Koi: A Location-Privacy Platform for Smartphone Apps

Saikat Guha
Microsoft Research India
saikat@microsoft.com

Mudit Jain
Microsoft Research India
t-muditj@microsoft.com

Venkata N. Padmanabhan
Microsoft Research India
padmanab@microsoft.com

Abstract — With mobile phones becoming first-class citizens in the online world, the rich location data they bring to the table is set to revolutionize all aspects of online life including content delivery, recommendation systems, and advertising. However, user-tracking is a concern with such location-based services, not only because location data can be linked uniquely to individuals, but because the low-level nature of current location APIs and the resulting dependence on the cloud to synthesize useful representations virtually guarantees such tracking.

In this paper, we propose *privacy-preserving location-based matching* as a fundamental platform primitive and as an alternative to exposing low-level, latitude-longitude (lat-long) coordinates to applications. Applications set rich location-based triggers and have these be fired based on location updates either from the local device or from a remote device (e.g., a friend's phone). Our Koi platform, comprising a privacy-preserving matching service in the cloud and a phone-based agent, realizes this primitive across multiple phone and browser platforms. By masking low-level lat-long information from applications, Koi not only avoids leaking privacy-sensitive information, it also eases the task of programmers by providing a higher-level abstraction that is easier for applications to build upon. Koi's privacy-preserving protocol prevents the cloud service from tracking users. We verify the non-tracking properties of Koi using a theorem prover, illustrate how privacy guarantees can easily be added to a wide range of location-based applications, and show that our public deployment is performant, being able to perform 12K matches per second on a single core.

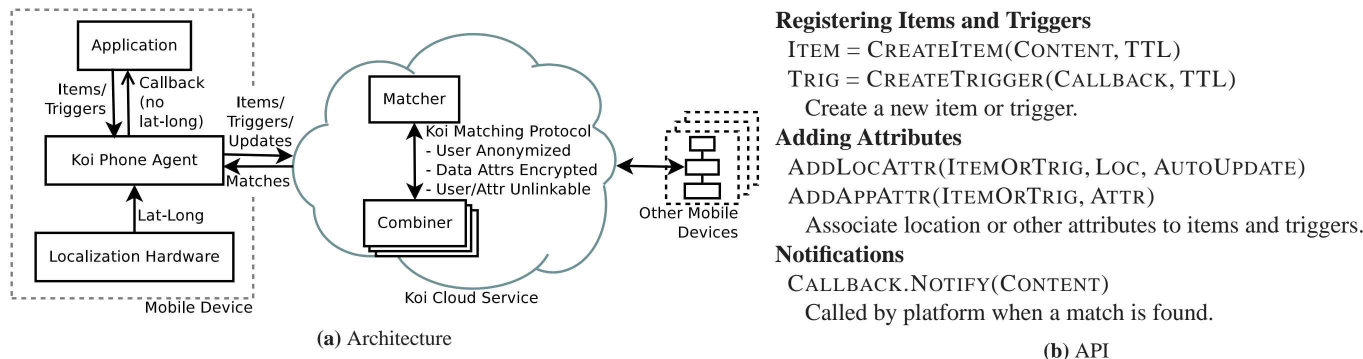
1 Introduction

The skyrocketing popularity of smart-phones has all but eviscerated the notion of "location-based services" as a separate class of applications. Today, virtually *all* applications and services must leverage location information. These applications and services include search, social networking, and multi-player games, to name just a few. Juxtaposed with this broad need for location information is the inherent complexity of individual applications operating on location data and its implications on user privacy. Recent high-profile incidents have put the spotlight on location privacy, with even governments and regulatory bodies asking questions of technology providers [6].

A popular approach in the research literature to providing location privacy is *obfuscation*, wherein a user's location coordinates are made imprecise by adding noise or are entirely suppressed, based on such factors as user density and the sensitivity of a location [19, 23]. While this approach has merits, obfuscation creates an unnecessary tension between the quality of location-based functionality and user privacy. Moreover, this approach does not address the crux of the privacy challenge: trusted applications (e.g., navigation app), which have a legitimate need for access to user location information, inadvertently leaking this to third-parties (e.g., Google maps), who are then in a position to link a user's ID to their location, and possibly track the user over time.

In this paper, we argue for a different approach, Koi, which is based on one key idea: switching to location *matching* instead of location *lookups*. In other words, rather than having an application look up mobile node's lat-long coordinates, Koi lets the application specify a location event of interest (e.g., proximity to fixed location such as a grocery store or the dynamic location of a friend) and notifies the application when there is a location match. Not only does the Koi approach relieve the application developer from having to work with the nitty-gritty of low-level lat-long information, it avoids polluting the application with lat-long information, thereby preventing accidental leakage of this information. Indeed, privacy incidents in the past have arisen from the carelessness (e.g., a third-party library used by the app developer, such as an advertising control, that constantly sends the user's lat-long to the third-party when lat-long is irrelevant to the original application [14]) rather than malice (e.g., an application that actively tries to subvert the location privacy of the mobile user).

The design of Koi comprises three main elements. The first, which arises directly from the approach outlined above and builds on prior concepts of triggers and symbolic locations [4, 30], is a *callback-based matching API* for applications. This API allows the application to specify the location event of interest, whether in terms of a static location or a dynamic location. The second element is a *privacy-preserving cloud-based matching service*, which uses a novel design comprising two non-colluding entities that together implement matching, while ensuring that neither entity learns the association between a user's ID and their location. This property is important since the matching service is a third-party as



(a) Architecture

(b) API

Figure 1: Koi Platform

far as the user is concerned and so should not be in a position to learn the user's location. The third element is support for *rich, semantically-meaningful, multi-attribute matching* that arises from the observation that applications have diverse requirements. Indeed, location-based applications are about more than just location. For example, a user might want to know when grocery store is nearby, so whether a store qualifies as a "grocery store" is equally important as proximity of the user to the store. Further, what qualifies as a "grocery store" depends on the application; for example, one could choose to treat a store tagged as a "supermarket" also as a grocery store. Even within the narrow domain of the location attribute, what qualifies as "nearby" is app-dependent in a way that opaque location tokens cannot accommodate.

We have designed and implemented the Koi system, and have deployed the matching service running on a production cloud platform. We show through micro- and macrobenchmarks that our implementation of Koi is performant enough for practical use despite the overhead imposed by privacy constructs. As we elaborate on in Section 7, Koi's location-based matching API naturally accommodates applications such as social networking, local search, recommendations, and advertising, which are essentially based on matching. Finally, as discussed in Section 6, we have verified using the ProVerif theorem-prover [33] that Koi protocols preserve location-privacy.

2 Overview of Koi

Koi comprises two components, one which runs on the user's mobile device and the other in the cloud (Figure 1a).

The mobile component of Koi interfaces between applications and the cloud component. To applications, it exposes a simple API (Figure 1b), which allows registration and updating of *items* and *triggers*, and provides notification through a callbacks. An *item* is a statement of fact. It contains information about an entity such as a user, a business, a vehicle (e.g., bus), etc., in the form of one or more *attributes* of the item, including its location. The *location attribute* is special in that an application can set it to be updated automatically by Koi, for example, a user's location as they move. *Triggers* are similar to items except that these represent queries, specifically a request for a callback when a match is found.

The mobile component of Koi interfaces with the cloud component by communicating with it to register items and triggers, and to set and update their attributes. The Koi cloud service comprises two sub-components — the *matcher* and the *combiner*[1] — which are assumed to be non-colluding. In broad terms, the *matcher* knows about identities of users (and other items) and also their attributes (including location), but it does *not* know the association between the users and their location or other attributes. On the other hand, the *combiner* knows the association between (anonymized) users and (encrypted) locations (and other attributes), but it does not know the actual identities or the attribute values. A privacy-preserving protocol enables the matcher and the combiner to perform matching without either of them learning about the association between the users and their locations, or more generally between the identities of items and their attributes. At the same time, knowledge of the (plain-text) location and other attributes enables the matcher to perform rich, semantically-meaningful matching based, for example, on geocoding, location proximity or spelling correction.

