Zerosquare: A Privacy-Friendly Location Hub for Geosocial Applications

Sarah Pidcock and Urs Hengartner
Cheriton School of Computer Science, University of Waterloo
{snpidcoc,uhengart}@cs.uwaterloo.ca

***Abstract*—The localization abilities of smartphones have provided a huge boost to the popularity of geosocial applications, which facilitate social interaction between users geographically close to each other. However, today's geosocial applications raise privacy concerns due to application providers storing large amounts of information about users (e.g., profile information) and locations (e.g., users present at a location). We propose Zerosquare, a privacy-friendly location hub that encourages the development of privacy-preserving geosocial applications. Our primary goal is to store information such that no entity can link a user's identity to her location. Other goals include decoupling storing data from manipulating data for social networking purposes, designing an architecture flexible enough to support a wide range of use cases, and limiting client-side computation.**

Zerosquare consists of two separate server components for storing information about users and about locations, respectively, and optional cloud components for supporting applications. We describe the design of the API exposed by the server components and demonstrate how it can be used to build several sample geosocial applications. We provide a proof-of-concept implementation using Python for the server components and the Android platform for the mobile devices and build several real-world geosocial applications on top of Zerosquare. Finally, we present experimental results that demonstrate the practicality of Zerosquare.

I. Introduction

With the evolution of smartphones into powerful computing devices, a broad range of applications that make use of the phones' localization features has materialized. Early location-based applications enabled users to retrieve content relevant to their current location (e.g., points of interests (POIs)). More recently, geosocial applications have started to appear [37]. These applications facilitate social interaction between users who may be geographically close to each other. For example, friend location and proximity detection help users who are already friends in an online social network meet in person when they are close (e.g., Google Latitude [4] and Foursquare [2]). Interest matching can 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 (e.g., Badoo [1], Skout [7], and MeetMe [5]).

The most concerning drawback of today's geosocial applications is the privacy problem presented by the frequent use of localization features on smartphones. A user must provide her location to an application provider in order to request information from an application or use its features. An application provider can build up large amounts of location data along with timestamps indicating when the user made a request. This data allows the provider to localize users (know their location at a particular point in time), build up traces of users' movement, and even track users. The 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. This data can be stored over a long time period and aggregated or statistically analyzed. Knowledge of a user's location at particular times, especially when combined with other information about the user, can be used by the provider to learn more about the user than she may have intended when signing up. An alarming amount of information, such as a user's home and work locations, activities, and relationships can be inferred from data that a provider has [30], [39]. Pseudonymizing or anonymizing location data may not be sufficient. An adversary can determine a user's home location and identity when given pseudonymized location data [31] and some outside information (such as a movement profile [16]). Trajectory information can be used to link a user's periodic anonymous location samples [24]. Given an inference of an individual's home and work locations, her anonymity set has been shown to be unique within a census block [23], [48]. Even when a user only occasionally exposes her location to a location-based service, the user still faces privacy risks because her home location and points of interest remain identifiable from the limited set of data points [20].

Both research (some recent examples are [18], [21], [33], [36] and [42]) and media reports [10], [26] have demonstrated that location privacy is of concern to people. However, recent research [19], [40] has also shown that people are not particularly concerned about making their location available to a location-based service (whereas there is concern about making location available to non-location-based services). This observation makes sense considering that today's location-based services need location information and maybe identity information to provide their service and that many people assume that this is the only way a location-based service can function. However, it is an open question whether people would still be willing to provide their location to a location-based service if they became aware of the kind of analytics (see above) or data leaks [44] that this data gathering enables. In general, the assumed willingness of people to provide their location to a location-based service should not stop researchers from investigating whether there are more privacy-friendly ways to provide these services. Importantly, the principle of "privacy by design" [28] suggests to proactively embed privacy into the design of any service. If research demonstrates that location-based services can be built in more privacy-friendly ways, this in turn may shift people's thinking about and expectations of the inner workings of location-based services.

Some providers of geosocial applications or of location-based services in general have started to become *location hubs*. A location hub is a centralized repository of location information, collected through servicing application requests, that lets other providers take advantage of this information to provide additional location-based services. For example, Foursquare has introduced Connected Apps [3]. Here, a user can choose from a set of applications developed by third parties and have all her location check-ins forwarded to her chosen applications, which in turn can provide additional information to the user (e.g., the Weather Channel application provides weather forecasts for the user's location). A location hub stores lots of identifying information about the users making requests, which is undesirable from a privacy standpoint. Our research focuses on designing a privacy-friendly location hub that can be accessed as needed by applications requiring geosocial functionality. We also aim to make our design flexible and simple enough to encourage development of privacy-friendly applications.

We propose to disassociate user identity information from user location information in our privacy-friendly location hub. No entity should know both a user's identity and her location. The foundation of our location hub, Zerosquare, are two non-colluding entities, one that stores information about users and another that stores information about locations. These entities expose basic API functions that can be used as building blocks for a wide range of geosocial applications. Users can either directly retrieve data from these entities or ask an application's supporting cloud component to retrieve data on their behalf to participate in social networking activities. Zerosquare 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). We make the following contributions:

- We present the design of a privacy-friendly location hub for building geosocial applications that stores data about users and data about locations separately.
- We implement the location hub using Python for the server components and the Android platform for the mobile devices.
- We emphasize the flexibility of our design by building various geosocial applications on top of it. Our experimental results demonstrate its practicality.

We look at related work next. We describe our design goals in Section III and our system and threat model in Section IV. In Section V we explain our location hub. We go through several use case for our location hub in Section VI. Section VII provides a security analysis. We discuss our implementation in Section VIII and provide experimental results in Section IX.

II. Related Work

Lots of location privacy research has focused on retrieving POIs from a service such that the privacy of the accessing user remains protected from the service provider. Geosocial applications cover a broader range of services and raise additional privacy concerns. For example, in a POI application, the locations of the POIs are typically not sensitive and do not change. Neither of these properties holds for a geosocial application, where the POIs are people, whose location may be sensitive and dynamic. In turn, privacy-enhancing technologies developed for the retrieval of non-sensitive, static POIs (such as technologies based on private information retrieval [11], [12]) have limited applicability to protecting the privacy of users in geosocial applications. Techniques based on Oblivious RAM [22] may be applicable, but are still orders of magnitude slower than non-oblivious databases [45]. Techniques for anonymous communication [12], [17] are useful building blocks in Zerosquare, but are not sufficient by themselves.

