. However, they do not have the ability to control the flow of data outside of their network, change the firmware of the sensor device, or change the software running on the registering device (e.g., smartphone) or server. Privacy is not a concern of our system since we are assuming the device readings are uploaded to a publicly available community map.

The main scenario we consider is an adversary who buys a sensor device and wants to trick our system into thinking their device is in a different place than where it actually is located. This scenario covers malicious behavior, but it also covers a person who accidentally sets the location incorrectly or moves the device to a new location and forgets to update its location. An adversary can perform man-in-the-middle (MITM) attacks on the packets that are sent and received within their network as illustrated in Figure 2c. They can also set up relays between different networks to make it look like a device originates from another network. In Section IV-B1, we present specific ways an adversary might try to achieve this goal and how our system protects against such attacks.

Spoofing GPS is outside of the scope of the adversary model of this work. While it is a viable attack vector for a malicious actor, research has been done on this topic [11] [12], along with commercial software products [13], and supplementary databases to ensure reported GPS data is accurate, such as GeoIP services [6]. Our solution, by design, focuses on using networking from user space to verify the location of the senor and if it has moved. Our solution could be integrated with GPS spoofing technologies, but that is not the focus of this paper.



(a) Current Registration Process (b) New Registration Process (c) Adversary Configuration

Fig. 2: Different registration schemes for associating air quality devices with a location-based service

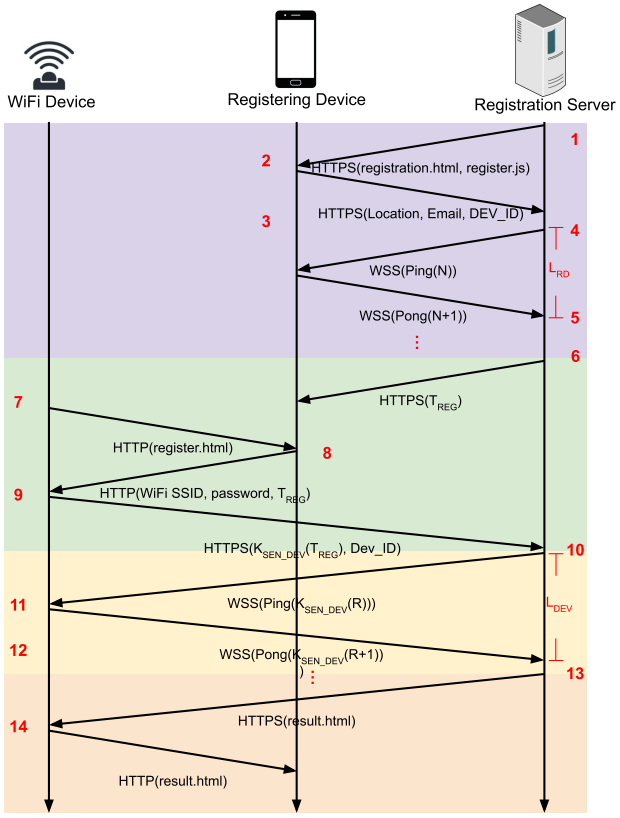


Fig. 3: The registration flow between the devices

B. Device Registration

The entities that participate in the registration of a device are: **(1) Registration Server** where the registration service is hosted and where the database of active users is held; **(2) Registration Device (RD)** that has location services to prove the location of the sensor. Generally, this will be a smartphone, but any device with location services, such as a web browser on a laptop, can also be used. **(3) WiFi Device** which is the device to be registered.

By design, our device registration process does not require any additional hardware and uses the same entities and general flow as the current state of practice for sensor device registration. Figure 2b outlines how the different components communicate to verify the sensor device's location. The novelty of our system is providing additional functionality without changing the devices themselves.

The first step is to prove that the RD is close to the WiFi device. We utilize the RD's location services (e.g., a smartphone's GPS) to get an approximate location of the device. Once the location has been established, we use the central server to monitor that the location has not changed. If the server detects the location has changed, the user is required to redo the registration process. Throughout this process, we design our protocol in such a way that an adversary can not manipulate or read any of the data.

In order to decide whether we accept that the RD and the device are in the same location, we need a common factor that is specific to both of those devices all done within a web browser. We find that the best way to determine the device's locality with respect to the RD was by measuring the latency of a packet sent to the central registering server from both devices. A benefit of this approach is it adapts to changes in network conditions because both devices are affected equally if they are on the same network. A device on a separate network would not see those same changes.