Multiple applications can use a common Koi service. Attributes names are name-spaced to the application registering the item to avoid name collisions. Applications may inter-operate by registering triggers for another application's attributes. Multiple Koi services (we envision a handful) may operate independently.

[1] As discussed in Section 5, our design also accommodates multiple combiners.

3 Goals, Non-goals, and Assumptions

Having sketched an overview of Koi, we now discuss the specific goals of the system, some non-goals, and the assumptions made.

The goal of Koi is to provide location functionality to applications that need it while ensuring that no third party (i.e., entity other than the user and the application) is in a position to learn the association between a user's identity and their location (or other attributes).

It is *not* a goal of Koi to prevent a malicious application from leaking a user's location information[2]. A malicious application is one where the application-developer intentionally and deliberately violates privacy. Our position is that the fact that a user has chosen to run an application implies an implicit trust in the application's non-maliciousness, even if the application might be buggy or may include arbitrary third-party code not audited by the application developer. Applications where the developer does not intend to violate privacy, but has a bug, or includes a third-party library (such as an ad control) that leaks user information (with our without the app-developer's knowledge) are not considered malicious. Koi protects against this latter class.

We assume that the location or other attribute itself is not sensitive, rather it is the *linkage* between the user identity and the attribute that is sensitive and needs to be protected. We leave it to the application developer to decide which non-sensitive attributes to register with Koi, depending on what matching functionality is desired.

The matcher and the combiner are assumed to be non-colluding with each other[3]. Furthermore, we assume an honest-but-curious attacker model for each of the matcher and the combiner. This means while the internal functioning of each entity is beyond scrutiny (e.g., they could try to glean information from the messages that come their way), the external interface of each entity must be conformant. We believe that this assumption reflects the real-world situation, where a service such as Google or Facebook would be wary of the PR backlash that might result from any externally-visible non-compliant behavior (e.g., active attacks) attributable to them. For example, if the matcher were to create and register fake users in an attempt to get matched with and thereby learn the location or other attributes of a real user, it would run the risk of being exposed when the real user realizes that they have been matched with a fake user.

4 Design

We present first the Koi platform API exposed to apps on the phone, followed by the Koi cloud service. We present the detailed protocol description in Section 5.

4.1 Platform API

The platform service model is similar to a database trigger. The app registers *item*s with the Koi phone-agent. Items may correspond to users or content (e.g., photos). The app associates *attribute*s with an item. Attributes may be locations, keywords, or arbitrary data. The app also registers *trigger*s. Triggers specify one or more attributes that must match. When an item matching the trigger is registered (by another user or app, or by the same app), the app registering the trigger is notified of the item through a callback. Figure 1b lists the Koi API.

The app specifies location attributes symbolically (e.g., loc:self, or within 1 block of loc:self). The Koi phone-agent internally replaces loc:self with the actual lat-long. The agent optionally automatically updates the lat-long if the user's location changes. This amortizes the cost of acquiring user location across multiple apps, and avoids having to wake each app up whenever the user moves. By never exposing lat-long data to the app, Koi minimizes the app accidentally leaking it to third-parties.

The app may associate arbitrary content with an item. A social networking app may, for instance, register the user's push-notification service handle so another user can contact this user. A citizen-journalism app may, for instance, register a photo or URL etc. The content may additionally be encrypted, for instance in the social networking app, only friends with the appropriate key may recover the push-notification handle.

4.2 Privacy-Preserving Matching Service

The operation of the Koi cloud service is best illustrated through an example. Consider the scenario in Table 1 where users Alice and Chuck register an item indicating they are tour-guides for Bangalore and Boston respectively. Bob registers a trigger looking for a tour-guide at his present location. The goal is to match Bob, who happens to be in Bangalore, with Alice. Note that Bob's phone-agent uses his lat-long without first geocoding it to Bangalore. For simplicity, we assume each of them register some unique user ID (as the item content, or the trigger callback) through which they can be reached. Our location-privacy goal is to prevent the cloud service from associating this user ID with the user's location.

At a high-level the Koi cloud service operates as follows. Each item or trigger is treated as a collection of rows — one for each attribute; Table 2a presents this *logical* view, i.e., it is never actually constructed or stored, and is shown here only to aid the description. Note a simple database approach that stores the user and attribute,

[2] We present coping strategies for malicious apps in Section 10.

[3] We discuss practical disincentives to collusion in Section 10.

ITEM (AliceID)	TRIGGER (BobID)	ITEM (ChuckID)
· TourGuide	· TourGuide?	· TourGuide
· loc:Bangalore	· loc:12.58N,77.38E?	· loc:Boston

Table 1: Original Data

Content/Callback	Attribute	RegId	AttrId
AliceID	TourGuide	P	A
AliceID	loc:Bangalore	P	B
BobID	TourGuide?	Q	C
BobID	loc:12.58N,77.38E?	Q	D
ChuckID	TourGuide	R	E
ChuckID	loc:Boston	R	F

(a) Logical View

Content/Callback	RegId
AliceID	P
BobID	Q
ChuckID	R
Attribute	**AttrId**
TourGuide	A, E
TourGuide?	C
loc:Bangalore	B
loc:12.58N,77.38E?	D
loc:Boston	F

(b) Matcher Partition

RegId	AttrId
P	A
P	B
Q	C
Q	D
R	E
R	F

(c) Combiner Partition

Table 2: Actual partitions stored by the Matcher and Combiner

while sufficient for matching, does not satisfy the privacy constraint. To achieve its privacy goals, Koi first associates a (random) RegId with each registered item or trigger, and a (random) AttrId with each attribute in the registration. For example, in Table 2a the TourGuide attribute in Alice's and Chuck's registrations are assigned different AttrIds A and E respectively, and both rows corresponding to Alice's item are assigned RegId P. How the RegId and AttrId are picked (and by whom) is described later in the Koi protocol section.

As mentioned, Koi partitions the logical table above into two halves that are placed on two different (non-colluding) entities, neither of which is able to link a registration with attribute(s) associated with it. The *matcher* stores Table 2b, and a *combiner* stores Table 2c. In the example, the matcher cannot place Chuck in Boston since he does not know the association between RegId R and AttrId F. At the same time, the combiner, which knows the link between R and F, doesn't know which user or what attribute they correspond to.

Matching is initiated by Bob when he registers his trigger (Q). Given a RegId Q, the combiner first queries the matcher for an associated AttrId (say, C); the matcher responds with AttrIds A, E, which the combiner maps to RegId P, R respectively. The matcher can answer this query since in Table 2b C maps to the query TourGuide? and A, E map to the answer TourGuide. The combiner then queries the matcher for another AttrId (D) associated with the original RegId; the matcher responds with AttrId B. This is because the matcher, which runs in the cloud, can geocode the plain-text attribute (loc:12.58N,77.38E?) associated with D to loc:Bangalore?. Note the matcher can support arbitrary matching algorithms here without affecting privacy; this rich matching is a feature unique to Koi. The combiner maps the matcher's response B to RegId P. At this point the combiner notes that RegId P matches both trigger attributes in Q, while R matches only one (without knowing what any of those attributes are). The combiner then informs the matcher that RegId P and Q match, and the matcher invokes the callback for Q (BobID) with the content registered for P (AliceID). Note that the combiner's queries for C and D (from Bob's trigger Q) are mixed with other ongoing queries for other users' trigger registrations (or mixed with cover traffic generated by the combiner) so the Matcher doesn't learn which AttrIds contributed to a given match, or even which AttrIds correspond to a single registration.

5 Koi Protocol

In this section we describe the three Koi protocols: registration, matching, and combining. We describe first the honest-but-curious matcher and single-combiner case. We then relax the restrictions on the matcher, and extend to multiple combiners.

5.1 Registration

The goal of the registration protocol is to create necessary state in the matcher and combiner for matching items to triggers (Table 2 illustrates this state, and Table 3c lists the protocol).

R1. The client encrypts each attribute ($attr_j$) associated with the item or trigger first with the matcher's public-key (M), and then with the combiner's public-key (C). Probabilistic encryption (e.g., by adding randomized padding) is used to defend against dictionary attacks. The double-encrypted attributes are sent to the matcher along with arbitrary $user$ data, which is the item content or trigger callback handle. As mentioned, item content may be encrypted so only certain users can decrypt it.