In this section, we focus on related work in the area of geosocial applications. Krumm [32] and Terrovitis [46] survey location privacy research with a focus on retrieving POIs.

Several researchers have developed privacy-enhancing technologies for specific geosocial application use cases. For example, protocols have been developed to determine if two users are within a threshold distance without either user revealing her location to the other [34], [35], [49], to find a fair meeting point without revealing any of the users' locations to a third party or other participants [9], to locate nearby people with shared interests without broadcasting personal information [14], [47], or for presence sharing without becoming trackable by a third party or by strangers [13]. The main drawback of these solutions is that they are application-specific and do not generalize to a wide range of geosocial application use cases.

In terms of privacy for (not necessarily geo) social applications, several solutions have been developed. For example, Persona [8] is a social network designed specifically with privacy in mind. All user data stored in the social network is encrypted using attribute-based encryption. However, by having only users (but not locations) become first-class citizens in the architecture, the applicability of these architectures to geosocial applications remains limited because storing or retrieving information about locations is difficult. Vis-à-Vis [43] addresses this concern to some degree. It is a privacy framework for online social networks in which each user has her own Virtual Individual Server (VIS) in the cloud. 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, the user's cloud provider can learn both her identity and her location, which is inconsistent with our privacy goal (see Section III).

Most relevant to our research are privacy-preserving frameworks for geosocial applications that provide support for various use cases and that prevent storage providers from learning both a user's identity and her location [41], [29], [25].

Similar to Persona, these frameworks decouple the storage of data from the actual social networking functionality.

In Puttaswamy and Zhao's framework [41], an untrusted server acts as a data store for encrypted data, where only authorized users (like friends) have decryption keys. The server keeps two databases, one for storing encrypted user profiles, the other one for storing encrypted place information. The storage interface allows users to put/get encrypted attributes in/from their profile and store/retrieve encrypted data to/from a location. From a privacy point of view, the framework is susceptible to traffic-analysis attacks. Data about users and data about locations are stored on the same server. If a user first updates her encrypted location attribute in her profile and then adds an encrypted entry for herself to the location database, the server can link the two updates and learn the user's location. The framework may also suffer from performance problems because to read information produced by a friend about a place (e.g., a review), a user must download *all* encrypted data about the place and attempt to decrypt each item with *all* keys in her possession. Functionality-wise, the architecture supports only applications that are targeted at people with pre-existing relationships, but it does not support applications that involve strangers (e.g., an application that matches nearby strangers with common interests).

Both Trust No One [29] and Koi [25] store information in plaintext form and rely on double indirection between a user and her attributes (e.g., her location) for protecting privacy. This double indirection allows to store knowledge of the link between the user and her attributes with a separate party, but without this party learning any information about the actual user or attributes.

Trust No One is a privacy-preserving platform for locating entities and relies on three non-colluding components. The mobile operator maintains mappings between actual locations and pseudonymized locations. The LBS provider maintains mappings between business identifiers and pseudonymized identifiers. The matching service links pseudonymized identifiers to pseudonymized locations. For a query, a user gets the pseudonymized identifier of her desired business and her pseudonymized location from the LBS provider and from the mobile operator, respectively, and asks the matching service for a match. Whereas the authors mention that their approach can be employed for locating nearby friends, this claim is questionable. The problem is that to generate the links kept by the matching service, the architecture assumes that the LBS provider knows the locations of the entities that can be located. Whereas this is a reasonable assumption for businesses, it does not hold 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 entities. It is unclear whether and how other geosocial applications can be implemented in the architecture.

Koi is a privacy-preserving platform for geosocial applications and relies on two non-colluding components. The matcher maintains mappings between actual user identifiers and pseudonymized user identifiers and between user attributes and pseudonymized attributes. The combiner links pseudonymized user identifiers to pseudonymized attributes. For a query, the two parties interact to determine whether there is a user having the desired attributes. 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 and can track and ultimately re-identify a user.

Finally, we are not the first ones to propose to separate knowledge of a person's identity and her location between two non-colluding parties. For example, the Virtual Triplines architecture [27] also exploits this idea. However, the earlier works focus on specific, non-geosocial applications and are not as flexible as our architecture.

III. Design Goals

We envision that our privacy-friendly location hub provides more privacy than widely deployed location hubs (such as Foursquare), which typically provide no privacy from the hub provider, but it may provide less privacy than application-specific privacy-enhancing technologies for geosocial applications.[1] Consider the case of a proximity service for alerting of nearby friends. The application-specific solutions in Section II are ideal from a privacy point of view since they do not require trusted third parties and let a user learn only whether a friend is nearby and no other information. However, the solutions cannot be used for other geosocial applications. Widely deployed location hubs, such as Foursquare, can be used for different kinds of geosocial applications. However, they are highly problematic from a privacy point of view since they let the hub provider continuously learn a user's identity and location. In our location hub, we strive for a solution that supports different types of applications and that has acceptable, though not perfect privacy properties. In particular, we have the following goals:

1) Privacy-Friendly By Design: 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. Our goal is to create a location hub that encourages an approach to application development in the spirit of the second option.

2) Privacy based on Separation of Information: In Zerosquare, we protect privacy by separating between a user's identity and her location. This approach is based on the observation that for public locations, such as a hospital or

[1] We leave the ideal solution of a privacy-friendly location hub that is as privacy-preserving as application-specific privacy-enhancing technologies for geosocial applications for future work.

a train station, a user's location is not sensitive with no other information associated with it. The same is true for a user's identity. However, when a user's location is linked to her identity, then there are privacy concerns. Therefore, our goal is to store and provide data in our location hub such that no entity can learn both a user's identity and her location.

3) Support of Various Application Use Cases: Researchers have developed privacy-enhancing technologies for specific geosocial applications (see Section II). However, many of these solutions are inappropriate for use in applications other than the ones that they were developed for. Our goal is to build a flexible architecture that exposes basic functions that can be combined to meet the needs of various geosocial applications. This simple model for building functionality eases the development of privacy-friendly applications, which is desirable in real-world deployment.