To start the process, the user connects to the central server allows location sharing. The server initiates a secure WebSocket (WSS) session, sending a sequence of "pings" with a payload of a nonce R to the client. The client responds with "pong" and the payload with $R+1$, and the server records the round-trip time (RTT). The server selects the minimum RTT value as the latency between the server and RD (L_{RD}), stored for future use. The nonce is required to avoid a redirect attack (see Section IV-B1). This process is illustrated in Figure 3, steps 1–5.

After the latency between the server and the RD has been measured, the server responds with a session token (T_{REG}). This token allows the server to distinguish between different device registration requests. The server then prompts the RD to navigate to a page that the sensor device hosts on its own HTTP server, requiring the RD to switch to the WiFi device's AP. The user is then clicks on a link redirecting them to the WiFi device's credential input webpage. During this page redirect, T_{REG} is transferred from the RD to the device. This stage of registration was heavily affected in design by RFC1918 [14] which prohibits the flow of requests from public address spaces to more private ones. Adherence to this rule allows us to

maintain our framework solely in a browser, providing a wider range of compatibility across different devices. After submitting the WiFi network credentials, the WiFi device sends a request to the server with a version of T_{REG} that is encrypted with a symmetric key (K_{DEV_SRV}) along with DEV_ID. This symmetric key must be determined and burned into flash memory beforehand by the manufacturer of the device. It is the entity that attests to the unique identity of the device during registration. This process is illustrated in Figure 3 in steps 6–9.

The server receives $K_{DEV\_SRV}(T_{REG})$ and decrypts it. If the T_{REG} value matches what the server sent to the RD, the server responds with a confirmation. If T_{REG} does not match due to an adversary modifying the data or generating a fake request, the server invalidates the registration token and the registration process must start from the beginning. If successful, the same latency measurement process as before happens between the server and the sensor device, yielding L_{DEV}. First, the server compares the public IP address of the RD and the public IP address of the sensor to ensure that they are from the same network. We assume that the LAN of the device uses a NAT and that all traffic coming from the home network has the same external IP address. If the public IP address matches both the RD and the WiFi device, we can reasonably assume that they are part of the same LAN, and therefore *could* be in the same location. If not, we assume that they are on different LANs and invalidate the registration. This process is illustrated in Figure 3 in steps 10–13.

Next, to ensure the WiFi device and RD are geographically on the same network, the server compares L_{DEV} and L_{RD} to see if they are similar to each other. If L_{DEV} and L_{RD} are not within a certain threshold, this could indicate that there is extra hardware with a delay of L_{BR} between one of the devices and the server, yielding a value of $L_{DEV} + L_{BR}$ or $L_{RD} + L_{BR}$ which is the added latency of a bridge (Figure 2c). We experimentally derive the threshold in Section VI-A. This process is illustrated in Figure 3, steps 13 and 14. If the public IP addresses, latency, and encrypted randomized sequence match, we can reasonably assume they are in the same location.

1) Design Security: We design our protocol to protect against the interception of data by a man-in-the-middle (MITM) attack between the RD and server, server and WiFi device, or RD and WiFi device is an additional concern. We use the Transport Layer Security (TLS) protocol to encrypt and authenticate all transactions going outside the LAN to the server. However, we do not use TLS for data sent between the RD and device due to certificate management overhead. Nonetheless, an attacker who removes or alters the payload (T_{REG}) between the RD and device would invalidate the registration session.

An adversary may try to deceive our system by falsifying the public IP address of the WiFi device, making it appear as if the device and RD are on the same network. For instance, the adversary could send traffic from a different network but with a spoofed source IP address that matches the same IP address the RD device is on. To mitigate this threat, our ping messages are encrypted with K_{DEV_SRV} and contain a random number. The device must first decrypt the message, increment the random number by one, encrypt it, and send it back to the server in response to the ping message. The server verifies that the encrypted number has been incremented by one before it calculates the round trip time. This ensures that only the WiFi device is capable of responding to pings, preventing spoofing attempts.

C. Change of Location Detection

With the location verified in the registration process, we are now confident that our WiFi device is indeed at the location of registration. However, that guarantee means little if someone moves the device after registration. The question becomes *how do we ensure the device has stayed in the same location?* To answer this question, we develop an algorithm for detecting changes in the location of a device, which we call Change of Location Detection (CoLD). The novelty of this algorithm is that it requires neither extra hardware nor assistance from the device itself. We are able to detect changes in location purely from the perspective of the server. Detecting a location change is different from determining the geolocation of a device. From the perspective of the algorithm, the only thing that matters is whether the device has moved or not. If we detect the device has moved, then we rerun the registration process to determine its new location.

