Insider: Towards Breaking Down Mobile App Silos

Vageesh Chandramouli, Abhijnan Chakraborty [*], Vishnu Navda, Saikat Guha,
Venkata Padmanabhan, Ramachandran Ramjee

Microsoft Research India

Abstract

User data is siloed in mobile apps today. Where one app may hold the user's flight booking, another app the user's cab reservation, with little data sharing between the two, resulting in a fragmented user experience. For instance, if a user, who has just booked a flight using one app, wishes to pre-book a cab from the airport, they would have to do so by manually re-entering the data from the flight app (e.g., location, date, time) into the cab app.

To enable the user to retake control of their in-app data, the Insider platform extracts structured user data from the presentation layer of arbitrary apps, without any modifications to the app code or binary, and makes it easy for such data to be shared across apps, when the user so desires. At its core, ordinary users create and share models for the apps they care about; the app model relates the data from the presentation layer of the app to attributes in a *task template*, which corresponds to the specific tasks users perform in the app (e.g., booking a flight). At runtime, Insider combines the app model with the raw stream from the presentation layer to produce structured and semantically-meaningful information. Insider then publishes this information through a set of APIs, enabling creation of novel apps that could not have been built previously. For example, we prototype the automatic population of information in a cab app after a flight booking as well as three other such novel apps that rely on cross-app data sharing. Finally, we report on a detailed evaluation of running Insider on 150 apps across 11 categories. We show that we are able to successfully extract 83% of the key attributes (e.g., UI widget corresponding to source airport) from these apps. Further, we also track app version changes during a 6 month period from Sep 2014 to Mar 2015 and find that even though 40% of apps were updated, only 6% of apps required rebuilding of the app model.

1. Introduction

All data generated by users today through their smartphone apps remain siloed in the app. This includes data generated both explicitly, such as the user actually booking a flight through the Expedia app, as well as implicitly, such as browsing the reviews of a few restaurants on Yelp. APIs exposed by these data silos are designed to protect and extend the silo's control over the *user's data*. For instance, Expedia offers APIs for other apps to invoke new flight bookings, but does not provide an API for other apps to take action on the user booking. Arguably, information about a flight the user paid for is data the user owns. Nevertheless, data silos make it hard for users to take control of their own data and share it with innovative new apps. For instance, the user today must manually re-enter his flight information (e.g., arrival date, time, city) into a cab app to make a cab reservation.

When it makes business sense to do so, two sufficiently large app silos may negotiate deals where one app refers its users to the other app in exchange for monetary compensation (called traffic acquisition costs), e.g., if Expedia were to add a Lyft cab button on the booking page that launches the Lyft app with pickup details pre-populated. Such deals limit user choice, e.g., the user may prefer that their flight information be shared with perhaps a small local cab service in their city, who is unlikely to be in a position to negotiate favorable terms with the data silo (i.e., Expedia in this example). Overall, the current data sharing model between siloed apps has the effect of removing choice for the user, whose very data fuels the in-app data economy in the first place.

Our goal, in general, is to create a platform that enables users to take control of their data contained within apps and share it in an effortless manner with other (explicitly authorized) apps. We believe that this approach not only unleashes user choice but can also light up compelling new experiences that would not be possible otherwise.

This paper focuses on three key design challenges we encountered in creating the Insider user data platform: 1) how user data can be extracted with no effort on the developer's or user's part, 2) what the data abstraction should be to enable meaningful sharing of user data across apps, and 3) devising an approach to inferring app models for a large number of apps and maintaining these models automatically as apps are updated. We discuss each of these challenges next.

Extraction. A naïve approach to extracting user data from apps is to ask app developers to instrument their app and explicitly tag user data. However, as discussed earlier, siloed apps have little business reason to proactively part with the

[*] Presently at IIT Kharagpur

user's data. Instead we believe that the user must take control of their data. While there are various layers where app data can be extracted (Section 2), we tap into the presentation layer of the mobile to extract the raw user data since the siloed app already contains UI logic to present the most relevant data in a structured way to the user. We develop Insider as a third-party app/service, which takes advantage of the limited accessibility system service in Android, and has the benefit of enabling easier deployment.

Abstraction. Representing the data as a document — a loosely structured bag of words — we quickly realized is only suited to the web, where the primary goal is to retrieve information. Apps are often used to perform tasks (e.g., call a cab), which inherently requires structured information e.g., the pick-up and drop-off locations are similar in a bag-of-words abstraction but have crucial semantic differences.

Our data abstraction is that of a *task*. A task contains an *action* (e.g., booked flight, viewed review) and a number of action-specific *attributes*, e.g., origin, destination, flight attributes for a booked flight action. Apps subscribe to tasks that they are interested in (e.g., a cab app might subscribe to a flight booking task). When the user books a flight in *any* flight app, the user is notified of the option to share the flight booking data with a list of apps that have subscribed to this data. If the user explicitly gives permission to share his/her flight booking data with a chosen cab app, the cab app is then passed this information automatically, eliminating the tedium of manual data re-entry.

Inference. While Insider is oblivious to the semantics of the attributes in the task template, internally representing them as generic key-value collections, Insider must nevertheless provide a mechanism to label data extracted from the presentation layer with semantically meaningful labels in the structured task template. An approach we rejected early on is the use of natural language processing techniques for automated extraction of app structure because of the enormous diversity in UI modalities in task completion apps (Section 2).

Instead, we rely on a one-time guided human execution where the user is asked to perform a task using the app. Insider logs the presentation layer changes in the app from the execution, infers the mapping from task attributes to UI widgets and creates an *app model*, which can then be shared with other interested users. Insider uses the generated app model at runtime to extract user data from the app, automatically label it with (opaque) semantic labels from the task template, and allow seamless sharing of the extracted information with other apps. Thus, the Insider *app model* is similar to the IFTTT (If This Then That) [8] macros used for automating tasks on phones, with the added ability to extract and pass structured in-app data between apps and to do so locally on the client (unlike IFTTT's cloud-based approach).

One concern with this approach is the stability of app model to app updates. To address this, we perform a combination of static model checking and position/value-based relearning to improve the longevity of the app model. For example, we tracked the version changes of 150 apps from September 2014 to March 2015 and found that 60 apps were updated but only 9 app models need to be relearnt through manual effort.

A key aspect of Insider is the creation of app models by ordinary users, with no programming skills. To evaluate the efficacy of this approach, we recruit crowd workers from Amazon Mechanical turk as our "users". We design and implement a cloud-based app task execution platform that runs an Insider-instrumented mobile OS in a VM in the cloud. Crowd workers connect to it using a web browser on their unmodified client device (e.g., phone, tablet or PC). We evaluate the extraction, abstraction, and learning mechanisms on 150 apps across 11 categories by leveraging 686 crowd workers and 1028 task executions in the cloud and find that we were able to successfully extract over 83% of semantically meaningful attributes relevant to these apps.