4) Decoupled Data Storage and 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 location hub should store data in a manner consistent with our privacy goals and provide access to that data. Applications that perform social networking-related functions should be able to request data from the hub and should only be granted access to information as desired by the original creator.

5) No Significant Client-Side Computation: Although the resources available in smartphones are steadily increasing, asking a device to perform a large number of computations (especially cryptographic ones) can still cause problems (e.g., drain the battery). Our goal is to 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 of her decryption keys on each item in the batch.

IV. System and Threat Model

Zerosquare can be used to develop geosocial applications in which users can store both information about themselves (e.g., their profile) and information about locations (e.g., users present at a location or a review a user has for a business at a location). To ensure that no entity learns both a user's identity and her location (separation of information goal), Zerosquare has two separate entities for storing information about users and information about locations. In addition, these entities only store information and do not provide any social networking functionality. Information is requested from both based on the needs of applications running on other entities. In more detail, Zerosquare consists of the following entities:

- $\mathbf{U}$: User-indexed database storing information about users.
- $\mathbf{L}$: Location-indexed database storing information about locations.
- $\mathbf{C_k}$: Optional component of geosocial application $\mathbf{k}$ running in the cloud under the control of $\mathbf{k}$'s provider. We will use $\mathbf{C}$ to denote the set of all $\mathbf{C_k}$.
- $\mathbf{D_i}$: Device of user $\mathbf{i}$ running geosocial application $\mathbf{k}$ by interacting with $\mathbf{U}$ and $\mathbf{L}$ and maybe with $\mathbf{C_k}$.

An outline of communications between the entities is shown in Figure 1. All communications are secured with TLS.

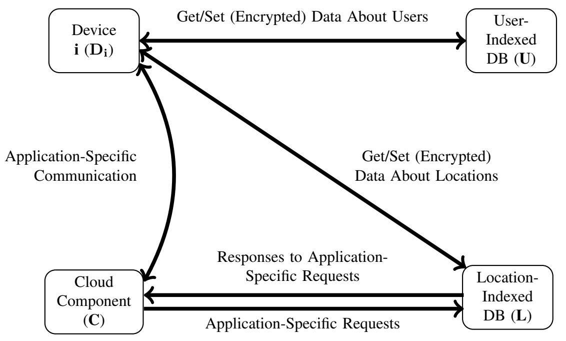
Device
i (Di)
Get/Set (Encrypted) Data About Users
User-Indexed DB (U)
Application-Specific Communication
Get/Set (Encrypted) Data About Locations
Responses to Application-Specific Requests
Cloud Component (C)
Application-Specific Requests
Location-Indexed DB (L)

Fig. 1. System Architecture.

$\mathbf{U}$, $\mathbf{L}$, and $\mathbf{C}$ are *honest-but-curious* and do not collude. The entities will perform according to specification, but any entity may try to discover additional information using any data they store or requests they handle. $\mathbf{U}$ is trusted to store information about users, but should not learn sensitive user information, such as a user's location. $\mathbf{L}$ is trusted to keep information about locations, but should not learn information about users.[2] A $\mathbf{C_k}$ can learn sensitive information about a user (e.g., her interests), depending on the application that $\mathbf{C_k}$ is a component of, but only if authorized by the user, and this information can never include a user's location. Learning information about users enables $\mathbf{C_k}$ to offer the expected application functionality, analyze user data (e.g., to display ads, see Section VI), and potentially make revenue from this analysis, so there is an incentive for $\mathbf{C_k}$'s provider to participate in Zerosquare. In turn, this revenue can be shared with the organizations running $\mathbf{U}$ and $\mathbf{L}$ to give them an incentive to participate. $\mathbf{U}$, $\mathbf{L}$, and $\mathbf{C}$ will not become users of Zerosquare or collude with existing users in an attempt to learn more information.[3] $\mathbf{D_i}$ is trusted to store and manipulate information for user $\mathbf{i}$ and will share information that it learns about other users only with user $\mathbf{i}$. Preventing the leakage of sensitive information from a device is an orthogonal research topic.

A user storing sensitive information about herself in $\mathbf{U}$ encrypts this information with a key specific to the user and the type of information. We assume that users have access to out-of-band methods for managing and exchanging these keys, as key distribution and key revocation are out of the scope of this research. Users share keys with other users who they trust (i.e., their friends) or with the cloud component $\mathbf{C_k}$ of an application that they want to use so that their data can be used for social networking purposes. Importantly, $\mathbf{U}$ does not have access to these keys. This design meets the goal of decoupling data storage from social networking functionality.

[2]Since IP addresses can reveal identity or location information, we require $\mathbf{D_i}$ to communicate with $\mathbf{L}$ and $\mathbf{U}$ through an anonymization network or a proxy. We will ignore these intermediaries in the description of the architecture. Such intermediaries are also required, though not mentioned, in related work (e.g., [25], [27]).

[3]There are several other privacy-preserving infrastructures that rely on the honest-but-curious model and that assume that a semi-trusted entity does not become a user of the infrastructure (e.g., [15], [27]).

A user storing sensitive information about a place in $\mathbf{L}$ (e.g., a review that should be accessible only to her friends) encrypts this information with a key specific to the user and the type of information. To meet the separation of information goal, the encrypted information stored in $\mathbf{L}$ cannot be associated with the identity of the user. Therefore, a device $\mathbf{D_i}$ that retrieves ciphertexts about a location (e.g., all encrypted reviews) from $\mathbf{L}$ would have to try (maybe unsuccessfully) all decryption keys that it got from other users to decrypt a ciphertext and repeat this for every ciphertext. This would violate the goal of limiting client-side computation. Therefore, a user attaches a *tag* to encrypted information that she stores in $\mathbf{L}$. The tag does not allow $\mathbf{L}$ to learn the identity of the user, but an authorized $\mathbf{C_k}$ can turn a tag back into an identity and can tell $\mathbf{D_i}$ which decryption key to use (see Section V).

