Proxensus: Anti-Spoofing Through Location Crowdsourcing

Android Team
Andrew Canonigo
Frans Timothy Juacalla
Martin Thai
Aryaman Sinha

Abstract

Our capstone project deals with the topic of detecting user location spoofing, or the faking of one's current location. When using different applications, users may opt to spoof their locations for a variety of privacy and security reasons, such as to protect their user data and from being tracked. Location spoofing can also be used maliciously or for dishonest purposes. Location spoofers may intend to gain unauthorized access to content that is normally restricted based on location, conduct fraud or cyberattacks, and even gain advantages in video games that offer rewards to players based on location.

Proxensus is an application that uses proximity detection to tag location spoofers through a crowdsourcing voting method. Location spoofers who come within contact of honest users get put on a 'blacklist' that servers handle. Proxensus adapts the framework of Decentralized Privacy-Preserving Proximity Tracing (DP-3T), a proximity tracing system that determines whether an individual has been exposed to COVID-19 without revealing location data or user data. The creators of DP-3T designed the system with the belief that slowing the spread of SARS-CoV-2 doesn't just come down to wearing masks and isolating when positive. Rather, they recognized that understanding the general interactions between people and then notifying them if they have been exposed to people with COVID-19 was integral in order to "break [the] transmission chain"[1]. Our application uses Bluetooth Low Energy (BLE) technology for proximity detection. A key value of Proxensus lies in its reliability in maintaining data privacy. Our modified tracing system will ensure that the data sent to the central server will be minimal (only location data and no other sensitive user data) and the authenticity of location information will be guaranteed[2] (no fake/misrepresented data). Our vision for Proxensus is to be able to extend the algorithm to applications that rely on the integrity of user location data to improve their functionality.

Introduction

Proxensus relies on gathering data related to the legitimacy of mobile device locations without compromising the privacy of the user. Our project was split into two teams: a server team and an Android team. The Android team utilized DP-3T's framework for exchanging anonymous unique identifiers between user devices. When users activate Proxensus, they will have a unique identifier attached to them that does not reveal any further information about the user. The server team handles all of the unique identifier data and utilizes crowdsourcing - or "majority voting' - to determine which of the unique identifiers are location spoofing. Decentralized location census through location crowdsourcing refers to an approach to conducting a census where data is collected from a dispersed network of users who are located in various locations. This is typically done through the use of mobile devices or web-based platforms that allow users to report information about their location and other relevant

information. Using the data collected, we can generate an accurate picture of the population, each location, and more importantly in our case, who, if anyone, is spoofing their location.

Literature Review

Our project is greatly influenced by the DP-3T systems, therefore it makes sense to fully comprehend and take a deep look at their proposed methodology and techniques. In general, the DP-3T application allows users to quarantine before any signs of symptoms. Users with the applications on their smartphones are able to exchange temporary IDs that are generated every epoch (user-setted) through proximity. Infected users notify the decentralized backend of their status, and users are able to cross check with the stored IDs that they came into close proximity to with the 'flagged' IDs shown from the database.

The DP-3T project team has developed three different protocols that support our goal of secure exposure detecting and tracing. The three methods, called low-cost decentralized proximity tracing, unlinkable decentralized proximity tracing, and hybrid decentralized proximity tracing, all share a common structure; they locally generate random ephemeral (temporary) identifiers (EphIDs) and broadcast them through Bluetooth Low Energy (BLE) beacons. However, these methods have tradeoffs regarding user privacy and computation cost, which allows us as developers to weigh each factor against each other.

Low-cost decentralized proximity tracing offers good privacy properties and low computation cost/small bandwidth usage. A random initial seed is generated for the day and put in a cryptographic hash function. For privacy purposes, devices change the EphId that they broadcast often and the duration for which the smartphone sends this unique EphId is called an epoch.The smartphone starts each day off generating a list of seeds that it will broadcast for the day. These seeds are then put in a pseudo-random function which creates the EphIds.