Finally, we prototype four novel cross-app data sharing apps. We automatically extract information from various categories of apps and prototype: 1) a history app that collates and shows a timeline of user activities across task categories such as shopping, movies, flight, etc., 2) a travel assistant app that automatically schedules price alerts for flights searched for (but not booked) and deletes the alert when the booking is made, 3) a Limo Yo app that automatically launches the user's preferred limo app with pick-up time based on flight departure (or arrival) and the drop-off (or pick-up) location set to the appropriate airport when a flight is booked and 4) a smart messaging app that presents a unified view of users and messages combined from different messaging apps, a demo of which is available on YouTube [12].

Overall, this paper makes three contributions. First, it argues for de-siloing of user's data from apps, and creating a platform that enables users to take control of in-app data and share it effortlessly with other apps. Second, it proposes a data abstraction centered around tasks, and presents the design and implementation of an architecture that achieves this goal without requiring app modifications or user effort beyond the one-time model creation. And finally, third, it reports on our pilot deployment and experimental validation across a wide range of apps and users.

2. Design Challenges

In this section, we discuss the key challenges we faced in designing Insider.

The first challenge is how to extract in-app data. As shown in Figure 1, there are a variety of locations where app data resides. For example, one could tap app data at the application layer by modifying the application binary in a way similar to how AppInsight [20] modifies the app binary for profiling performance. While developer involvement may potentially be avoided in the binary instrumentation process for extracting in-app data, one would still need developer

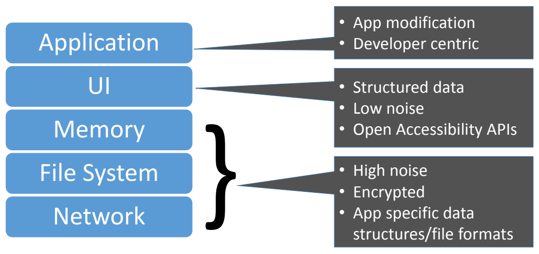


Figure 1: **Options for app data Extraction**

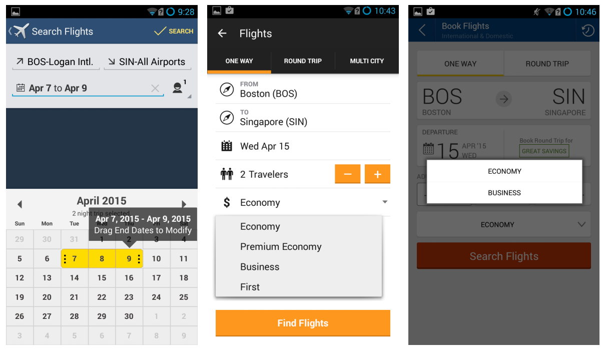


Figure 2: **Three flight booking apps: Expedia, Kayak and MakeMyTrip**

consent for distributing such instrumented app binaries. As discussed earlier, developers are not incentivized to allow such modifications.

App data can also be inferred from network traffic. For example, Narus [21] uses app's network communication traffic to learn about app usage. Apart from challenges with network traffic encryption, the large amount of noise (e.g., extraneous communication sessions to advertising systems) and app-specific encoding of information makes extracting even high-level app information (e.g., the identity of the app) from network traffic challenging [22], leave alone fine-grained semantic information à la Insider. Tapping the memory or file system also suffers from similar issues.

Instead, we choose to extract data from the presentation layer. Given the limited screen real-estate, developer typically choose to only display the most relevant information in the UI, significantly reducing noise in the data. Further, information is presented in a structured way to make it easy for users to consume it and execute tasks. Finally, the presence of accessibility APIs in the OSes means that the presentation layer information can be extracted without requiring any OS or app modifications, thereby facilitating deployment.

The second challenge derives from our need to tap into the presentation layer. While it simplifies deployment, the sheer diversity in how app developers choose to present information through the UI makes even automated exploration of apps using a monkey extremely challenging [18], leave alone automated inference of semantic information.

For example, we show screenshots of three popular flight booking apps, Expedia, Kayak and MakeMyTrip in Figure 2. Notice that the Expedia app does not identify source or destination explicitly (except through ascending and descending arrow icons), has no class of travel or one-way option, and requests the departure and arrival dates to be input by dragging the highlighted dates at the bottom of the screen. On the other hand, Kayak identifies the "from" and "to" airports, has separate options for one-way, round-trip and multi-city, and multiple options for class of travel. Finally, MakeMyTrip does not identify source or destination, allows one-way and round-trip but no multi-city travel and has only two options for class of travel. Such significant variation across apps makes it extremely challenging to infer semantics or execute tasks programmatically. Hence, we settled on having the user perform a one-time guided execution of the app for creating the app model.

The next challenge is in designing an approach that is simple enough that everyday app users, with no programming expertise, can build accurate app models with ease. We experimented with two different approaches for the user guided one-time execution. The first approach was to ask the user to point and label the semantically meaningful parts of the UI. The second approach was a guided task execution-based model where the user is asked to execute a set of simple primary and secondary task instances for the app. Surprisingly, even though the former approach is intuitive, given the various ambiguities in UI interactions, we found that the latter approach had a much higher efficacy in extracting an accurate app model, based on our evaluation with crowd workers from Amazon Mechanical Turk (Section 7).

The final challenge arises from our desire to reduce user effort in building app models. Having one or a small number of users build the model for an app and then share it with other users of the app helps amortize the effort. However, updates to an app can render the model obsolete. Requiring the user to perform a guided execution afresh would be prohibitive since apps could be updated frequently. Instead, we leverage static checking, and position and value-based matching to automatically update app models in most of the cases when apps are updated (Section 4.2).

3. Insider Overview

We now present a high-level architectural overview of the Insider platform. We will refer to Figure 3 in our description.

3.1 Extraction

Insider taps in-app information at the UI layer as it provides a rich view of the user's activities inside any app. It turns out that most modern OS platforms support accessibility APIs to enable features such as audio narration of on-screen con-

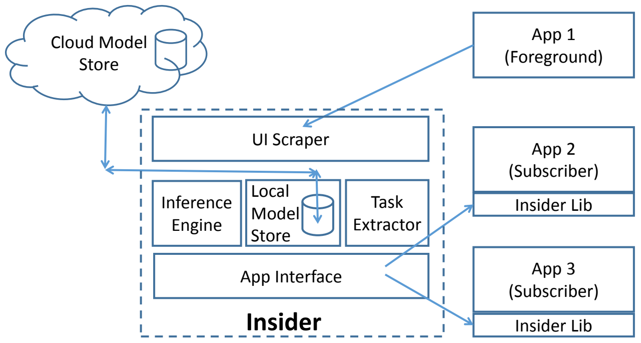


Figure 3: **Insider Overview**

tent. We leverage these APIs to tap on-screen content from unmodified 3rd-party apps. Specifically, using accessibility APIs on (unmodified) Android (version 4.2), we record the UI widgets displayed on the screen, along with their various properties, including value, position, activity ID (i.e., app page ID), whether the user interacted with the widget (e.g., via touch, swipe, keyboard entry) and the timestamp of the interaction. This yields the *syntactic* structure of the on-screen app content. The *UI Scraper* module of Insider shown in Figure 3 runs as a background service and scrapes content from the foreground apps (App1 in the example).

