A Privacy-Friendly Architecture for Mobile Social Networking Applications

by

Sarah Pidcock

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

Abstract

The resources and localization abilities available in modern smartphones have provided a huge boost to the popularity of location-based applications. In these applications, users send their current locations to a central service provider and can receive content or an enhanced experience predicated on their provided location. Privacy issues with location-based applications can arise from a central entity being able to store large amounts of information about users (e.g., contact information, attributes) and locations (e.g., available businesses, users present). We propose an architecture for a privacy-friendly *location hub* to encourage the development of mobile location-based social applications with privacy-preserving features. Our primary goal is to store information such that no entity in our architecture can link a user's identity to her location. We also aim to decouple storing data from manipulating data for social networking purposes. Other goals include designing an architecture flexible enough to support a wide range of use cases and avoiding considerable client-side computation.

Our architecture consists of separate server components for storing information about users and storing information about locations, as well as client devices and optional components in the cloud for supporting applications. We describe the design of API functions exposed by the server components and demonstrate how they can be used to build some sample mobile location-based social applications. A proof-of-concept implementation is provided with in-depth descriptions of how each function was realized, as well as experiments examining the practicality of our architecture. Finally, we present two real-world applications developed on the Android platform to demonstrate how these applications work from a user's perspective.

Acknowledgements

I thank my supervisor Urs Hengartner for guiding me through the work on this thesis and providing helpful feedback along the way. I also thank Bisheng Liu for help he provided, especially with the use cases, and Doug Stinson for assisting me with administrative tasks. Finally, I thank my thesis readers (Ian Goldberg and Ian McKillop) for showing interest in my work and for providing their time and useful feedback.

Dedication

I dedicate this thesis to the people in my life who always believed that there was light at the end of the tunnel.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

With the evolution of smartphones into powerful, pervasive computing devices that people can always have at their disposal, a broad range of applications making use of capabilities offered by smartphones has materialized. Many of these applications make use of localization features (which can determine a device's geographical location with varying levels of accuracy) that the majority of smartphones contain. Localization in varying degrees of accuracy can be done using the strengths of signals from nearby cellular network base stations, broadcasts from visible Wi-fi access points or by calculating latitude and longitude coordinates from signals emitted by Global Positioning System (GPS) satellites. With GPS, the location of a smartphone can be determined to within 10 metres [21]. This level of accuracy provides rich opportunities for location-aware applications and services whose functionalities are enhanced by the usage of users' location data [47].

The availability of localization has encouraged the development of smartphone applications driven by users' location data. To make use of these applications, users send requests to a service provider that include their current location. One possible outcome is that the service provider can reply to the user with content that is associated with or filtered based on her location. The service provider can also update the user's state in the application, which may trigger events at the current time or in the future. Many popular location-based applications fall into the following categories:

- **Mobile Social Networking.** These applications facilitate social interaction between users that are geographically close to each other. Friend location and proximity detection features help users that are already connected by the social network's notion of friendship meet in person. Interest matching can also be employed to make more specific matches between friends, as well as provide a social discovery service for users in close proximity with similar interests. FourSquare [5] and Google Latitude [8] are real-world examples of applications that provide friend location services. Location-based social networking applications have received a lot of attention recently in the smartphone development community [43]. Popular applications for meeting nearby people in the Google Play [10] store for Android smartphones include Badoo [3], Skout [13], and myYearbook [11].

- **Content Storage and Delivery.** These applications are often based on providing information about points of interest or enriching data by attaching a location (geotagging). Content that is served to users (either on request or through push notifications) is filtered based on their current locations and additional criteria they add. Functionalities that can be provided in this manner include location-based search, location-based recommendations relating to points of interest and location-targeted advertising. A popular implementation of location-based search is the functionality provided by Google Maps [9] that finds points of interest matching keywords provided by a user near her current location. Location-based advertising is often served through mobile frameworks such as the Google AdMob Ads SDK for Android [7].

- **Navigation.** These applications calculate the best possible route from the user's current location to her desired destination. Turn-by-turn directions are provided to the user and obstructions such as traffic and construction may be taken into account. Applications for the Android smartphone platform that provide navigation features include Waze [15], Garmin Mobile [6] and Google Maps [9].

The benefits of location-based services stem from the inherent relevance that proximity has in many use cases. When a user is searching for content that she would like to use in the near future, results that are more than a short distance away from her current location are unlikely to be useful. A similar argument applies to a user wishing to meet others with similar interests. From the perspective of mobile advertisers, a user is more likely to be engaged by an advertisement related to a nearby point of interest. The availability of location data for users also allows for the serving of ads that are both location-sensitive and time-sensitive (i.e. a short-lived deal that is only valid at a certain location).

The most concerning drawback of location-based services is the privacy problem presented by the frequent use of localization features on smartphones. A user must provide her location in order to request any information from an application or use any of its features. The service provider component of the application that receives all of the users' requests can build up large amounts of location data along with timestamps indicating when the user made the request. This data allows the service provider to localize users (know their location at a particular point in time), build up traces of users' movement and even track users. The service provider may also have identifying information and attributes (such as interests) that users have provided during registration or are part of their profiles in a social application. Knowledge of a user's location at particular times, especially when combined with other information about the user (such as their interests and identity), can be used by malicious entities to violate the user's self-determination of their own information. Data about a user can be collected from application requests, stored over a long period of time, and aggregated or statistically analyzed to learn more about the user than she may have intended.

In existing research, there are many examples of techniques that can be used to violate a user's privacy given some collection of her location information that has been obtained by an adversary (either a malicious service provider or a third-party that has received data a user did not intend for it to have). An alarming amount of information about a user's home and work locations, activities and relationships can be inferred from data that the service provider has available [36, 45]. An adversary can even determine a user's home location and identity when given pseudonymized [38] or anonymized [53] location data and some outside information (such as a movement profile [23, 53]). Golle and Partridge used data from the U.S. Census Bureau to determine the size of an individual's anonymity set in the general population given an inference of home and/or work locations [44]. Within a census block, a user was unique if both home and work locations were known. A user still faces privacy risks from only occasionally exposing her location to a location-based service as her home location and points of interest were identifiable from such a limited set of data points [26]. Gruteser and Hoh showed how trajectory information could be used to link a user's periodic anonymous location samples [32].

From a practical standpoint, there are recorded cases of users' sensitive data, including location, being misused for an unintended purpose (i.e. a purpose users did not consent to when they originally agreed to provide data) [4]. Static analysis (i.e. tracing of sensitive data through decompiled code) of 24,350 Android applications done by Gibler *et al.* [30] found 57,299 potential privacy leaks in 7,414 applications. 939 applications contained a total of 3,405 potential location data leaks. This analysis only traced through code without any consideration for its structure, which means the number of leaks during a live run of the code could be much more significant. For example, consider that if location data was being processed in a loop (which is quite common because many applications receive data from GPS hardware that makes a phone's location available as much as once per second) and being leaked during each iteration, the number of leaks generated by one small piece of code could be very large. Many of the leaks were found to be in advertising libraries that allow developers to generate revenue by serving in-application ads. These advertising libraries use users' locations to target them with specific ad categories in order to maximize the chance that the user will click on the ad. Dynamic taint analysis (tracking of sensitive data from its source through applications to an outgoing network transmission) also presented some alarming results [25]. Among a random sample of 30 applications taken from a collection of the top 50 applications in each Android Market (now the Google Play Store) category (a total of 1,001 applications), half of them exposed location information to advertisers without declaring this intention in the End-User Licensing Agreement (EULA) shown at installation time. The leaking of data to advertisers is especially concerning because many applications that are not location-based services (such as Angry Birds [1,2], a very popular mobile game) are offered in a free, ad-supported version as well as a paid, ad-free version [39]. In a sense, this practice is putting a price on users' privacy without them even realizing it. This problem emphasizes the need for a privacy-friendly architecture for building any application that touches a user's location data.

1.2 A Privacy-Friendly Architecture for Mobile Social Networking Applications

Providers of location-based services (either as the primary focus of an application or one of many functions) essentially become *location hubs*. Essentially, this means that they are centralized repositories of location information collected through servicing application requests. These service providers also store identifying information about the users making requests, which is undesirable from a privacy standpoint. This research focuses on designing a privacy-friendly location hub (i.e. a location hub which provides features to protect users' location data from misuse) that can be accessed as needed by applications requiring location-based functionality. We also aim to make our design flexible and simple enough to encourage development of privacy-friendly applications.

Our design consists of an architecture built with protecting the association between a user's identity and her location in mind. The foundation of our architecture is two entities, one which stores information about users and another which stores information about locations. These entities expose basic API functions that can be used as building blocks for mobile social networking applications that prioritize privacy preservation. Users communicate directly with these entities to store data in such a way that no entity learns both their identity and location and can either directly retrieve data or ask an application's supporting cloud component to retrieve data on their behalf to participate in social networking activities. In addition to basic get/set functions for data, our architecture also provides a callback framework to support scenarios where a user wishes to be notified when a condition is met (e.g., another person that wishes to play basketball checks in) at their location. A wide range of location-based mobile social applications can be built using the constructions provided by our architecture. We make the following contributions:

- We present the design of a privacy-friendly architecture for building mobile social applications that stores data about users and data about location separately.
- We implement the complete architecture and provide experimental results regarding the performance of the architecture. Our implementation is deployed using Python code for server components and the Android platform for the mobile devices.
- We build two real-world applications to demonstrate the practical use of our architecture.

In the next chapter, we look at some of the previous research that provides a foundation for our work. In Chapter 3 we describe the architecture of our system in detail. We detail our implementation of the system and provide results from experiments on the implementation in Chapter 4. Finally, we discuss possible future work in Chapter 5 and conclude in Chapter 6.

Chapter 2

Related Work

This chapter discusses existing research into strategies for protecting privacy in location-based services, online social networks and on smartphones, which are areas that either feature prominently in our work or show interesting parallels. Then it explores privacy-preserving methods for specific applications that focus on location information. We conclude by examining architectures that incorporate some of the previously discussed research and are similar in spirit to our work.

2.1 Location Privacy in Location-Based Services

Privacy issues in location-based services have been addressed in the past by data protection mechanisms and location privacy frameworks, but many of these approaches either fail to provide the necessary protection for users [51] or are inflexible in terms of supporting a wide variety of use cases. For example, the protocols presented for private proximity detection of friends [40, 41, 54] do not support any other location-based applications (e.g. matching of strangers, local search).

Reducing precision

The user's coordinates are replaced with a larger geographical area in a pre-determined division of space. For example, a location-based service could provide users with a grid that maps GPS coordinates onto labelled 1 kilometre-wide squares and users would only provide the identifier of the square they are located in while making a request [24]. This approach is not appropriate because the results from most location-based applications become less useful (or completely useless in the case of navigation) as precision is reduced. Without some form of anonymization, the amount of precision that would have to be removed in order to guarantee sufficient privacy would significantly reduce the flexibility to support a wide range of applications of any system employing this approach [38].

Spatial and Temporal Cloaking

The goal of this type of approach is to ensure that every request to the location-based service cannot be differentiated from a threshold number of other requests. This scheme has the same goal as the k-anonymity principle, in which every record in an anonymized database must be indistinguishable from $k-1$ other records. A trusted anonymizer [31] or a decentralized mix network [17,49] is employed to remove any identifying information in the request and to reduce the precision of either the spatial (location coordinates) or temporal (timestamp of the request) data based on how tolerant the particular location-based service is of imprecisions in either dimension. For example, a social networking application where a user is trying to meet other people in her area is fairly intolerant of location imprecision because the user probably does not want to travel far to meet up with a match provided by the application. On the other hand, the application can tolerate time imprecision (the delay necessary for the anonymizer to build up enough similar requests) because the user has already committed to spending time in the area to meet people and a little bit of extra time will not be an issue. The benefit of the multi-dimensional cloaking approach is that it can be tuned to meet the needs of different types of applications. However, the k-anonymity approach has been shown by research to be inappropriate as a method for protecting location privacy [51].

Private Information Retrieval

Queries to a location-indexed database owned by an application are executed using Private Information Retrieval (PIR) protocols [28, 37, 42]. When PIR is used for a query, the database cannot learn any information about which records were retrieved [20]. Approaches using PIR typically tile the geographical area that the location-based service covers with a grid of a pre-determined (by either the user or the application) granularity. Each location that has data associated with, or Point of Interest (POI), is stored as one record in a location-indexed database and each record is mapped to the region containing it. To retrieve points of interest, a user maps her location to grid square and requests all records mapped to that region using PIR techniques. The main drawback of using PIR in location-based services is that it is not a good solution for services where data changes frequently, as writes to a PIR-enabled database require expensive recalculation to re-map records.

Secure Multiparty Computation

Calculations on spatial data are performed such that the inputs remain private to whoever provided them but all entities know the result. Typically, inputs are encrypted and algorithms with the ability to manipulate ciphertext are used. Approaches based on Secure Multiparty Computation (SMC) have been used in the context of location data to determine the intersection of spatial datasets such that all parties only know the result [29], to determine if two users are within a threshold distance without either user revealing her location to the other [40, 41, 54], and finding a fair meeting point without revealing any of the users' locations to a third party or other participants [18]. The main drawback of SMC is that techniques are application-specific and would not generalize well to a wide range of location-based services.

Summary

The location privacy protection mechanisms presented above are either not flexible to support many use cases or are insufficient for protecting users' location privacy. Both of these problems motivate a different approach to providing location privacy protection because this work aims to provide the means to develop applications with effective privacy features fitting a wide selection of use cases. Reducing precision only supports use cases that can tolerate imprecision in location data. Spatial and temporal cloaking addresses some of the problems of imprecision tolerance by allowing both location data (i.e. geographical coordinates) and temporal data (i.e. the time when a location data point was acquired) to be obfuscated based on the needs of a particular application [31]. However, this approach relies on the concept of hiding a user among a set of other users, which is shown to be susceptible to attacks presented by Shokri *et al.* [51]. Private Information Retrieval only efficiently supports use cases where the database of location data is not frequently changed. Approaches that utilize SMC are tailored to fit the needs of particular applications. This is impractical for a generalized architecture because each new use case would need to have an SMC-based algorithm added to accommodate it.

2.2 Data Protection in Online Social Networks

Protecting users' sensitive data in online social networks has also been the subject of considerable research. While our work does not focus on privacy in traditional online social networks, ideas along the lines of privacy by design and separating storage of data from social networking functionality are still applicable.

Tootoonchian *et al.* [52] proposed Lockr, which employs the principle of decoupling content that a user stores in online social networks (OSNs) from the functionality it provides. Lockr obfuscates the social graph from OSNs by using social attestations to form relationships between users. When a user requests content belonging to another user, she must present a social attestation. The social attestation is proven to be a property of the presenter in zero knowledge and compared to access control lists associated with content by their creators to determine if the owner of the attestation should be able to view the content.

Baden *et al.* [16] presented Persona, which is a social network designed specifically with privacy in mind. Each user partitions her friends into groups based on logical combinations of attributes. All data stored in the social network is encrypted using attribute-based encryption, in which each access structure, formed from a logical combination of attributes, is assigned a key and an item of data is encrypted with the key matching the desired access structure.