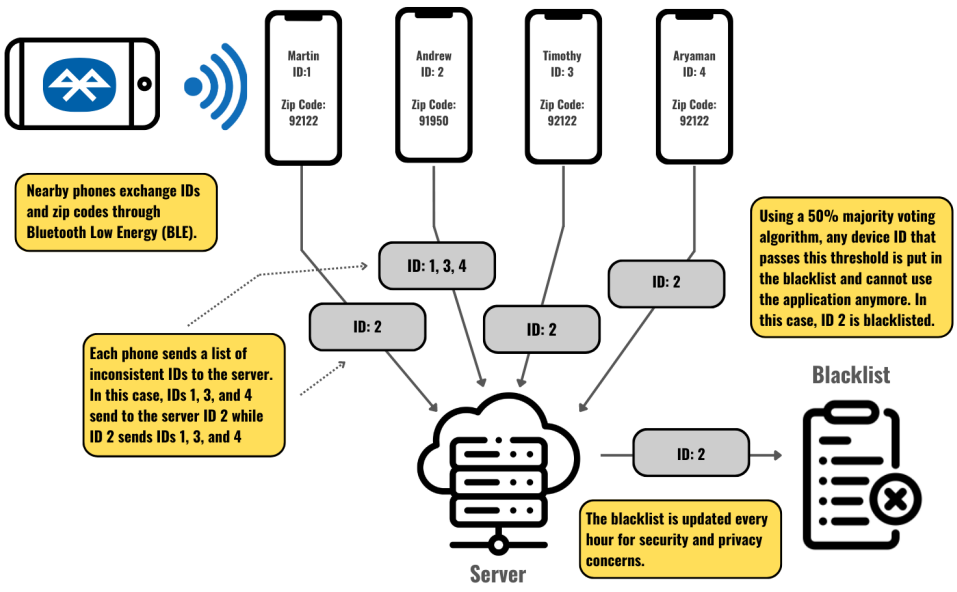
Unlinkable decentralized proximity tracing provides even better privacy properties at the expense of computation cost and higher bandwidth. This method uses a bloom filter to store hashed EphIDs of positive users which is disseminated to other users and this prevents any linkage of EphIds to COVID-19 positive users as there's no list of seeds of positive users (unlike low-cost decentralized proximity tracing). How EphIds are generated is that the smartphone draws a random 32-byte seed which gets put through a hash function. The leftmost 128 bits are then chosen as the EphId and are stored for however long health authorities recommend. A bloom filter is generated by the backend every two hours for every seed is put through it to check if it saw any EphIDs from COVID-19 positive users.

Hybrid decentralized proximity tracing uses ideas from both methods proposed above and how it works is that it generates random seeds within a time frame and uses the seeds in the same way as the low-cost design to create EphIds which are only uploaded to the server if they could have possibly been at risk for COVID-19. In regards of privacy properties, this design performs better than the low-cost model but worse than the unlinkable model. In terms of computation cost, it's more efficient than the unlinkable model but takes more bandwidth than

the low-cost model. In this design, EphIds are generated by a random 16-byte seed put through a pseudo-random function. These EphIds are then broadcasted during an epoch of a fixed duration. These epochs are grouped together in a time window.

Methods

For our project, we have decided for simplicity and certain constraints to implement a rudimentary unique identifier generation system in which the seed is static, meaning that devices will essentially only have a singular unique identifier. Our tracing system will then adopt a more "majority rules" based crowdsourcing approach to determine whether a user has been faking their location or not. This means that the tracing system will not only have devices share ID's between them but also their current locations as an attempt to locate any location spoofers. This is based on the assumption that the **hackers/spoofers are in the minority of users**. The picture below shows the algorithm we are implementing.



As the graphic illustrates above, smartphones will be sending ephemeral ID's to one another. These devices will then send a list of IDs to the server of devices with a different location from them. In this example, the phone with ID 3 will send a list of IDs [1,2,4] since they are in UCSD instead of NYU while IDs 1,2 and 4 will send a list of IDs [3] since 3 is the only one who is in a different location relative to them. With this information, the server could then

immediately tell who is the liar since their ID will appear numerous times on their end, hence why it is a “majority” crowdsourcing algorithm.