3.2 Task Abstraction and Inference

Recall our earlier observation that unlike webpages, task-oriented apps tend to be organized to help users complete tasks. Each task leads to a page in the app that is organized akin to a web form, comprising key-value pairs.

Insider uses training data to learn (opaque) semantic labels for these key-value pairs. Insider builds app-specific models (such as the one shown in Figure 4) that are used to assign semantically meaningful labels to appropriate key-value pairs. As mentioned, the training data is acquired by asking a small subset of the users of an app (our evaluation shows that 3 users suffice) to perform a guided task execution within the app.

The *Inference Engine* analyzes the logs from *UI Scraper* to build the app model, which is then stored locally in the *Local Model Store*. The analysis is completely carried out on the client-side, and only the generated models are uploaded to a central repository — the *Cloud Model Store* — from where other users can download models for apps that are installed on their devices. For less populat apps that do not have an existing model in the store (or whenever a model gets obsoleted in a way that cannot be fixed through our automated process (Section 4.2)), the training process needs to be carried out (again).

The training process involves the user executing a sequence of tasks in the app. An example of a (primary) task template is: “Book a round-trip flight between any two cities on any date”. A subsequent (secondary) task in the sequence may be: “Book a round-trip flight between the same two cities as before departing on the same day as before, but change the `return date`.” The attributes shown in teletype font are (opaque) semantic labels that will be associated with the UI widgets in the app. In this example, Insider would diff the values extracted from the presentation layer across the primary and secondary tasks.

Ideally, there would be a change in the value of a single UI widget, in which case that widget would be labeled as “`return date`”. In practice, however, the values corresponding to multiple UI widgets could change even when only a single task attribute is changed. Section 4 details the Insider mechanism for robustly inferring the attribute-UI widget mapping even in such cases.

3.3 Insider Apps

Insider platform is designed to extract siloed user data from within any foreground app and then give the user the option of making it available to other 3rd party apps. To ensure that the user is in full control of his data, our design is based on two key principles – transparency and access control. At any point in time, the user can look at which apps are subscribing to notifications corresponding to which task categories, and can update the permissions to (dis)allow such subscriptions. We have developed a library that the developer can use within his app to subscribe to specific task categories. When such an app is launched for the first time, the user is notified of the task categories that the app wishes to subscribe to, and is asked either allow or deny permission. (This is akin to how access control for hardware sensors is performed today.)

When a particular foreground app (App1 in the illustration) is running, the Task Extractor processes UI logs from the UI Scraper and extracts semantically meaningful structured data using app-specific models available either locally or downloaded from the cloud. Then, apps that subscribe to the particular task are invoked via the Android Intent interface and the structured data is transferred to the new app (App2 and App3 in the example).

4. Learning Insider Models for Apps

Ordinary users can build app models for new apps by executing a sequence of simple tasks inside the app as instructed by Insider platform. The *Inference Engine* module in Insider then processes the UI traces gathered from *UI Scraper* to build a new app model, which is stored locally as well as shared with other users via the cloud store. In this section, we describe how the *Inference Engine* module maps UI data to entities in the task abstraction template.

4.1 Semantics from Training Data

Insider includes a learning engine to glean the *semantics* of the in-app content that is displayed on the screen, i.e., the meaning of the various UI widgets and key-value pairs, or at least the ones that matter. Gleaning such semantic information is challenging because of the lack of standardization. Different "keys" could be used in different apps to refer to the same semantic information, e.g., the keys "origin", "source", and "dep city" could all be used in various flight booking apps to refer to the same semantic entity — departure city. Sometimes the key is implicit, say denoted by an image (e.g., separate icons for adult and child passenger counts) or even just screen position (e.g., the UI widget for the departure city is generally placed to the left of that for the arrival city, without the need for either field to be labeled).

In view of these challenges, Insider uses training data to learn the semantics of the app information that is displayed. Such training data is obtaned from training tasks executed by ordinary users running the app, who wish to help in generating app models for new apps. By matching attribute values set in the training task to those obtained from the instrumentation of the presentation layer, Insider is able to relate the in-app content to the attributes contained in the task template.

We settled on the above methodology for training tasks after much iteration. Our first attempt — having users simply point and label various UI elements — proved to be problematic, both in terms of inaccuracy (e.g., mislabeling) and ambiguity (e.g., an individual UI element might defy a label picked from a predetermined set). On the other hand, driving an application to execute a task turned out to be more natural for the workers. Such a "learning from example" methodology has been applied with great success in other domains as well, e.g., Excel FlashFill [4], wherein users provide example data points and the system automatically learns the intended formula.

4.1.1 Training Tasks

With regard to the nature of the training tasks themselves, our initial attempt was to have an expert define one or more tasks, with all of the attribute values specified. However, this proved problematic since a particular app might not support specific settings of an attribute (e.g., a flight booking app targeted at the Indian market may not support non-Indian airport codes, and likewise a shopping app would not allow an item that is not in its catalogue to be added to the cart).

In view of these challenges, the expert defines one of more *task templates* for each category (e.g., booking a flight or adding items to a shopping cart), with placeholders for one or more attributes. Table 1 shows a few examples of task templates. The users each then execute a task conforming to the specified template, with the attribute values of their choice. They are then led through an automatically-constructed sequence of secondary tasks, each involving executing the original task but with one or more of the attribute values changed.

Primary task instances: We let the users pick the attribute values for the specific primary task instances that they execute. We rely on user's ability to pick suitable values for attributes that are admitted by an app, ignore attributes that are not supported by the app (e.g., a particular flight booking app might not allow the number of infants to be set), and do what is necessary to interact with the UI widgets (e.g., to set a date, some apps might present a textbox while others a calendar widget).

Secondary task instances: Then, for the secondary (i.e., follow-on) task instances, we had the users re-execute the task, with one or more attribute values changed. The baseline for such secondary task instances was to have the worker change exactly one attribute value from the previous task instance they had executed. While this conservative approach helps minimize ambiguity in deducing the relationship between the attributes and the UI elements displayed by the app, it would mean that the number of secondary task instances would be as many as the number of attributes.

To enable inferencing with fewer secondary tasks, or equivalently to maximize the amount of information gleaned from each secondary task, we use two techniques: (a) we consider the entity type of each attribute in the task template and allow multiple attributes to be changed simultaneously in a secondary task, so long as they can be disambiguated based on their entity types (e.g., the source airport and number of passengers can be changed simultaneously but the source airport and destination airport cannot), and (b) even when the entity type for two attributes is the same, we allow these to be changed simultaneously so long as the change happens in a controlled manner that enables disambiguation (e.g., a secondary task could involve incrementing the number of adults by 1 and the number of children by a different delta, say 2, or as an example of non-numeric attributes, a secondary task could involve setting the source city to the value of the destination city in the primary task, and the destination city to a new value of the worker's choice.