Shakimov *et al.* [48] presented Vis-à-Vis, a privacy framework for online social networks in which each user has her own Virtual Individual Server (VIS). A VIS runs in the cloud, stores the user's social network data, and interacts with entities requesting the user's data. Location is treated as a special attribute and users can define hierarchical groups that they are willing to share their location in varying granularities with. However, running a VIS in the cloud has a financial cost for a user, which is undesirable because existing social networks are free to use and a user may not be willing to pay for privacy. Also, the user's cloud provider can see her data, which may not provide sufficient privacy for some users.

2.3 Privacy for Smartphones

Application-agnostic solutions have been developed at the operating system level for dealing with leakage of sensitive data. The idea behind low-level tracking is to monitor the flow of sensitive data and react appropriately without modifying any applications. However, our work focuses on privacy built into applications and reactive approaches that act externally on applications (that were not developed with privacy in mind) at runtime do not fit in with that model. Also, reactive approaches require modification of a smartphone's operating system to provide hooks for data monitoring. Such a requirement presents a barrier to widespread deployment because users may not want to perform the risky operation of loading a custom operating system on their smartphones. The two approaches presented below both provide monitoring of the propagation of sensitive data but differ in how they address reacting to possible privacy leaks.

Enck *et al.* [25] presented TaintDroid, an operating system extension for the Android smartphone platform that tracks which running applications are handling and sending sensitive data. Data that has the potential to be sensitive (e.g., GPS coordinates) is labelled (tainted) at the source and interactions with applications and other data are dynamically monitored to determine if sensitive information is being leaked. The goal is to be able to identify misbehaving third-party applications without incurring too much of a performance penalty. TaintDroid takes advantage of the different layers in the Android system architecture to track data without having to perform static analysis of untrusted application code. At each level, the monitoring method is designed to exploit the semantics of data transmission at that level (e.g., between declared variables, messages between applications, between methods, between files). This approach only addresses monitoring of sensitive data and notification of possible privacy leaks but leaves the reaction to such notifications to the discretion of whoever is performing the monitoring.

Hornyack *et al.* [34] designed AppFence, which acts as an agent that determines whether or not an application requesting particular data should be granted access. In addition, if an application is granted access to data, AppFence also decides if the application should be allowed to transmit the data out on the network interface. Applications that only manipulate data locally on the smartphone (such as a contact list viewer) can function normally under this model. If an application is denied access to data, shadow data is substituted in place of private information in a manner such that a consistent view of data is presented to each application (but may be differ across multiple applications) and the modification is difficult to detect. External transmission of data is detected by monitoring network stack activity and outgoing network communication. Experiments assess user-visible differences between applications run with and without AppFence privacy controls.

2.4 Specific Mobile Social Applications Built With Location Privacy in Mind

Location privacy is a focus in the area of mobile social applications. Interesting solutions for location sharing with privacy by design have been presented. These solutions are not flexible and are tied to specific applications, which make them inappropriate for building a comprehensive privacy framework that supports many use cases.

Cox *et al.* [22] designed SmokeScreen, which is a decentralized application designed specifically for the privacy-preserving location sharing use case. Each user assigns each of their friends to cliques representing levels of trust and negotiates a secret key for each clique. When the user would like to share her location, she computes a clique signal for each clique she would like to share their location with using that clique's secret key and broadcasts it using short-range signalling capabilities (such as Bluetooth or Wi-Fi) on her device. Any users that are in the area will detect the clique signals and can determine who created them if they in are in any of the cliques represented by the signals. In order for a user to share a location with a stranger, she must broadcast an opaque identifier in addition to her clique signals. Any user that detects this identifier can make a request to a centralized broker for the identity of the user who created it. The broker coordinates exchanges of location information based on which opaque identifiers users have made requests for.

Protocols using secure multiparty computation mentioned in section 2.1 have also been incorporated into specific location-privacy friendly applications. Examples include private proximity testing [40, 41, 54] and Fair Rendez-Vous Point determination [18].

2.5 Privacy-Friendly Frameworks for Mobile Location-Based Applications

There are many academic solutions that utilize privacy by design for mobile location-based applications that are application-specific. However, the need for solutions that support a wide variety of use cases has led to the creation of more generalized frameworks. These frameworks provide data storage and public API functions that are used to manage data. Also, the concepts of preventing third parties from knowing enough information to determine both a user's identity and her location and decoupling storage of data from social networking functionality are employed.

Puttaswamy and Zhao [46] designed a framework for location-based mobile social applications with the goal of avoiding location data being stored unencrypted on untrusted third-party servers. In their solution, servers in location-based services are encrypted data stores and the users employ offline key exchange to share their decryption keys with their friends. Two cryptographic building blocks and a narrow storage system interface form the basis of the system. A friendship proof is used as a cryptographic validation of the social connection between two users and a transaction proof is attached to each item of encrypted data that is used to cryptographically establish ownership. The storage system interface simply allows users to put encrypted attributes (information about users) in their profiles, get encrypted attributes from stored profiles, store encrypted data to a location, and retrieve encrypted data from a location. One problem with this system is that a user must download all encrypted data at a location and attempt to decrypt each item with all keys in their possession to read data produced by her friends, which can be quite expensive computation-wise. Another problem with the design is that it can only be used to build applications where users request data and receive results immediately. Functionality where users can register triggers for certain conditions (like other users arriving at their location) and have data pushed to them when the conditions are met is not supported. Applications that are supported only include those that operate on existing real-world connections and not scenarios involving strangers being matched. Data about users and data about locations are stored on the same server, which presents a traffic analysis problem. If a user first updates her encrypted location attribute that is part of her profile and then adds an entry for herself to the location database, the server can link the two updates.

Jaiswal and Nandi [35] proposed an approach that divides knowledge of a user's location and a user's queries to a location-based service (which may contain identifying information and data about interests and social relationships) between two non-colluding entities. One entity is responsible for creating and maintaining a mapping between actual locations and pseudonymized locations (unique identifiers assigned to regions in a grid overlaying the physical location space) while the other is responsible for creating and maintaining mappings between actual identifiers and pseudonymized identifiers (unique identifiers assigned to each user and business in the location-based service). Users register triggers with an application that specify which pseudonymized identifiers they are interested in. A matching service determines if any pseudonymized identifiers of interest are located in the same region as the user. The matching service is provided by multiple non-colluding components (possibly contributed by the end users themselves) because a single entity with access to all user requests could infer the mapping between pseudonymized locations and actual locations. This system supports returning results to the user at the time of their query (pulling) as well as pushing matches to users when they change location. However, it does not provide functionality for users registering triggers related to the activities of businesses or other users and having matches pushed to them when the conditions of the trigger are met. Whereas the authors mention that their approach can be employed for locating nearby friends, this claim is questionable. The problem is that, as mentioned above, the architecture assumes that the LBS provider knows the locations of the entities that should be located. Whereas this is a reasonable assumption for businesses, the assumption cannot be made if the located entity is a user. The assumption would conflict with the goal of the architecture, that is, preventing the LBS provider from knowing users' locations. Functionality-wise, the focus of Trust No One is on locating nearby businesses (or friends). It is unclear whether and how other geosocial applications can be implemented in the architecture. The system also does not prevent an adversary from registering many fake users, contributing the majority of the components in the decentralized matching service and collecting enough queries to use statistical techniques like those presented by Shokri *et al.* [50] to mount attacks on the pseudonymized locations.

Guha *et al.* [33] presented Koi, a platform that uses privacy-preserving location-based matching to avoid exposing fine-grained GPS coordinates to smartphone applications. Koi follows a similar principle to the work of Jaiswal and Nandi [35], which specifies that no entity in the system can know both a user's identifying information and location. The two main components in Koi are a phone agent that runs on the user's device and a service based in the cloud. The phone agent provides functionality to the applications for registering and updating items, which are attributes of an entity in the system, and triggers, which instruct the cloud service to complete a callback when a match to a desired item is found. The cloud service employs two non-colluding sub-components. The matcher assigns and maintains a mapping of random unique identifiers to each registered item or trigger (regID) as well as a mapping of unique identifiers to each registered attribute of an item or trigger (attrID). The combiner performs matchings on the obfuscated data when a trigger is registered by querying the matcher for attrID's associated with those mapped to by the trigger and the user that created the trigger (identified by regID's) and iteratively eliminating regID's that don't match. Once the process is complete, the combiner only provides the matcher with matching regID's and not the attrID's that contributed to the match to ensure that the matcher does not learn any associations between users and locations (since location is an attribute). The phone agent on the device of the user that created the trigger is then notified of the match through the registered callback. Koi can support any matching algorithm because the matcher has plaintext attributes available while answering the combiner's queries about relationships between attributes. Applications implemented using the Koi platform include a private mobile social network, as well as proof-of-concept navigation and local search and advertising services. Koi's main weakness is its susceptibility to traffic-analysis attacks, as the matcher can link attributes (which could include location) to a user by observing which attributes and users get updated/matched close in time. Similarly, since the matcher sees all location updates, it may be able to link nearby updates as being submitted by the same user and can track and ultimately re-identify a user [27].

Summary

The works presented by Puttaswamy and Zhao [46], and Jaiswal and Nandi [35] are workshop papers that do not address how a privacy-friendly architecture could be deployed in practice, which is a fundamental contribution of this work. The Jaiswal and Nandi work also does not address the same variety of uses cases that this work covers and does not sufficiently address the possibility of an adversary being able to reverse the mapping from actual locations to pseudonymized locations. The research by Guha *et al.* [33] was developed concurrent to this work and includes a full implementation, but is susceptible to traffic-analysis attacks in which the matcher can learn a user's location in addition to already knowing her identity. In the architecture presented by this work, the goal is to avoid all entities from learning both a user's identity and her location.

Chapter 3

Architecture

3.1 Introduction

In this chapter, we present the design of our privacy-friendly architecture for mobile social networking applications. The chapter is organized as follows. We state our design goals in section 3.2 and follow by presenting a high-level system model in section 3.3. In section 3.4, we describe the threat model and trust assumptions. We elaborate on the basic system model and describe the functionality of all entities in detail in section 3.5. The security analysis is presented in section 3.6 and we finish by discussing use cases for our architecture in section 3.7.

3.2 Design Goals

The main goal of our system is to design a privacy-friendly location hub that supports many types of mobile social networking applications. In this section, we elaborate on finer details of this overall goal.

Privacy-Friendly By Design

At the highest level, we classify being privacy-friendly as having less privacy than theoretical application-specific, privacy-preserving protocols but more privacy more privacy than widely deployed location hubs for geosocial applications (such as Foursquare). [1] There are two possible approaches to building privacy-friendly applications. One option is to attempt to limit data access and transmission to and from existing applications through external privacy controls. The other option is to build applications with privacy in mind from the beginning. The first option is reactive and may cause some applications not to function properly because they are being denied information that the original developers assumed they would have, which is undesirable from a functionality standpoint. Our goal is to create a location hub that encourages an approach to application development in the spirit of the second option. The location hub should provide the functionality applications need in a privacy-friendly way by design.

Decouple Data Storage From Social Networking Functionality

Existing social networks store all of a user's information and also manipulate stored data to provide social networking functionality for the user. Our goal is to present an alternative to this model that enhances privacy. Essentially, our architecture should store data in a manner consistent with our privacy goals and act as a gatekeeper for that data. Applications that perform social-networking-related functions should be able to request data and should only be granted access to information as desired by the original creator.

Division of Information

A user's location is not sensitive with no other information associated with it, and the same is true for a user's identity. However, when a user's location is linked to a user's identity, then there are privacy concerns. Our goal is to store and provide data in our architecture such that no entity should know both a user's identity and her location.

Avoid Significant Consumption of Client-Side Device Resources

Although the resources available in smartphones are steadily increasing, asking a device to perform a large number of computations (especially cryptographic operations) can still cause problems. Many users run power and memory-hungry applications (e.g., music players, games) that may not behave as users expect if another application with significant

[1] A privacy-friendly architecture that is both general and as privacy-preserving as existing academic application-specific protocols is future work.

resource-based needs is running at the same time. In our framework, cryptographic operations are likely to cause the most computational load and drain on power resources. Our goal is to reduce or avoid decryption by trial and error. More specifically, we would like to avoid a scenario where a client downloads a batch of data and has to try many or all of her decryption keys on all items of data in the batch in order to produce a human-readable collection of data.

Support Many Application Use Cases

Privacy protocols that are designed for specific applications exist in research. However, many of these solutions are complex and inappropriate for use as a foundation for developing privacy-friendly applications. Our goal is to build a flexible architecture that exposes basic functions that can be combined to meet the needs of many applications. This simple model for building functionality eases the development of privacy-friendly applications, which is desirable in real-world deployment.

3.3 System Model

This section gives a high-level overview of the entities in our system and the communications between them. We intend for our system to be used to develop location-based mobile social applications in which users will want to store both information about themselves and information about locations (e.g., if the user is present at a location, a review a user has for a business at a location). To ensure that no entity can know both a user's identity and her location (division of information goal), there are two separate entities, **U** and **L**, for storing information about users and information about locations. In addition, **U** and **L** only store information and do not provide any social networking functionality. Information is requested from both based on the needs of protocols of applications running on other entities.

3.3.1 Overview of Components

Our architecture consists of the following entities:

- **U**: The user-indexed database that stores information about users.
- **L**: The location-indexed database that stores information about locations.

- Device $\mathbf{D_i}$: Runs mobile social networking application **k** by interacting with **U** and **L** and maybe with $\mathbf{C_k}$ under the control of user **i**.
- $\mathbf{C_k}$: Optional component of mobile social networking application **k** that runs in the cloud under the control of application provider **k**. For simplicity, **C** will be used to indicate a particular $\mathbf{C_k}$ or a notable set of some or all of them.

A high-level outline of communications between the entities, all of which are secured with SSL, is shown in figure 3.1.

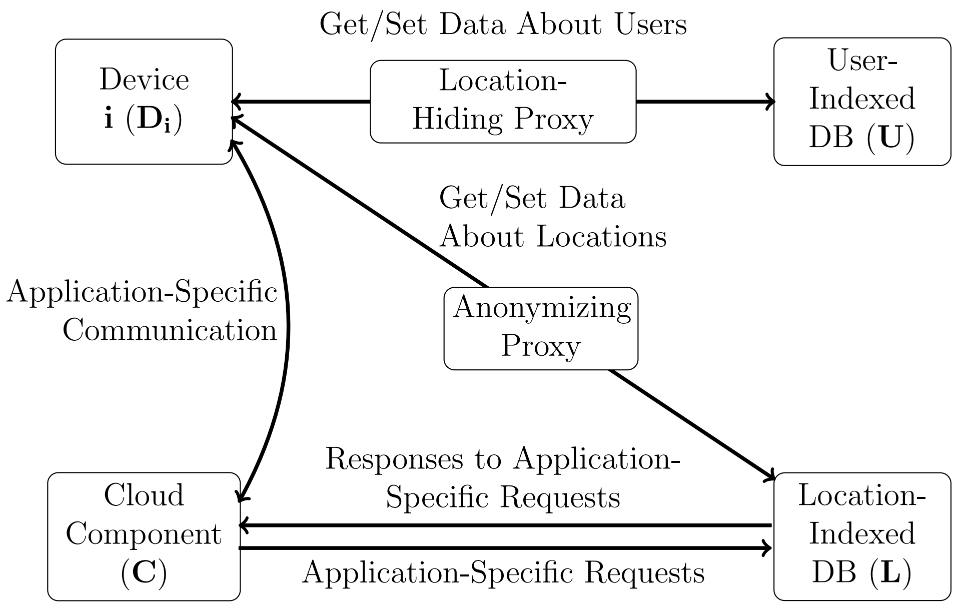


Figure 3.1: System Architecture

3.3.2 Notation

The following notation is used frequently when discussing our framework:

- uid_i: A unique identifier with user **i** in **U**. For simplicity, *uid* will be used to refer to unique identifiers for users in general.
- cid_k: The unique identifier associated with application **k** in **U**. For simplicity, *cid* will be used to refer to unique identifiers for applications in general.