R2. The matcher picks a random registration ID rid for the registration, and forwards the double-encrypted attributes along with the rid to the combiner. Note multiple items/triggers from the same user are assigned different (random) rids.

The matcher stores in a table `R2U` a mapping from the rid to the arbitrary $user$ data and the ID of the registering user U.

R3. Upon receiving the double-encrypted attributes, the combiner decrypts them to reveal j attributes for that registration encrypted by the matcher's public-key. The combiner picks a random attribute ID aid_j for each of the j encrypted attributes. It sends aid_j and the jth encrypted attribute to the matcher (one at a time). These messages are mixed with aid/encrypted-attribute pairs from other ongoing registrations so the matcher can't link them back to which registration (rid) they are associated with. If enough natural cover traffic doesn't exist, the combiner can generate cover traffic.

The combiner stores the set of aid_j associated with the registration (rid) in the table `R2A`, and a reverse mapping from the aid_j to the rid in table `A2R`.

The matcher, upon receiving each aid/encrypted-attribute pair, decrypts the encrypted attribute to reveal the pain-text attribute $attr$. At this point the matcher may arbitrarily process the attribute to construct a set of equivalent $attr'_k$ For instance, the (misspelled) $attr$ Bretney may be corrected to include Britney in the equivalence set, or the location loc:12.58N,77.38E may be geocoded to include loc:Bangalore in the equivalence set. The matcher then updates table `T2A` for each $attr'$, which given a plain-text attribute returns the set of aids associated, to include the aid sent by the combiner. It also stores a reverse-mapping from aid to the plain-text set of equivalent attributes $attr'$ in table `A2T`.

This concludes the registration protocol.

5.2 Matching Attributes

Matching is event-driven: when a trigger (or item) is registered, the combiner looks for items (or triggers) matching it. Finding a match is relatively straight-forward (Table 3d). Given a registration (rid), the combiner looks up in table `R2A` the associated set of aid_j. It then executes the following two-message protocol individually for each aid_j.

M1. The combiner sends the aid_j to the matcher. As before, these messages are mixed with aids from other ongoing matches requests so the matcher can't link which sets of aids are associated with a single registration. And if enough natural cover traffic does not exist, the combiner can generate this cover traffic.

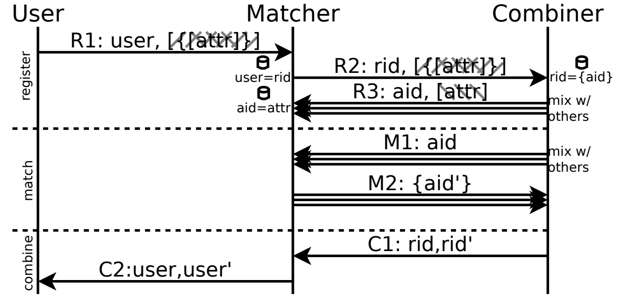
The matcher, upon receiving the message, looks up in table `A2T` the plain-text equivalent attribute set $\{attr'_k\}$ associated with that aid. For each $attr'$ in the equivalence set, the matcher looks up table `T2A` for the set of aid's registered. The matcher unions together these sets of aid's to construct the response.

M2. The matcher sends the response back to the combiner.

For each aid', the combiner queries table `A2R` to retrieve the registration rid' associated with that aid'. The combiner marks rid' as a match for the jth attribute for

$[m]_X$	Probabilistic encryption of m using public-key of X
U, U', M, C —	User, User 2, Matcher, Combiner
$user$	Arbitrary user data for item content; or callback key for trigger
$attr$	Attribute (plain-text)
rid, aid	Registration ID, Attribute ID
`R2U`, `R2A`, `A2R`, `T2A`, `A2T`$[x] \leftrightarrow y$ —	Table $key : x$, $val : y$; store ($\leftarrow$), lookup ($\rightarrow$)

(a) Notation



(b) Protocol. Encrypted parts hatched; Matcher key: \, Combiner key: /

#	From → To Message	Processing
R1.	$U \rightarrow M$ $user, [\{[attr_j]_M \vert \forall_j\}]_C$	
R2.	$M \rightarrow C$ $rid, [\{[attr_j]_M \vert \forall_j\}]_C$	**Matcher:** `R2U`$[rid] \leftarrow \langle U, user\rangle$
Repeat $\forall_j$ mixed with messages from other ongoing registrations.		
R3.	$C \rightarrow M$ $aid_j, [attr_j]_M$	**Combiner C:** `R2A`$[rid] \leftarrow$ `R2A`$[rid] \cup \{aid_j\}$ `A2R`$[aid_j] \leftarrow rid$ **Matcher** $\{attr'_k\} \leftarrow process(attr_j)$ $\forall_k$`T2A`$[attr'_k] \leftarrow$ `T2A`$[attr'_k] \cup \{aid_j\}$ `A2T`$[aid_j] \leftarrow \{attr'_k\}$

(c) Registration

#	From → To	Message	Processing
			Combiner C `R2A`$[rid] \rightarrow \{aid_j \vert \forall_j\}$
Repeat $\forall_j$ mixed with messages from other ongoing match requests.			
M1.	$C \rightarrow M$	aid_j	**Matcher** `A2T`$[aid_j] \rightarrow \{attr'_k\}$ $\bigcup_{\forall_k}$ `T2A`$[attr'_k] \rightarrow \{aid'_l \vert \forall_l\}$
M2.	$M \rightarrow C$	$\{aid'_l \vert \forall_l\}$	**Combiner C, $\forall_l$** `A2R`$[aid'_l] \rightarrow rid'_l$ $\mathcal{M}_{rid}[j] \leftarrow \mathcal{M}_{rid}[j] \cup rid'_l$

(d) Matching

#	From → To	Message	Processing
For each $rid' \vert \forall_j rid' \in \mathcal{M}_{rid}[j]$			
C1.	$C \rightarrow M$	rid, rid'	**Matcher** `R2U`$[rid] \rightarrow \langle U, user\rangle$ `R2U`$[rid'] \rightarrow \langle U', user'\rangle$
C2.	$M \rightarrow U$	$user', user$	

(e) Combining

Table 3: Koi Protocol Description

the rid for which the matching protocol was initiated (in match-set $\mathcal{M}_{rid}$).

Note, by design, there are no cryptographic operations in the match protocol to support high matching throughput.

5.3 Combining Matches

The simplest criteria the combiner can use to determine if rid' matches rid is if rid' is present in the match-set $\mathcal{M}_{rid}[j]$ for all j. The combiner can support richer criteria (e.g., boolean match expressions), which we omit for brevity.

C1. Once a match is found, triggering the callback is straightforward. The combiner notifies the matcher that registrations rid, rid' match. The matcher looks up in table `R2U` the associated users and content/callback data. We discuss below a slight modification of this that hides which pair of rids matched from the matcher.

C2. The matcher sends to the user registering the trigger, say user U, the $user$ data for the callback handle that fired, and the $user'$ data for matched item content.

User U's phone-agent, upon receiving notification C2, invokes the app-registered callback with the matched item content. The app is free, at this point, to present the content to the user (e.g., if the content is a photo), or initiate direct communication with the other user (e.g., if the content is a push-notification ID for the other user). The action the app takes in the callback function are external to Koi.

5.4 Extensions

We now discuss two extensions that further reduce the information learned by the matcher without affecting functionality.

Keeping the matched pair secret. In message C1 above, although the matcher doesn't learn which location-attribute contributed to the match, the matcher still learns which pair of users matched (which might implicitly leak some information). To exchange $user$ data without either the combiner or matcher learning what was exchanged and with whom, and doing so without establishing shared keys between the matched users, we use a commutative cryptography scheme [41]. In commutative crypto, a message encrypted first by key k_1 and then by key k_2 can be decrypted using the (inner) key k_1 to reveal the message singly encrypted by only the (outer) key k_2.

