My OS ought to know me better: In-app behavioural analytics as an OS service

Earlence Fernandes
University of Michigan

Oriana Riva Suman Nath
Microsoft Research

Abstract

What a user does in an app (e.g., viewing the menu of a restaurant or listening to the same song several times) is key to understanding user interests and preferences, and ultimately to enabling personalised experiences. This kind of behavioural analytics information, as we call it, is rarely used today (and if it is used, it remains siloed in one app). This paper makes a case for the OS to provide an in-app behavioural analytics service which monitors user activities within an app to extract such analytics and to share them with other apps in a secure, private and uniform way. All this must be achieved with zero developer effort and with low resource overhead.

1 Introduction

Today there are apps for almost everything, from education to entertainment, from travelling to shopping, from cooking to exercising. In fact, while using mobile devices users spend most of their time within apps [3]. Knowing what a user does within an app, such as purchases she makes, restaurants she books, or songs she listens to is valuable information that reflects her interests, habits, and behaviours. We collectively call this information *in-app behavioural analytics* due to its analogy with web analytics today collected (directly or indirectly) by web publishers each time a user visits their website.

In-app behavioural analytics data is key to personalised user experiences. Amazon, for instance, keeps statistics on items bought or viewed by each user to suggest items related to her purchase and browsing history. Other apps that utilise a user's behavioural information to personalise contents and experiences include Netflix (for movies), Pandora (for music), Ness (for restaurants), Polyvore (for fashion), and Prismatic (for news). When done well, personalisation drives higher conversion and customer satisfaction. For example, 35 percent of Amazon's product sales originate in personalised recommendations [14].

In-app behavioural analytics can benefit not only third party apps, but also various first party services offered by today's mobile OSes. For example, iOS, Windows Phone and Android all come with digital assistants (Siri, Cortana and Google Now, respectively). Today, when a user asks her digital assistant to order a pizza, it returns results from Google or Bing for keywords such as "pizza". On the other hand, if the digital assistant could tap into the user's in-app activities to learn that the user often uses the OpenTable app to make reservations at the "Via Tribunali" pizzeria, it would readily know which restaurant, and perhaps which pizza, the user is referring to.

Despite their value, in-app behavioural analytics are rarely used today, mainly for two reasons. First is the cost and complexity of extracting in-app behaviour. This requires an app developer to carefully instrument her code to capture the user's activities within the app, and then to extract semantically meaningful information about the activities. This includes both the semantics of *what* content a user consumes in an app (e.g., a restaurant) and *how* she interacts with it (e.g., she views the menu or reserves a table). Second, apps today work in silos. Due to security and privacy reasons, mobile OSes such as iOS and Windows Phone do not allow apps to locally share data with each other. Therefore, even if there is a third party library that a developer can use to easily extract in-app behavioural information, one app will not be able to access information from another, severely limiting the utility of behavioural analytics. Even if the apps could share data with each other, through the cloud or locally (e.g., on Android), there is no central service or framework that co-ordinates exchange of behavioural data thus making interoperability hard.

In this paper, we take the unorthodox view that the OS should take the responsibility for extracting in-app behavioural analytics and making them available to apps, through what we call a *Behavioural Analytics Service* (BAS). The BAS would 1) transparently monitor how the user interacts with her apps, 2) extract semantically

meaningful information about her interests, habits, and behaviours, and 3) expose the information to various apps in a secure, private, and uniform way. Behavioural data captured by the BAS is represented as a collection of *Behavioural Data Items* (BDIs), each of which is an object containing name, type and a list of qualifying metadata and usage statistics for some "thing" that the user consumes or interacts with in an app. For instance, a concrete BDI might contain the name of a business, its type ("restaurant"), address, price range, user actions associated with it (e.g., "Like", "View menu" or "View map"), and time the user spent viewing it.

Our proposal raises many open questions for which we do not have concrete answers. For example, we do not propose a concrete vocabulary for BDI attributes and methods to extend the vocabulary to support new attributes. We hope that our examples throughout the paper will initiate a discussion for a base vocabulary and extension methods. Users and app developers also have conflicting interests in what BDIs may be shared, and balancing these is an open problem (more in §4).

Putting the BAS in the OS has many unique benefits over alternative designs, as we discuss in §2. It also poses constraints: the BAS must work without any help from app developers (which implies zero developer effort) and be efficient. We outline a design proposal in §3 and discuss associated research challenges in §4.