- `ttl`: The time-to-live value required by some API functions that indicates how long the creator would like the data item to persist before it expires and can no longer be returned in responses to API requests to retrieve data.

3.4 Threat Model

In our system, **U**, **L**, and **C** are *honest-but-curious* and do not collude. The public interfaces of these entities will perform according to specification, but any of them may try to discover additional information using any data they store or requests they handle. **U** is trusted to store information about users, but should not learn information about users' locations or other sensitive user information. **L** is trusted to keep information about locations, but should not learn information about users. $\mathbf{C_k}$ can learn information about users' identities but not users' locations. $\mathbf{D_i}$ is fully trusted to store and manipulate information for user **i** and will not share information that it knows about other users. **U**, **L**, and **C** will not register users in the system or collude with existing users in an attempt to learn more information.

We assume that all entities in the system have a key pair with all public keys certified by a common trusted Certification Authority in a Public Key Infrastructure. In addition, we assume that no entity will share its private key with another entity and entities will have the necessary public keys and certificates available when needed for SSL connections and verifying signatures.

To meet the goal of decoupling data storage from social networking functionality, attribute values in **U** must be encrypted with information-specific keys. We assume that users have access to out-of-band methods for managing and exchanging information-specific keys, as key distribution and key revocation are out of the scope of this research. Users will share information-specific symmetric keys with other users that they trust (i.e., their friends) so that their data can be used for social networking purposes, but **U** (which stores data) cannot decrypt it. We do not address the possibility of the existence of certain attribute names or the lengths of encrypted attributes revealing any extra information.

There are a few assumptions made about how users interact with a location-based service built using our architecture. We assume that users do not update their location with **L** (check in) at a location that is considered sensitive. For example, we assume a user will not check in at her own home or the home of any other user because background information external to the architecture can tie users to home locations. If users wish to store their home location in the system, they can put an encrypted value in **U** and give the key out when they wish to share it. Another assumption that we make is that users do not check in frequently along a contiguous route (i.e., we do not address the possibility of a user being re-identified through a tracking attack).

3.5 Architecture Details

This section presents each entity in our system in full detail. First, we present the design and the public API of the User-Indexed Database. Then we discuss the the design and the public API of the Location-Indexed Database. We go on to describe the responsibilities of cloud components and devices at a high level. To finish, we detail some operations that are useful to many applications.

3.5.1 User-Indexed Database

U provides storage for information about users in the form of a set of attributes. Attributes may be general and usable for many different applications (e.g., location) or application-specific (e.g., Facebook could register an attribute type for wall comments that is only requested by operations in their application components). A piece of data stored with **U** is indexed by a unique identifier (which could be based on the users' public keys) associated with its creator and an attribute name (e.g., "interest", "location", "friend"). A user could also have attribute names specific to which other users have access (e.g., "location_friends", "location_colleagues", "location_family") as defined by particular applications. Each attribute name can have multiple values and values can be encrypted with a symmetric key specific to the particular identifier and attribute name. User attribute entries contain a timestamp that indicates when they expire. For simplicity, user attribute entries cannot be explicitly deleted but their expiry times can be updated. Therefore, an entry could effectively be deleted by updating its expiry time to the current time.

When $\mathbf{D_i}$ makes a request to **L**, its IP address (which could have geolocation performed on it) must be masked. This can be accomplished by accessing **U** through a proxy. All requests made to **U** by $\mathbf{D_i}$ must be signed using $\mathbf{D_i}$'s private key in order to prevent other users from adding information about user **i** to the database. Requests made by $\mathbf{C_k}$ must be signed using $\mathbf{C_k}$'s private key to ensure that a user's data is not retrieved by an unauthorized cloud component. A timestamp with the current time is also included in all requests and U maintains a log of timestamps a user has presented. Every request is checked against the log to avoid replay attacks. Anyone can retrieve a piece of encrypted information but will only be able to decrypt it if the creator of that data has given them the correct keys.

Tags

Tags are attached to data stored in **L** to mark data as having been created by a user without revealing that user's identity. For example, a user can provide her tag when she checks in at a location so that **L** cannot know her identity but her entry can later be retrieved and used for location-based social networking operations by an entity that has the means to turn her tag back into her identity.

Tags are generated on request by **U** and mappings between tags and identities are maintained (either implicitly in the design of the tag or stored explicitly). **U** is the only entity that knows how to reverse the mapping. A user must request a new tag for each time she stores information in **L**. **U** also provides an access-controlled means for **C** to determine which user is associated with a given tag if needed for social networking functionality. Typically, $\mathbf{D_i}$ initiates a location-based social networking operation with **C**. **C** then retrieves data that is annotated with tags from **L**, requests the identities associated with tags from **U** and attaches these identities to data in the response to $\mathbf{D_i}$. $\mathbf{D_i}$ can use the identities to select the correct decryption key and each item of data in the batch returned from **C** only needs to have one decryption operation applied. This meets our goal of avoiding unnecessary decryption operations that do not result in human-readable data.

Public API Provided by U

U provides the following public interface to allow other entities to access and manipulate data:

- `tag` $\leftarrow$ `createTag(`uid_i`)`: Create a tag for user **i**, remember the mapping from tag to uid_i and expiry time (using a standard time-to-live), and return the tag. The mapping could be remembered by storing uid_i within the tag (e.g., in an encrypted way). This function is typically called by $\mathbf{D_i}$.

- `whitelist(`uid_i`, ` cid_k`)`: Add application **k** to the whitelist for user **i**. By whitelisting application **k**, a user informs **U** that she is using this application and hence **U** is allowed to hand over this user's tag-to-uid_i mapping to $\mathbf{C_k}$. This function is typically called by $\mathbf{D_i}$.

- uid_i ← `getIdentity(tag,` cid_k`)`: Return the identifier associated with a tag. This function will succeed only if application **k** has been whitelisted by user **i**. This function is typically called by an individual $\mathbf{C_k}$.

- `setAttribute(`uid_i`, attribute, value, ttl)`: **U** stores the attribute/value pair indexed by uid_i until it expires. The expiry time of the entry is calculated using the `ttl`. If this function is called multiple times for the same attribute/value pair the expiry time of the entry is updated using the new `ttl`. Values are encrypted, and only authorized users (e.g., friends) would have the decryption key. However, this is transparent to **U**. This function is typically called by $\mathbf{D_i}$.

- `value` ← `getAttribute(`uid_i `or` cid_k`,`uid_j`,attribute)`: Return the value(s) for the given uid_j/attribute pair as requested by user **i** or application **k**. There is no access control for this function but it may be added later since the observation that an attribute changes its value may leak information (e.g., if a user's value for the attribute name "location" changes at 8AM, then it can be reasonably assumed that the user has just left her home). This function is typically called by $\mathbf{D_i}$ or $\mathbf{C_k}$ and the identifier parameter is only used to make sure the entity making the request is registered with **U**.

3.5.2 Location-Indexed Database

L provides storage for information about locations in the form of a set of attributes. As with **U**, attributes can be applicable to different use cases (e.g., "checked in") or application-specific (e.g., Yelp could register an attribute type for reviews of particular locations). A predetermined set of locations is registered with **L**. A piece of data stored with **L** is indexed by a representation of a registered location (e.g., GPS coordinates, address, labeled region) and an attribute name (e.g., "checked in", "for sale", "tourist information"). Values can be compound data types that contain a user's tag so that an application that retrieves the data can trace back to the original creator as well as other relevant information (e.g., a description of an item for sale or a tourist guide). Attributes also have associated timestamps to indicate when they expire. For simplicity, attribute entries cannot be explicitly deleted but will be ignored and removed automatically from the database after their expiry time.

When $\mathbf{D_i}$ makes a request to **L**, its IP address (which could be used as identifying information) must be masked. This can be accomplished by building an SSL tunnel through **U** or accessing **L** through a proxy. $\mathbf{D_i}$ must also encrypt location data in all requests with **L**'s public key to avoid **C** from learning user **i**'s location during social networking operations where **C** makes requests to **L** on $\mathbf{D_i}$'s behalf. **C** cannot learn user **i**'s location because of our goal that no entity learn both a user's identity and her location.

L provides a callback operation for social networking applications. The idea behind this functionality is to allow **C** (on behalf of $\mathbf{D_i}$) to register with **L** for a particular condition (e.g., a new value being added for an attribute name) and to be notified when that condition is met. At the most basic level, the condition could be a new value being added for an attribute name at a location. Like attribute entries, callback entries have an expiry time and cannot be explicitly deleted. However, entries will be ignored once they expire and are deleted automatically. During registration, **C** must provide a URL for a public API function (handler) to which **L** can send an HTTPS request (in a predetermined format) for callback handling when the condition is met (for more details, see section 4.2.2). After each data storage operation, **L** must check all non-expired callbacks to see if any have been triggered (i.e., they match the attributes that were set). For any callbacks that are triggered, then **L** must call the associated handler. Once **L** has called the handler to complete the callback operation, **C** can do application-specific processing and provide results to users.

Public API Provided by L

L provides the following public interface to allow other entities to access and manipulate data:

- `setAttribute(tag, location, attribute, value, ttl)`: **L** remembers the attribute/ value pair for the given location until it expires (the expiry time is calculated using the `ttl` value). Sample attribute are "present", "reviews", "ads", or "coupon". **L** may have to remember multiple values for a particular location/attribute combination (e.g., multiple reviews). For the attribute "present", the value would be a tag. For the attribute "review", the value would be a review (annotated with a tag if it is encrypted). The `tag` parameter is recorded by **L** only for later reference to prevent tag replay attacks where false information is added for an existing tag.

  The value could also include a tag that identifies the creator of the value (e.g., for an encrypted review, this would make it easier to locate the decryption key). If there is such a tag, the tag could come appended with an additional entry that has more fine-grained expiration information (than ttl) about the stored information, but without **L** being able to learn this expiration information. This is a defence against traffic analysis attacks by **L**, where an entry expires and a user then inserts a new one. To implement this feature, a tag could be a public key, with the key pair created by

U, and **U** remembers the corresponding private key (and identity). Here, the fine-grained expiration information would be encrypted by $\mathbf{D_i}$ with the tag (i.e., public key).

L does not consider validity of the submitted values (e.g., whether a tag is valid). This function is typically called by $\mathbf{D_i}$.

- `token` ← `registerCallback(location, attribute, handler, ttl)`: **L** remembers a callback for the given location/attribute pair until it expires (the expiry time is calculated using the `ttl` value). The `location` parameter must be encrypted with **L**'s public key and may accept a description of a geographical region (defined by GPS coordinates and a radius). Namely, **L** calls the handler whenever an attribute for the location/attribute pair is set using the above function. The handler is a public API function provided by **C**. The function returns a token that can be used to uniquely identify the callback.

- `value(s)` ← `getAttribute(location, attribute)`: Return the value(s) stored for a location/attribute pair. The `location` parameter must be encrypted with **L**'s public key. The location data provided must be either consistent with **L**'s representation of location or **L** must be able to recognize the representation and convert it appropriately to a format that is internally recognizable (e.g., the `location` parameter may accept a description of a geographical region). To return a particular value, the location provided should map to the same location that was used for the value when it was stored with `setAttribute`. If GPS coordinates are used to classify locations, then **L** may also employ GIS support that allows it to determine the nearest neighbour(s) to a location provided in a query.

3.5.3 Cloud Components and Devices

$\mathbf{C_k}$ is the optional component of mobile social networking application **k** that runs in the cloud if required for scalability or privacy reasons. For example, consider an application for matching users with similar interests at the same location that only reveals information about a user to other users that she has been matched with. If a device were to perform matching operations on behalf of a user, the protocol would either need to run on unencrypted data from all colocated users (which violates the privacy requirement of only revealing such data if a match is made) or secure multiparty computation on encrypted data (which imposes too much of a computational load on the device). A cloud component supporting this application that has been whitelisted by individual users (i.e., can decrypt their attributes) can match users on plaintext attributes and only reveal information about a user to users she has been matched with. In addition, a cloud component has more computational power (that can also be easily scaled) than a device and can process a large number of matching operations.

At the very least, a cloud component must support identification of users by *uid* as all information it receives will contain a *uid* or a tag that maps back to a *uid*. Also, a cloud component must store a mapping for each user **i** from uid_i to the user's certificate and verify the identity of requestors. Some cloud components may also manage callbacks on behalf of users and manipulate data returned from callbacks to provide social networking functionality.

The device $\mathbf{D_i}$ (owned by user **i**) is responsible for accepting requests through a user interface, communicating with the necessary entities for the application the user is running, and presenting the results to the user in a human-readable form (on the user interface). The design of the device component is left to the developers of the application. There is no public API for the communication between $\mathbf{D_i}$ (for any user **i**) and **C**. It is up to the developers of each application to design their own protocol.

3.5.4 Operations Common to All Applications

Our framework is designed to support the functionality of many applications but there are some operations that are useful to all applications.

Registration

To register a user **i** with the architecture, $\mathbf{D_i}$ generates a symmetric key for each attribute type (e.g., interest, friends, contact information). Then $\mathbf{D_i}$ contacts **U** to provide their public certificate and identity as well as a list containing cid_k for each application **k** that they will be using. **U** verifies that the certificate belongs to the user, assigns a new unique identifier uid_i to the user (which could be based on the user's public key), adds the desired applications to the user's whitelist and returns the identifier to the $\mathbf{D_i}$. Then $\mathbf{D_i}$ uses uid_i to register its public certificate, and symmetric keys as appropriate with **C** (i.e., a user only provides keys for attributes that they want the cloud component to know). All of these operations must be cryptographically signed so that **U** and **C** can verify that $\mathbf{D_i}$ is the owner of the private key matching the certificate it provided.

Making Social Connections

Social networking applications are built on a foundation of users making connections to each other. In most social networks, social connections are made by becoming "friends". In our architecture, users connect by sending each other their *uid*s, and attribute keys (used to encrypt attribute values stored in **U**) for the attribute types they would like their friends to view. This process does not have to be symmetric, as user **i** can send her "location", "birthday", and "interests" attribute keys to user **j** but user **j** can send a subset of those keys or keys for a completely different set of attributes to user **i**. For two users **i** and **j** that are friends to use functionality based on existing social connections of a particular application **k** that has a supporting cloud component $\mathbf{C_k}$, both $\mathbf{D_i}$ and $\mathbf{D_j}$ must whitelist $\mathbf{C_k}$ and send it their "friend" attribute keys. To register user **i** as a friend of user **j**, $\mathbf{D_i}$ must call **U**'s `setAttribute(`uid_i`,‘‘friend’’,<encrypted `uid_j`>,ttl)` and $\mathbf{D_j}$ must call **U**'s `setAttribute(`uid_j`, ‘‘friend’’,<encrypted `uid_i`>,ttl)` (where *uid* in the `value` parameter are encrypted using the sender's "friend" attribute key) and the `ttl` value is high to allow the connection to persist for a long time.