As the Android team, we are handling the smartphone interaction/communication (via BLE) side of the algorithm. For the foundation, we utilized an old DP-3T prestandard codebase as a framework. We heavily modified and built upon this codebase, only preserving and changing the bluetooth framework of the existing DP-3T which would have these 5 basic functionalities:

1. *Unique ID generation*
2. *Bluetooth service implementation for both client and server for message transmission between devices*
3. *Retrieval of location data, specifically zip code*
4. *Posting IDs of devices with distinct locations as self*
5. *Getting blacklist from server via GET request to tell if user is spoofing their location or not*

To handle the first problem of generating a unique identifier, we decided to make it to be constant and simply generate an 8 byte randomly generated number. This unique ID serves to not only protect the user’s information but also make it unique enough that there is no possible overlap between the devices. This means that there would be no devices that have the same identifier. We chose 8 bytes long because we had to be able to fit the IDs in the advertising packets (explained in the next section), which were only to hold 32 bytes of data. This means 32 bytes of data can only be communicated between phones at a time. Considering the other types of data that needed to be transmitted, such as the zip code (5 bytes) and UUID (Universally Unique Identifier; 16 bytes), we had only 11 bytes to use. We did not want to use all 11 bytes in order to not go over the limit, so we decided on 8 bytes. For storage, we simply used SharedPreferences to store them once the ID was generated, which happened only once.

For the second issue, we built upon DP-3T’s codebase which had existing bluetooth related code. In general, what we needed for the proximity tracing to work was to have a working data advertiser and a data receiver. These are the primary mechanisms needed to make communication between the phones possible.

Via BLE, a data advertiser would be the mechanism that sends data (or payload as it is called by convention) to another smartphone device. In our case, the payload that would be sent to a receiving device would be the unique ID and zip code. The advertiser would advertise to all available bluetooth devices that have the same UUID. For this requirement, we simply used the UUID which was utilized by the pre-standard codebase, which was 16 bytes or 128 bits long, a standard for UUIDs. As mentioned before, the advertised payload with BLE is only 32 bytes in size. In addition, the payload needed to be in byte array (byte[]) form. This meant, as we were sending two types of data, we had to combine them into one byte array. Fortunately this was made trivial with the use of the string methods, such .getBytes(), which allowed us to convert

both data and combine them. Finally, due to the always changing nature of zip codes, the advertiser always needs to update its data to the current zip code location the user is in. To do this, we made sure to destroy and re-instantiate the advertiser every few seconds. We decided 5 seconds was a good interval. This means that every 5 seconds the advertiser would cease its service and then start again with a brand new zip code.

The other component of the BLE system is payload receiver/scanner. The scanner would receive data from a device with the same UUID. With the bluetooth receiver/client, we had to make sure that payload was the correct type. This was done by slicing the byte[] received from the advertiser. The combined ID and zip code data was ordered, meaning the byte array could simply be split and sliced. The first 5 elements of the byte array (5 bytes) corresponded to the zip code while the 6th up to the rest was for the unique ID (8 bytes). Overall, the bluetooth portion of the project was the most complicated task we had to complete

For the third task, which is retrieving zip code location, we achieved this by using Android/Google's Geocode API, which essentially allowed us to retrieve the zip code by getting the latitude and longitude values. The use of this API required our app to check location permissions from the user. Furthermore, we had the setting for zip code retrieval to be frequent, with the checks occurring every 5 seconds or so. Overall, getting the zip code was relatively simple.

For the 4th and 5th tasks, Achieving them was pretty straightforward. Getting the list of IDs with differing zip codes as the user was easily accomplished by comparing the zip code strings and checking if they are the same or not. If they weren't, then that zip code would be added to the list published to the server. As for the posting part, it was also quite easy. We used the Java Net library in order to create the Post request mechanisms. We published JSON Object to the server which followed the format {"from_user":..., "spotted_users":..., "time":...} created by Server Team, where "from_user" is the ID of the device, "spotted_users" the list of IDs that have a different zip code compared to the user and "time" as the timestamp. Posting to the server had to be done on an hourly basis due to server limitations. The server, which was hosted by Heroku, allowed only hourly updates of the blacklist. With this in mind, we had to make it so that we were publishing data every 60 minutes.