Verification task instances: Based on the execution of the primary task and possibly secondary task(s), Insider builds a semantic model for the app and also learns some specific settings of attribute values supported by the app (e.g., the airport codes that the app admits as input). We use this learning to run one or more *verification* task instances at the end, wherein all of the attributes values are specified (drawn from the values we know are supported by the app) and we look for consistency between the specified task and the inferences made from its execution using the semantic model built by Insider.

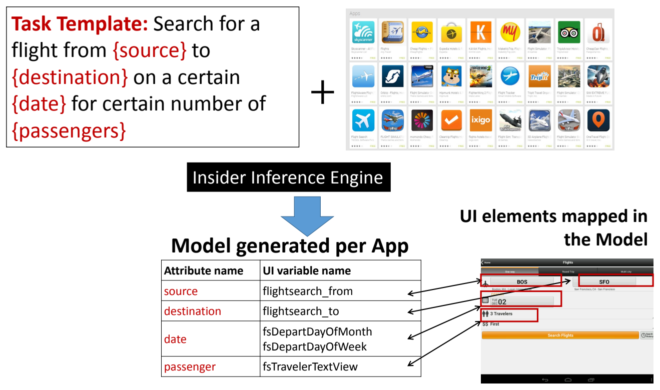
4.2 Inferencing from Training Task Data

The training tasks generate raw traces comprising the values of all UI variables (e.g., textboxes) in the app. This trace is filtered and processed in multiple steps, as shown in Algorithm 1, to build the Insider app model. The model, which is represented by the variable `UIDiffset` in the algorithm, maps each attribute in the task template to the corresponding UI variable(s). An example is shown in Figure 4.

Briefly, the inference algorithm processes the raw UI traces from a user's execution of the app. From each trace,

App category	Task type	Task template
Flight	Primary	<u>Search</u> for a one-way flight from a `source city` to a `destination city` on a certain `date` for a certain `number of passengers` in a certain `class of travel`
	Secondary	<u>Search</u> for a one-way flight using the same attribute values as in previous task but only changing the `source city`
	Secondary	<u>Search</u> for a one-way flight using the same attribute values as in previous task but only changing the `destination city`
	Secondary	. . .
Shopping	Primary	Browse for an `item` and <u>add</u> it to shopping cart
	Secondary	<u>Delete</u> the `item` from shopping cart
	Primary	<u>Add</u> a `quantity` of two of a different `item` to the shopping cart
	Secondary	<u>Delete</u> the `items` from the shopping cart
Recipe	Primary	<u>Search</u> for a `recipe`
	Secondary	<u>Search</u> for a different `recipe`
Transit	Primary	<u>Search</u> for a route from a `source location` to a `destination` using a certain `mode of transport`
	Secondary	<u>Search</u> for a route using the same attribute values as in the previous task but only change the `source location`
	Secondary	. . .
Stock	Primary	<u>Search</u> for a `stock quote`
	Secondary	<u>Search</u> for another `stock quote`
	Primary	<u>Search</u> for a `stock quote` and <u>add</u> it to the `favourite list`
	Secondary	<u>Search</u> for another `stock quote` and <u>add</u> it to the `favourite list`
Hotel	Primary	<u>Search</u> for a hotel for `people` from `start date` to `end date` in a certain `city`
	Secondary	<u>Search</u> for a hotel using the same attribute values as in previous task but only changing the `people` count
	Secondary	. . .
Weather	Primary	<u>Search</u> for weather conditions in `city`
	Secondary	<u>Search</u> for weather conditions in different `city`
News	Primary	<u>Search</u> for a news `topic`
	Secondary	<u>Search</u> for a different news `topic`
Music	Primary	<u>Search</u> for a song by `artist` in `genre`
	Secondary	<u>Search</u> for a song by using the same attribute values as in previous task but only changing the `artist`
	Secondary	. . .
Social	Primary	<u>Post</u> a `status` on the social medium
	Secondary	<u>Post</u> a different `status` on the social medium
	Primary	<u>Search</u> for a `topic` on the social medium
	Secondary	<u>Search</u> a different `topic` on the social medium
Communication	Primary	<u>Post</u> a `message` to someone
	Secondary	<u>Post</u> a different `message`

Table 1: Task templates used by apps in our experiments. The `attributes` are shown in teletype and the <u>actions</u> are underlined. The primary task templates are specified by an expert; the secondary ones are derived from it automatically.


Task Template: Search for a flight from {source} to {destination} on a certain {date} for certain number of {passengers}

Attribute name	UI variable name
source	flightsearch_from
destination	flightsearch_to
date	fsDepartDayOfMonth fsDepartDayOfWeek
passenger	fsTravelerTextView

Figure 4: **App Model Example**

Algorithm 1: Inference Algorithm

```
InferAppModel(UITrace)
begin
    NumTasks = sizeof(UITrace)
    NumAttributes = NumTasks-1
    for $i = 0; i <= NumAttributes; i++$ do
        UITrace[i] = FilterOutUnnamedAndNonStringVariables(UITrace[i])
        UITrace[i] =
        FilterOutAllButFinalValueOfVariables(UITrace[i])
    for $i = 1; i <= NumAttributes; i++$ do
        UIDiffSet[i] = Diff(UITrace[i-1], UITrace[i])
        i++
    UIDiffSet =
    FilterOutVariablesWithRepeatedOccurences(UIDiffSet)
```

we first filter out the UI variables for which the developer has not assigned a name and also variables whose values are not text strings. The rationale is that for the relatively small subset of variables that matter, developers typically do assign a name. Further, in our experience, restricting to variables with textual values did not eliminate any variables of interest. Even UI widgets such as radio buttons tend to have textual values (e.g., "Economy", "Business", etc. in the context of class of travel), although we had to additionally record the state of the radio button to know which textual value was selected.

A variable's value is logged in the raw UI trace whenever it changes, so a variable might occur multiple times in the trace (e.g., the user might set the passenger count to 1 and then change it to 2, before hitting "submit"). So we filter out all but the final value of each variable.

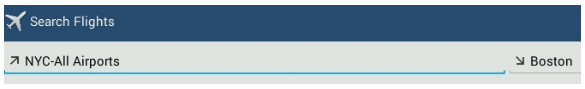
Next, we walk through the set of traces corresponding to the primary task and the subsequent secondary tasks. For each pair of successive tasks (which corresponds to the user executing a task and then the same task again, but with a change in the setting of one attribute, say `i`), we compute the difference, `UIDiffSet`, comprising only the variables whose values changed across the pair of tasks. This difference set represents the candidate set of variables that might correspond to attribute `i`.

We refine `UIDiffSet` by filtering out variables that appear in the difference set for multiple attributes. Such a repeated occurrence suggests thatthe changes occurring in the variables from one task to the next are likely not due to the change in the attribute value. For example, a variable corresponding to the current time will see a change in value each time and hence will occur in the difference set of all attributes. However, clearly these changes are not due to changes in the attribute value, hence this variable is filtered out.