When storing information about a location in $\mathbf{L}$, a user may inadvertently reveal her identity if the location is private and bound to an identity (e.g., her home). Therefore, we assume that users do not store information about private places in $\mathbf{L}$. For example, we assume that users do not update (check in) their location with $\mathbf{L}$ if the location is private. (For a public location, a user would associate a tag with her checked-in location to hide her identity from $\mathbf{L}$.) If users are at a private location and wish to store their whereabouts in Zerosquare, they can always encrypt the location, store the ciphertext as part of their information in $\mathbf{U}$, and give the decryption key to whomever they wish to share their location with. Note that Lindqvist et al. [33] have shown that many Foursquare users already choose not to check in at private places so our assumption of users not checking in with $\mathbf{L}$ when being at a private location is justified.

Another assumption is that users do not frequently update their location with $\mathbf{L}$ while moving along a contiguous route. This assumption prevents the user from being re-identified through a tracking attack by $\mathbf{L}$. Again, if necessary, users can continuously update their (encrypted) location with $\mathbf{U}$.

We assume that all entities have a key pair with all public keys certified by a Certification Authority. No entity will share its private key with another entity, and entities will have the necessary public keys and certificates available when needed for opening TLS connections or verifying signatures.

V. Architecture Details

This section presents each entity in our system in detail.

A. User-Indexed Database

$\mathbf{U}$ provides storage for information about a user $\mathbf{i}$ in the form of a set of attributes. An attribute is indexed by a unique identifier uid_i associated with the user and an attribute name (e.g., "interests", "location", "friends"). The associated attribute value can be encrypted with a symmetric key chosen by the user. The user can choose a different key for each attribute. An attribute can have multiple values. Attribute values also contain an expiration time.

As mentioned earlier, a user attaches a tag to data stored in $\mathbf{L}$ to mark the data as having been created by this user without revealing the user's identity to $\mathbf{L}$. A user must use a new tag whenever she stores data in $\mathbf{L}$ to avoid becoming trackable by $\mathbf{L}$. The user requests tags from $\mathbf{U}$, which also maintains mappings between tags and identities. $\mathbf{U}$ also provides an access-controlled means for $\mathbf{C_k}$ 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 $\mathbf{C_k}$, which then retrieves data that is annotated with tags from $\mathbf{L}$. $\mathbf{C_k}$ requests the identities associated with these tags from $\mathbf{U}$. For a particular tag, $\mathbf{U}$ will inform $\mathbf{C_k}$ of the associated identity only if the corresponding user has previously whitelisted $\mathbf{C_k}$. $\mathbf{C_k}$ will then attach the received identities to the data (e.g., encrypted reviews) in its response to $\mathbf{D_i}$. $\mathbf{D_i}$ can use the identities to select the correct decryption keys. See Section VI for examples.

API provided by **U**
tag ← createTag(uid_i)
whitelist(uid_i, cid_k)
uid_i ← getIdentity(tag, cid_k)
setAttribute(uid_i, attr, val, ttl)
val(s) ← getAttribute(uid_i or cid_k, uid_j, attr)

API provided by **L**
setAttribute(loc, attr, tag, ttl)
token ← registerCallback(loc, attr, handler, ttl)
(val(s),tag(s)) ← getAttribute(loc, attr)

TABLE I
APIs PROVIDED BY **U** AND **L**.

The API provided by $\mathbf{U}$ is shown in Table I. All requests must be signed by their caller. Requests also include a timestamp, which is used by $\mathbf{U}$ to defend against replay attacks. We assume that user $\mathbf{i}$ has registered with $\mathbf{U}$ to establish a mapping between uid_i and the public key of her device $\mathbf{D_i}$. The same applies to geosocial application $\mathbf{k}$ and the mapping between its identifier cid_k and $\mathbf{C_k}$'s public key.

- `tag ← createTag(`uid_i`)`: $\mathbf{U}$ creates a tag for user $\mathbf{i}$, remembers the mapping from tag to uid_i, and returns the tag. This function will succeed only if called by $\mathbf{D_i}$.
- `whitelist(`uid_i`, `cid_k`)`: $\mathbf{U}$ adds application $\mathbf{k}$ to the whitelist for user $\mathbf{i}$. By whitelisting application $\mathbf{k}$, the user informs $\mathbf{U}$ that she is using this application and hence $\mathbf{U}$ is allowed to hand over this user's tag-to-uid_i mapping to $\mathbf{C_k}$. This function will succeed only if called by $\mathbf{D_i}$.
- uid_i `← getIdentity(tag, `cid_k`)`: $\mathbf{U}$ returns the identifier associated with a tag. This function will succeed only if $\mathbf{C_k}$ was previously whitelisted by $\mathbf{D_i}$.
- `setAttribute(`uid_i`, attr, val, ttl)`: $\mathbf{U}$ stores the attribute/value pair indexed by uid_i until it expires (indicated by `ttl`). The value may be encrypted. This function will succeed only if called by $\mathbf{D_i}$.
- `val(s) ← getAttribute(`uid_i `or` cid_k`, `uid_j`, attr)`: $\mathbf{U}$ returns the (maybe encrypted) value(s) stored for the given uid_j/attribute pair. This function will succeed only if called by $\mathbf{D_i}$ or $\mathbf{C_k}$ (to defend against DoS attacks).

B. Location-Indexed Database

L provides storage for information about a location in the form of a set of attributes. An attribute is indexed by a representation of the location (e.g., GPS coordinates, address, labeled region) and an attribute name (e.g., "checked in", "reviews, "ads"). The associated attribute value can be encrypted with a symmetric key chosen by the creator of the value. An attribute can have multiple values (e.g., multiple reviews of a restaurant), each from a different creator. Attribute values can be associated with a tag to trace the value back to its creator and locate the decryption key for an encrypted value. Attribute values also contain an expiration time.

$\mathbf{D_i}$ must encrypt location data with **L**'s public key in all requests to **L** to avoid $\mathbf{C_k}$ from learning user **i**'s location during social networking operations where $\mathbf{C_k}$ makes requests to **L** on $\mathbf{D_i}$'s behalf (see Section VI for examples). $\mathbf{C_k}$ cannot learn user **i**'s location because of our goal that no entity learns 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 $\mathbf{C_k}$ (on behalf of $\mathbf{D_i}$) to register a handler with **L** for a particular condition (e.g., a new value being added for an attribute name at a particular location) and to call the handler when that condition is met. Like attribute values, callback entries have an expiry time. After each data storage operation, **L** must check all callbacks to see if any have been triggered. Once **L** has called the handler to complete the callback operation, $\mathbf{C_k}$ can do application-specific processing and provide results to users. The API provided by **L** is shown in Table I.