To keep the matching secret from the matcher, in message R1, the user encrypts the $user$ data with a per-registration randomly chosen key k_{rid} using a commutative encryption scheme; it sends k_{rid} to the combiner by putting it inside the R1 message component encrypted with the combiner's public-key (which the matcher forwards to the combiner in R2). The matcher additionally encrypts the encrypted $user$ data with a secret key (k_M), and includes the double-encrypted $user$ data in message R2. The combiner retrieves k_{rid} from the message and uses it to decrypt the double-encrypted $user$ data. By the commutative encryption property, the combiner now has the $user$ data for each rid encrypted with the matcher's secret key.

When notifying the user of the match, instead of sending message C1, the combiner retrieves the (encrypted) $user'$ data associated with rid', encrypts it using the k_{rid} registered by user U, and sends it to the matcher directing it to forward to the user for rid. The matcher decrypts the double-encrypted data using his secret key k_m, revealing $user'$ single encrypted with k_{rid}, which it sends to user U who can decrypt it to reveal the $user'$ data registered by the matched user.

During the protocol neither the matcher nor combiner learn the content of $user$ data, and during the notification phase encryption prevents the matcher from learning which other user's $user'$ data was sent to user U.

Multiple combiners. It may be tempting for the matcher to collude with the combiner if there is only one combiner. Having many combiners allows the user to pick which combiner he trusts to not collude with the matcher. Supporting multiple combiners requires a small change to the protocol. The user, in message R1, indicated to the matcher his choice of combiner C_x and uses C_x's public-key to encrypt the second component of the message. The matcher then forwards the message to C_x in R2. Random aids chosen by the combiner are namespaced to the combiner, e.g., by prefixing the combiner's domain name to the aid.

During matching, in message M2 a combiner receives aids registered both by itself and other combiners. For aids the combiner itself registered (which it can tell by the namespace prefix), it follows the protocol as before. For aids registered by other combiners, the combiner constructs all possible sets of the form $\{aid'_j | aid'_j \in \mathcal{M}_{rid}[j]\}$ where the aid' all belong to combiner C_y; each set corresponds to a *possible* rid' registered with C_y because the combiner doesn't know which aid's belong to a single registration, vs. which are from different registrations. It then engages in a private set intersection protocol [15] with C_y to determine if any of the aid' sets matches an actual registration; if so, the two combiners exchange the encrypted $user$ data (above) and notify the users of the match. If none of the aid' sets match, by the properties of the private set intersection protocol, neither combiner learns anything in the process.

6 Privacy Analysis

In this section we first define informally what we mean by location-privacy and our trust assumptions. We then model Koi in applied pi-calculus [35], a language for formally modeling distributed systems and their interactions, which then allows us to use the ProVerif [33] automated cryptographic protocol verifier tool to provide machine-generated proofs of Koi's privacy properties. We then discuss privacy concerns *external* to the Koi service, arising from poor application design, and offer coping strategies for applications.

6.1 Defining Privacy

Our privacy goals are based on Pfitzmann and Köhntopp's definition of anonymity [31] which is unlinkability of an *item of interest* (IOI) and some logical user identifier. Pfitzmann and Köhntopp consider anonymity in terms of an *anonymity set*, which is the set of users that share the given item of interest — the larger this set, the "better" the anonymity. In the related-work section (Section 11) we compare this definition of privacy to k-anonymity, l-diversity, and differential privacy.

In the Koi context, anonymity translates to unlinkability between an attribute ($attr$) and the registration ID (rid) (and by extension, the registering user U) the attribute is associated with. We assume that the individual attributes themselves are *not* sensitive; rather, it is when these attributes are *linked* to the user, or to the user's other attributes narrowing down and possibly identifying the user, that it becomes sensitive. Thus for location-privacy, as long as location data is present only in item/trigger attributes, it cannot be linked back to the user or his other attributes, thereby preventing the user from being tracked by third-parties.

6.2 Proving Privacy Properties

Before we model Koi, we recall basic ideas and concepts of applied pi-calculus needed for our analysis. A more comprehensive description (in the ProVerif context) is available in [9].

6.2.1 Applied Pi-Calculus Primer

Messages. Messages are obtained by applying *constructors* on *names*, *variables*, and other messages. Constructors are function symbols e.g., $\mathsf{REncrypt}(\dots)$. Names are symbols for atomic data e.g., Alice. Variables e.g., x, may be bound to names or messages. Messages are taken apart by *destructors*. Destructors are function symbols e.g., $\mathsf{RDecrypt}(\dots) \rightarrow \dots$.

Equations of the form $x = y$ can be used to establish the equivalence of two messages.

As illustration, we model probabilistic asymmetric encryption as the following pi-calculus relation:

$$\mathsf{RDecrypt}\left(\mathsf{REncrypt}(m, \mathsf{PubKey}(k), r), k\right) \quad \rightarrow \quad m \quad (1)$$

In Equation 1, the constructor for probabilistic encryption $\mathsf{REncrypt}(m, pk, r)$, the r component is fresh for every encryption thus preventing dictionary attacks. The destructor $\mathsf{RDecrypt}(\dots, k) \rightarrow m$ succeeds only if pk (from the constructor) and k are related in the form $pk = \mathsf{PubKey}(k)$. If so, the destructor discards r and yields the message m.

Channels. Channels are named sources/sinks for messages. A message m sent to channel c using $\mathtt{out}(c, m)$ can be received using $\mathtt{in}(c, x)$, where x will be bound to m. Channels model asynchronous out-of-order message exchange.

Processes. Processes are built from the grammar below. We omit discussion of the syntax (see [33] for details).

P,Q,R :=	processes
0	null process
P | Q	parallel composition
!P	replication
new a; P	name restriction
let N = D in P else Q	term evaluation
if N = M then P else Q	conditional
in(c, N); P	message input
out(c, N); P	message output

As illustration, we model a simplified version of the matcher fragment that processes message R1 and generates message R2 (from Table 3) as follows:

```
MatcherR1R2 :=
    in(net, m);
    let (user, eattrs) = m in
    let rid = RID(m) in
    out(net, (rid, eattrs))

Matcher := ... | !MatcherR1R2 | ...
```

The matcher is modeled above as the parallel combination of smaller processes that each process one type of message. The process MatcherR1R2 receives a message m from the channel net; deconstructs m to extract the $user$ data, and the encrypted attribute $eattrs$; picks a fresh registration ID for the message m using the RID constructor; and sends out the message $(rid, eattrs)$ back on the net channel. In our full Koi model, messages are tagged with a message type to avoid ambiguity in processing.

6.2.2 ProVerif Primer

ProVerif [33] can verify, among other security properties, the secrecy of information in protocols expressed in the

applied pi-calculus in a fully automated manner. Applied pi-calculus models distributed protocols as a collection of messages, parallel processes, and channels.

ProVerif is sound. ProVerif performs a brute-force exploration through the proof space. When it says that a security property is true, then it actually is so. The security proofs obtained through ProVerif are valid for an unbounded number of sessions of the protocol. However, ProVerif is not complete (i.e., false attacks can be found). When ProVerif finds an attack it provides a derivation tree that can be used to manually verify whether the attack found is false or not.

ProVerif is *not* always the preferred choice for verifying new cryptographic constructions because it assumes perfect cryptography, but is useful in verifying straightforward uses of existing cryptographic primitives, as is the case with Koi. This is because if an attack were to be found against a specific instantiation of a crypto primitive (e.g., RSA), its use in Koi is trivially replaced with another instantiation (e.g., ElGamal).

ProVerif's default attacker model models a Dolev-Yao attacker [11] that can overhear, intercept, and synthesize any message sent on any channel it has access to. This is too weak since the attacker doesn't have access to internal state of the participants. ProVerif allows for more powerful attacker models by allowing the user to specify what internal state the attacker can access or modify.

Proverif cannot model traffic analysis attacks. As mentioned, we assume there is enough natural (or generated) cover traffic for creating a mix [5] that mixes, delays and reorders messages to defeat traffic analysis and timing attacks.

6.2.3 Modeling Koi

We have modelled Koi in applied pi-calculus. Our model includes the extension that keeps matched pairs secret from the matcher. The model was constructed by one of the authors, and then manually and independently checked by the other two authors. One of the authors checking the model had experience with implementing Koi, and used this knowledge to verify correctness. The other author checking the model used only Section 5 of this paper to cross-verify the applied pi-calculus model and the protocol description here. As a final sanity-check, we introduced a number of bugs in the protocol (e.g., using deterministic encryption, not encrypting something that should be, etc.) and ensured that ProVerif found attacks that exploited the bugs introduced.