2 Why the OS?

We compare our proposal for putting the BAS in the OS with two alternative designs based on third party libraries: (a) In-app, where the behavioural analytics library extracts behavioural data and stores it locally to be later consumed by the same app, and (b) Cloud-based, where behavioural data is extracted and stored in the cloud to be consumed by the same or other apps. The high- level architectures of these designs are shown in Figure 1. We compare these designs on various dimensions summarised in Table 1. We first discuss the unique benefits the OS-based design has over other approaches.

Sharing across apps. With a centralised behavioural analytics service in the OS, apps can share behavioural analytics data among each other. This can give an app a broader view of the user's interests, habits, and behaviours than what is visible to that single app, and hence enable the app to personalise a user's experience based on her activities in other apps. This is not possible with in-app libraries that cannot share data with other apps (e.g., in iOS and Windows Phone). Note that Android allows in-app libraries to share data across apps (e.g., via a shared sqlite database), but, without a unifying service, sharing must be implemented and maintained on a per-app basis. A cloud-based library can also allow such sharing. However, unless *all apps* use the same analytics library or *all libraries* follow the same representation of behavioural data, interoperability and sharing across libraries can be a nightmare.

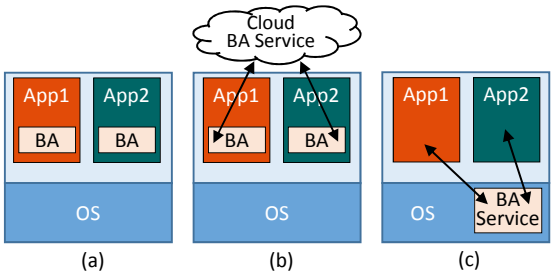

Figure 1: Three design choices for implementing in-app behavioural analytics (BA).

	3rd party library		(c) OS
	(a) In-app	(b) Cloud	service
Sharing	×	√	√
Low resource overhead	×	×	√
App independence	×	√	√
Privacy	√	×	√
Deep learning	×	√	×

Table 1: Trade-offs of various designs.

Low resource overhead. Behavioural analytics can come with a smaller overhead by being an OS service. Resource footprint of the service can be independent of the number of apps, unlike in-app libraries where the overhead can grow linearly with the number of apps (i.e., the library is replicated in each app). Without developers' annotations, the process of automatically inferring semantics of user actions can be expensive. Similar techniques are used for processing web documents and are computation-intensive and use large databases. Pushing the task of inferring semantics to the cloud can avoid the computational and memory overhead, but would cost network overhead on battery-constrained mobile devices and introduce privacy risks, as discussed below.[1]

App independence. Behavioural analytics provided by the OS can be independent of the apps installed on the device. This nicely decouples publishers and consumers of analytics data. For instance, if a recipe app such as Epicurious wants to leverage the cuisine type of restaurants the user searches most often, it does not need to depend on any specific restaurant app and can use data extracted from any currently installed restaurant app. This approach can scale more easily with new apps and does not break when apps are removed or updated. This is also

[1] For many use cases, content semantics must be extracted on the fly, so data must be transferred to the cloud continuously—data aggregation techniques cannot be used to lower the energy overhead. Instead, as discussed later, the processing and memory overheads of the BAS on the device are amenable to reductions by optimising the semantics extraction algorithms.

possible with a third party cloud library, but, as discussed above, it is conditional on the willingness of different apps or different third party libraries to interoperate.

Privacy. An OS service generating and managing behavioural analytics provides a privacy-preserving architecture by design. The OS is implicitly trusted and it can grant or deny access to analytics data as specified by the user. Furthermore, users and app developers do not have to worry about third party libraries surreptitiously tracking in-app interactions and transmitting them to the cloud for nefarious purposes other than analytics extraction. Ultimately, behavioural analytics can be treated as other device resources such as GPS or network and be managed using a similar permission model. Although, an OS design does *not* guarantee privacy of behavioural analytics (e.g., a malicious app with granted permissions can still leak behavioural analytics), it aids, instead of compromising, user privacy.

New functionality. Behavioural analytics data collected by the OS also enables novel system functionality. As discussed earlier, modern personal digital assistants can dynamically learn about a user's preferences and habits from her in-app activities and help users complete their tasks. Many other first party services bundled with mobile OSes can use the analytics to know more about the user and personalise the experience. For example, the music player can automatically select songs that the user likes, and the app store can discover and recommend apps that the user might need. An app, which does not exist today, can mash-up various information about an upcoming trip such as hotel, flights and restaurant details that the user consumes in apps like Expedia and Yelp, so that she can later retrieve them in one place.