Location Update

Our framework is designed as a location hub in which a user updates her location from any compatible application and that information can be used by other applications. When the user wishes to update her location, $\mathbf{D_i}$ must obtain coordinates from its localization hardware and request a new tag from **U**. The next step, which is only done if $\mathbf{D_i}$ is running an application that requires this functionality, is to register a callback for the user's location with the appropriate **C**. Then the user must call **L**'s `setAttribute` function with her tag, location, the attribute name "present", her tag as the attribute value and a time-to-live value (indicating how long the location update will persist). As mentioned in section 3.4, the user should update her location with **L** only sporadically and only if the location is public to avoid tracking and re-identification attacks. Finally, the user must call **U**'s `setAttribute(`uid_i`, ‘‘location’’,<encrypted location>,ttl)` where her location is her coordinates encrypted with a symmetric key specific to her "location" attribute and the `ttl` value is short to allow for frequent location changes (a user can refresh her location update by registering again for the same location but with a different tag).

3.6 Security Analysis

The main focus of our architecture security-wise is the division of information goal, which states that none of **U**, **L** or **C** can know both a user's identity and her location. Using our previously stated trust assumptions, we study each of these entities in turn and discuss if this goal is met.

U knows a user's identity because providing identifying information was required during registration and each request must include a user's identifier (*uid*). When a user stores her location in **U**, she encrypts it with a symmetric key specific to the "location" attribute. For **U** to discover her location, it would have to collude with **L** to look up the location where the user's most recent tag was seen or collude with the user's friends to decrypt her stored location, both of which we rule out using our threat model. We also ruled out **U** registering a user in the system in the threat model. If **U** could register a user in the system, it could become friends with a user to obtain her decryption keys and also use the attack described later in this section (registering finely-spaced triggers) to get matched with a user and determine her location. **U** could query several locations in **L** around a known home location for a user (which could be public) and compare results with recently registered tags. However, we assumed that **U** is honest-but-curious and does not become a user of the system so this does not fit into our model and **U** would get caught by **L**.

L receives requests to set attributes about locations from users that are annotated with tags. Users make requests through an anonymizing proxy so **L** cannot use any connection-based information (such as an IP address) to identify them. Only **U** can reverse map tags back into identities so **L** cannot find out a user's identity in this manner. **L** could also collude with **U** or a **C** that the user has whitelisted (and can therefore call **U**'s `getIdentity` to get the identity behind the tag) but we ruled out collusion in the threat model. **L** could also attempt to query **U**'s `getIdentity`, which is prevented because **U** ensures that a registered cloud component is calling the function. We ruled out **L** registering a user in the system in the threat model. If **L** could register a user in the system, it could set a large number of common attribute name/attribute value pairs with **U** and register for matching at many locations with **C** in an attempt to get matched with other users (and therefore learn their identities).

C receives a user's identifier during registration and can see attributes in plaintext if a user has provided **C** with decryption keys. In all requests to **C** containing a location, the coordinates are encrypted with **L**'s public key and are passed to **L** during a subsequent operation. **C** could query several locations in **L** around a known home location for a user (which could be public) and call **U**'s `getIdentity` to see if any of the returned tags match the user's identifier. **C** could also register and become friends with the user or collude with the user's existing friends to get the decryption keys for the user's location attribute that has been retrieved from **U**. We rule out both of these scenarios in the threat model.

Separating **U** from **L** addresses the traffic analysis threat that exists in related work by Guha *et al.* [33], and Puttaswamy and Zhao [46] because requests with identifying information and requests with locations are handled by different entities. If an adversary is a user of the system, she can register many callbacks for different attributes in locations that are close together to try and get matched with a particular user. Rate-limiting at **C** could solve this problem. However, she cannot impersonate another user without their private key because to communicate with **U** all requests must be signed and when communicating with **L**, a tag is required (which is issued by **U**). The adversary also cannot find out the location of a user who is not a friend because she does not have the decryption key for the attribute stored in **U** and we assume in the threat model that no other entity will collude with the adversary to provide them with the decryption key. Threats presented by **C** being able to register users in the system are identical those presented in the threat assessment for **U**.

As for threats not coming from entities in the system, a passive adversary can only learn which entities are communicating with each other but cannot learn anything because all communications are secured with SSL. An active adversary cannot launch a replay attack at **U** because a timestamp must be included in all requests (which are signed, so the timestamp can't be modified by the adversary). **U** verifies that a timestamp has not been presented before by the user making the request. A non-user can check in with **L** with a fake tag because there is no way to check if a provided tag is valid. However, a non-user cannot register callbacks unless a malicious **C** does so on their behalf but this violates the honest-but-curious model.

3.7 Use Cases

Our first goal was to present an architecture that exhibited privacy by design to encourage development of applications that consider privacy preservation a priority. We later stated an additional goal of making our system flexible enough to support many use cases. In this section, we describe how a variety of applications can be realized using our framework to show how these two goals have been met.

3.7.1 Friend Locator

Suppose user **i** and user **j** are friends (i.e., they have exchanged decryption keys), user **j** has made a recent location update, and user **i** would like to locate user **j**. $\mathbf{D_i}$ makes a `getAttribute` request to **U** with uid_i, uid_j and the attribute name "location". **U** returns user **j**'s encrypted location and $\mathbf{D_i}$ can decrypt to learn user **j**'s location.

3.7.2 Friend Proximity Detection

A proximity detection application notifies a user if she is within a threshold distance of any of her friends. Suppose $\mathbf{D_i}$ and $\mathbf{D_j}$ are friends and are frequently updating their locations. If $\mathbf{D_i}$ wishes to be notified when $\mathbf{D_j}$ is within a threshold distance, one of the following procedures could be used:

1. $\mathbf{D_i}$ calls **U**'s `getAttribute(`uid_i`,`uid_j`,‘‘location’’)`, decrypts the returned location and compares it to the user's current location. If the distance between the two locations is within a threshold value, $\mathbf{D_i}$ is notified. This solution does not scale well if user **i** has many friends and $\mathbf{D_i}$ has to compare her location to all of their locations.

2. Suppose $\mathbf{D_i}$ and $\mathbf{D_j}$ have both whitelisted a supporting cloud component **C**. $\mathbf{D_i}$ sends its proximity detection requests with its location and threshold distance (encrypted with **L**'s public key), and values indicating how often and for how long they would like proximity detection protocols to run to **C**. **C** calls **U**'s `getAttribute(`uid_i`,` uid_j`, ‘‘friend’’)` to get a list of $\mathbf{D_i}$'s friends and stores it for later use. **L**'s `getAttribute` function must be designed such that the `location` parameter accepts a region (defined by GPS coordinates and a radius). In addition, **L**'s location-aware database must be able to execute a query that returns all entries at locations within the specified distance of a set of coordinates. **C** calls `getAttribute(<encrypted (coordinates,distance)>,‘‘present’’)` and gets a list of tags within `distance` of `coordinates`. **C** calls **U**'s `getIdentity` with the list of tags and gets a mapping of tags to *uid*s for *uid*s that have whitelisted it. **C** determines which of these *uid*s belong to users that are friends with user **i** (which may include user **j**) and sends a notification with this list of *uid*s to $\mathbf{D_i}$. If uid_j is included in the set, then $\mathbf{D_i}$ notifies user **i**. This solution could work but investigation to determine how well it scales as the number of users, requests and entries in **L**'s database increases would have to be done.

3. Suppose $\mathbf{D_i}$ and $\mathbf{D_j}$ have both whitelisted a supporting cloud component **C**. $\mathbf{D_i}$ sends its proximity detection requests with its location and threshold distance (encrypted with **L**'s public key), and values indicating how often and for how long they would like proximity detection protocols to run to **C**. **C** calls **U**'s `getAttribute(`cid_k`,` uid_i`, ‘‘friend’’)` to get a list of $\mathbf{D_i}$'s friends and caches this data for later use. **L**'s `registerCallback` function must be designed such that the `location` parameter accepts a region (defined by GPS coordinates and a radius). In addition, **L**'s location-aware database is able to execute a query that returns all registered locations within the specified distance of a set of coordinates.

   L's callback storage is modified to permit locations to be regions and queries are able to determine if coordinates for a check-in are inside callback regions. **C** calls **L**'s `registerCallback(<encrypted (coordinates, distance)>, ‘‘present’’, <handler>, ttl)` (where the `ttl` is the value the user specified for how long to run proximity detection protocols) **L** adds a callback for attribute “present” to the database for a circle-shaped region with `coordinates` as the centre and `distance` as the radius, and returns the token for the callback to **C**. A mapping γ from callback tokens to *uid* is stored for later reference.

   When the callback is triggered (by a check-in to a location inside its associated region), **L** calls **C**'s callback handler with a list of tokens for triggered callbacks as well as tags registered with the attribute “present” at that location.

   The handler calls **U**'s `getIdentity` function to get a list λ of mappings from tags to *uid*. **C** uses cached attributes for user **i** to determine if any *uid* in λ or γ are friends. If any friends are found, **C** stores the result for later retrieval by $\mathbf{D_i}$. If the smartphone platform $\mathbf{D_i}$ is running on supports it, **C** can send a push notification to $\mathbf{D_i}$ immediately when a match is found.

   The main drawback of this approach is that the query to determine which callbacks' regions are matched by location check-in can take a long time if there are many callbacks in the database.

3.7.3 Matching Service

The purpose of this type of application is to match users at the same location (who can be friends or strangers) with each other based on attributes. For example, if Alice wishes to play basketball at a nearby park, she can indicate to the application that she is interested in basketball and give her location. She can then be matched with other users at the same location that are also interested in basketball.

A cloud component **C** provides a public API function that allows $\mathbf{D_i}$ to register user **i** for matching at a particular location. We assume that $\mathbf{D_i}$ has whitelisted **C** and given **C** the necessary decryption keys for user **i**'s attributes stored with **U**. When a user **i** registers for matching, they provide uid_i, coordinates encrypted with **L**'s public key (so that **C** cannot learn her location) as well as a value indicating how long they would like their callback request to persist before it expires. A signature over all parameters is also required. Once **C** has received a registration request, it verifies that the signature matches the certificate associated with uid_i, retrieves user **i**'s attributes by sending a `getAttribute` request to **U** and caches them for later use in matching. Then **C** sends a request to **L**'s `registerCallback` function with the encrypted location, the attribute name "present" (which indicates that the attribute represents a user checked in at a location), the time-to-live value and the URL of a public API function that **L** can call to complete the callback. **L** returns a token, and **C** records the association between that token and uid_i before sending the token to $\mathbf{D_i}$. $\mathbf{D_i}$ then sends a `setAttribute` request with their coordinates (encrypted with **L**'s public key), the attribute name "present" (which indicates that they would like to check in to the location), their tag and a time-to-live value. This is done to immediately trigger callbacks and generate results for user **i**.

When callbacks are triggered by **L**, **C**'s handler is called with a list of tags that are present at the location as well as all matching callback tokens. The handler calls **U**'s `getIdentity` function to get a list λ of mappings from tags to uid. Then each callback token is mapped to a uid_i, and the attributes associated with uid_i are then matched against the attributes of all users in λ. Any matching algorithm can be used because $\mathbf{C_k}$ can see plaintext attributes for users. Then the results of the match are stored and **C** provides a public API function that allows $\mathbf{D_i}$ to retrieve this data.

Example

Suppose user **i** and user **j** are both at location ℓ and have at least one common interest stored with **U**. Matching application **k** is supported by cloud component $\mathbf{C_k}$.

1. $\mathbf{D_i}$ calls `whitelist(`uid_i, cid_k`)` and gives the decryption key for the attribute "interest" (stored with **U**) to $\mathbf{C_k}$.

2. $\mathbf{D_i}$ performs a location update with the callback registration step included.

3. $\mathbf{D_j}$ does steps (1) and (2).

4. $\mathbf{D_j}$'s location update triggers callback handling for location ℓ. **L** calls $\mathbf{C_k}$'s callback handler with the list of all tags checked in at location ℓ.

5. $\mathbf{C_k}$ calls **U**'s `getIdentity` with the list of tags received from **L** and gets a set λ of mappings from tags to identities for all users that have whitelisted it.

6. $\mathbf{C_k}$ uses users' interests that were cached during callback registration to perform matching. Since both user **i** and user **j** were at location ℓ and whitelisted $\mathbf{C_k}$, they are in the set to be matched. Matching occurs and results (matching attributes for sets of users) are stored. At least one result is stored for users **i** and **j** because they have (a) common interest(s).

7. If the platforms $\mathbf{D_i}$ and $\mathbf{D_j}$ are running on support push notifications, $\mathbf{C_k}$ can send them their match results immediately. Otherwise, $\mathbf{D_i}$ and $\mathbf{D_j}$ periodically poll $\mathbf{C_k}$ to retrieve their matching results.

3.7.4 Local Search

A local search application provides information about points of interest. A user provides her location and keywords (e.g., "restaurant", "coffee shop", "shopping") and the application returns results that are close to her current location. Points of interest can register using **L**'s `setAttribute` function with an empty tag (since they are not users), the attribute name "point of interest", useful information as the value and a time-to-live value that will allow the entry to persist far into the future. We assume that data is not encrypted or, in the case of a paid application, only paying users have the decryption key. [2] A device can retrieve local search information by calling **L**'s `getAttribute` function with its current location and attribute name "point of interest".

3.7.5 Social Recommendations

A social recommendations application allows users to write a review or give an opinion (e.g., ratings of 1–5 stars) of points of interest (e.g., restaurants, stores, attractions). Describing how to build this functionality using our framework is best described by the following example:

Scenario

User **i** writes a review and $\mathbf{D_i}$ stores it in **L**. User **j** would like to read reviews about a location. User **i** and user **j** are friends, which means they have each others' decryption

[2]How an application manages keys to maintain this restriction is out of the scope of this work.

keys. The social recommendations application **k** is supported by cloud component $\mathbf{C_k}$.

Generating a Review

A review is generated as follows:

1. $\mathbf{D_i}$ calls `whitelist(`uid_i, cid_k`)` and gives the decryption key for the attribute name "friend" (stored with **U**) to $\mathbf{C_k}$.
2. $\mathbf{D_i}$ calls `createTag(`uid_i`)`.
3. $\mathbf{D_i}$ stores an encrypted review with **L** using `setAttribute(tag, location, ‘‘review’’, (tag,<review text>),ttl)` where the `ttl` value indicates how long they would like their review to persist. The value `<review text>` is encrypted with $\mathbf{D_i}$'s attribute key for attribute name "review". Alternatively, $\mathbf{D_i}$ can store a public review with **L** using `setAttribute(tag, location, ‘‘public_review’’, <review text>),ttl` and leaving `<review text>` unencrypted.

Reading a Review

To retrieve public reviews (which are unencrypted), $\mathbf{D_j}$ calls `getAttribute(location, ‘‘public_reviews’’)`. For encrypted reviews, $\mathbf{D_j}$ follows these steps:

1. $\mathbf{D_j}$ sends its location (encrypted with **L**'s public key to avoid $\mathbf{C_k}$ learning its location) to $\mathbf{C_k}$.
2. $\mathbf{C_k}$ calls `getAttribute(location, ‘‘reviews’’)` with the encrypted location and gets back data in the form {`(<review text>, tag)`}, where `<review text>` is encrypted with the attribute key of the writer.
3. For each tag that is attached to an encrypted review, $\mathbf{C_k}$ calls `getIdentity(tag,` cid_k`)` and receives the identity of the review writer if she has whitelisted $\mathbf{C_k}$.
4. For each received identity, $\mathbf{C_k}$ calls **U**'s `getAttribute` function with the attribute "friends" to get a list of the writer's friends.
5. If a writer (e.g., $\mathbf{D_i}$) lists $\mathbf{D_j}$ as a friend, **C** passes on the encrypted review and the identity of the writer to $\mathbf{D_j}$.
6. $\mathbf{D_j}$ uses the review decryption key received from $\mathbf{D_i}$ to decrypt the review.