We highlight below some non-trivial aspects of our model, which we view as methodological contributions of our work.

Unlinkability. ProVerif does not natively model unlinkability. That is, if process u_1 were to send out message m_1, and process u_2 were to send out m_2, ProVerif tracks only that the attacker knows all four names (u_1, u_2, m_1, m_2) but does not track the *link* between u_1 and m_1 or u_2 and m_2.

We model unlinkability by creating a new constructor $\mathsf{LINK}(x, y)$ that explicitly models the linkability of x and y, and a new destructor INFER that allows the attacker to infer new links as follows.

$$\mathsf{LINK}(a, b) = \mathsf{LINK}(b, a) \quad (2)$$

$$\mathsf{INFER}\left(\mathsf{LINK}(x, a), \mathsf{LINK}(a, y)\right) = \mathsf{LINK}(x, y) \quad (3)$$

Equation 2 states that links are symmetric, i.e, if a is linked to b, then b is linked to a. Equation 3 states that links are transitive, i.e, if x is linked to a, and a is linked to y, then x is linked to y.

For each message sent on a channel we emit LINK messages that the attacker may use to draw inferences and new conclusions from. Thus unlinkability of x, y translates to ProVerif's notion of secrecy of $\mathsf{LINK}(x, y)$, which ProVerif is well-equipped to prove in an automated manner.

Adversary. The standard ProVerif attacker can observe only messages sent on public channels, and cannot observe internal state (e.g. if a process decrypts a message and then re-encrypts it before sending it out on a channel, the attacker would not have access to the decrypted message). To model scenarios where the process itself may be compromised by an attacker (since in our model the matcher and combiner could themselves be adversaries), we use the notion of a "spy" channel on which all internal internal process state is echoed (e.g., the decrypted message above) as well as give the spy write access to all channels the process has access to, in effect allowing the ProVerif attacker to completely supplant the compromised process. We model a spy channel for each Koi entity. By giving the ProVerif attacker access to the appropriate spy channel we can model an adversarial matcher, or combiner. By giving the attacker access to multiple spy channels we can model collusion between adversarial parties. By setting the ProVerif attacker to passive we model honest-but-curious adversaries (i.e., can receive but not send messages).

Datastore. ProVerif's constructors/destructors do not modify the environment and therefore cannot be used to model stateful operations such as a datastore. The only place to "store state" is in an (asynchronous) channel, where a message is "stored" between when it is sent and when it is received. We model storing into a datastore $\mathcal{M}[x] \leftarrow y$ as $\texttt{out}(\mathcal{M}, (x, y))$. We model performing a lookup on x as the sequence $\texttt{in}(\mathcal{M}, (= x, y))$; $\texttt{out}(\mathcal{M}, (x, y))$. The $(= x, y)$ syntax is ProVerif syntactic sugar that uses pattern-matching to deconstruct the input message into a tuple, check that the first element

equals x, and binds the variable y to the second element. Since `in` removes the message from the channel (the equivalent of deleting the mapping from the datastore each time a lookup is performed), after each lookup we add back the mapping into the datastore (by using `out`). Giving the attacker access to the appropriate datastore channel is an easy way to model an adversary that has read/write access to this internal state.

6.2.4 ProVerif Results

We model a configuration with one (honest) user U_1 — the intended victim of privacy violations — and an unbounded number of other users, a matcher, and a combiner. The honest user registers two attributes A_1 and A_2. We ensure at least one other user (U_2; honest or not) registers A_1 so he is matched with U_1.

We then ask ProVerif whether the attacker can conclude the following under various assumptions regarding which parties are being adversarial.

- P1. $\mathsf{LINK}(U_1, A_1)$, i.e., U_1 registered A_1
- P2. $\mathsf{LINK}(A_1, A_2)$, i.e., there exists some user that registered both A_1 and A_2
- P3. $\mathsf{LINK}(U_1, U_2)$, i.e., U_1 was matched with U_2.

Using ProVerif we were able to generate proofs for the following privacy properties:

Result 1: A honest-but-curious combiner cannot violate P1, P2, or P3. This is easy to see since the combiner is never sent a message that contains user ID, $user$ data, or attributes in an unencrypted form.

Result 2: A honest-but-curious matcher cannot violate P1, P2, or P3. As with all properties proved by Proverif, this holds for an unbounded number of messages in arbitrary order, subject to the mixing and crypto assumption mentioned earlier.

Result 3: A honest-but-curious combiner in collusion with a honest-but-curious matcher can violate P1, P2 and P3. As expected, if both the matcher and the combiner collude, the Koi service, as a whole, is analogous to existing cloud services that are organized as a monolithic database. Such a database can trivially violate all privacy properties above since it would have full knowledge. Thus as long as the combiner and matcher do not collude, our privacy goals P1, P2 and P3 are assured.

7 Koi Use-Cases

We examined 10 most popular location-based applications on the Android platform. The functionality provided by these applications can broadly be classified into 6 classes as listed in Table 4. We describe in pseudocode example applications we have built for two of these classes, and discuss briefly the other classes of apps.

Application	SN	TC	LS	LR	LA	ND
BrightKite	✓	✓	✓	✓		
Facebook Places	✓	✓	✓	✓		
Foursquare	✓	✓	✓	✓		
Google Latitude	✓	✓				
Google Maps						✓
Google Search			✓			
Google Ads					✓	
Gowalla	✓	✓	✓	✓		
Loopt	✓	✓	✓	✓	✓	
Routes	✓	✓				

Table 4: Location-based functionality provided by popular applications. SN: Social Networking, TC: Tagging Content, LS: Local Search, LR: Local Recommendations, LA: Location-based Advertising, ND: Navigation Directions.

7.1 Private Mobile Social Network

A mobile social network facilitates interaction between nearby friends and friends-of-friends. With the existing instantiations (e.g., Foursquare or Facebook Places), the need to perform matching on *friends* who are *nearby* violates privacy in two significant ways: first, the cloud service learns the identities of a user's friends; and second, the service also learns each user's location.

We have implemented a novel mobile social networking application on Koi, which enables nearby friends and friends-of-friends with common interests to get in touch, in a privacy-preserving manner. The user's profile is hidden from the OSN service, which also means that a user's profile can only be seen by others whom the user allows.

Here is how the application operates. The user creates his profile on his phone by running the application. He adds to his profile information including his interests, photos, etc. He also picks a random key that others must possess before the application will grant them access to data in his profile. He adds friends by simply exchanging profile keys with them. The application registers an item for the user (Algorithm 1, line 2– 8) with the user's push-notification mailbox as the item content, an encrypted attribute with the user's name prefixed by `me:`, an (auto-updating) attribute with the user's location, and an encrypted attribute for each of his friends' names prefixed by `friend:`. The app then registers a trigger for each friend (line 9–13). One attribute identifies the friend of interest (line 12), which matches the name registered by the friend's app on line 4, and another targeting 1 mile from the user's current (auto-updating) location (line 13), which matches the location registered by the friend's app on line 5. As the user and his friends move around, their phone-agents update the location attributes. When a friend moves within 1 mile of the user, the corresponding trigger fires with the friend's push-notification mailbox, which the app can use to exchange messages.