Limitations. We acknowledge that the above mentioned benefits do not come without limitations. Understanding a user's habits, preferences, and behaviours from her in-app actions could employ expensive machine learning and data mining algorithms that correlate user activity data with external data sources to identify rich semantics and subtle patterns in user activities. Therefore, a cloud-based solution is likely to provide richer behavioural data than an in-device solution. We argue that this is a reasonable trade-off because (1) recent commercial systems have shown that simple, low-footprint techniques are sufficient to extract very useful behavioural data [8], and (2) we are hopeful that the machine learning and data mining community will develop lower-footprint behavioural analysis algorithms when the need becomes evident.

3 Towards an OS-managed BAS design

For simplicity, we discuss the BAS design for a single-device OS. Our arguments, however, also apply to a multi-device OS where the OS collects behavioural analytics of a user on multiple devices (based on user's app usage on those devices) and sync the BDIs securely across devices.

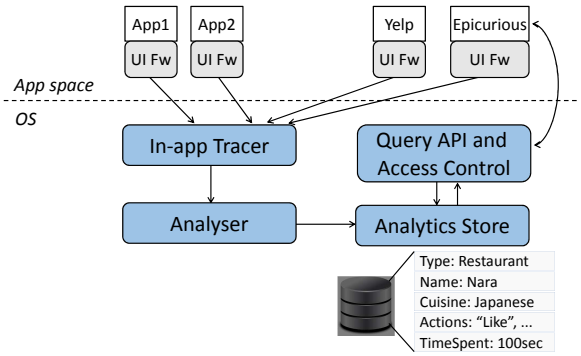

Figure 2: Components of the OS Behavioural Analytics Service (BAS) and example of a behavioural analytics item (BDI) extracted from the Yelp restaurant app.

A BAS in the OS needs to satisfy at least three key requirements. First, it should not rely on any help, such as semantics of the app or its source code, from app developers. Apps should enjoy the benefits of the BAS with zero developer effort. Requiring developer effort would not only impose extra burden on developers, but also create the danger of compromising the accuracy of the BDIs collected by the BAS. For example, if the BAS allows apps to specify as hints the categories of the BDIs they provide, a malicious app actually providing BDIs of category X can claim BDIs of incorrect category Y and make the BAS think that a user using the app likes BDIs of category Y instead of X. Second, since BDIs may contain sensitive information, the BAS needs to be careful about which app may get access to them. Without any access control, an app can read BDIs of the user and use them for undesirable purposes such as ad targeting or sell them to third parties. Finally, the BAS should run efficiently, with a small resource footprint, even when extracting BDIs from a large number of apps.

3.1 A design proposal

We now outline a design for the BAS that satisfies the above requirements. As shown in Figure 2, it consists of three key components.

In-app Tracer: This component extracts raw in-app data by extending the app UI framework (UI Fw). Data include contents a user consumes on various app pages (e.g., text contained in the ListItems of a List UI element) and user actions (e.g., taps of various UI elements) with timestamps.

Analyser: This is the "brain" of the BAS that analyses raw data collected by the tracer to generate BDIs. The analysis can happen as soon as new user traces are made available, periodically or at night when the device is idle

or perhaps charging. The analyser performs the following tasks. First, for each app page, it locates and classifies all texts appearing in the raw data into pre-defined categories such as names of businesses, persons, or locations. Second, it groups various categorised texts into groups such that each group represents one logical entity. For example, a restaurant name, its type ("restaurant"), its address, its phone number, can all be combined together. Each group is represented as one BDI. Third, it discards all but the BDIs that are relevant to the user; e.g., the ones that the user clicks on, visits frequently, or spends a long time on. Finally, it considers previous pages visited by the user to enrich the selected BDIs with additional information. For example, it can associate the name of a restaurant in a page with the cuisine type the user selected in the previous page. The BDIs generated by the Analyser are stored in the Analytics Store.

Query API and Access Control module: This module decides which apps can access the BDIs in the Analytics Store and at what granularity. Apps can submit one-time queries as well as subscribe for BDI events of interest (e.g., a restaurant reservation was made).