We utilize two tools, gap detection and `traceroute`, to detect the change of location of a WiFi device. Devices, especially environmental sensors, are programmed to upload data at specific intervals. When a device is moved, it will typically be powered off and disconnected from the Internet, creating a gap in data. We develop an algorithm that detects gaps in sensor data and then checks for change in the position of the device. To perform this check, we use `traceroute` which tracks the hops a packet takes to construct a path through the Internet starting from the server and ending at the device. When the system detects a gap, we can compare the path from before the gap to after the gap. If the path has stayed the same the device has not moved.

Using `traceroute` reliably is a challenging problem because the path a packet can take through the Internet can be dynamic and take multiple paths [15]. The CoLD algorithm cannot simply compare a single route from before a gap to a single route after the gap because a route can change even when the location of the device has not changed. We deal with this by comparing many routes collected before the gap to multiple routes after the gap. We describe the details of the algorithm in Section V-B.

1) `Traceroute` Logging: The traceroute logger runs `traceroute`, collects path data, and sends it to the database for all devices in the fleet. It measures the path between the logger and the WiFi device's router, not the device itself. If the external IP address changes, the path is checked for significant differences to determine if the device has moved.

2) Gap Detection: When data from a sensor device has no long gaps and the public IP address remains unchanged, we can

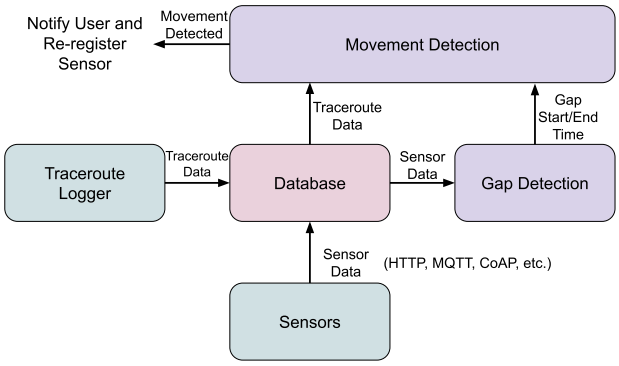


Fig. 4: CoLD algorithm modules and connections

assume the device is in the same location. However, gaps in data or changes in public IP address indicate a system change, which could be due to a device relocation or other reasons such as power or internet outages. Gap detection module is used to detect gaps in the data and report them. The module compares the most recent timestamp with the current time and if the difference between the two is greater than T_{GAP} (the typical time between data readings for a device), a gap is detected. We set T_{GAP} to be three times the amount of time between expected uploads from the device, and when a gap is detected, the movement detection module is notified with information about the start and end of the gap.

3) Movement Detection: The movement detection module is responsible for detecting when a WiFi device needs to be re-registered with the CoLD algorithm. When a data gap is identified, the module checks if the ISP has changed based on the source IP address of the data packet. If the ISP has changed, it may indicate a device has moved and needs to be re-registered. Next, the module compares the `traceroute` data before and after the gap to determine if there has been a change in the device's location. If the `traceroute` data is similar and no movement is detected, nothing happens. However, if the `traceroute` data is dissimilar, the module quarantines incoming data from that device and alerts the user to re-register the device. More details on how we determine similarity in the `traceroute` data are explained in Section V-B.

V. Implementation

We take care great care in choosing the technologies and hardware that are included in the implementation and testing of our framework to ensure that our setup reflected actual conditions that could be found in real world deployments.

A. Device Registration

The proximity and location verification system consists of a WiFi device, a RD, and a registration server. The Raspberry Pi 4B is chosen for the WiFi device, and the Google Pixel 6A is chosen for the RD. A virtual private server is used for the registration server, running on Ubuntu 20.04.5 and using Nginx as the web server and Python Flask with SocketIO library to handle the registration process. The system relies solely upon HTTPS requests and WebSockets for communication, which makes it portable and easily accessible by any RD with internet access. The WiFi device's firmware also communicates over HTTPS requests and WebSockets. The design is not tied to HTTPS or WebSockets and could easily support other protocols like MQTT or CoAP depending on the device's needs.