For matching friend of friends, the `me:` on line 11

```
1:  procedure MATCHFRIENDS(USER)
2:      I ← CREATEITEM(USER.PUSHMAILBOX, TTL)
3:      U ← ENCRYPT(me: + USER.NAME, USER.PROFILEKEY)
4:      ADDAPPATTR(I, U)
5:      ADDLOCATTR(I, loc:self, TRUE)
6:      for all F in USER.FRIENDS do
7:          P ← ENCRYPT(friend: + F.NAME, F.PROFILEKEY)
8:          ADDAPPATTR(I, P)
9:      for all F in USER.FRIENDS do
10:         T ← CREATETRIGGER(F.ONNEAR, TTL)
11:         V ← ENCRYPT(me: + F.NAME, F.PROFILEKEY)
12:         ADDAPPATTR(T, V)
13:         ADDLOCATTR(T, loc:self+1 mile, TRUE)
```

Algorithm 1: Private Mobile Social Network Application

```
1:  procedure ROUTE(DEST)
2:      I ← CREATEITEM(NULL)
3:      ADDLOCATTR(I, route.direction:⟨loc:self;DEST⟩, TRUE)
4:      N ← RANDOMNONCE()
5:      ADDAPPATTR(I, N)
6:      for all ACT in ACTIONS do
7:          T ← CREATETRIGGER(ACT.ANNOUNCE)
8:          ADDAPPATTR(T, N)
9:          ADDAPPATTR(T, ACT)
```

Algorithm 2: Turn-by-turn Directions Application

is changed to read friend:, which then matches the attribute for the common-friend registered by the friend-of-friend on line 8. Matches can be restricted based on shared-interests by associating attributes with the user's item, and querying for them in the triggers. Groups (e.g. photography-club) is supported in an identical manner with the group's name and group's secret key used instead of the friend-name and friend's key.

Note that neither the cloud, nor non-friends can track the user. As per Koi's location-privacy guarantees, the user's location cannot be tracker by the cloud service. A stranger cannot track the user's location since matching the user's item requires the stranger to construct a trigger with the encrypted username attribute (line 11), which only those that have the user's profile key can construct.

7.2 Turn-by-turn Directions

We created a proof-of-existence turn-by-turn directions application to demonstrate how Koi can add privacy to navigation applications that make heavy use of location information and cloud knowledge. Today any query for driving directions reveals to the cloud third-party (typically Google, Bing, or MapQuest) the user's current location and intended destination. Koi avoids both.

A key challenge in building a turn-by-turn direction application using Koi is that the Koi APIs do *not* provide a way to directly query for information such as the route to the destination. To get around this problem, we leverage that fact that the matcher sees attributes in clear-text and could employ application-specific logic while performing matching. So the end-to-end route is decomposed to waypoints, which are then be conveyed to the application one-by-one, via triggers registered with Koi.

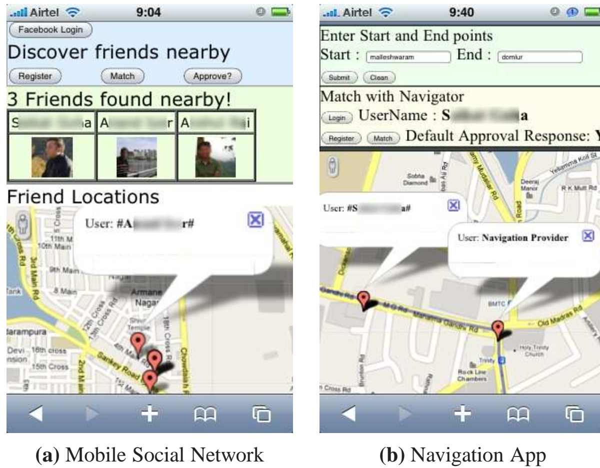
In particular, the application registers an item with an attribute that indicates the user's current location and the destination (Algorithm 2, line 3). This item implicitly corresponds to the next waypoint on the end-to-end route. The application also anticipates and registers one trigger for each possible directive corresponding to the next waypoint (e.g., "go straight", "turn left", "turn right", etc.; lines 2.6–2.9). Based on the current location and destination contained in the attribute, the application-specific matcher logic looks up the route and replaces the item's attribute with the actual directive corresponding to the next (i.e., first) waypoint. (Such a replacement is akin to the matcher applying spelling correction or using synonym information as part of its application-specific matching logic.) Say the directive at the first waypoint is "turn left"; the trigger corresponding to "turn left" fires and so the application learns that the next directive to present to the user is "turn left". As the user travels along the route, the phone-agent auto-updates the user's location, which causes the matcher to compute the direction to the next waypoint, which fires the trigger corresponding to the next directive, and so on. In this manner, the turn-by-turn directions are conveyed to the user.

7.3 Local Search, Tagging, Advertising

The four remaining classes of apps identified in Table 4, i.e., location-based content tagging, local search and recommendations, and location-based advertising, essentially all boil down to registering an item with the appropriate content and attaching location and other attributes to it. The item can be found by others using the appropriate triggers.

8 Implementation

We have implemented the Koi platform and the two proof-of-concept applications mentioned. The cloud-component is written in 1040 lines of C# code. The phone-based agent is available as a 230 line Javascript library that can be used by HTML5 applications on all modern smartphones, and as a C# library that can be used by native applications on the WP7 platform. The C# agent can leverage the platform's native push-notification service to receive trigger updates, while the Javascript version resorts to polling. The cloud service exposes an open REST-based API (over HTTP), allowing agents to be written for other platform and programming languages.



(a) Mobile Social Network **(b)** Navigation App

Figure 2: Two applications implemented on the Koi platform

In addition to the core Koi API, our agent exposes a GUI API since location-based apps naturally use map widgets that require lat-long information. Our GUI API is a bare-boned API to allow the app to put push-pins at the user's current location (specified symbolically as loc:self), and overlay simple geometric shapes. Figures 2a and 2b show screenshots of early prototypes of our proof-of-concept applications. These applications consist of 50–60 lines of Javascript code, and required around 6 person-hours each to create.

9 Experimental Evaluation

We view a significant portion of the contribution of this paper as being architectural, in terms of proposing a higher-level abstraction (location-based triggers) as an alternative to the lat-long location API that is commonly used. As such, the evaluation of this architectural contribution rests on showing the ease of building applications on the Koi platform, which we have done to a small extent in Section 8. In this section we focus on the performance of the Koi cloud service, which we evaluate experimentally using both real-world traces, and micro-benchmarks. In order to exclude evaluation artifacts arising from load on the Azure platform, all experiments are run on one core of a 3 GHz dual-core machine with 4 GB of memory; the matcher and combiner share the one core while the second core is used by the benchmarking process. We use the loopback device for all communications to test the raw performance of our implementation independent of network bandwidth and latency.

9.1 Macro-Benchmark: Mobile Ads

We benchmark a location-based advertising application where business-owners register advertisements with location attributes, and users are matched to ads for businesses near them (i.e., within a hundred meters). We use a 2 GB real-world mobility trace of 264K users from around the world collected over a period of 1 year. The trace contains 22 million timestamped latitude-longitude updates across all users. Since we lack advertiser information, we simulate 10K businesses near popular locations visited by many users.

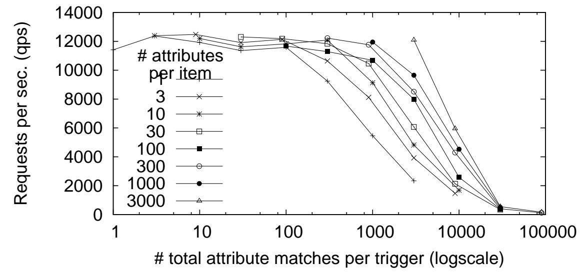


Figure 3: User matching performance

Processing 12 months worth of trace data through our implementation takes around 2 hours and 50 minutes. The system generated 2.2 billion notifications, all within 30ms of the location update that triggered the notification. The peak memory consumption is around 2.5 GB over the entire run. Even considering the small size of our trace we find our implementation running on a single-core can easily handle a mobile advertising application while leaving plenty of headroom for more demanding applications.

That said, macro-benchmarks, even for a handful of applications, are by nature inadequate for evaluating performance limits. We turn to micro-benchmarks to stress our implementation to its limits.

9.2 Micro-Benchmarks

Matching. We focus first on the matching throughput of our Koi implementation since matching is the primary mode of use. Our implementation combines messages R3 and M1 (Table 3) into a single message to amortize communication costs. Figure 3 plots the number of matching queries processed successfully per second (qps) as a function of how many attributes matched. That is, if the trigger matched 100 items each on 10 attributes, or if the trigger matched 1000 items on 1 attribute each, the x-axis value for both datapoints is 1000. The y-axis value is the mean query throughput for processing 100K requests (i.e., each datapoint represents an experiment lasting between 10s to a few minutes); standard-deviation error bars are negligible.