3.7.6 Advertising

An advertising application allows a business to post advertisements to specific locations annotated with keywords indicating the area of interest (e.g., coffee, clothing, daily deals). Users that check in at a location can be served advertisements registered at that location that match their interests. In a deployed system, **L** would, in general, keep track of who stores what information and how often this information is accessed. Then the owner of the information could billed accordingly. However, the details are future work.

Scenario

Assume **A** is the advertiser and has posted an ad to $\mathbf{D_i}$'s location.

Approach I

1. **A** registers the ad with **L** with one entry for each interest they wish to attach to the ad as follows:
   - `setAttribute(<empty tag>, location, ''ad_health'', <ad text>, ttl)`.
   - `setAttribute(<empty tag>, location, ''ad_soccer'', <ad text>, ttl)`.
2. $\mathbf{D_i}$ directly queries **L** based on its current location and user **i**'s interests.

The advantage of this solution is that it is simple to implement. The drawback is that **L** can link requests if a person asks for multiple interests at each location. Therefore, users should send only limited number of requests per location.

Approach II

We assume that the identity of the advertiser does not reveal the location where his ads are posted (e.g., the brand name of McDonald's does not reveal the locations where the company places ads). One goal is to avoid **L** linking multiple requests that contain interests, which we achieve by using a cloud component to make requests to **L** on the user's behalf. Another goal is to prevent the cloud component from seeing the text of the ad, as the text could reveal the location where the ad was placed.

1. **A** registers the ad with **L** using `setAttribute(<empty tag>, location, ''ad'', <advertiser, advertise_ID, <interest1, interest2, ...>>)`, where `advertiser` represents
contact information (e.g., URL) for **A**, the `advertise_ID` represents the ID of the specific ad regarding that location, and the interest list `<interest1, interest2, ...>` is used to match potential clients' interests. We assume **A** could run multiple ads at different locations, which are indexed by `advertise_ID`.

2. $\mathbf{D_i}$ calls `whitelist`(uid_i, cid_k) and gives the decryption key for the attribute "interest" (stored with **U**) to $\mathbf{C_k}$.

3. $\mathbf{D_i}$ makes a request to $\mathbf{C_k}$ with her current location (encrypted using **L**'s public key).

4. $\mathbf{C_k}$ passes the encrypted location to **L** and asks for ads registered at that location.

5. **L** sends $\mathbf{C_k}$ a collection of data where each entry is of the form `<advertiser, advertise_ID, <interest1, interest2, ...>>`. Note that the actual contents of ads are not contained within the list.

6. $\mathbf{C_k}$ performs interest matching between $\mathbf{D_i}$ and the ads. If $\mathbf{C_k}$ finds a matching ad placed by advertiser **A**, $\mathbf{C_k}$ sends **A**'s public key and contact information, as well as the `advertise_ID` of the ad, to $\mathbf{D_i}$ (we assume that $\mathbf{C_k}$ has stored all advertisers' public keys).

7. Upon receiving **A**'s public key and contact information from $\mathbf{C_k}$, $\mathbf{D_i}$ verifies the identity of **A**.

8. $\mathbf{D_i}$ requests the ad from **A** using the `advertise_ID` and downloads the ad text from **A** through an anonymization network or proxy.

9. $\mathbf{D_i}$ presents the ad text to the user.

During the whole process, **A** only knows $\mathbf{D_i}$'s location but not $\mathbf{D_i}$'s identity (since all communication between $\mathbf{D_i}$ and **A** is either assisted by $\mathbf{C_k}$ or anonymized) and $\mathbf{C_k}$ knows $\mathbf{D_i}$'s identity but not $\mathbf{D_i}$'s location.

The main drawback of this scheme is that a business that only posts ads at a small number of locations (e.g., a small local business) cannot participate without causing privacy problems for users. If $\mathbf{C_k}$ can find out an advertiser's locations using their contact information, they can narrow down or pinpoint the location of any user requesting an ad for that advertiser. The simplest solution to this problem is to have a third party with contact information that does not trace back to any companies post ads on behalf of smaller companies with few locations.

Chapter 4

Implementation and Experiments

4.1 Introduction

This chapter presents the implementation and evaluation details of the key components of our framework. In section 4.2 the implementations of the user-indexed database server (**U**), location-indexed database server (**L**), a sample cloud component (**C**) and device client (**D**) are discussed. In section 4.3 we present results from performance measurements executed on our framework. Section 4.4 describes proof-of-concept applications implemented using our framework.

4.2 Implementation

The server and cloud components (**U**, **L**, and **C**) are written in the Python programming language using the CherryPy web application framework and use the PostgreSQL object-relational database system for storage of persistent data. Each of these components exposes a WSGI interface with SSL support that can be accessed using HTTPS requests. All communications between components package data using protocol buffers [12] in the bodies of HTTPS POST requests. For symmetric-key encryption operations, AES in CFB mode (with random initial vector values) with 128-bit key size is used. RSA with 2048 bit key size is used for public-key encryption and the SHA-256 hash function is used for all hashing operations. Signing operations use SHA-256 with RSA. Self-signed certificates and associated private keys have been generated for each entity (users, cloud components, servers) using OpenSSL and are used for signing API requests, verifying signatures and establishing HTTPS connections. In our prototype self-signed certificates are sufficient for experiments, but certificates verified by a known certificate authority would be necessary for a real-world implementation.

4.2.1 Implementation of the User-Indexed Database Server

All users and cloud components provide a certificate during registration and are assigned a unique identifier by **U**. Mappings between *uid* and user certificates as well as between *cid* and cloud component certificates are stored in the database. All requests to API functions provided by **U** must include the unique identifier of the sender (*uid* or *cid*) and a timestamp (to avoid replaying of messages). Authentication to verify the identity of the user or cloud component making the request is also used for all API functions. To authenticate a request, **U** queries the database for the certificate associated with the identifier (*uid* or *cid*) provided in the request and verifies that the signature was produced by the corresponding private key. **U** rejects the request if it contains a timestamp for an earlier or identical time to the most recent timestamp (in the Timestamp column of the User Verification table) they have remembered for the provided identifier. Otherwise, **U** continues with the request and updates the stored timestamp for the identifier to match the one in the request. For simplicity in our prototype implementation, each user has only one symmetric key used to encrypt all of her attribute values. In a real-world deployment, each user would have a different key for each attribute name.

A set of database tables supports the operations of **U**. The main table is the collection of user attributes. Each row represents one attribute for a user, identified by *uid*. See Table 4.1 for more details about the composition of each row. Other tables used by **U** include a table listing which users have whitelisted which applications (Whitelist) and lists of users' and applications' certificates (User Verification and Application Certificates, respectively) to be used for verifying signatures on requests. See Table 4.2 for more information about the User Verification table and Table 4.3 for more details about the other tables.

Individual API operations are implemented as follows:

- `getTag`: This function accepts uid_i from $\mathbf{D_i}$ and concatenates this identifier with the current time in milliseconds. This value is then encrypted with symmetric-key encryption using a secret key only known by **U** and returned to $\mathbf{D_i}$.

- `whitelist`: This function accepts uid_i from $\mathbf{D_i}$ as well as all the *cid* of the cloud components for the applications they would like to whitelist. **U** stores an entry in the Whitelist table indicating that the device client has whitelisted **C**.

uid	**Attribute Name**	**Attribute Value**	**Expiry Time**
12345	interest	< encrypted value >	2012-09-19 00:00:00
12345	friend	< encrypted value >	2012-09-19 00:00:00
12346	location	< encrypted value >	2012-08-01 12:15:00
12347	interest	< encrypted value >	2012-12-31 15:00:00
12347	interest	< encrypted value >	2012-12-31 15:00:00

Table 4.1: Table of User Attributes in **U**

uid	**Certificate**	**Timestamp**
12345	< Certificate Data >	1348594480
12346	< Certificate Data >	1348137280
12347	< Certificate Data >	1348137255
12348	< Certificate Data >	1348458329

Table 4.2: Table of User Verification Information in **U**

Whitelist

uid	**cid**
12345	9000
12345	9002
12346	9000
12347	9002

Cloud Component Certificates

cid	**Certificate**
9000	< Certificate Data >
9001	< Certificate Data >
9002	< Certificate Data >
9003	< Certificate Data >

Table 4.3: Other Tables of Data in **U**

- `getIdentity`: This function accepts a list of tags from a cloud component **C** as well as its *cid*. **U** decrypts all tags using its secret key and queries the Whitelist table in the database with the list of identifiers derived from decrypting the tags. The database responds with only the identifiers for users that have whitelisted **C**. A set of mappings indicating which tags correspond to which identifiers is returned.

- `setAttribute`: This function accepts uid_i from $\mathbf{D_i}$ as well as the name of the attribute $\mathbf{D_i}$ would like to modify, the desired value encrypted with $\mathbf{D_i}$'s attribute key and a timestamp indicating when $\mathbf{D_i}$ would like the attribute to expire. **U** calls a stored procedure in the database to determine if an attribute is new for $\mathbf{D_i}$ or being updated and inserts a record into the User Attributes table or modifies an existing one as appropriate.

- `getAttribute`: This function accepts a unique identifier (*uid* or *cid*) from the entity making the request (could be **C** or $\mathbf{D_i}$) as well as lists of *uid*s and attribute names. For each requested *uid*, **U** retrieves the attribute values for the requested attribute names from the User Attributes table. A set of mappings from *uid* to a list of all of the (attribute name,attribute value) pairs returned for that *uid* (of the form {(*uid*,{`(attribute name, attribute value)`}}) is returned to the user. This operation was implemented to support batch operations because a wider variety of use cases is supported. For example, our implementation supports situations where a user wants particular attributes for one of her friends or many attributes for multiple friends.

4.2.2 Implementation of the Location-Indexed Database Server

The database backend of **L** uses the PostGIS extension to add support for indexing records by location and performing calculations on the location data in records. All locations in requests to **L** must be encrypted with **L**'s public key. The database has a table that lists the GPS coordinates (latitude, longitude) of all places that **D** can check into (using the `setAttribute` function) as well as a unique identifier for each location to make bookkeeping easier. During API calls, **L** performs a nearest-neighbour query on provided GPS coordinates to get the identifier for the nearest location that can be checked into. This function could easily do a k-nearest-neighbours search to implement the functionality of many location based services, which present a user with a list of nearby locations and allow the user to choose which one to check into.

A proxy server is available that permits clients to issue an HTTPS CONNECT command (with **L**'s hostname and port) in order to hide their IP addresses from **L**. For security reasons, the configuration file only permits connections from the IP addresses of our test clients to **L**. This is sufficient for experiments but in a real-world implementation a more sophisticated authentication scheme would be required to handle a large number of clients.

A set of database tables supports the operations of **L**. The first table is a list of registered locations that users can check into, indexed by coordinates. Each row represents one location and lists a unique identifier that is used to identify the location in other tables and a common (human-understandable) name in addition to the GPS coordinates. See Table 4.4 for sample data. Location updates are stored in the Location Check-In table, which is indexed by the identifier assigned in the Location Directory table. Each entry includes an attribute name as well as an attribute value and expiry time. See Table 4.5 for more details about the data in this table. The final table used by **L** stores information about registered callbacks. Each entry represents a particular callback and includes a unique identifier for the callback, the location, contact information for the cloud component that registered the callback and an expiry time (to the nearest minute). See Table 4.6 for sample data. To help ensure that **L** is only dealing with legitimate cloud components, all applications must register their cloud components' public certificates, which are stored in a table identical to the Cloud Components Certificates table in **U**.

Individual API operations are implemented as follows:

- `registerCallback`: This function accepts a location, an attribute name for which to register a callback and the URL of a callback handler The callback handler URL must reference a public API function exposed by **C** that accepts HTTPS requests in a predetermined format. **C** must also provide a signature over all parameters so that **L** can verify (using the Cloud Components Certificates table) that it is a registered cloud component. In addition, **L** should verify that the hostname provided by **C** matches the common name registered in **C**'s certificate. **L** adds a record to the Location Callbacks table and returns the automatically assigned row identifier to **C** (this acts as a callback identifier).

- `setAttribute`: This function accepts a location from $\mathbf{D_i}$ as well as a name representing which attribute $\mathbf{D_i}$ would like to modify, the desired value, and a `ttl` value (in hours). **L** inserts a record into the Location Check-In table with the attribute name, attribute value, and expiry time provided in the request. Then **L** triggers callback processing by querying the Location Callbacks table for all callbacks registered at the location. Callback details are grouped using contact information (e.g., callbacks are grouped together only if they have identical cloud component hostname and handler values) and **L** sends one message to each cloud component/handler combination with a list of identifiers for the callbacks that were triggered.

Location ID	**Coordinates**	**Common Name**
1	43.462856,-80.520644	LCBO Uptown Waterloo
2	43.463394,-80.520869	Starbucks Uptown Waterloo
3	43.463678,-80.521175	David's Tea Uptown Waterloo
4	43.470238,-80.530128	Laurier University Stadium
5	43.469833,-80.542316	University of Waterloo Dana Porter Library
6	43.472792,-80.542102	University of Waterloo Davis Centre

Table 4.4: Location Directory Table in **L**

Location ID	**Attribute Name**	**Attribute Value**	**Expiry Time**
1	present	$< tag >$	2012-09-03 15:15:00
1	present	$< tag >$	2012-09-03 12:00:00
6	badge	*cid*,"September's Studious Scholar"	2012-10-01 00:00:00
2	review	$< tag >$,$<$ encrypted review text $>$	2012-12-31 00:00:00

Table 4.5: Location Check-In Table in **L**

Call. ID	**Loc. ID**	**Attribute**	**Call. URL**	**Call. Handler**	**Expiry Time**
1	1	present	https://match.cloud.com:443	/api/callHandler	2012-09-03 15:15
2	1	present	https://match.cloud.com:443	/api/callHandler	2012-09-03 12:00
3	4	present	https://match.cloud.com:443	/api/callHandler	2012-10-01 00:00
4	6	present	https://match.cloud.com:443	/api/callHandler	2012-12-31 00:00

Table 4.6: Location Callbacks Table in **L**

- `getAttribute`: This function accepts a location from $\mathbf{D_i}$ as well as a key representing which attribute $\mathbf{D_i}$ would like to retrieve values for. **L** queries the Location Check-In table to get all records for the identifier and the desired attribute.

4.2.3 Implementation of the Cloud Service

The main function of **C** is to act as an agent for a particular application and manage callbacks on behalf of devices. **C** receives a callback request from $\mathbf{D_i}$ with the user **i**'s unique identifier (assigned by **U**), retrieves the user's attributes from **U** and caches them for later callback processing. A callback (which must include a URL of an API function exposed by **C** that can accept an HTTPS request to complete the callback) is registered with **L**. The returned callback identifier is stored with the user's identifier in a database table and the identifier is returned to $\mathbf{D_i}$ so that it can be used to retrieve results at a later time (see Table 4.7).