After all of this filtering, we have a refined difference set, which only comprises variables that are likely to be related to the corresponding attributes. However, there is still the possibility of making an incorrect inference due to human error, e.g., the task may have asked the user to change the source airport but the user changed the destination airport instead. To guard against this, we do not finalize the app model when uploading it to the cloud-store. We let a small subset of users (3 users, by default in our experiments) execute training tasks for each app and look for a majority agreement in the app models that are inferred by each user. For instance, with 3 users, if a certain variable `v` appears in the difference set for attribute `i` in the traces from at least 2 workers, it is retained; otherwise, it is filtered out. After a few users have built the app model for a version of the app, and uploaded them to the cloud store, the model is finalized. Other users can simply consume the finalized version of the app without needing the training phase.

Aliasing: We are now left with the Insider model for the app, `Attribute2UIMapping`, which maps each attribute to the corresponding UI variable(s). In many cases, an attribute maps onto just one variable and we are done. In some cases, the attribute could map onto multiple variables, which we term aliasing. For example, when a user executes the task of adding a new item to the cart in a shopping app, a textbox corresponding to the item name and another one corresponding to the price might both change. Furthermore, for the small number of training tasks executed by users, the values appearing in both of these textboxes might be unique. In such cases, Insider's inference algorithm would not be able to tell which of the two textboxes constitutes an appropriate description of the item. In these cases, we use verification tasks to help refine the app model: a verification task to "add Apple USB Superdrive to the cart" would be executed by the user whereas one to "add $74.99 to the cart" would not be.

Finally, in a few cases, the Attribute2UIMapping could map to the null set, say, because of a lack of consensus among the (erroneous) user runs. This will result in Insider being unable to learn the corresponding attribute to UI widget mapping for that app.



UITrace snippet:
function: package property variable_name location text_value
onDraw: com.expedia.bookings.widget.AlwaysFilterAutoCompleteTextView VFEDCL..I id/departure_airport_edit_text xy=(7,77) text=**NYC-All Airports**
onDraw: com.expedia.bookings.widget.AlwaysFilterAutoCompleteTextView VFEDCLF.I id/arrival_airport_edit_text xy=(204,77) text=**Boston**
Attribute2UIMapping:
(Attribute:Source, UIVariable:**departure_airport_edit_text**),
(Attribute:Destination, UIVariable:**arrival_airport_edit_text**)

Figure 5: **UI widgets in Expedia app, UI trace snippet, and Attribute2UIMapping inferred by Insider**

Figure 5 shows a snippet of UI trace for a flight search task inside Expedia app, which is uniquely identified by its package name (com.expedia.bookings) and the activity title(activity.FlightSearchActivity). The trace contains information about the UI variables and the text values contained in them. The `Attribute2UIMapping` table describes the the app model inferred by Insider. The *Inference Engine* automatically associates Source attribute with value 'NYC - All Airports' and Destination attribute with value 'Boston' during this run.

Handling App Updates: Developers often update their apps on the marketplace with new features and bug fixes. In Insider, the app models that we use to extract in-app data can potentially get invalidated during such updates. This can happen mainly due to either (a) change in variable names and/or (b) changes in UI structure. Although, Insider does not deal with an adversarial model, where the developer is intentionally trying bypass Insider, we however employ a number of techniques to deal with updates that are typically seen in App Stores, without requiring the users to re-learn the model.

We leverage additional information with each key-value pairs in the app model such as normalized x,y location on the screen and a set of values observed for the key during actual user runs. When an app update occurs, if the developer changes the variable names (key names) and/or location on screen, we look for the nearest key in spatial and value domain to find the new key corresponding to the task attribute. In Evaluation section we show that this approach is able significantly cut down the process of re-learning app models during updates.

5. Insider Applications

This sections discusses four proof-of-concept apps we built on top of the Insider platform. These apps mashup the user's data de-siloed from various travel and messaging apps including Expedia, ClearTrip, Kayak, Skype, WeChat and Line. Our sample apps "subscribe" to task completion events that the Insider runtime extracts automatically as the user uses the silo apps.

The "subscribe" part requires apps to explicitly declare their subscriptions and for the user to consent to it (or not). We use an approach patterned after how access to hardware sensors is typically managed in mobile OSes, by having the subscriptions be declared as part of the app manifest and be presented to the user for their consent at app installation time. The difference here is that instead of declaring hardware sensors to which access is requested, apps declare user actions from a defined list, e.g., flight booking, movie search, etc.

There may be the concern that such an approach often devolves into a situation wherein users blindly hit "OK" and proceed with installing an app, without quite understanding what accesses the app is requesting. However, we argue that this concern would be mitigated in the context of Insider because the user actions to which access is requested are high-level, semantically-meaningful entities (e.g., "flight arrival information"), which we expect the typical user can relate to more easily than obscure hardware sensors (e.g., "location data"). This higher-level abstraction also enables the application requesting access to spell out the reason for its request, e.g., "This app is requesting access to your flight arrival information to book a cab on arrival". That said, we defer to future work a user study to evaluate the effectiveness of such notifications.

We present four simple applications that take advantage of the capabilities enabled by Insider:

History: History app enables users to see a timeline of activites that have been performed inside various apps. It subscribes to multiple task categroies such as shopping, movies, cuisine search, flight, hotel, and rental car bookings. The app allows searching feature to index into tasks using filters such as time, keywords and task type. Note that none of the silo apps provides this unified view across the other competitor apps, nor offer APIs for third-party apps to directly mashup information for the user's convenience without any user effort.

Price Alert: The price alert app subscribes to the "flight search" and "flight booking" events from multiple flight apps. If it receives two or more "flight search" events for the same origin and destination airports (on potentially different dates), but does not receive a "flight booking" notification within a day of the search event, it automatically registers a price alert for the range of dates seen in the search event. If the price alert triggers with a price at least 10% lower than the price seen from the search events, it notifies the user. Alternatively, if it receives a "flight booking" event before the price alert triggers, it deletes the alert.

Limo Yo: The limo yo app subscribes to the same events as the history app. It infers the user's home and office address using location data at different times of day. When the user books a flight from their home city, it provides a single-touch limo booking that launches the user's preferred limo app with pick-up location set to their home or office address (depending on the day of week and time of the flight), destination set to the airport, and pick-up time set based on travel time estimates between home/office and the airport; return flights to the home city result in a limo to home. In a non-home city, it offers to book limos between the airport and hotel (if a hotel booking event is received) and rental-car booking event is not seen. The entire limo-booking experience requires a single click, far less than the Uber app requires.

Smart Messaging: The Smart Messaging app subscribes to messaging events, which trigger whenever a new message is sent to or received from any messaging app. Insider infers the application name, user name and the message content whenever a messaging event is triggered. The Smart Messaging app does a mashup of all messaging apps, wherein a combined view of all users and messages can be viewed in one place. In addition, for apps that support deep linking (e.g., Skype), the smart messaging app can launch the corresponding app and directly navigate to the appropriate page in that app so that the user may continue his messaging in that app, if he so desires. We have a uploaded a demo video of the Smart Messaging app in YouTube [12].