As is evident from Figure 3, end-to-end performance saturates to its peak as long as the average number of matching attributes (per request) is below 100. The bot-

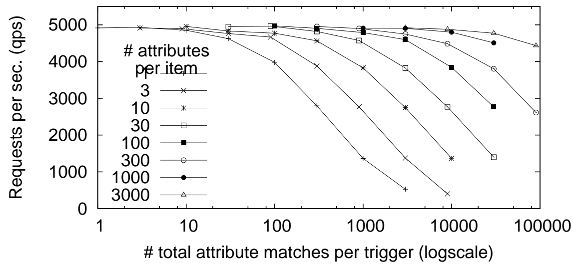


Figure 4: Attribute registration (and indexing) performance

tleneck here is the connection throughput of the HTTP library (which saturates are 12K qps.) Between 100 and 10K matching attributes, performance depends on the number of items matched (with fewer items resulting in higher performance); this is because our in-memory index layout is optimized for triggers matching a few items on many attributes. Beyond 50K attributes per match performance plummets as we run into memory issues. To put this in perspective, for the macro-benchmark above, this quantity is in the low tens, representing several orders of magnitude of headroom.

Registration. We focus next on registration throughput which includes both cryptographic operations, and building the various tables and indexes (e.g., `R2U`, `R2A`, `A2R` etc.) Figure 4 plots the number of registration requests processed per second (qps) as a function of the average number of matching attributes per trigger registered (which governs our in-memory index build process). As above, datapoints represent mean qps for processing 100K requests.

The figure is qualitatively similar to Figure 3 in that performance peaks for workloads similar to the advertising benchmark. The peak performance in Figure 4 is half that of Figure 3 since twice the number of HTTP requests are involved and the bottleneck is the HTTP connection throughput. As before, performance degrades as our index runs into memory issues, though more gradually due to different access patterns.

Combining. Our implementation can process trigger notifications at the rate of 12K qps (same peak as Figure 3). This throughput is independent of the number of items and attributes.

Overall we find our prototype implementation performs well enough even for real-time apps, and moves the performance bottleneck outside of Koi and into the communication layer.

10 Discussion

We discuss briefly privacy-related issues beyond Koi.

The first issue pertains to malicious applications registering a large number of finely-spaced triggers or triggers at sensitive locations, and reverse-engineering a victim user's location based on triggers matched. A weak defense against this attack would be for Koi to rate-limit the number of trigger registrations from an application (either per-device, or across all devices). This would force the attacker to Sybil himself to remain below threshold. Distributing apps through mobile marketplaces today requires developers to purchase a developer key. If the Sybils were to re-use this key, or re-use the credit-card used to purchase this key, they would be easily linked. Rate-limiting combined with an economic burden could serve to proactively mitigate against malicious applications.

The second issue pertains to collusion between the matcher and combiner. We mitigate the possibility from both the matcher and combiner side. On the combiner side, we allow for privacy-advocacy firms (e.g., EFF and ACLU), anti-virus companies (e.g., McAffee), certificate agencies (e.g., Verisign), non-profits (e.g., Mozilla), and other such outfits to run the combiner. Since the existence of these outfits is entirely dependent on public trust, it creates a strong disincentives to collusion — if they collude and anyone finds out (e.g., during an audit, or through whistle-blowers), the company would lose all credibility and be forced to shut down (as happed with the DigiNotar certificate agency recently). On the matcher side, by allowing the user to pick the combiner (of which there may be hundreds) we make it infeasible for the matcher to collude with any significant fraction of these combiners before it becomes public. The matcher would then be guilty of intentionally and deliberately circumventing privacy technology, which they could be legally liable for, creating a strong disincentive.

The third concern pertains to incentives for apps to adopt Koi. At a high level, it is up to the platform to drive adoption. Incentives may include positive reinforcement (e.g., higher placement in the mobile marketplace for Koi-enabled apps), negative feedback (e.g., more frequent nagging popups for apps using legacy location APIs instead of the Koi API), or strong enforcement (e.g., blocking the legacy location API for free applications). Overall, the platform can use a combination of these and other incentives to drive adoption.

11 Related Work

Existing Location APIs in Production Use. As mentioned, Apple's iOS Core Location [1], Android's Location Manager, and the Windows Phone 7 Location Service all expose only low-level lat-long information to apps which, when carelessly passed by apps to third-party code (e.g., Google's AdMob or Apple's iAds widget), poses a privacy risk as reported in TaintDroid [14] where the third-party service can track the user across

multiple applications.

Location APIs in Prior Research. There is a rich body of research on location APIs. For programming ease, Brown et al. [4] propose the notion of a stick-e note that applications can register, which is triggered whenever the user's present context (e.g., location) matches that specified in the note. Other APIs have focused on fusing location information from multiple sensors to provide applications a unified view [28]. The Location Stack [21] proposes a 7-layer model that combines sensing and fusion with the inference of user activity and intentions.

While the above work focuses on important issues such as programming ease, sensor fusion, and activity inference, we view these as complementary to our focus on privacy in Koi.

Location Privacy. There is a rich body of work on location privacy. A survey some of this work appears in [17] and some challenges are discussed in [2, 36, 38].

Myles et al. [30] propose a rich policy API that includes support for symbolic locations as well as geographic locations, and for triggered callbacks. However, their proposal depends on a trusted external entity to enforce privacy constraints. In contrast, PlaceLab [24] takes a decentralized approach, wherein end devices acquire location information locally and allows users control over whether and how to share this information. As an extreme form of decentralization, Anonysense [8] ensures that the mobile nodes in a participatory sensing context do *not* share their location at all. Instead, the nodes download all tasks registered with the system and then determine locally which ones match their location and should be executed. Besides imposing a bandwidth cost, such an approach is not suitable for location-based applications that depend on the location of other users, e.g., mobile social networking as in Four Square.

There is a large body of work on techniques to cloaking user location while still enabling location-based services. Gruteser et al. [20] propose a centralized location broker that performs temporal and spatial cloaking of user location information, keeping in mind such factors as the density of users in a given area. In the context of a traffic monitoring application, the authors in [22] propose having virtual trip lines to trigger the reporting of location updates by mobile nodes, say based on the privacy sensitivity of locations. All of these techniques assume a trusted intermediary.

The research that is perhaps closest to ours in spirit is the recent work of Jaiswal and Nandi [26]. As in our work, they steer away from having a trusted intermediary by having multiple distinct entities, each holding a part of the sensitive information, work in unison to provide location-based services. However, location updates can still be linked, which opens up the possibility of attacks.

In comparison to the above work, Koi is designed explicitly to ensure privacy even with respect to the cloud service, which is not trusted. Also, unlike prior work, we do not seek to anonymize users or hide their location information. Rather, we consider the *linkage* between a user and their location as the privacy-sensitive information and focus on protecting this from the cloud service.

Notions of Privacy. Several notions of privacy have been proposed in the databases literature. k-anonymity [40] ensures that each output row is identical to at least $k-1$ other rows. However, k-anonymity is susceptible to attacks that exploit the lack of diversity in the value of sensitive attributes amongst the k rows. To address this, l-diversity [29] ensures that the output contains l well-represented values for each sensitive attribute. k-anonymity and l-diversity are complementary to Koi in that the combiner could suppress matches when the anonymity set size drops below k (i.e., fewer than k items match); the downside, however, is reduced functionality since matches cannot be too specific (e.g., cannot match nearby friend if only one friend is nearby).

Differential privacy [13] adds noise to ensure the output (typically aggregate queries) is independent of the presence or absence of a particular record. The differential privacy model is fundamentally a bad fit for Koi since the output of a matching service cannot be independent of whether the matched item is present or absent.