When a callback registered by **C** is triggered at **L**, a list of identifiers representing triggered callbacks are sent to the handler and application-specific processing can be applied by **C**. Results are stored for later retrieval by the application running on a device.

Callback ID	uid
1	12345
2	12346
3	12347
4	12348

Table 4.7: Callbacks Table in **C**

4.2.4 Implementation of the Device Client

The Device Client (**D**) was built using the Android platform. Cryptographic keys are stored in a Bouncy Castle keystore object (the only format supported by Android). The keystore is added to the application as an asset and loaded in the code to assist with creating SSL sockets and performing cryptographic operations. A basic GUI is provided with a field where an identifier can be entered to cause the application make requests as if it is the user specified by the identifier.

Since the Android operating system does not permit network operations in the main thread of the application, all HTTPS requests must be executed in a separate task or thread. Requests are first constructed and then passed along with the method name of a handler for processing the specific results from the request to an asynchronous task to be executed. The asynchronous task uses Java reflection to call the results handler with the raw data returned from the request. This design allows an asynchronous task to remain flexible enough to handle all types of requests and avoids having to write a different asynchronous task for every API function that could be called.

4.2.5 Challenges

The M2Crypto library was used for cryptographic operations in Python code because it can be instructed to not automatically add padding to plaintext when doing AES encryption. PyCrypto, the standard python cryptography library, exhibited unexpected behaviour with respect to padding plaintext and was producing ciphertext that could not be decrypted using Java cryptography libraries and was not producing reasonable decryptions of ciphertext produced by Java libraries. We were unable to determine what was causing the problem so using a different library was required. However, M2Crypto's automatically added padding was also not correct, but a library function was provided that did pad plaintext correctly. We wrote our own function for removing padding since M2Crypto's automatic removal of padding is also incorrect.

The implementation of the Bouncy Castle open-source cryptography library provided in Android is outdated and does not operate properly with keystores created with later versions of the library. Updated functionality cannot be provided by adding an external JAR file to the Android application without causing class loading conflicts with the Bouncy Castle library objects built into the Android operating system. However, a repackaged Bouncy Castle library, Spongy Castle [14], is available which renames the namespace to eliminating class loading errors and causes no problems when added to an Android application. Spongy Castle can also be provided to external cryptographic tools to create keystores that can be accessed easily from inside applications. When trying to add longer keys to a keystore using command line tools on the development computer, invalid key exceptions were raised. This problem was fixed by installing the Java Cryptography Extension (JCE) Unlimited Strength Policy Files in the Java Virtual Machine (JVM) on the development computer. Fortunately, the Android JVM supports unlimited strength cryptography by default.

One of our original implementation goals was to use both SSL server and client authentication during HTTPS requests. The CherryPy framework does not provide native support for SSL client authentication and the only extension available requires a file containing certificates for all possible clients that can be searched to determine if a connection at socket level is coming from a legitimate client. This does not scale well, as there could be many clients using our system. Also, client connections are handled at the socket level and an exception is thrown in the client after a failed attempt instead of being handled gracefully in an application-specific way. Our solution was to assign all users of the system private and public keys and use cryptographic signatures at the application level to authenticate clients.

The usage of asynchronous tasks to handle network connections can cause several threads access a connection object at the same time. The documentation of the suggested client class, AndroidHttpClient, claims that the client connection manager provided by default is thread-safe. However, the application only properly executed half of the requests being attempted by 10 threads simultaneously and threw exceptions for the rest. Using our own custom client class, which is extended from the Apache HttpClient class to use the thread-safe client connection manager provided by the standard Java libraries to eliminated the problem.

The application polls the cloud service for results from callback processing. Although native push-messaging is available on the Android platform, use is restricted heavily and it is difficult to configure.

4.3 Experimental Evaluation

In this section, we discuss the experimental evaluation of our framework. We performed experiments to determine the server processing overhead as well as the end-to-end (E2E) processing time at the client for each of the API functions offered by **U** and **L**. Analysis of the time taken to run operations in the cloud component **C** associated with the matching service application was also done. During experiments, **L** and **C** were run on a 3.4 GHz quad-core machine with 4 GB of RAM and **U** was run on a 2.4 GHz dual-core machine with 4 GB of RAM. The client was run on a Nexus One device with Android 2.3.6 installed and comparison measurements were collected using a client written in Python connected to the same wireless network and running on a 2.4 GHz dual-core machine with 4 GB of RAM. For Python code, execution times were measured using the cProfile library. In the Android client, the system function for getting time in milliseconds was used to calculate execution times. For each function in the server entities and Python client, we performed 500 trials. Due to issues with instability over a large number of consecutive executions of each function, only 250 trials were performed for each function on the Android client.

4.3.1 Experiments on U

In the first round of experiments on API functions provided by **U**, `getIdentity` was executed with only one tag in the request and `getAttribute` was called for a *uid*/attribute name pair that only returned one attribute value. Since `getIdentity` is only called from a cloud component, measurements were only done using Python test code (to mimic a cloud component written in Python) and not performed on the Android client. The results are shown in Table 4.8.

Function	Server Time	Python Client E2E	Android Client E2E
`createTag`	1.0 ms ± 0.0 ms	27 ms ± 7 ms	180 ms ± 30 ms
`whitelist`	7 ms ± 2 ms	39 ms ± 7 ms	180 ms ± 20 ms
`getIdentity`	4.0 ms ± 0.0 ms	40 ms ± 5 ms	–
`getAttribute`	2.9 ms ± 0.5 ms	23 ms ± 4 ms	180 ms ± 20 ms
`setAttribute`	7 ms ± 1 ms	31 ms ± 9 ms	180 ms ± 30 ms

Table 4.8: Server Processing and end-to-end times for baseline measurements on API functions offered by **U**. Each column lists the mean execution times (± the standard deviation) across all trials.

More details about what operations were taking significant time during server processing are as follows:

- `createTag`: The only operation that registered measurable server processing time was verification of the user's identity because it involves cryptographically verifying the signature of the requestor.
- `whitelist`: 78% of the server processing time was spent inserting a new entry into the Whitelist table. The most significant portion of the rest of the time (about 15%) was spent verifying the user's identity.
- `getIdentity`: Verifying the identity of the cloud component making the request took about 40% of the server processing time and checking the whitelist took about 35% of the time.
- `getAttribute`: Verifying the user's identity takes up 35% of the server processing time while retrieving the attribute from the database occupies 40% of the rest of the time.

- `setAttribute`: 70% of the server processing time was spent calling the stored procedure in the database to insert or update the attribute. The only other operation to register in the measurements was verifying the user's identity (15%).

Some of the differences between the Android and Python clients can be attributed to the amount of computation time the Android application uses to maintain its state. Traceview (an Android profiling tool) showed that a lot of time was spent making calls to UI-related functions and other functions for keeping the application running. The rest of the differences are probably caused by the differences in computational power available (i.e. the Android device has less resources available and a lot of applications competing for them).

For one batch experiment, we tested `getAttribute` for a user/attribute pair with an increasing number of results to determine how the amount of data being returned affects the execution of the function. The number of returned results was increased until the test became unstable on the Android device. In a real-world deployment, if a user requested all attributes for a large number of her friends (e.g., if she had 200 friends averaging 50 attributes each), it could be possible to get 10000 attributes returned from the server. The timing results are shown in Table 4.9 and the breakdown of the server processing time is displayed in Figure 4.1.

Attributes Returned	Server Time	Python Client E2E	Android Client E2E
10	3.0 ms ± 0.0 ms	23 ms ± 1 ms	290 ms ± 90 ms
100	10.1 ms ± 0.4 ms	40 ms ± 2 ms	400 ms ± 200 ms
500	45 ms ± 2 ms	82 ms ± 6 ms	800 ms ± 400 ms
1000	85 ms ± 3 ms	139 ms ± 7 ms	1050 ms ± 70 ms
5000	420 ms ± 80 ms	590 ms ± 20 ms	4700 ms ± 600 ms
10000	600 ms ± 200 ms	900 ms ± 300 ms	11000 ms ± 3000 ms

Table 4.9: Server processing and end-to-end times for measurements on **U**'s `getAttribute` function with varying numbers of attributes returned. Each column lists the mean execution times (± the standard deviation) across all trials.

For the other batch experiment, we tested `getIdentity` with an increasingly large list of tags to determine how the amount of data being processed affects the execution of the function. The number of tags in the request was increased until the test became unstable. With more than 350 tags in the request, the CherryPy web server framework had trouble

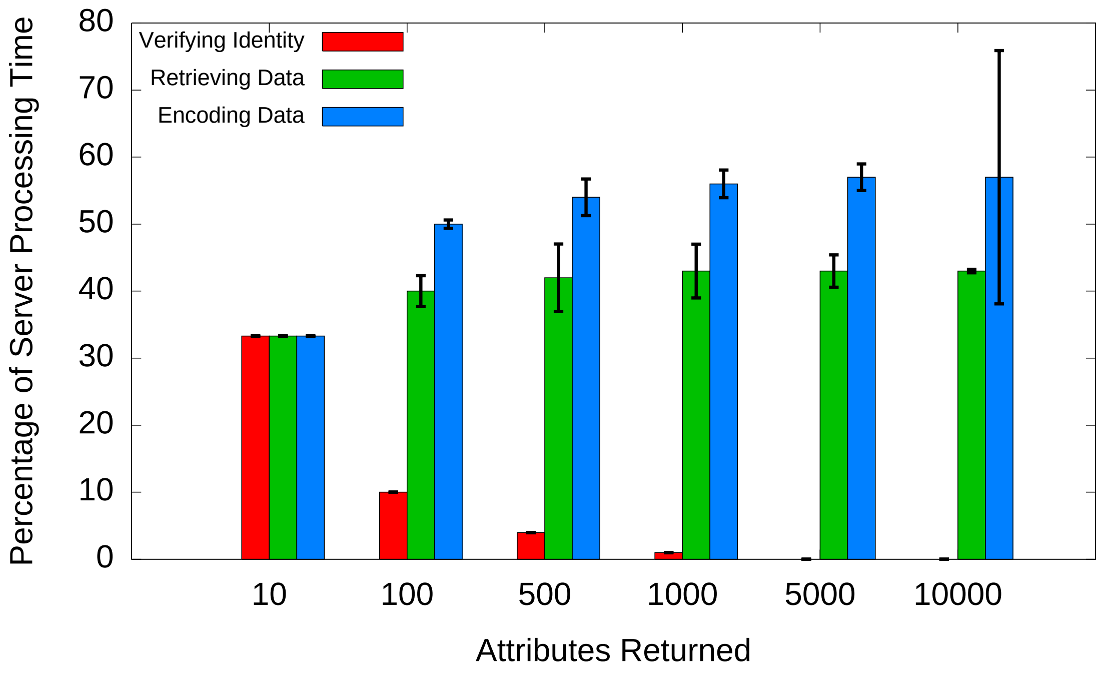


Figure 4.1: Server processing time (mean percentage of total processing time taken with error bars showing standard deviation) breakdown for **U**'s `getAttribute` function with varying numbers of attributes returned.

Number of Tags	Server Time	Python Cloud E2E
10	3.0 ms ± 0.4 ms	32 ms ± 1 ms
50	7.0 ms ± 0.4 ms	36 ms ± 3 ms
100	11.1 ms ± 0.7 ms	42 ms ± 2 ms
250	25 ms ± 1 ms	52 ms ± 5 ms
300	29 ms ± 1 ms	61 ms ± 5 ms
350	34 ms ± 2 ms	64 ms ± 6 ms

Table 4.10: Server processing and end-to-end times for measurements on **U**'s `getIdentity` function with a varying number of tags in the request. Each column lists the mean execution times (± the standard deviation) across all trials.

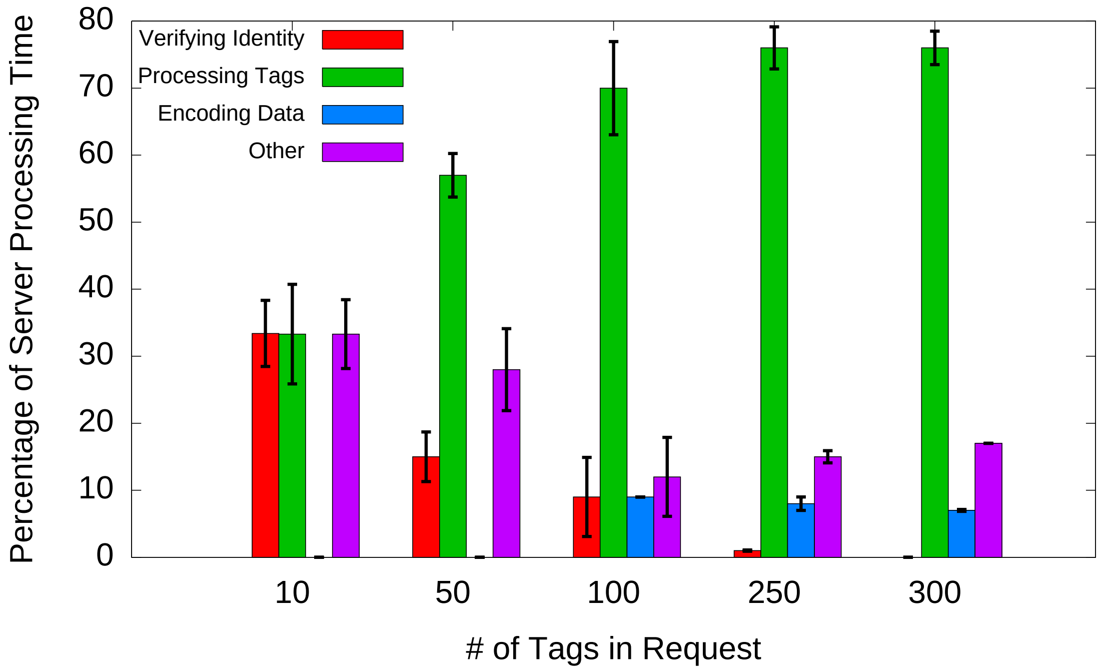


Figure 4.2: Server processing time breakdown (mean percentage of total processing time taken with error bars showing standard deviation) for **U**'s `getIdentity` function with a varying number of tags in the request. The data point for a batch of 350 tags has been omitted because its breakdown is identical to the results shown for a batch of 300 tags.

decoding the large amount of incoming data. The timing results are shown in Table 4.10 and the breakdown of the server processing time is displayed in Figure 4.2.

For basic functions involving the minimum amount of data, our architecture performs well with unnoticeable latency at the clients. In functions involving addition of data to the system, database insert operations occupied the greatest portion of the execution. Otherwise, execution time was split between retrieving information from databases (if necessary) or cryptographically verifying a requestor's identity.

Increasing the batch size of data returned from `getAttribute` causes encoding data for transport and retrieving data from the database to dominate the server's execution time and cryptographic operations used to verify the identity of the user making the request to become negligible. From the standpoint of the Android device, the time spent waiting for a request to return a large number of attributes becomes too high for impatient users (the waiting time is almost 5 seconds for 5000 attributes). Either the speed of the function needs to be increased or expectations of users have to be managed (e.g., perhaps only allow users to retrieve attributes for a limited number of friends at one time).