In each of the above cases, it is important to note that structured information is sourced from any app used by the user, so long as an Insider model has been built for it. This sets it apart from existing approaches wherein functionality such as price alerts are tied to what the developer of a specific app implements.

6. Implementation

We briefly discuss some noteworthy aspects of our implementation of Insider on Android.

Third-Party App to Tap Presentation Layer: Insider is implemented as a regular third-party app for the purpose of easily deploying it on unmodified Android phones. The *UI Scraper* is implemented as a background user-level service. As a one-time step during first launch, the user grants Insider app permission to subscribe to events from the `AccessibilityService` system service in Android. Thereafter, the *UI Scraper* receives callbacks from the system whenever an `AccessibilityEvent` is fired. This class represents events such as touch, other gestures, and changes in page content. The service logs the events that it receives a callback for.

From the viewpoint of the `AccessibilityService`, the displayed content is represented as a tree of `Access-ibilityNodeInfo` nodes, which contain the ID of the corresponding `View` widget, its value, and its screen position. Each `AccessibilityEvent` (e.g., button click) refers to the event's source, which is an `Accessibility-NodeInfo` node. Thus, we are in a position to log all the events of interest, together with the ID, value, and position of the corresponding UI elements.

In order to keep Insider runtime to have a lightweight footprint, the **Task Extractor** module is invoked to process UI logs only when there exists app models for the currently running foreground app. In addition, we also ensure that there are subscribing apps for the tasks associated with the foreground app.

Capturing operations: There can be multiple kinds of tasks that can be performed in the same application. For example, a shopping app supports adding an item to a cart or deleting an existing item from the cart. Capturing the values set in an app alone (e.g., the content of various textboxes) is sometimes not sufficient to determine what task is being performed. We may also need to capture the operations invoked (e.g., clicking the "add" vs. "delete" button in a shopping app). So during training phase, if we are unable to distinguish between two or more tasks for the same application, then we look for sequence of user interactions that are unique to a particular task. We include such operations along with the task attributes in our app model for the corresponding task.

Application Interface: To make it simple for developers to build apps with Insider, we have built a wrapper library that can be included in the app to start subscribing for new tasks. Inter-app communication is implemented via Intent interface in Android. The applications interested in subscribing to specific tasks first need to *Register* with Insider app. Insider then takes users consent via a toast message before storing the mapping between the app package and the task category (each identified by a unique id). In addition, the subscribing app also needs to specify the tasks requested in the manifest file of the package. This allows Insider app to explicitly check for compliance by reading the app's manifest file. When *Task Extractor* picks up a new task performed by the user, it sequentially invokes *OnData* call for each app that has subscribed to that task category.

7. Evaluation

We now evaluate Insider along multiple dimensions.

7.1 Overhead on the Client Device

Insider employs accessibility API to tap into in-app data. We wanted to understand how much overhead this approach imposes on compute and energy consumption.

We ran two popular mobile benchmarking suites available on Android Play store – Vellamo 3.0 and AndEBench. We compared the scores for different benchmarks related to 2D and 3D rendering, text based animations, page load performance and Java runtime. We also enabled *UI Scraper* when the benchmarking applications are running to evaluate the overheads. Compared to no instrumentation, the phone with Insider has 2% across all benchmarks for accessibility APIs based approach. We also measured energy consumption during these runs. Since Insider runtime is only active when both the screen and the CPU are active, the average energy consumption increased by only 1% compared to no the instrumentation case. Thus, running the accessibility APIs based Insider runtime on phones has negligible impact on both CPU and energy consumption, while also having the advantage of deployability. This makes it suitable for real-world deployment.

7.2 Extraction Accuracy

For the purpose of evaluating Insider with large number of users, we recruited 686 crowd-workers on Amazon MTurk platform. Since, these users are mostly desktop workers, we ran Android x86 AOSP on cloud VMs and made them accessible to the crowd-workers in a browser window using VNC streaming. Our setup consisted of 40 VMs running on a public cloud provider for a period of 9-months. We evaluate the effectiveness of Insider in its central objective, which is to extract structured and semantically-meaningful information from app executions. To perform this evaluation, we first build a model for an app by launching training tasks with 3 separate crowd workers and apply our inferencing algorithm on the data. Next, we launch multiple verification tasks to check the correctness of the learned mapping between UI variables to corresponding attributes in the task template. For each verification task, we know the specific task instance provided as input (e.g., "search for a flight from London to New York on December 1 for 3 passengers"). We then use the Insider model already built for the app to verify if the UI widgets identified in the app model has the corresponding values specified in the verification task. If the two match, Insider is deemed to have successfully extracted the corresponding attributes for that app.

Note that this is a conservative evaluation since we could have cases where Insider has a correct model and reconstructs the task details correctly, yet the verification task fails because of human error during the verification task (e.g., the worker may have searched for a flight from London to New Delhi instead of New York). On the other hand, it is highly unlikely that Insider inferred an incorrect app model during training and somehow the verification task, executed by a different worker, also recreated the same error in execution.

We evaluate Insider on 150 different apps that span 11 different task completion categories. Our metrics are extraction accuracy (percentage of UI widget to task attribute mapping learnt correctly, i.e., that pass verification) and false positives (percentage of UI widgets to task attribute mapping that fails verification), computed after the training phase. Note that all the false positives get eliminated after the verification step.

The results are shown in Table 2. The total number of attributes that can be learnt from these 150 apps is 285,

Number of apps	150
Total number of attributes in all apps	285
Number of attribute to UI mapping learnt correctly (accuracy)	237 (83.2%)
Number of attribute to UI mapping learnt incorrectly (false positive)	34 (11.9%)
False positive after verification	0%

Table 2: Extraction accuracy of Insider

accounting for the fact that all task attributes specified in the task template are not present in all the respective apps. We find that insider is able to extract 237 attributes correctly resulting in an extraction accuracy of 83.2%. We also find that Insider incorrectly learns UI mappings for 34 attributes for a false positive rate of 11.9% but these are removed from the app model once they don't pass the verification step.

Figure 6 shows a breakdown of accuracy and false positive values for each individual category as well as overall before the verification step. While there is variation in extraction accuracy across the different categories, we note that all app categories have an extraction accuracy of at least 60%.

We analyzed the traces to understand the reasons for Insider being unable to extract the remaining 17% of attributes. Extraction errors arise for a variety of reasons. For example, there is the inherent error due to relying on crowd-sourced human workers. A second source of errors is due to the differences in apps' UI design that impede our automatic framework from extracting effectively. For example, one of the lessons we learned from some of our early evaluations was that some of the heuristics about UI design that was built into Insider's earlier extraction algorithm turned out to be incorrect. Insider worked under the assumption that most UI variables of interest will have a unique identifier associated with it. However, we noticed that for some apps, many UI variable names were reused in the same app screen. As a result, Insider's earlier inference algorithm treated them as a single variable and extracted a single final value from one of the many variables present on the screen. If this value does not change between two secondary tasks, it gets eliminated from further consideration. Another reason for missing variables is that for certain UI elements like radio-buttons and check-boxes, between two secondary tasks, the text values associated with the UI variables don't change but their internal properties do. Insider's earlier inference algorithm only looked for changes in the values and hence eliminated these variables. To address the above two drawbacks, we augmented the UI variable name with the location property (top left, top right), and augmented the UI variable value with the internal state (e.g., isEditable, isSelected, and isClicked), which then improved our extraction accuracy. Nevertheless, note that even having partial attribute related information for a few apps can still be beneficial (e.g., extracting the source, destination airport and date, but missing the number of passengers can still be effective for a price alert app).