B. Change of Location Detection

The CoLD system consists of three components that work together to detect if a WiFi device has changed location. We outline the implementation details of each of the components.

1) Traceroute Logger: We write a Python script that uses `traceroute` to determine the route between the server and the sensor device. We send `traceroute` to all devices every 10 minutes and record the responses. We ignore invalid responses from routers or if the amount of hops was too high. The `traceroute` logger records the results along with the timestamp in the central database.

2) Gap Detect: A separate Python script monitors the database looking for gaps in data. The length of time that constitutes a gap (T_{GAP}) depends on the time interval between data uploads, as programmed into the device by the device developer. We set T_{GAP} to be three times the interval of data transmission. When a gap is detected, the start and end date of the gap is collected and this information is sent to the final component of our system.

3) Movement Detection: The movement detection subsystem is a third Python application that gets activated when a gap is detected by the gap detection module. It starts by pulling the `traceroute` data from the database before and after the gap. It pulls a week's worth of `traceroute` data before the gap to get a historical baseline of how the traces should look. We call this our trusted data. Next, we compare our trusted data with the data after the gap, which we call our questionable data. The movement detection module scores the questionable data against the trusted data. If the score is greater than 90%, then we consider the questionable data to be the same as the trusted data. The threshold can be adjusted depending on the tolerance to false positives/negatives. We opt to have less false negatives and more false positives. We rather have a device need to be re-registered than miss someone moving a device.

VI. Evaluation

Our system consists of two major components which we evaluate in the sections below.

A. WiFi Device and RD Proximity Verification

The device registration subsystem in our framework ensures that the RD and WiFi device are in the same location. This is accomplished by measuring the L_{RD} and L_{DEV} and making sure those values are within a certain tolerance L_{TOL} of one another. This premise only functions if $|L_{RD} - L_{DEV}| \leq L_{TOL} \leq L_{BR}$. To establish a good value for L_{TOL}, we conduct an experiment that compares the latency between the RD and the server (L_{RD}) and the device and the server (L_{DEV}) while the RD and device are on the same network.

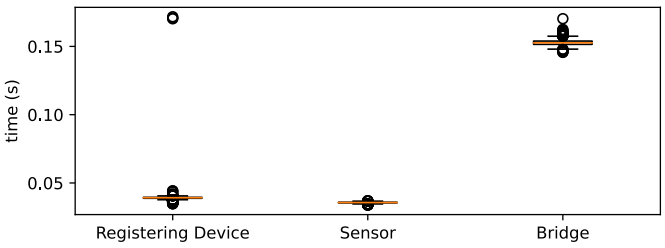


Fig. 5: The measured latency between a WiFi sensor, the registering device, and the registering device using a bridge

To determine the best value for L_{TOL}, we create a website with a JavaScript application that sends and receives a series of ten requests to and from the server. We take the minimum of the recorded latency values and use that as the general value to represent the time it took to communicate with the server for that device. We run this test on the Google Pixel 6A, which acts as our registering device, for an hour and a half.

Similarly, for our WiFi device, we write a Python script to send ten requests to the server that runs on a Raspberry Pi. The Raspberry Pi acts as our device. The series of ten requests and their responses were executed and latency values were recorded and the minimum is selected as the value that represents the time it took to talk to the server. The latency values are then compared to see if they lie within a reasonable tolerance of each other.

In a perfect system, the latency between the smartphone (RD) and Raspberry Pi (WiFi device) should be the same ($|L_{RD} - L_{DEV}| = 0$) since they come from the same network and go to the same server. However, due to inconsistencies in the network, these values are generally not the same. We use a tolerance, L_{TOL}, to allow for some difference between measured values. The results of our experiments, Figure 5, show that the latency between the RD and device is almost the same. The difference in minimum latency between the two devices is approximately 15 ms.

We test our method of comparing latency values against traffic tunneling using network bridging, a common method of spoofing network location that allows an RD to provide false geolocation information by being in a different location from the device. An adversary can set up a network bridge between two networks that are geographically far apart, and there are many ways to set up this type of network topology, such as using port forwarding, a VPN, or a proxy. We evaluate this potential attack vector using a proxy service called ngrok, which forwards network traffic to a server in the cloud and then sends it to a desired location. The results show (Figure 5) that the latency between the server and the device when going through a bridge is much higher, around 110 ms, which is expected since any bridging method adds extra latency to the response time. Our framework detects this added latency, making it sensitive to spoofing.