- `setAttribute(loc, attr, val, tag, ttl)`: **L** remembers the attribute/value pair for the given location until it expires. For example, for the attribute "checked in", the value would be a boolean. For the attribute "reviews", the value would be a (maybe encrypted) review. The tag, created by **U**, contains information about the creator of the attribute/value pair but without revealing the identity of the creator to **L**. For example, for the attributes "checked in" and "reviews", the tag can be used to identify the person who checked in and who wrote the review, respectively. A tag can be used only once in a `setAttribute` request, and **L** will reject all future `setAttribute` requests with the same tag to defend against replay attacks.
- `token ← registerCallback(loc, attr, handler, ttl)`: **L** remembers a callback for the given location/attribute pair until it expires. Namely, **L** calls the handler whenever an attribute for the location/attribute pair is set using the above function. The handler is a function provided by $\mathbf{C_k}$. The location can be a specific place or a geographical area.
- `(val(s),tag(s)) ← getAttribute(loc, attr)`: **L** returns the value(s) and tag(s) stored for a location/attribute pair. The location can be a specific place or a geographical area.

C. Cloud Components and Devices

$\mathbf{C_k}$ is the optional component of geosocial application **k** that runs in the cloud if required for scalability or privacy reasons. For example, consider an application for matching strangers with similar interests at the same location. 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 would violate user privacy) or run secure multiparty computation on encrypted data (which would impose too much of a computational load on the device). A cloud component of this application that has been whitelisted by individual users and that can decrypt their "interests" attribute can match nearby users on decrypted attributes without learning these users' location and inform users of a match. 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.

$\mathbf{D_i}$ (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.

VI. Use Cases

We have realized a variety of applications to demonstrate the flexibility of Zerosquare. In particular, our applications consist of friend locator, friend proximity detection, interest matching, local search, social recommendations, and advertising services. Due to space reasons, we focus on a simple service (friend locator) and two more complex services (interest matching and social recommendations) in this section. For the remaining services, we refer to the extended version of this paper [38]. We have implemented the friend locator and the interest matching services. Again, for details, including screenshots, please see the extended version. It is also possible to exploit Zerosquare for "traditional" location-based applications, such as a POI service. Here, the POIs would be stored in **L**, and a user can retrieve them from **L**. If the user does not want to reveal her location to **L**, she can exploit existing privacy-enhancing technologies for the retrieval of POIs [32], [46].

A. Friend Locator

Suppose user **i** and user **j** are friends (i.e., they have exchanged decryption keys for the attribute "location"), user **j** has made a recent (encrypted) location update with **U**, 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.

B. Interest Matching

The purpose of this 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 who are also interested in basketball.

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

1) $\mathbf{D_i}$ whitelists $\mathbf{C_k}$ with **U** by calling `whitelist(`uid_i, cid_k`)` and gives the decryption key for the attribute "interests" (stored with **U**) to $\mathbf{C_k}$. $\mathbf{C_k}$ retrieves user **i**'s interests by sending a `getAttribute` request to **U**, decrypts them, and caches them for later use in matching.
2) $\mathbf{D_i}$ asks $\mathbf{C_k}$ to register a callback for her current location ℓ. For this purpose, she provides **C** with her location encrypted with **L**'s public key so that **C** cannot learn it. Then **C** sends a request to **L**'s `registerCallback` function with the encrypted location, the attribute name "checked in", a time-to-live value, and the URL of a function that **L** can call to complete the callback.
3) $\mathbf{D_i}$ updates her location with **L**. Namely, $\mathbf{D_i}$ retrieves a tag for uid_i from **U** using a `createTag` request and sends a `setAttribute` request with the location, the attribute name "checked in", the tag and a time-to-live value to **L**. This is done to immediately trigger callbacks and generate results for users at user **i**'s location (including user **i** herself).
4) $\mathbf{D_j}$ does steps 1)-3).
5) $\mathbf{D_j}$'s location update has **L** trigger callback handling for location ℓ. **L** calls $\mathbf{C_k}$'s callback handler with the list of all tags checked in at location ℓ.
6) While executing the callback, $\mathbf{C_k}$'s handler issues `getIdentity` requests for the tags received from **L** to get a list λ of mappings from tags to user identities for all users who have whitelisted $\mathbf{C_k}$.
7) $\mathbf{C_k}$'s handler matches the interest attributes associated with the users in λ against each other. Any matching algorithm can be used because $\mathbf{C_k}$ can see plaintext interests. Since both user **i** and user **j** are at location ℓ and whitelisted $\mathbf{C_k}$, they are in the set to be matched. Matching occurs and matching attributes for sets of users are stored. At least one match is stored for users **i** and **j** because they have (a) common interest(s).
8) If $\mathbf{D_i}$ and $\mathbf{D_j}$ support push notifications, $\mathbf{C_k}$ can alert them of the match immediately. Otherwise, $\mathbf{D_i}$ and $\mathbf{D_j}$ periodically poll $\mathbf{C_k}$ to retrieve their matches.

C. Social Recommendations

A social recommendations application allows users to write reviews of POIs that are accessible (maybe only) to their friends. Suppose user **i** writes a review about a place and $\mathbf{D_i}$ stores it in **L**. User **j** would like to read reviews about the location. User **i** and user **j** are friends, which means they have each others' decryption keys for the attribute "reviews". The social recommendations application **k** is supported by cloud component $\mathbf{C_k}$.

1) $\mathbf{D_i}$ calls `whitelist(`uid_i, cid_k`)`. It also gives the decryption key for the attribute "friends" (stored with **U**) to $\mathbf{C_k}$.
2) $\mathbf{D_i}$ creates a review. Namely, $\mathbf{D_i}$ retrieves a tag for uid_i from **U** using a `createTag` request and stores an encrypted review with **L** using `setAttribute(loc, ''reviews'', <review text>, tag, ttl)`. The value `<review text>` is encrypted with $\mathbf{D_i}$'s key for attribute "reviews". Alternatively, $\mathbf{D_i}$ can store a public review with **L** using `setAttribute(loc, ''public_reviews'', <review text>, tag, ttl)` and leaving `<review text>` unencrypted.