With `getIdentity`, increasing the number of tags in the request causes processing the tags (which involves decrypting each tag and determining if it is expired) to dominate the server processing time. Encoding data for transport becomes noticeable at requests containing 200 tags and other operations (mostly consists of checking the whitelist with the *uid* of all unexpired tags and the identifier of the requesting cloud component) become less significant. Verifying the identity of the requesting cloud component is negligible for requests containing more than 250 tags.

4.3.2 Experiments on L

For the first round of experiments on API functions provided by **L**, `getAttribute` (which returns results in a batch) was executed for a location and attribute for which there was only one entry to determine a baseline. `setAttribute` was executed for a location that did not have any callbacks registered to get an application-independent measurement (callback processing time is included in this function). Since `registerCallback` is always called on a user's behalf by **C**, no measurements were done with the Android client. The results are shown in Table 4.8.

More details about what operations were taking significant time during server processing are as follows:

Function	Server Time	Python Client E2E	Android Client E2E
`getAttribute`	17 ms ± 5 ms	35 ms ± 8 ms	240 ms ± 60 ms
`setAttribute`	43 ms ± 5 ms	63 ms ± 7 ms	500 ms ± 100 ms
`registerCallback`	24 ms ± 4 ms	110 ms ± 10 ms	–

Table 4.11: Server Processing and end-to-end times for API functions offered by **L**. Each column lists the mean execution times (± the standard deviation) across all trials.

- `setAttribute`: 43% of the server processing time was spent inserting a record for the attribute. The most significant portion of the rest of the time was spent decrypting the location parameter (42%) and a small amount of time (2%) was spent determining which registered location was closest to the provided coordinates.
- `getAttribute`: The most significant amount time (73%) was spent performing cryptographic operations. The only other notable operation was retrieving information from the database (8%).
- `registerCallback`: 47% of the server processing time was spent inserting a record for the callback being registered into the database. Notable portions of the rest of the time were spent performing cryptographic operations (42%) and mapping GPS coordinates to registered locations (4%).

As we observed in experiments on **U**, database operations consumed significant time in functions. However, cryptographic operations took up much larger portions of time than in **U** because **L** uses public-key cryptography to decrypt locations received in requests (**U** only uses symmetric cryptography in its functions). Nearest-neighbour queries to determine the registered location in the database closest to the coordinates provided in a request also were noticeable in server processing times. From the viewpoint of the user, none of the operations required noticeable time on the Android device, which contributes to the practicality of our architecture.

4.3.3 Experiments on C

We built a cloud component to support a matching service. A public API exposes operations for matching registration and retrieving match results (for $\mathbf{D_i}$), and processing callbacks (for **L**). The motivation for performing execution time measurements on **C** is to observe how a real-world use case implemented using our architecture behaves in terms of practicality. Matching of two users is done by directly comparing attribute/value pairs until one common pair is found (match was successful) or each pair belonging to a user has been compared against all of the other user's pairs and no common pairs were found (match was unsuccessful).

We tested the function exposed to $\mathbf{D_i}$ for matching registration and measured an execution time of 190 ms $\pm$ 20 ms at **C**. The Python test client and Android client observed end-to-end times of 240 ms $\pm$ 40 ms and 600 ms $\pm$ 70 ms, respectively. From a user's perspective, neither of these times are noticeable enough to be impractical.

To test callback processing at **C**, we set up a scenario where 2 users with 5 attributes each were checked in at and had registered for matching for the same location. The execution time for the callback processing operation initiated upon the registration and check in of the second user was measured at **C** (77 ms $\pm$ 7 ms) and **L** (130 ms $\pm$ 6 ms). The matching protocol performed well for this simple scenario. More experimentation with larger and more complicated matching scenarios is future work. We also used the same scenario to measure the time needed at a client to complete all of the operations necessary to receive a match with another user (registering for matching at and checking into a location, and retrieving results). From a user's perspective, the Android client takes a noticeable amount of time (1400 ms $\pm$ 200 ms) which needs to be improved upon in the future to make our matching service practical. In comparison, the Python client took less than half as long (510 ms $\pm$ 50 ms) to complete the same operations.

More details about what operations were taking significant time during server processing are as follows:

- Registering Callbacks: 63% of the server processing time was spent registering a callback with **L**. The only other notable operation was requesting user attributes from **U** (35%).
- Processing Callbacks: The greatest portion of the processing time (75%) was consumed by calling **U**'s `getIdentity` and waiting for the results. The only other notable operation was matching the two users (21%).

Neither of these processing time breakdowns are surprising, as the most significant portion of time in both cases is spent waiting for calls to API functions in other entities to complete.

4.4 Proof-of-Concept Applications

In this section, we describe proof-of-concept applications implemented using the Android platform. We created a friend locator and a matching service to demonstrate real-world applications that can be built using our framework.

4.4.1 Friend Locator

The friend locator application provides a GUI interface (shown in Figure 4.3) with a map and a text box where uid_i, where user **i** is whose location we are retrieving, can be entered. Symmetric keys for decrypting data are pre-loaded into the device. To initialize the friend locator protocol, the user must use the Android menu key on the device to pop up the menu (shown in Figure 4.4) and press the "Locate!" button. Then the device retrieves encrypted coordinates from **U** for the desired user. The application decrypts the coordinates and displays a graphic on a map widget to indicate the location of user **i** (shown in Figure 4.5).

4.4.2 Matching Service

The matching service provides a GUI interface (shown in Figure 4.6) with a text box where uid_i, where user **i** is registering for matching, can be entered. There is a button that can be pressed to initiate the registration for matching by sending a request to **C**. For testing purposes, coordinates were hard-coded but in a real-world application coordinates would be requested from localization hardware on the device. Once **C** has returned a callback ID, the application stores it for later reference. The user can press another button to request matching results from **C** using the stored callback ID. The set of uid_j where user **j** was matched with user **i** are displayed in a list (shown in Figure 4.7) . Selecting individual items in the list pops up a dialog box with more information about the user associated with the *uid* in the list item (shown in Figure 4.8).

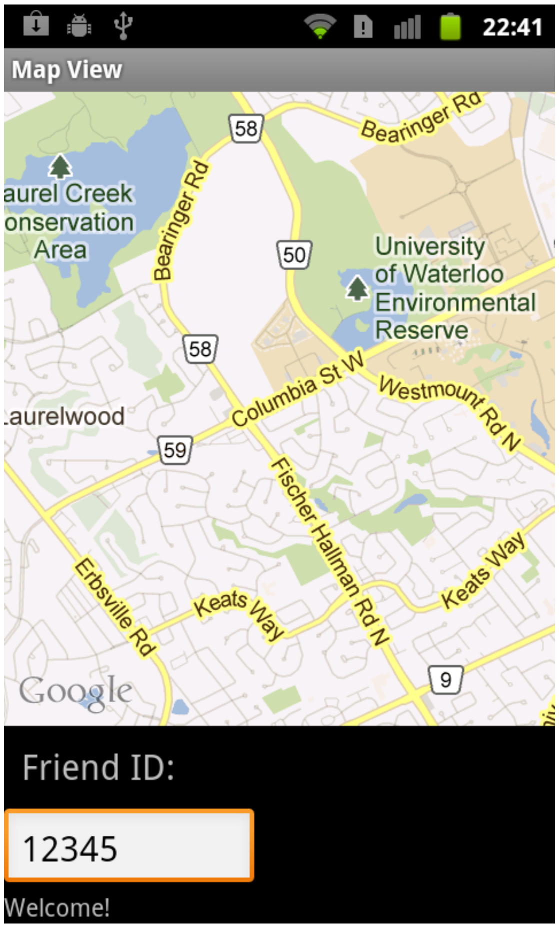


Figure 4.3: Friend Locator Welcome Screen

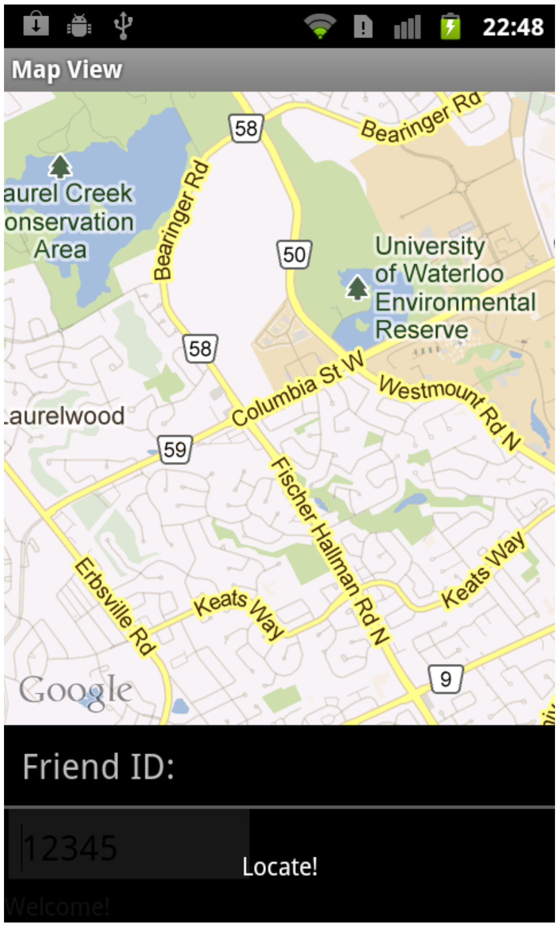
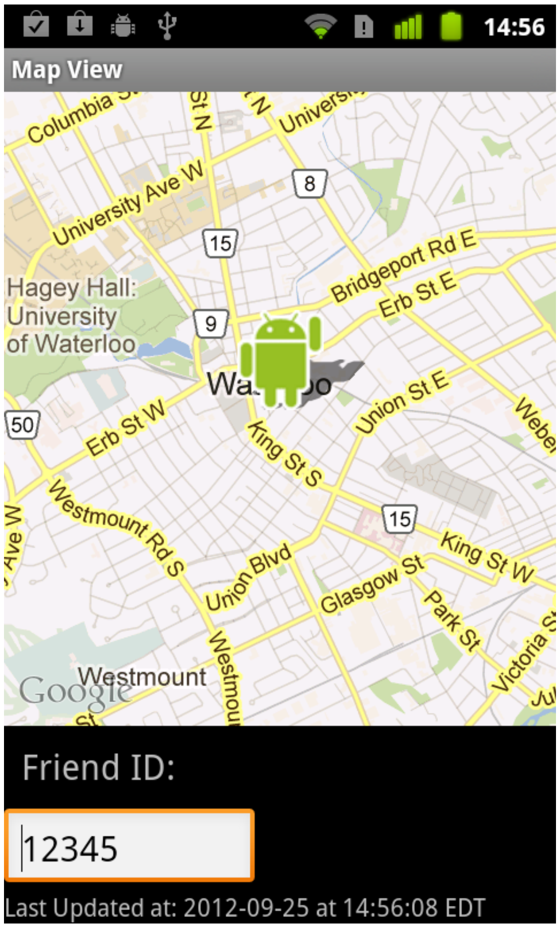


Figure 4.4: Pop-Up Menu Button For Initializing Friend Locator

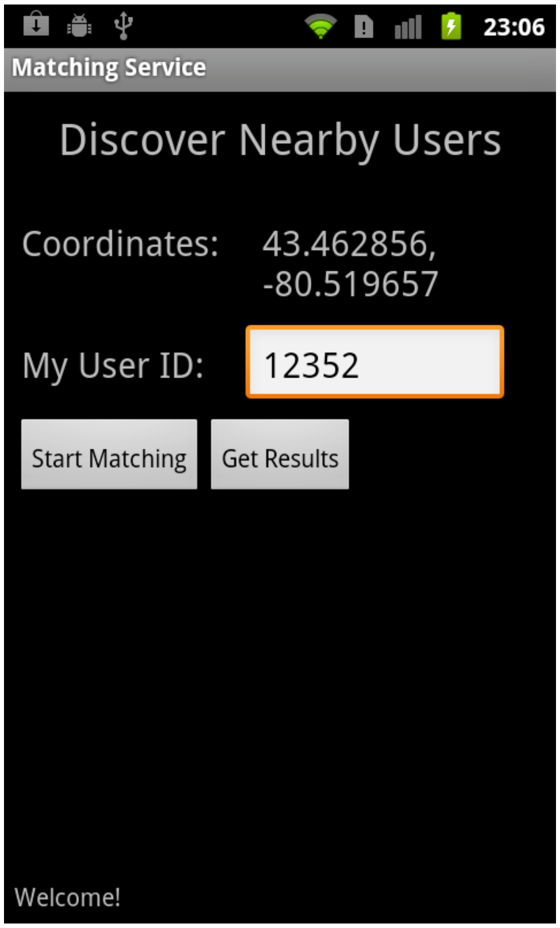


Friend ID:

12345

Last Updated at: 2012-09-25 at 14:56:08 EDT

Figure 4.5: Map View Showing Friend's Location



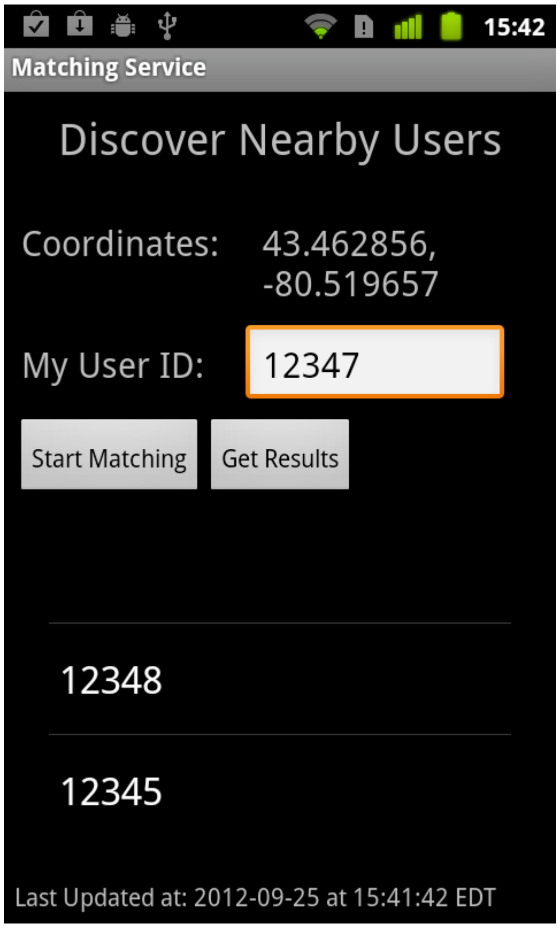
Discover Nearby Users

Coordinates: 43.462856, -80.519657

My User ID: 12352

Welcome!