Our method detects any bridging method by taking advantage of the added propagation delay when redirecting traffic to another network. It can detect the bridged network only if it adds a noticeable latency to the response time. If the bridged network is physically close to the original network, the propagation delay is negligible, and our algorithm works as expected. We use a threshold of 15 ms (L_{TOL}) to determine if the difference between the RD and device's latency is significant enough to indicate they are not on the same network. If the difference is more than the threshold, registration cannot be completed, and the user must try again. We take the minimum value of 10 measurements to prevent spurious latency from falsely detecting that the devices are on different networks. Out of 4238 attempts, only three failed, yielding a 99.93% success rate (as seen in Figure 5).

B. Change of Location Detection

We evaluate the change of location algorithm by measuring `traceroute` data from a central server to 15 device locations for 20 weeks. At each location, we collect `traceroute` data every 10 minutes. To test the effectiveness of the CoLD algorithm, we test nodes that exist in three common geographic configurations: rural, inter-city, and intra-city. In the rural set we measure the `traceroute` at four geographic points roughly 40 miles (64 km) apart from each other. In the second set, inter-city, we measure the `traceroute` at five geographic points that are roughly 8 miles (13 km) from each other. In the last set, intra-city, we measure the `traceroute` at six locations that are a couple of city blocks apart. The device placements are shown in Figure 6.

To evaluate our the CoLD algorithm's accuracy, we artificially trigger a gap event and provide data from one node at a targeted location then data from a different node and see if the algorithm can detect the "movement" from a difference between the sets of data. Incrementally closing the distance between nodes in the different geographical data sets allows us to determine the limitations of the algorithm. The results for each of these tests are shown in Table I. In each table, the rows represent the nodes used as trusted data and the columns represent the nodes used as questionable data. In the cells of the table is the average percentage of questionable data over the 20 weeks. The results of testing a location against itself (the diagonal of the table) determines how well our algorithm can detect true positives. A location compared to another location that is not itself (non-diagonal) tests the false positive rate. We describe the results of each table in more detail in the subsections below. In the results, some of the percentages are above zero when emulating a sensor movement. These percentages do not come close to the established threshold of 90% so none of the sensors would be miscategorized.

1) Rural Area Test: This test evaluates devices that are around 40 miles (64 km). Excluding the diagonal where the locations check against themselves, we can see that most of the tests are close to 0%. Nodes N and O did score non-zero, and this is attributed to them being on the same ISP. This evaluation shows that we are able to accurately determine when a device has moved a large geographic distance.

TABLE I
COLD ALGORITHM ACCURACY BETWEEN NODES FOR DIFFERENT GEOGRAPHIC DISTANCES

	L	M	N	O
L	99.1	0	0	0
M	0	99.9	0	0
N	0	0	99.9	23.9
O	0	0	1.8	98.0

(a) Rural Area

	A	B	C	D	E
A	99.9	0	0	0	14.6
B	0	96.39	0	0	0
C	0	0	99.7	0	0
D	0	0	0	98.0	0
E	11.3	0	0	0	99.8

(b) Inter-City

	F	G	H	I	J	K
F	99.9	0	66.6	0	0	66.6
G	0	96.3	0	0	0	0
H	66.6	0	99.9	0	0	66.6
I	0	0	0	94.9	0	0
J	0	0	0	0	99.9	0
K	66.6	0	66.6	0	0	99.9

(c) Intra-City

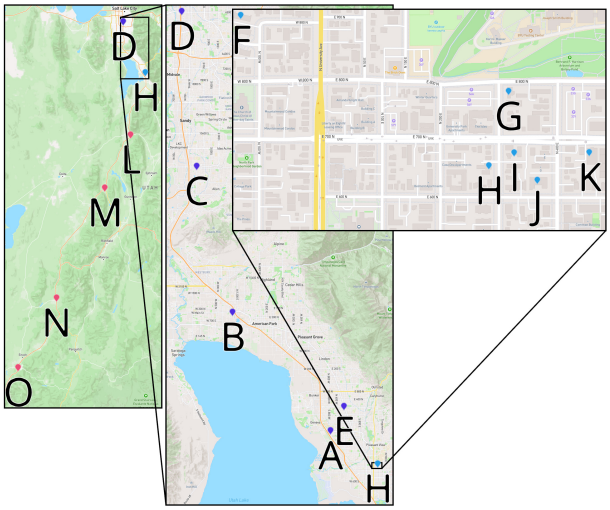


Fig. 6: Device test location maps: rural (left), inter-city (center), and intra-city (right)