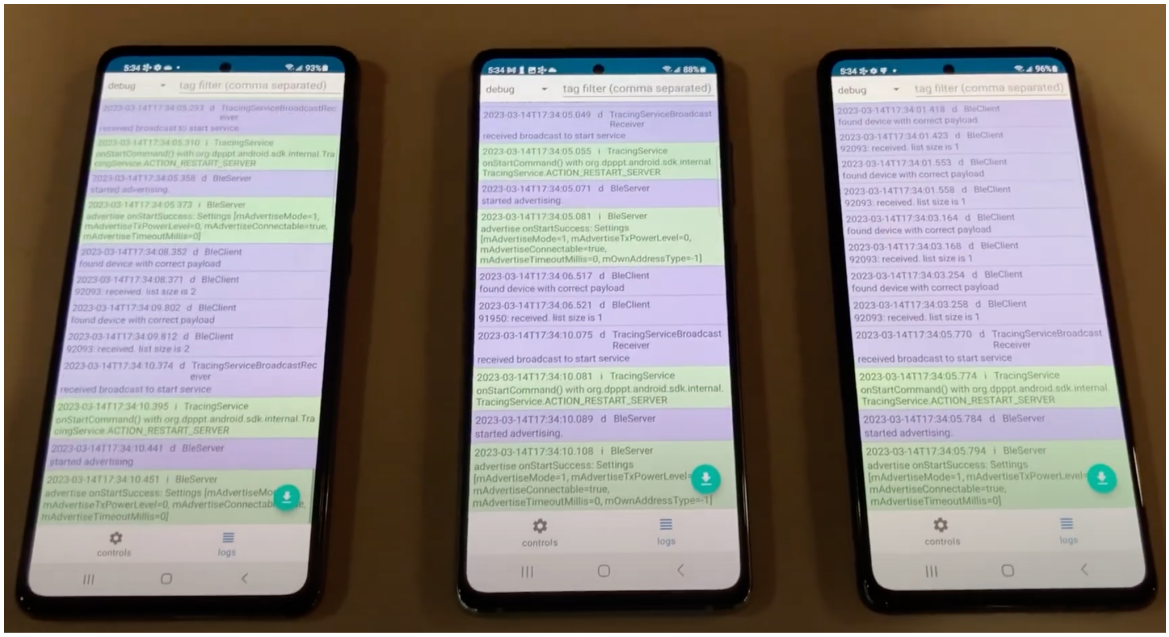
Finally, making a GET request to the server was also relatively simple and we had to only check the output, which was the blacklist, and whether the ID of the user was there. If the ID of the user was found in the blacklist, then they were blocked from participating in the tracking process.