Finally, the reason for false-positives in the inferencing algorithm is the aliasing effect described in Section 4. We

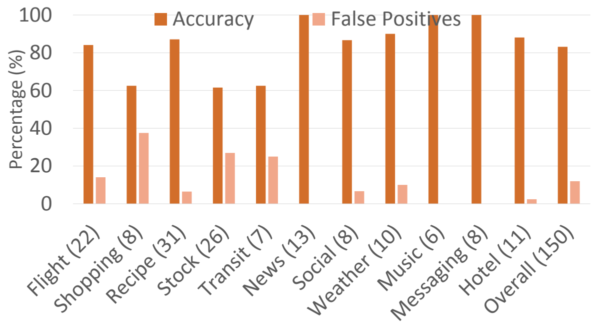


Figure 6: **Accuracy across app categories**

eliminate these false positives from the app model by invoking verification tasks. This process is analogous to taint tracking where we provide values for certain attributes and ask the human worker to perform the task. If we see the values appear in the UI variables in our app model then the verification step is successful. Otherwise, we eliminate these variables from our model. By employing such verification tasks we were able to eliminate all false positives.

7.3 Updating Model for New App Versions

Mobile apps tend to be updated frequently, so having to update Insider's model as well each time would add to the cost. We performed two kinds of analysis to study the impact of app version updates.

Number of apps	150
apps with version updates	60 (40%)
apps requiring app model update	9 (6%)

Table 3: Study of app version updates from Sep 2014 to Mar 2015 and their impact on the learnt app model

First, we ran the MTurk experiments to build app models for 150 apps in two stages, 94 apps in September 2014, and 56 more apps in November 2014. In early March we performed a longitudinal study on the 150 apps, tracking the validity of the earlier learnt Insider app model over time. As shown in Table 3, out of the 150 apps, 60 apps had a newer version released in the span of 6 months. Our robust model detection technique of the updated apps revealed that only 9 of the 60 apps required a new app model to be learnt as they had completely new UI structure. Thus, only a small fraction of the apps (6%) required new training tasks to be deployed

#Task Groups	1	3	5	7
Accuracy (%)	77.55	89.80	89.80	89.80

Table 4: Accuracy vs. #Task Groups

to relearn the app model in a span of 6 months. Even for these 9 apps, only 5 apps had extensive UI changes that rendered all the attributes learnt in the previous app model ineffective; for the other four apps, the previous app model was still effective except for one or two attributes that needed relearning.

Second, we compiled a total of 34 versions of 4 popular flight booking apps – Expedia(8), TripAdvisor (4), Kayak (13), and Orbitz (8) spanning a period of over 24 months downloaded from a well known archive [2]. Note that some of the older versions no longer run correctly. Nevertheless, when we statically analyzed the app binaries (APK files), we found that developer-assigned variable names matching those in the Insider model built for the latest version of the app, were present in all previous versions of the apps except in the case of Orbitz app, where apps dated more than 15 months earlier had those variable names missing. This suggests that the app model and the mapping of attributes to variable names likely remains valid across multiple versions. In other words, the Insider model would not have to be learnt afresh despite the version updates, which is good news.

7.4 Varying the Number of Training Task Groups

The evaluation presented in Section 7.2 was based on Insider models built using 3 training task group per app, each executed by a different human worker (a primary task instance and the corresponding secondary instances constitute a task group.) Can we get away with fewer training tasks groups or is there a benefit in having a larger number of training task groups? This boils down to a question of how dependable human workers are in executing Insider tasks correctly.

For a subset of 10 flight apps, we evaluate this question by varying the number of training task groups for each app from 1 to 7, and quantifying the extraction accuracy of Insider using the methodology noted in Section 7.2. As seen in table 4, increasing number of human workers per app from 1 to 3 improves accuracy, however it does not provide gains beyond 3 workers. Note that cost of training per app will increase with number of task groups. Thus we choose a minimum value of 3 task groups per app to strike a balance between accuracy and cost of training.

8. Discussion

We discuss a couple of extensions to the base Insider system, aimed at reducing the amount of human effort needed.

8.1 Leveraging Implicit Training Data

In our discussion thus far, all of the training data was generated explicitly by having human workers execute tasks. However, there is also the opportunity to obtain training data implicitly, specifically in cases where a user session involves a completed transaction that results in an external manifestation such as a confirmation email. Such confirmation emails, conforming to a standardized format [11], are already used by services such as Google Now and Microsoft Cortana to learn about bookings and such made by users. Insider matches this information with that obtained from UI instrumentation, to learn the semantics of the in-app content. If any ambiguity remains after the matching step, say because the same value occurs multiple times in a transaction (e.g., **01** passengers travelling on **01** July), data from confirmation emails for additional transactions, whether by the same user or a different user, can help with disambiguation. Once the mapping between UI elements displayed and the components of the structured confirmation email has been established, Insider is in a position to glean the semantics of in-app information, even when these do not result in a completed transaction (and hence there is no confirmation email).

The attraction of working with such implicit training data is that it does not impose any additional human burden for generating the data. The main limitation, though, is that it requires the app developer to make the relevant information available in a standardized format. Consequently, the list of supported apps is still modest in size; e.g., the list for Google Now, available at [9], shows fewer than one hundred apps.

We evaluated one such app called Chope App for online restauant reservation which sends a confirmation email with well defined schema.org template (21 attributes). Using a simple value based matching algorithm, 15 UI variables were accurately mapped to appropriate task attributes defined in the schema. The remaining matches required more sophisticated string matching functions which ignore some extra delimiters that get embeded in the email text. Overall, the implicit training approach turned out to be very promising for the apps that use structured confirmation emails.

8.2 Reducing the Number of Secondary Tasks

Our evaluation thus far has had as many secondary tasks launched as there are attributes, each such task involving varying the value of just one attribute while keeping the rest unchanged. However, as noted in Section 4.1.1, we can cut down the number of secondary tasks by using entity type information to perform some disambiguation.

We consider the effectiveness of such an approach here in the specific context of airport entities in flight booking apps. Rather than launching separate secondary tasks to identify the origin aiport and the destination airport, we launch a single secondary task to identify the origin airport and attempt to automatically identify the only other airport entity appearing in the app as the destination airport. We find that such disambiguation produces correct results in 25% of the flight apps. This suggests that using entity type information is promising for cutting down the number of secondary tasks, although we defer an extensive evaluation to future work.