To retrieve public reviews, $\mathbf{D_j}$ calls `getAttribute(loc, ''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(loc, ''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 the writer has whitelisted $\mathbf{C_k}$.
4) For each received identity, $\mathbf{C_k}$ invokes **U**'s `getAttribute` 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, $\mathbf{C_k}$ 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.

VII. Security Analysis

The main focus of Zerosquare security-wise is the separation 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 as each request must include a user's identifier. When a user stores her location in **U**, she encrypts it with a symmetric key specific to her "location" attribute. For **U** to discover her location, it would have to collude with the user's friends to decrypt her stored location or collude with **L** to look up the location where the user's most recent tag was seen, both of which we rule out in our threat model. We also rule out other types of actively malicious behaviour, like **U** creating a fake user and becoming friends with the user to obtain her decryption keys, or **U** querying several public locations in **L** around the known home location for a user (which could be learned from the phonebook) and compare results with recently generated tags for the user.

L receives requests to set attributes about locations from users that are annotated with tags. 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

$\mathbf{C_k}$ that the user has whitelisted (and can therefore call **U**'s `getIdentity` to get the identity behind the tag) but we rule out collusion in the threat model. **L** could also attempt to query **U**'s `getIdentity`, which will fail because **L** is not whitelisted with **U**.

Separating **U** from **L** addresses the traffic-analysis threat that exists in related work [25], [41]. All updates to a user's attributes in **U** are seen only by **U**, not by **L**. Sensitive attributes of a user, such as her location, have their values encrypted with a key not known to **U**. Updates to a location in **L** are seen only by **L**, not by **U**, and the update reveals no information about the creator to **L**. Therefore, as long as **U** and **L** do not collude, which is forbidden by our threat model, neither **U** nor **L** can link between updates to a user and updates to a location.

C receives a user's identity and can see attributes in plaintext if a user has provided it 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 launch active attacks similar to the active attacks that we discussed for **U** to learn location information, but our threat model rules out active attacks.

If an adversary is a user of the system, she can register many callbacks in locations that are close together to try and get matched with a particular user. Rate-limiting at **C** could solve this problem. She cannot impersonate another user without their private key because to communicate with **U** all requests must be signed. A user 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.

As for threats not coming from entities in the system, a passive adversary can learn only which entities are communicating with each other but cannot learn anything because all communications are secured with TLS. An active adversary cannot launch a replay attack at **U** because a timestamp must be included in all requests, which are signed, and **U** verifies that a timestamp has not been presented before by the user making the request. An active adversary can check in with **L** with a tag generated by the adversary (instead of by **U**) because **L** is oblivious to tag semantics. However, the fake tag will be detected when **C** asks **U** to map it to an identity. An active adversary that learns a tag associated with a user (e.g., using **L**'s `getAttribute` call) will not be able to replay the tag in a `setAttribute` call since **L** remembers tags that it sees and discards re-used tags.

VIII. Implementation

This section presents the implementation of the key components of our framework. **U**, **L**, and **C** are written in Python 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 TLS support that can be accessed using HTTPS requests. All communications between components package data using protocol buffers [6] in the bodies of HTTPS POST requests. For symmetric-key encryption operations, AES with 128-bit key size is used. RSA with 2048 bit key size is used for public-key encryption and signatures.

U's `getTag` function creates a tag by encrypting a user's identity with a probabilistic, symmetric-key encryption scheme whose secret key is known only to **U**. We extended **U**'s `getAttribute` and `getIdentity` functions to support batch operations for efficiency and let queries include multiple attributes or multiple users and multiple tags, respectively,

The devices $\mathbf{D_i}$ were built using the Android platform. A device polls $\mathbf{C_k}$ 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]

IX. Experimental Evaluation

In this section, we discuss the experimental evaluation of Zerosquare. We describe experiments to determine the server processing overhead as well as the end-to-end processing time at the client for each of the API functions offered by **U** and **L**. We also analyze the time taken to run operations in the cloud component $\mathbf{C_k}$ associated with an interest matching application. During the experiments, **L** and $\mathbf{C_k}$ run on a 3.4 GHz quad-core machine with 4 GB of RAM, and **U** runs on a 2.4 GHz dual-core machine with 4 GB of RAM. The client runs on a Nexus One smartphone with Android 2.3.6 installed, and comparison measurements are 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 each API function in the server entities, we perform 500 trials with the Python client. Due to instability issues in the Android platform over a large number of consecutive executions of a function, we perform only 250 trials for each API function with the Android client. We present mean and standard deviation across all trials.

A. Experiments on U

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

Function	Server	Python Client	Android Client
`createTag`	1.0 ± 0.0	27.4 ± 6.5	181.4 ± 28.3
`whitelist`	6.6 ± 2.5	39.4 ± 7.0	181.8 ± 21.4
`getIdentity`	4.0 ± 0.0	39.6 ± 5.4	–
`getAttribute`	2.9 ± 0.5	23.4 ± 4.3	182.1 ± 18.7
`setAttribute`	6.7 ± 1.4	31.4 ± 9.1	176.0 ± 28.3

TABLE II
Server processing latency and client end-to-end latency for API functions offered by **U** [ms].

[4]To hide their IP address from **U** and **L**, devices access **U** and **L** via proxies that support the HTTP CONNECT command (see Footnote 2).

Zerosquare performs well with unnoticeable latency at the clients. Executing the operations involves verifying a digital signature to verify the caller's identity. For the `createTag` function, we find that this verification consumes most of the server's processing time. For functions involving addition of data to the system (`whitelist` and `setAttribute`), the database insert operations occupy the greatest portion of the latency. For the remaining functions, execution time is split between retrieving information from databases and cryptographically verifying a requestor's identity.

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) shows that a lot of time is 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).

For a batch experiment, we test `getAttribute` for a user/attribute pair with an increasing number of attributes to determine how the amount of data being returned affects the execution of the function. The number of returned results is increased until the test becomes unstable on the Android device. The timing results are shown in Table III.