2) Inter-City Test: In this test the nodes are about 8 miles (13 km) apart. This location emulates the movement of a device from one city to another in a metropolitan region. This test is important because environmental sensors, like air quality devices, can vary dramatically from city to city. It is imperative that we are able to accurately detect when a movement of this scale has occurred. The results for this experiment are shown in Table I. We can see that the algorithm performed incredibly well when comparing a node to a node that is not itself (not the diagonal). Nodes A and E did score non-zero, and this is attributed to them being on the same ISP.

3) Intra-City Test: Up to this point, our algorithm has been able to differentiate between movement regions and between cities. The final test is to see if the CoLD algorithm can detect movements within part of a city. This establishes a limit to the granularity that can be expected within the borders of a city. The results show (Table I) that the accuracy for some nodes is very good, but for other nodes is not great. This is to be expected since we are using `traceroute` and it is not sensitive enough to notice small physical movements in a city.

VII. CONCLUSION

We presented a framework that allows a user to register the location of a WiFi device and determine whether it has moved from its original location without the aid of any additional hardware. The change of location detection process can successfully distinguish when a device has changed location between cities. We showed this by testing our algorithm's ability to recognize a device's own path data compared to those in its own city, neighboring cities, and distant cities. The experimental results and lessons learned indicate that it is possible to register and track a device's location without the need for extra hardware accurate to the radius of a city, which for applications in citizen science is more than sufficient. Our system provides the necessary ingredient for automatic verification of location for citizen science devices.

REFERENCES

[1] PurpleAir Inc., "Purpleair: Real-time air quality monitoring," 2022. [Online]. Available: https://www2.purpleair.com/

[2] "Field evaluation purple air (pa-ii) pm sensor - south coast air quality ..." 2017. [Online]. Available: http://www.aqmd.gov/docs/default-source/aq-spec/field-evaluations/purple-air-pa-ii—field-evaluation.pdf?sfvrsn=11

[3] "Using airnow during wildfires," 2020. [Online]. Available: https://www.airnow.gov/fires/using-airnow-during-wildfires/

[4] C. Kitras, C. Pollan, K. Myers, C. Wirthlin, and P. Lundrigan, "LaMDA," 2023. [Online]. Available: https://github.com/NET-BYU/LaMDA

[5] D. Maier and A. Kleiner, "Improved gps sensor model for mobile robots in urban terrain," in *2010 IEEE International Conference on Robotics and Automation*, 2010, pp. 4385–4390.

[6] IP2Location, "Geolocate your user's location," 2022. [Online]. Available: https://www.ip2location.io

[7] P. Callejo, M. Gramaglia, R. Cuevas, and A. Cuevas, "A deep dive into the accuracy of ip geolocation databases and its impact on online advertising," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2022.

[8] I. Poese, S. Uhlig, M. A. Kaafar, B. Donnet, and B. Gueye, "IP geolocation databases: unreliable?" *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 2, p. 4, 2011.

[9] Google, "Geolocation api," 2022. [Online]. Available: https://developers.google.com/maps/documentation/geolocation/overview

[10] PurpleAir Inc., "Register your purpleair device," 2022. [Online]. Available: https://www.purpleair.com/register

[11] D.-K. Lee, D. Miralles, D. Akos, A. Konovaltsev, L. Kurz, S. Lo, and F. Nedelkov, "Detection of gnss spoofing using nmea messages," in *2020 European Navigation Conference (ENC)*, 2020, pp. 1–10.

[12] D. Miralles, N. Levigne, D. M. Akos, J. Blanch, and S. Lo, "Android raw gnss measurements as the new anti-spoofing and anti-jamming solution," in *Proceedings of the 31st International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2018)*, 2018, pp. 334–344.

[13] "Mobile games protection," Feb 2023. [Online]. Available: https://irdeto.com/denuvo/mobile-games-protection/

[14] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. d. Groot, and E. Lear, "Address allocation for private internets," Feb 1996. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1918

[15] F. Viger, B. Augustin, X. Cuvellier, C. Magnien, M. Latapy, T. Friedman, and R. Teixeira, "Detection, understanding, and prevention of traceroute measurement artifacts," *Computer Networks*, vol. 52, no. 5, pp. 998–1018, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128607003428