9. Related Work

Table 5 summarizes the prior work on gleaning information from mobile app usage and the approach taken by Insider. AppInsight [20] instruments app binaries to do fine-grained tracking of the app's execution at the level of individual functions. Such instrumentation has been used for a variety of purposes, including performance profiling [19]), content-based advertising [17], and automated testing [18]. While binary instrumentation could, in principle, be done without depending on the developer, in practice it is difficult to distribute instrumented app binaries without the developer's consent.

Flurry [5] and Google Analytics [6] are popular services used by developers to learn about the usage and performance of their apps in the field. These involve instrumenting the app's source code and linking to a special library. The granularity of tracking is under the developer's control, with finer-grained tracking resulting in a larger overhead, e.g., in terms of network traffic.

Narus [21] analyzes an app's network communication to learn about app usage. This approach has the advantage of not requiring any instrumentation or presence on the client device, and also providing a view cutting across apps. However, it requires a vantage point in the network that can observe all traffic. Even with such a vantage point, the growing trend towards end-to-end encryption would stymie any attempt to glean information from network traffic. Nevertheless, gleaning information about apps from network traffic continues to be an active area of research [22].

A second strand of prior work centers on automated traversal of apps using a "monkey". Such monkeys have been used to *explore* the state space of an app, often exhaustively, for such purposes as testing [18], discovering ad fraud [16], and finding privacy leaks. The key challenge is in instructing the monkey to interact with UI widgets in an appropriate way, to make forward progress and to do so efficiently (e.g., without traversing the same app screens repeatedly). Customized monkeys have been developed for each analysis like the ones noted above. However, there has also been recent work on a generic framework for UI automation that allows a flexible combination of exploration and analysis [15].

While sharing the focus on the in-app context, our work, Insider, differs from the above strands of work in some significant ways. Insider takes a *task-oriented* approach, both in terms of gleaning *semantically-meaningful* information about tasks based on the user's in-app activity (instead of just focusing on such metrics as app launches and page views), and in terms of *directed traversal* of apps to accomplish specific tasks (instead of exploration). Hence its dependence on human workers to execute specified tasks. Furthermore, Insider employs a platform-based approach, involving *instrumentation of the UI layer* — on-screen presentation and user input — while leaving the apps themselves unmodified. This approach, therefore, is able to cut across apps.

[13] discusses how a shared memory side channel in window managers in Android as well as other OSes could be used by an attacker to detect UI events in a target application. Separately, TaintDroid [14] used taint tracking, enabled through instrumentation of the JVM interpreter, to perform variable-level tracking through untrusted app code. In contrast to the security angle in these prior works, in this paper we show that the well-documented accessibility APIs provided by the unmodified OS can be used, with user consent, to make UI information available to and enable the tracking of UI variables by a third-party app such as Insider, thereby providing users new functionality. While in this paper we have focused on Android, similar accessibility APIs also exist on other modern OSes such as Windows [1].

Finally, recent years have seen the advent of mobile personal assistants such as Apple Siri [3], Google Now [7], and Microsoft Cortana [10], which tap a range of signals to learn about user context. Among these are in-app signals, e.g., calendar entries, but this only happens for select applications, often first-party apps authored by the OS vendor themselves, where the developer has chosen to make information available through APIs.

10. Conclusion

The confinement of user data within app silos, as is the norm today, is limiting. In this paper, we have argued for breaking down these silos and liberating user data, so that it can be shared across apps for the user's benefit. To this end, we have presented a system, Insider, which takes a task-centric approach to defining and extracting structured and semantically-meaningful information from apps. To construct an app model that transforms the raw data feed from the presentation layer into structured and semantically-meaningful information, Insider utilizes a one-time human guided task execution for each app. We show the effectiveness of Insider across a spectrum of app categories and apps. We also present new applications that take advantage of the de-siloing of in-app information by Insider to create unique user experiences.

References

[1] Accessibility for windows runtime apps using csharp/vb/c++ and xaml. `http://msdn.microsoft.com/en-us/library/windows/apps/hh452680.aspx`.

[2] Android Apps repository. http://www.androiddrawer.com.

[3] Apple Siri. `https://www.apple.com/ios/siri/`.

[4] Excel Flash Fill. http://office.microsoft.com/en-in/excel-help/use-autofill-and-flash-fill-RZ103988276.aspx.

[5] Flurry Analytics. `http://www.flurry.com/solutions/analytics`.

Name	Approach	Tracking Granularity	Client Overhead	Infrastructure Overhead	Developer Effort	Ease of Deployment
AppInsight [20]	Binary instrumentation of app	Very Fine (functions)	Medium	Nil	Nil	High (but legal concerns)
Flurry [5]/ Google Analytics [6]	Javascript included with app	Variable (depends on developer effort)	High (network traffic)	High (aggregation)	Variable	Medium (developer has to sign up)
Narus [21]	Network traffic analysis	Coarse (apps)	Nil	High (deep packet inspection)	Nil	Low (traffic encryption)
Insider	Platform instrumentation	Fine (UI elements)	Low	Nil	Nil	High (third-party app)

Table 5: Prior work on tapping in-app information.

[6] Google Mobile App Analytics. `http://www.google.com/analytics/mobile/`.

[7] Google Now. `http://www.google.com/landing/now/`.

[8] IFTTT: If This Then That. `https://ifttt.com/`.

[9] Integrate with Google Now. `http://www.google.com/landing/now/integrations.html`.

[10] Microsoft Cortana. `http://www.windowsphone.com/en-in/how-to/wp8/cortana/meet-cortana`.

[11] Schema-org. `https://schema.org/`.

[12] Smart Messaging App Demo on YouTube. `https://www.youtube.com/watch?v=Hm4PtOA-Myw`.

[13] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into Your App without Actually Seeing it: UI State Inference and Novel Android Attacks. In *Proc. of the USENIX Security Symposium*, 2014.

[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] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-Automation for Large Scale Dynamic Analysis of Mobile Apps. In *MobiSys*, Jun 2014.

[16] B. Liu, S. Nath, R. Govindan, and J. Liu. DECAF: DEtecting and Characterizing Ad Fraud in Mobile Apps. In *NSDI*, Apr 2014.

[17] S. Nath, F. X. Lin, L. R. Sivalingam, and J. Padhye. SmartAds: Bringing Contextual Ads to Mobile Apps. In *MobiSys*, Jun 2013.

[18] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. In *MobiSys*, Jun 2014.

[19] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling User-Perceived Delays in Server-Based Mobile Applications. In *SOSP*, Nov 2013.

[20] L. R. Sivalingam, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Sayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *OSDI*, Oct 2012.

[21] A. Tongaonkar, S. Dai, A. Nucci, and D. Song. Understanding Mobile App Usage Patterns Using In-App Advertisements. In *PAM*, Mar 2013.

[22] Q. Xu, Y. Liao, S. Miskovic, M. Baldi, Z. M. Mao, A. Nucci, and T. Andrews. Automatic Generation of Mobile App Signatures from Traffic Observations . In *Infocom*, 2015.