Attributes	Server	Python Client	Android Client
10	3.0 ± 0.1	22.5 ± 1.1	287.5 ± 85.17
100	10.0 ± 0.4	39.7 ± 1.6	403.9 ± 184.0
500	45.4 ± 1.7	82.2 ± 5.8	782.1 ± 371.8
1,000	84.5 ± 2.6	139.1 ± 6.5	1,052.5 ± 71.4
5,000	418.1 ± 79.3	589.9 ± 16.0	4,670.4 ± 610.8
10,000	624.1 ± 208	873.4 ± 284.2	10,575.4 ± 2,812.7

TABLE III
SERVER PROCESSING LATENCY AND CLIENT END-TO-END LATENCY FOR **U**'S `GETATTRIBUTE` FUNCTION WITH VARYING NUMBERS OF ATTRIBUTES RETURNED [MS].

A breakdown of the results shows that for ten attributes, 33% of the server processing latency is due to verifying the requestor's identity. For 100 attributes, the percentage decreases to 10%. Increasing the amount of data returned by `getAttribute` even further causes encoding of the data for transport and retrieving the data from the database to dominate execution time. For the Android device, the time spent waiting for a request to return a large number of attributes becomes high for more than 5,000 attributes.

We also examine the batch behaviour of `getIdentity`. Due to space limitations, we refer to the extended version [38].

B. Experiments on **L**

For the experiments on API functions provided by **L**, `getAttribute` is executed for a location and attribute for which there is only one value. `setAttribute` is executed for a location that does not have any callbacks registered to get an application-independent measurement. Since `registerCallback` is always called on a user's behalf by **C**, no measurements are done with the Android client. The results are shown in Table IV.

Function	Server	Python Client	Android Client
`getAttribute`	17.3 ± 4.9	35.1 ± 8.1	235.1 ± 54.9
`setAttribute`	42.8 ± 4.8	62.6 ± 7.3	462.1 ± 108.7
`registerCallback`	23.5 ± 4.3	106.5 ± 13.7	–

TABLE IV
SERVER PROCESSING LATENCY AND CLIENT END-TO-END LATENCY FOR **L**'S API FUNCTIONS [MS].

Like in the experiments on **U**, we find that cryptographic operations consume a significant time on **L** because **L** uses public-key cryptography to decrypt locations received in requests. Nearest-neighbour queries to determine the registered location in the database closest to the coordinates provided in a request are also noticeable in the server processing latency. From the viewpoint of the user, none of the operations require noticeable time on the Android device, which contributes to the practicality of Zerosquare.

C. Experiments on **C**

We now evaluate the cloud component $\mathbf{C_k}$ built to support an interest matching application. An API provided by $\mathbf{C_k}$ allows $\mathbf{D_i}$ to register with the application and to retrieve matches and **L** to invoke callback handlers. Matching of two users is done by directly comparing attribute/value pairs until one common pair is found or each pair belonging to a user has been compared against all of the other user's pairs and no common pairs were found.

We test the registration function exposed to $\mathbf{D_i}$ and measure a latency of 188.1 ms ± 14.5 ms at $\mathbf{C_k}$. The Python client and Android client experience end-to-end latencies of 241.1 ms ± 38.6 ms and 602.6 ms ± 72.8 ms, respectively. From a user's perspective, neither of these times is noticeable enough to be impractical.

To test callback processing at $\mathbf{C_k}$, we set up a scenario where two users with five attributes each are checked in at and have 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 is measured at $\mathbf{C_k}$ (77.3 ms ± 6.8 ms) and at **L** (129.6 ms ± 5.9 ms).

D. Experiments on **D**

We use the same scenario to measure the time needed by a client to complete all of the operations necessary to receive a match with another user (registering for matching, checking into a location, and retrieving results). From a user's perspective, the Android client takes a noticeable amount of time (1,388.5 ms ± 151.2 ms). In comparison, the Python client takes less than half as long (506.6 ms ± 52.7 ms) to complete the same operations.

X. CONCLUSIONS

We have presented Zerosquare, a location hub for building geosocial applications that is designed with privacy as the primary feature. Secondary goals include being able to support various application use cases and not requiring unreasonable amounts of computation from a mobile device. We have discussed the API of the entities in our system and shown how applications can be built using these functions. We have also provided a working implementation of Zerosquare and proof-of-concept applications. Our experimental results demonstrate real-world practicality of Zerosquare.

Future work includes weakening the assumptions made in our threat model (e.g, defending against actively malicious **U**, **L**, or **C**) and enabling users to continuously check in with **L** without them becoming identifiable or trackable.

Acknowledgements

We thank Hassan Khan, Bisheng Liu, Danish Mehmood, and the anonymous reviewers for their helpful comments. This work is supported by a Google Focused Research Award and by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Badoo. http://www.badoo.com. Accessed Feb 2013.

[2] Foursquare. http://www.foursquare.com. Accessed Feb 2013.

[3] Foursquare Connected Apps. http://blog.foursquare.com/2012/06/28/. Accessed Feb 2013.

[4] Google Latitude. http://www.google.com/latitude. Accessed Feb 2013.

[5] MeetMe. http://www.meetme.com. Accessed Feb 2013.

[6] protobuf - Protocol Buffers - Google's data interchange format. http://code.google.com/p/protobuf/. Accessed Feb 2013.

[7] Skout. http://www.skout.com. Accessed Feb 2013.

[8] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: An Online Social Network With User-Defined Privacy. In *Proc. of SIGCOMM 2009*.

[9] I. Bilogrevic, M. Jadliwala, K. Kalkan, J.-P. Hubaux, and I. Aad. Privacy in Mobile Computing for Location-Sharing-Based Services. In *Proc. of PETS 2011*.

[10] B. X. Chen. iPhone keeps record of everywhere you go. http://www.wired.com/gadgetlab/2011/04/iphone-tracks, April 2011. Accessed Feb 2013.

[11] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private Information Retrieval. In *Proc. of FOCS 1995*.

[12] D. A. Cooper and K. P. Birman. Preserving Privacy in a Network of Mobile Computers. In *Proc. of IEEE Symposium on Security and Privacy 1995*.