Figure 4.6: Matching Service Welcome Screen



Discover Nearby Users

Coordinates: 43.462856, -80.519657

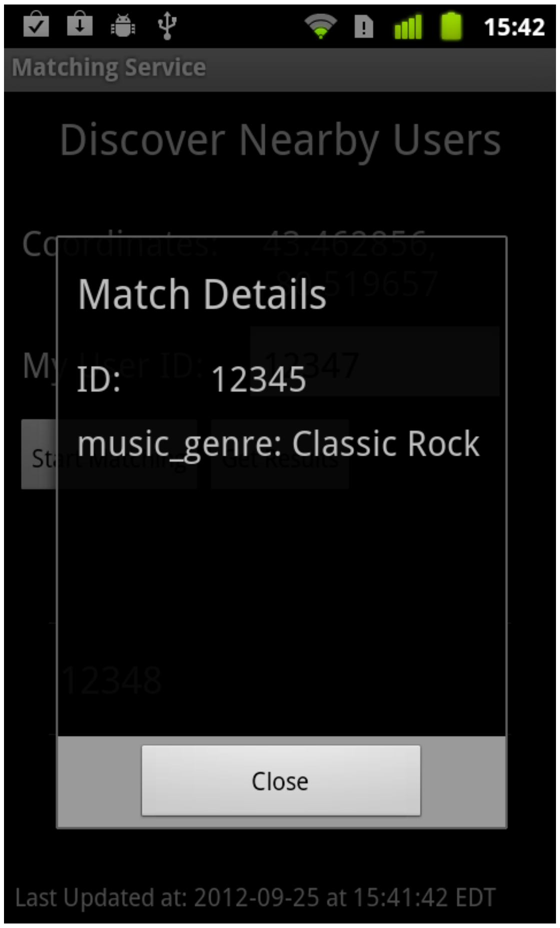
My User ID: 12347

12348

12345

Last Updated at: 2012-09-25 at 15:41:42 EDT

Figure 4.7: Matching Service Screen



Match Details

ID: 12345

music_genre: Classic Rock

Last Updated at: 2012-09-25 at 15:41:42 EDT

Figure 4.8: Details of a Successful Match

Chapter 5

Future Work

Our current design and implementation still leave areas that could be improved on in the future.

5.1 Design

The expectations in the threat model could be weakened. If we eliminate the tracking threat of a user checking in frequently along a route, our architecture could be used to implement a navigation application (currently our threat model precludes this type of application). We could also weaken the honest-but-curious threat model applied to **U**, **L**, and **C** if we could prevent entities in the system from making requests that are not part of defined protocols such as:

- **C** making requests to **L** that are not on behalf of a user
- **U** making any request to **L**

Other aspects of our design could be improved on in the future. Overall, we would like our architecture to be general enough to support many application use cases and have privacy preservation properties that are as strong as an application-specific solution. Adding rate-limiting at **C** would prevent rogue users from registering large numbers of callbacks over a small area in order to be matched with (and therefore learn information about) a lot of users. Figuring out a way for **L** to verify if a tag provided in a request is valid would prevent non-users from checking into registered locations. It would also improve the overall privacy of the system if we could avoid **C** learning the social graph (i.e., who is friends with whom) and an entity (that cannot decrypt a stored attribute value) learning extra information from the mere existence of an attribute name or the encrypted attribute value.

5.2 Use Cases

Our sample applications could be improved in a few areas. For some applications, billing or other monetization features would have to be added in the future for the business aspects of deploying the applications to be viable. In the matching service, we would like to add a way of avoiding stale matches in a scenario where a user moves on to a new location before the callback for their old location has expired.

5.3 Implementation

Our implementation is considered proof-of-concept and there are some features that need to be added. All requests to **U** from $\mathbf{D_i}$ must be routed through a location-hiding proxy. Registration is currently done by manually adding entries to the databases in **U** and **C**. In the future, a user should be able to register from their device or a web portal. The implementation of **L**'s `setAttribute` currently does not have the `tag` parameter so checking for tag replay attacks still needs to be done. We also wish to implement the remaining use cases we presented and examine their real-world functionality. On the device, location coordinates are hard-coded for testing purposes but a deployed application would need to make use of localization hardware to determine a user's location. Management of separate keys for each user and attribute is not currently implemented, but should be included. When **L**'s `setAttribute` is called, it triggers callbacks, makes requests to the appropriate handlers (at cloud components that support applications) and then blocks while waiting for these requests to complete. Ideally, `setAttribute` should trigger callbacks in a separate thread and then return immediately so that the user-observed execution time is not application-dependent.

It is possible that the lengths of encrypted values in **U** may leak information so more analysis needs to be done to see if unrestricted access to encrypted values is a privacy issue. If a large amount of data is sent in a request to any of our entities, the server framework has trouble decoding the request and often throws an exception. Research into the issue suggests that there is a bug in the CherryPy framework that needs to be worked around (perhaps by splitting requests with large amounts of data into multiple requests). We determined in experiments that database operations are often a bottleneck, so in the future we would like to investigate the possibility of reducing the number of database operations. Finally, more experimental analysis to determine how well our system would scale to a large numbers of users (and their associated attributes and callbacks), registered locations, and applications (with supporting cloud components) should be conducted.

Chapter 6

Conclusions

Location-based applications for smartphones are becoming very popular because an application can enhance a user's experience if it knows her location. However, allowing a service provider (which supports an application) to collect a user's location on a frequent basis (given that they already have identifying information for the user) is a privacy concern. Our architecture is designed with privacy as the primary feature and it provides a platform for building privacy-friendly mobile location-based social applications. Secondary goals include being able to support many application use cases and not requiring unreasonable amounts of computation from a mobile device. The public API functions of entities in our system and details showing how applications can be built using these functions are presented. We provide a working implementation of the complete architecture and proof-of-concept applications with the server components in Python and the device application on the Android platform. Finally, we present results from experiments investigating the execution time of basic operations and the real-world practicality of our architecture.

References

[1] Angry Birds Space - Android Apps on Google Play. https://play.google.com/store/apps/details?id=com.rovio.angrybirdsspace.ads. [Online; accessed October 2012].

[2] Angry Birds Space Premium - Android Apps on Google Play. https://play.google.com/store/apps/details?id=com.rovio.angrybirdsspace.premium. [Online; accessed October 2012].

[3] Badoo. http://www.badoo.com. [Online; accessed September 2012].

[4] Chronology of Data Breaches. http://www.privacyrights.org/data-breach. [Online; accessed October 2012].

[5] FourSquare. http://www.foursquare.com. [Online; accessed September 2012].

[6] Garmin Mobile. http://www8.garmin.com/mobile/mobilext/. [Online; accessed September 2012].

[7] Google AdMob Ads SDK. https://developers.google.com/mobile-ads-sdk/docs/android/intermediate. [Online; accessed September 2012].

[8] Google Latitude. http://www.google.com/latitude. [Online; accessed September 2012].

[9] Google Maps. http://maps.google.com. [Online; accessed September 2012].

[10] Google Play. https://play.google.com/store?hl=en. [Online; accessed September 2012].

[11] myYearbook. http://www.myyearbook.com. [Online; accessed September 2012].

[12] protobuf - Protocol Buffers - Google's data interchange format. http://code.google.com/p/protobuf/. [Online; accessed September 2012].

[13] Skout. http://www.skout.com. [Online; accessed September 2012].

[14] Spongy Castle by rtyley. http://rtyley.github.com/spongycastle/. [Online; accessed September 2012].

[15] Waze. http://www.waze.com. [Online; accessed September 2012].

[16] Randy Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee, and Daniel Starin. Persona: An Online Social Network With User-Defined Privacy. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 135–146. ACM, 2009.

[17] Alastair R. Beresford and Frank Stajano. Location Privacy in Pervasive Computing. *IEEE Pervasive Computing*, 2(1):46–55, January 2003.

[18] Igor Bilogrevic, Murtuza Jadliwala, Kübra Kalkan, Jean-Pierre Hubaux, and Imad Aad. Privacy in Mobile Computing for Location-Sharing-Based Services. In *Proceedings of the 11th International Conference on Privacy Enhancing Technologies*, PETS '11, pages 77–96. Springer-Verlag, 2011.

[19] A.J. Bernheim Brush, John Krumm, and James Scott. Exploring end user preferences for location obfuscation, location-based services, and the value of location. In *Proceedings of the 12th ACM International Conference on Ubiquitous computing*, Ubicomp '10, pages 95–104. ACM, 2010.

[20] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private Information Retrieval. *Journal of the ACM*, 45(6):965–981, November 1998.

[21] Ionut Constandache, Romit Roy Choudhury, and Injong Rhee. Towards Mobile Phone Localization Without War-Driving. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 2321–2329. IEEE Press, 2010.

[22] Landon P. Cox, Angela Dalton, and Varun Marupadi. SmokeScreen: Flexible Privacy Controls for Presence-Sharing. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 233–245. ACM, 2007.

[23] Yoni De Mulder, George Danezis, Lejla Batina, and Bart Preneel. Identification via Location-Profiling in GSM Networks. In *Proceedings of the 7th ACM Workshop on Privacy in the Electronic Society*, WPES '08, pages 23–32. ACM, 2008.

[24] Matt Duckham and Lars Kulik. A formal model of obfuscation and negotiation for location privacy. In Hans Gellersen, Roy Want, and Albrecht Schmidt, editors, *Pervasive Computing*, volume 3468 of *Lecture Notes in Computer Science*, pages 243–251. Springer Berlin / Heidelberg, 2005.

[25] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 1–6. USENIX Association, 2010.

[26] Julien Freudiger, Reza Shokri, and Jean-Pierre Hubaux. Evaluating the Privacy Risk of Location-Based Services. In *Financial Cryptography*, volume 7035 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2011.

[27] Sébastien Gambs, Marc-Olivier Killijian, and Miguel Núñez del Prado Cortez. Show Me How You Move and I Will Tell You Who You Are. *Transactions on Data Privacy*, 4(2):103–126, 2011.

[28] Gabriel Ghinita, Panos Kalnis, Ali Khoshgozaran, Cyrus Shahabi, and Kian-Lee Tan. Private Queries in Location Based Services: Anonymizers are not Necessary. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 121–132. ACM, 2008.

[29] Gabriel Ghinita, Carmen Ruiz Vicente, Ning Shang, and Elisa Bertino. Privacy-Preserving Matching of Spatial Datasets with Protection Against Background Knowledge. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, pages 3–12. ACM, 2010.

[30] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST '12, pages 291–307. Springer-Verlag, 2012.

[31] Marco Gruteser and Dirk Grunwald. Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 31–42. ACM, 2003.

[32] Marco Gruteser and Baik Hoh. On the anonymity of periodic location samples. In *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings*, Lecture Notes in Computer Science, pages 179–192. Springer-Verlag, 2005.

[33] Saikat Guha, Mudit Jain, and Venkata Padmanabhan. Koi: A Location-Privacy Platform for Smartphone Apps. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Apr 2012.

[34] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. "These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 639–652. ACM, 2011.

[35] Sharad Jaiswal and Animesh Nandi. Trust No One: A Decentralized Matching Service for Privacy in Location Based Services. In *MobiHeld '10*, pages 51–56. ACM, 2010.

[36] Lukasz Jedrzejczyk, Blaine A. Price, Arosha K. Bandara, and Bashar Nuseibeh. I Know What You Did Last Summer: Risks of Location Data Leakage in Mobile and Social Computing. Technical Report TR2009-11, Department of Computing, Faculty of Mathematics, Computing and Technology, The Open University, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom, November 2009. [Online; accessed May 2011 at http://computing-reports.open.ac.uk/2009/TR2009-11.pdf].

[37] Ali Khoshgozaran and Cyrus Shahabi. Private Buddy Search: Enabling Private Spatial Queries in Social Networks. In *CSE (4)*, pages 166–173. IEEE Computer Society, 2009.

[38] John Krumm. Inference Attacks on Location Tracks. In *Proceedings of the 5th International Conference on Pervasive Computing*, PERVASIVE '07, pages 127–143. Springer-Verlag, 2007.

[39] Ilias Leontiadis, Christos Efstratiou, Marco Picone, and Cecilia Mascolo. Don't Kill My Ads!: Balancing Privacy in an Ad-Supported Mobile Application Market. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, pages 2:1–2:6. ACM, 2012.

[40] Sergio Mascetti, Dario Freni, Claudio Bettini, X. Sean Wang, and Sushil Jajodia. Privacy in Geo-Social Networks: Proximity Notification with Untrusted Service Providers and Curious Buddies. *The VLDB Journal*, 20(4):541–566, August 2011.

[41] Arvind Narayanan, Narendran Thiagarajan, Mugdha Lakhani, Michael Hamburg, and Dan Boneh. Location Privacy via Private Proximity Testing. In *NDSS '11*. The Internet Society, 2011.

[42] Femi Olumofin, Piotr Tysowski, Ian Goldberg, and Urs Hengartner. Achieving Efficient Query Privacy for Location Based Services. In *Proceedings of the 11th International Conference on Privacy Enhancing Technologies*, PETS '10. Springer-Verlag, 2010.

[43] Sarah Perez. The New Social Network: Who's Nearby, Not Who You Know. http://techcrunch.com/2011/09/16/the-new-social-network-whos-nearby-not-who-you-know/, September 2011. [Online; accessed September 2011].

[44] Philippe Golle and Kurt Partridge. On the Anonymity of Home/Work Location Pairs. In *Pervasive Computing, 7th International Conference, Pervasive 2009, Nara, Japan, May 11-14, 2009. Proceedings*, Lecture Notes in Computer Science, pages 390–397, 2009.

[45] Tatiana Pontes, Marisa Vasconcelos, Jussara Almeida, Ponnurangam Kumaraguru, and Virgilio Almeida. We Know Where You Live: Privacy Characterization of Foursquare Behavior. In *4th International Workshop on Location-Based Social Networks (LBSN 2012)*, LBSN '12, 2012.

[46] Krishna P. N. Puttaswamy and Ben Y. Zhao. Preserving Privacy in Location-Based Mobile Social Applications. In *HotMobile '10*, pages 1–6. ACM, 2010.

[47] Bharat Rao and Louis Minakakis. Evolution of Mobile Location-Based Services. *Communications of the ACM*, 46(12):61–65, December 2003.

[48] Amre Shakimov, Harold Lim, Ramón Cáceres, Landon P. Cox, Kevin Li, Dongtao Liu, and Alexander Varshavsky. Vis-à-Vis: Privacy-Preserving Online Social Networking via Virtual Individual Servers. In *Communication Systems and Networks (COMSNETS), 2011 Third International Conference on*, COMSNETS '11, pages 1–10. IEEE, January 2011.

[49] Minho Shin, Cory Cornelius, Dan Peebles, Apu Kapadia, David Kotz, and Nikos Triandopoulos. AnonySense: A System for Anonymous Opportunistic Sensing. *Pervasive Mobile Computing*, 7(1):16–30, February 2011.

[50] Reza Shokri, George Theodorakopoulos, Jean-Yves Le Boudec, and Jean-Pierre Hubaux. Quantifying Location Privacy. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 247–262. IEEE Computer Society, 2011.

[51] Reza Shokri, Carmela Troncoso, Claudia Díaz, Julien Freudiger, and Jean-Pierre Hubaux. Unraveling an Old Cloak: k-anonymity for Location Privacy. In *WPES '10*, pages 115–118. ACM, 2010.

[52] Amin Tootoonchian, Stefan Saroiu, Yashar Ganjali, and Alec Wolman. Lockr: Better Privacy for Social Networks. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 169–180. ACM, 2009.

[53] Hui Zang and Jean Bolot. Anonymization of Location Data Does Not Work: A Large-Scale Measurement Study. In *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, Mobicom '11. ACM, 2011.

[54] Ge Zhong, Ian Goldberg, and Urs Hengartner. Louis, Lester and Pierre: Three Protocols for Location Privacy. In *Proceedings of the 7th International Conference on Privacy Enhancing Technologies*, PETS '07, pages 62–76. Springer-Verlag, 2007.