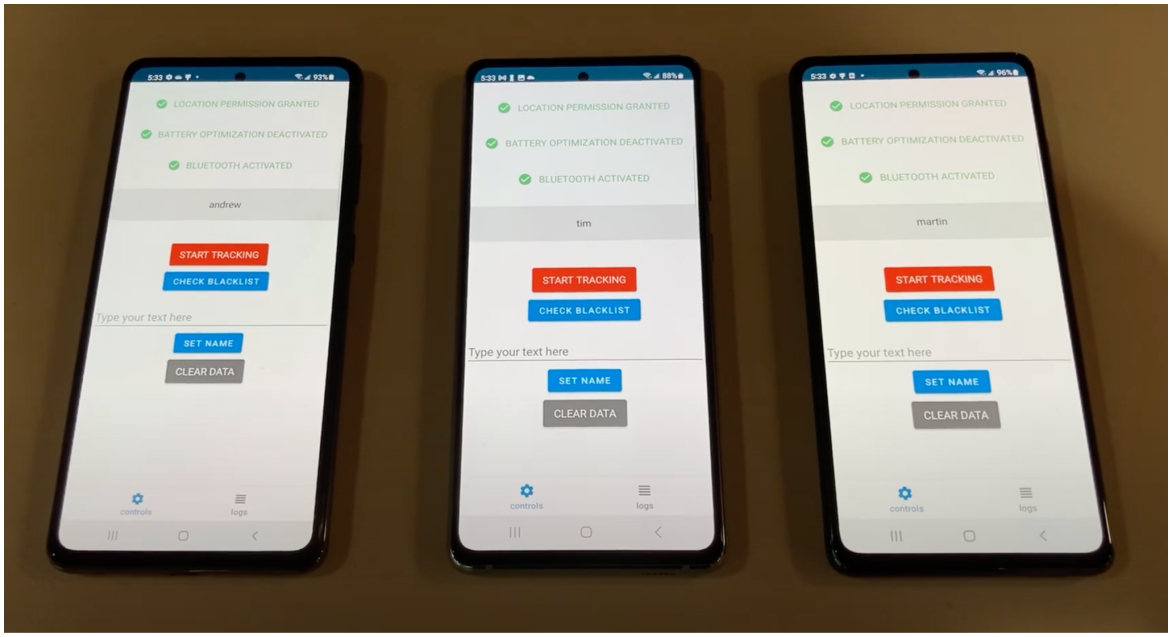
After completing these 5 goals, we had to thoroughly test the application. 3 smartphones for the whole process to work. It required 3 because the majority-based algorithm required the non-spoofer phones to be a majority. With two phones, having a majority was not possible. For testing purposes, we hardcoded 1 of the 3 phones to have a different zip code. We

Results

App demonstration: https://youtu.be/stuOTvUJUmk
Proxensus Github Page: https://github.com/acanonig/DSC180B-Proxensus-

Application User Interface





Logs page showing interaction between devices

Discussion of Results

Proxensus successfully creates unique identifiers per user - partially based on the username and partially based on random number generation - upon activation of the application. The application successfully uses Google API to grab a phone's projected location data. For honest users, the application will pull the user's current zip code. For location spoofers, the application will pull the fake location's zip code, as intended. One occasional annoyance involves a slight inaccuracy with retrieving the correct zip codes. Two honest phones in very close proximity to each other can display two different zip codes, but those two zip codes are adjacent to each other. Fine tuning with pulling location data would be necessary to fix this.

The application's user interface includes a start tracking button which turns on the phone's bluetooth low energy functionality. When phones are in proximity of each other and are currently tracing, the application creates logs that display successful communication between the phones. In other words, unique identifiers and corresponding zip codes are exchanged properly.

We have successfully established a link between the android phones and the server database. Upon exchange of data between users, any data discrepancies result in the user sending the data of the conflicting user to the database. Once the data is handled by the server side and a blacklist of spoofers are generated, users are able to check if they are found on this blacklist. The application has a button that cross checks with the database whether a user's unique identifier is found on this blacklist and notifies that same user their status as an honest user or spoofer.

Because our application relies on crowdsourcing data and majority voting, one weakness of the methods is that the majority vote is not necessarily the correct answer. For example, if there are more collaborating spoofers with the same lying location data than honest users within proximity of each other, the honest users would lose in the 'majority vote' on the server side and be put on the blacklist. In the future, our application can fine tune parameters such as thresholds necessary for the voting algorithm to be activated. There are many more improvements to be made about the application, such as having IDs be generated based on a changing seed or time, making it more secure and less vulnerable, similar to the literature that we have studied with DP-3T and other similar techniques.

References

Veale, and Reslbesl (2020). “DP3T White Paper.” Github 2021, https://github.com/DP-3T /documents/blob/master/DP3T%20White%20Paper.pdf.

DP-3T. “DP3T SDK Backend.” Github, 2021, https://github.com/DP-3T/dp3t-sdk-backend

S. Arunkumar, M. Srivatsa, M. Sensoy and M. Rajarajan, "Global attestation of location in mobile devices," *MILCOM 2015 - 2015 IEEE Military Communications Conference*, Tampa, FL, USA, 2015, pp. 1612-1617, doi: 10.1109/MILCOM .2015.7357675.