[13] L. P. Cox, A. Dalton, and V. Marupadi. SmokeScreen: Flexible Privacy Controls for Presence-Sharing. In *Proc. of Mobisys 2007*.

[14] A. De Cristofaro, Durussel and I. Aad. Reclaiming Privacy for Smartphone Applications. In *Proc. of PerCom 2011*.

[15] E. De Cristofaro, C. Soriente, and G. Tsudik. Hummingbird: Privacy at the time of Twitter. In *Proc. of IEEE Symposium on Security and Privacy 2012*.

[16] Y. De Mulder, G. Danezis, L. Batina, and B. Preneel. Identification via Location-Profiling in GSM Networks. In *Proc. of WPES 2008*.

[17] R. Dingledine, Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *Proc. of USENIX SEcurity 2004*.

[18] C. Efstratiou, I. Leontiadis, M. Picone, K. K. Rachuri, C. Mascolo, and J. Crowcroft. Sense and Sensibility in a Pervasive World. In *Proc. of Pervasive 2012*.

[19] D. Fisher, L. Dorner, and D. Wagner. Location Privacy: User Behavior in the Field. In *Proc. of SPSM 2012*.

[20] J. Freudiger, R. Shokri, and J.-P. Hubaux. Evaluating the Privacy Risk of Location-Based Services. In *Proc. of FC 2011*.

[21] P. Gage Kelley, M. Benisch, L. F. Cranor, and N. Sadeh. When Are Users Comfortable Sharing Locations with Advertisers? In *Proc. of CHI 2011*.

[22] O. Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proc. of STOC 1987*.

[23] P. Golle and K. Partridge. On the Anonymity of Home/Work Location Pairs. In *Proc. of Pervasive 2009*.

[24] M. Gruteser and B. Hoh. On the Anonymity of Periodic Location Samples. In *Proc. of SPC 2005*.

[25] S. Guha, M. Jain, and V. Padmanabhan. Koi: A Location-Privacy Platform for Smartphone Apps. In *Proc. of NSDI 2012*.

[26] L. Hickman. How I became a Foursquare cyberstalker. http://www.guardian.co.uk/technology/2010/jul/23/foursquare, July 2010. Accessed Feb 2013.

[27] B. Hoh, M. Gruteser, R. Herring, J. Bain, D. Work, J.-C. Herrera, A. M. Bayen, M. Annavaram, and Q. Jacobson. Virtual Trip Lines for Distributed Privacy-Preserving Traffic Monitoring. In *Proc. of MobiSys 2008*.

[28] Information and Privacy Commissioner of Ontario. http://privacybydesign.ca. Accessed Feb 2013.

[29] S. Jaiswal and A. Nandi. Trust No One: A Decentralized Matching Service for Privacy in Location Based Services. In *Proc. of MobiHeld 2010*.

[30] L. Jedrzejczyk, B. A. Price, A. K. Bandara, and B. 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, November 2009.

[31] J. Krumm. Inference Attacks on Location Tracks. In *Proc. of Pervasive 2007*.

[32] J. Krumm. A Survey of Computational Location Privacy. *Personal and Ubiquitous Computing*, 13(6), August 2009.

[33] J. Lindqvist, J. Cranshaw, J. Wiese, and J. Zimmerman. Im the Mayor of My House: Examining Why People Use foursquare - a Social-Driven Location Sharing Application. In *Proc. of CHI 2011*.

[34] S. Mascetti, D. Freni, C. Bettini, X. S. Wang, and S. Jajodia. Privacy in Geo-Social Networks: Proximity Notification with Untrusted Service Providers and Curious Buddies. *The VLDB Journal*, 20(4), August 2011.

[35] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location Privacy via Private Proximity Testing. In *Proc. of NDSS 2011*.

[36] S. Patil, G. Norcie, A. Kapadia, and A. J. Lee. Reasons, Rewards, Regrets: Privacy Considerations in Location Sharing as an Interactive Practice. In *Proc. of SOUPS 2012*.

[37] S. 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. Accessed Feb 2013.

[38] S. Pidcock. A Privacy-Friendly Architecture for Mobile Social Networking Applications. Master's thesis, Cheriton School of Computer Science, University of Waterloo, January 2013.

[39] T. Pontes, M. Vasconcelos, J. Almeida, P. Kumaraguru, and V. Almeida. We Know Where You Live: Privacy Characterization of Foursquare Behavior. In *Proc. of LBSN 2012*.

[40] A. Porter Felt, S. Egelman, and D. Wagner. I've Got 99 Problems, But Vibration Ain't One: A Survey of Smartphone Users' Concerns. In *Proc. of SPSM 2012*.

[41] K. P. N. Puttaswamy and B. Y. Zhao. Preserving Privacy in Location-Based Mobile Social Applications. In *Proc. of HotMobile 2010*.

[42] M. P. Scipioni and M. Langheinrich. To Share or Not To Share? An Activity-centered Approach for Designing Usable Location Sharing Tools. In *Proc. of SOUPS 2012*.

[43] A. Shakimov, H. Lim, R. Caceres, L. P. Cox, K. Li, Dongtao Liu, and A. Varshavsky. Vis-à-Vis: Privacy-Preserving Online Social Networking via Virtual Individual Servers. In *Proc. of COMSNETS 2011*.

[44] R. Singel. White Hat Uses Foursquare Privacy Hole to Capture 875K Check-Ins. http://www.wired.com/threatlevel/2010/06/foursquare-privacy, June 2010. Accessed Feb 2013.

[45] E. Stefanov and E. Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *Proc. of IEEE Symposium on Security and Privacy 2013*.

[46] M. Terrovitis. Privacy Preservation in the Dissemination of Location Data. *SIGKDD Exploratory*, 13(1), August 2011.

[47] Q. Xie and U. Hengartner. Privacy-Preserving Matchmaking For Mobile Social Networking Secure Against Malicious Users. In *Proc. of PST 2011*.

[48] H. Zang and J. Bolot. Anonymization of Location Data Does Not Work: A Large-Scale Measurement Study. In *Proc. of Mobicom 2011*.

[49] G. Zhong, I. Goldberg, and U. Hengartner. Louis, Lester and Pierre: Three Protocols for Location Privacy. In *Proc. of PET 2007*.