Privacy in Publish-Subscribe Systems. A publish-subscribe (pub-sub) system comprises publishers who post events, subscribers who register filters corresponding to events of interest to them, and a broker who *matches* events from publishers to the filters registered by the subscribers. Traditionally, the broker is assumed to be trusted, however, there has been recent work on the issue of confidentiality. Rich matching, while supporting confidentiality, is particularly hard for existing systems. [27] encrypts sensitive fields, but matching is restricted to the unencrypted fields. [37] employs a commutative encryption scheme [32] for confidentiality, however, matching is restricted to equality on a single keyword. [34] encrypts events and filters and then uses technique for search on encrypted data [12] to match these, but matching is limited to equality matching, keyword matching, and some limited numeric range matching. PSGuard [39] encrypt using hierarchical key derivation algorithms [42]; such key spaces are constructed for different types of matching, including topic or keyword matching, numeric attribute based matching, and prefix or suffix matching. Finally, [25] uses attribute-based encryption (CP-ABE) [3] to encrypt the events and key-policy attribute-based encryption (KP-ABE) [18] to encrypt the filters, and combine KP-ABE with searchable data encryption [12] to enable the broker to perform matching that can be expressed as conjunctions and disjunctions of equalities, inequalities and negations. Koi

supports rich matching that goes far beyond the functionality of prior schemes by enabling matching based on application-specific semantics (e.g., matching "barber" and "hairdresser") in addition to equality-matching, numeric range-matching, regex-matching, and conjunctions and disjunctions over them all.

Private Information Retrieval (PIR) and Secure Multiparty Computation (SMC). PIR allows a user to retrieve an item from a database server without revealing which item they are retrieving. Single-server PIR schemes necessarily have $\Omega(n)$ computational cost in n — the size of the database, and $\mathcal{O}(log^2 n)$ communication cost [7, 16], which is prohibitively large. The best known multi-server scheme still has a communication cost of $\mathcal{O}(n^{\frac{1}{\log\log n}})$ [43], which is still impractical in our setting. SMC is known to be harder than PIR [10]. Koi's privacy model differ from PIR and SMC, thus allowing for a much more efficient protocol that has constant communication and computational overhead.

12 Summary

We have presented Koi, a platform for supporting location-based applications in a privacy-preserving manner. Overall we find there is much to be optimistic about in raising the level of abstraction from a get lat-long API to a matching-based API. We have presented a privacy-preserving matching service, verified the privacy properties using the ProVerif theorem-prover, shown how a wide range of applications can be built using the API including implementing two concrete applications, publicly deployed the service and released client libraries, and have demonstrated the service to perform and scale well through macro- and micro-benchmarks.

References

[1] Apple iOS 4 Core Location API. http://tinyurl.com/49fyamf.

[2] Location Interoperability Forum (LIF) Privacy Guidelines. LIF TR 101, Sep 2002, http://tinyurl.com/4csy4rx.

[3] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-Policy Attribute-Based Encryption. In *IEEE Symp. on Security and Privacy*, 2007.

[4] P. Brown, J. Bovey, and X. Chen. Context-Aware Applications: From the Laboratory to the Markeyplace. *IEEE Personal Communications*, October 1997.

[5] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2), 1981.

[6] J. Chester, S. Grant, J. Kelsey, J. Simpson, L. Tien, M. Ngo, B. Givens, E. Hendricks, A. Fazlullah, and P. Dixon. Letter to the House Committee on Energy and Commerce. `http://tinyurl.com/y85h98g`, September 2009.

[7] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private Information Retrieval. October 1995.

[8] C. Cornelius et al. AnonySense: Privacy-Aware People-Centric Sensing. In *ACM MobiSys*, 2008.

[9] K. S. Cortier V. *Formal Models and Techniques for Analyzing Security Protocols*. IOS Press, 2010.

[10] R. Cramer, I. Damgård, and S. Dziembowski. On the complexity of verifiable secret sharing and multiparty computation. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, 2000.

[11] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198 – 208, March 1983.

[12] C. Dong, G. Russello, and N. Dulay. Shared and Searchable Encrypted Data for Untrusted Servers. In *DBSec*, 2008.

[13] C. Dwork. Differential Privacy. In *ICALP*, 2006.

[14] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[15] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient Private Matching and Set Intersection. In *EUROCRYPT*, 2004.

[16] C. Gentry and Z. Ramzan. Computational Private Information Retrieval with Logarithmic Total Communications. July 2005.

[17] W. W. Gorlach, A. Terpstra, and A. Heinemann. Survey on Location Privacy in Pervasive Computing. In *SPPC*, 2004.

[18] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based Encryption for Fine-grained Access Control of Encrypted Data. In *ACM CCS*, 2006.

[19] M. Gruteser and D. Grunwald. Anonymous Usage of Location-Based Services through Spatial and Temporal Cloaking. In *Proceedings of MobiSys'03*.

[20] M. Gruteser and D. Grunwald. Anonymous Usage of Location-Based Services through Spatial and Temporal Cloaking. In *MobiSys*, 2008.

[21] J. Hightower, B. Brumitt, and G. Borriello. The Location Stack: A Layered Model for Location in Ubiquitous Computing. In *IEEE WMCSA*, 2002.

[22] B. Hoh, M. Gruteser, R. Herring, J. Ban, D. Work, J.-C. Herrera, A. M. Bayen, M. Annavaram, and Q. Jacobson. Virtual trip lines for distributed privacy-preserving traffic monitoring. In *MobiSys*, 2008.

[23] B. Hoh, M. Gruteser, H. Xiong, and A. Alrabady. Preserving Privacy in GPS Traces via Density-Aware Path Cloaking. In *Proceedings of CCS '07*.

[24] J. I. Hong, G. Boriello, J. A. Landay, D. W. Mcdonald, B. N. Schilit, and J. D. Tygar. Privacy and Security in the Location-enhanced World Wide Web. In *Ubicomp*, 2003.

[25] M. Ion, G. Russello, and B. Crispo. Providing Confidentiality in Content-based Publish/Subscribe Systems. In *SECRYPT*, 2010.

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

[27] H. Khurana. Scalable Security and Accounting Services for Content-based Publish/Subscribe Systems. In *ACM Symposium on Applied Computing*, 2005.

[28] U. Leonhardt and J. Magee. Multi-sensor location tracking. In *Mobicom*, 1998.

[29] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1), March 2007.

[30] G. Myles, A. Friday, and N. Davies. Preserving Privacy in Environments with Location-Based Applications. *IEEE Pervasive Computing*, 2(1), January 2003.

[31] A. Pfitzmann and M. Köhntopp. Anonymity, unobservability, and pseudeonymity — a proposal for terminology. In *International workshop on Designing privacy enhancing technologies: design issues in anonymity and unobservability*, pages 1–9, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[32] S. Pohlig and M. Hellman. An Improved Algorithm for Computing Logarithms over gf(p) and its Cryptographic Significance. *IEEE Transactions on Information Theory*, 24(1):106–110, January 1978.

[33] Proverif. http://www.proverif.ens.fr/.

[34] C. Raiciu and D. S. Rosenblum. Enabling Confidentiality in Content-Based Publish/Subscribe Infrastructures. In *IEEE SecureComm*, 2006.

[35] M. Ryan and B. Smyth. *Formal Models and Techniques for Analyzing Security Protocols*, chapter Applied pi calculus. IOS Press, 2010.

[36] M. Scipioni and M. Langheinrich. I'm Here! Privacy Challenges in Mobile Location Sharing. In *IWSSI/SPMU*, May 2010.

[37] A. Shikfa, M. Onen, and R. Molva. Privacy-Preserving Content-Based Publish/Subscribe Networks. In *IFIP SEC*, 2009.

[38] M. Spreitzer and M. Theimer. Providing Location Information in a Ubiquitous Computing Environment. In *SOSP*, 1993.

[39] M. Srivatsa and L. Liu. Secure Event Dissemination in Publish-Subscribe Networks. In *ICDCS*, 2007.

[40] L. Sweeney. k-Anonymity: A Model for Protecting Privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):557–570, 2002.

[41] S. A. Weis. *New foundations for efficient authentication, commutative cryptography, and private disjointness testing*. PhD thesis, Cambridge, MA, USA, 2006. AAI0810110.

[42] C. K. Wong, M. G. Gouda, and S. S. Lam. Secure Group Communications using Key Graphs. *IEEE/ACM Transactions on Networking (TON)*, 8(1):16–30, February 2003.

[43] S. Yekhanin. New Locally Decodable Codes and Private Information Retrieval Schemes . *Electronic Colloquium on Computational Complexity*, 2006(127), 2006.