3.2 A use case

Consider two apps: Yelp, a popular app for searching restaurants, and Epicurious, an app for searching recipes, installed on a user's device. Each time the user interacts with the apps, the BAS extracts behavioural analytics from their interaction traces and saves them in the Analytics Store. As an example, suppose in Yelp, the user has browsed several Japanese restaurants, such as the "Nara" restaurant (for which Figure 2 reports the extracted BDI). Later, when the user launches Epicurious, it can query the BAS and find out that the user likes Japanese cuisine. With this information, Epicurious can rank recipe results to reflect Japanese dishes. This example demonstrates how apps can share data in a scalable way (i.e., Yelp does not know about the existence of Epicurious and vice-versa), with little effort.

Other scenarios that can leverage this kind of app sharing are Amazon recommending recipe books based on cuisine preferences inferred from Epicurious, the store app recommending kids apps based on toddler songs listened in Pandora, or TripAdvisor recommending restaurants based on hotel reservations made in Expedia.

4 Challenges and open questions

We now discuss a few challenges in realising our proposed design.

Correct and efficient app-tracing. A mobile app page contains various UI elements (such as buttons and text boxes) whose contents are often populated asynchronously (e.g., from the cloud). For *correctness*, the content must be captured after the page has reached a stable state after a user interaction. If the page is scanned for content too soon after a user interaction, the content may not yet be there. If we wait too long before scanning it, the target content may no longer be there either, because it might change due to another interaction or the user navigating to a new page. Mobile apps are highly asynchronous, further complicating the decision of when to capture app page content. To determine if the page has reached a stable state, the system needs to track and determine when all asynchronous calls due to a user interaction have completed and activate UI capture at that point. Systems for app performance analytics such as AppInsight [12] deal with similar problems and we can leverage their techniques.

Efficiently capturing all contents and activities can be tricky too. First, the structure of UI elements in an app page can be quite large and complex. In our preliminary investigation, we found that the AllRecipes app in Windows Phone contains on average 663 UI elements per app page! Second, content must be captured at *every* user interaction, not only when a page loads for the first time. Otherwise, one risks loosing relevant content such as the text entered in a search box before navigating to the next page or dynamic text that appears or disappears at user interactions. Further, note that the content of an app page must be captured on the UI thread (other threads do not have access to the UI tree); therefore, the delay introduced by content capturing must be minimal otherwise it can degrade the UI responsiveness.

App contents and user interaction logs collected on the device can grow large. Our preliminary exploration shows that a 10-minute interaction with the music app Spotify produces about 1 MB of logs. To save space, the BAS can discard the raw data after it has extracted BDIs from them. In the long term, the analytics store itself needs to remove BDIs that no longer apply to the user. New eviction strategies might be needed to determine such BDIs. One option could be to prefer the BDIs that are related to the user's long-term interests and preferences (e.g., cuisine types, news topics, artists, etc.) over those related to temporary ones (e.g., restaurants visited during a trip abroad, purchased items for small kids).

Automatic semantics with zero developer effort. Assigning semantics to unstructured textual data without supervision is by itself a hard problem. In the data mining community, the problem is known as *named entity recognition* [7, 9, 13] where entity extraction algorithms classify text contained in web documents into *entities*. In general, to work in an unsupervised manner, these algorithms rely on the "context" of an input string (text surrounding it) and compare it against a large knowledge base or dictionary. For instance, given a string "Italian", the entity extraction algorithm may classify it as a cuisine type or a language entity. If other cuisine types (i.e., the

context) are found in the same document, the ambiguity is resolved and "Italian" is classified as cuisine type.

Using existing entity extraction techniques in the BAS is non trivial for two key reasons. First, "context" in a mobile app page is limited. App pages contain little text to fit in small screens. For instance, only one restaurant and its associated cuisine type may be displayed per page. Such an insufficient context may result in a poor quality of entity extraction. Second, these techniques are computation-intensive and rely on large databases, making them unsuitable for resource-constrained devices.

However, there are other lightweight techniques that can be easily adopted into a BAS. For example, rule-based techniques [7, 9, 11, 13] can infer semantics of a text string or an app page. For instance, a string can be classified as a phone number or an address with high confidence, if it satisfies a regular expression or a pattern. Similarly, a user can be inferred to book a flight if he clicks on the "book" button on a page containing text fields with labels "From", "To", "Date" and the text "Alaska Airlines". Commercial solutions have shown that such hard-coded rule-based inference provides valuable, albeit not as rich as sophisticated data mining algorithms, behavioural information about the user [8].

Recent work on automated app crawling [6, 10] offers new opportunities between the two extremes of hard-coded rule-based techniques that are lightweight but less powerful, and sophisticated unsupervised data mining techniques that are expensive but more powerful. For example, one might envision automatically crawling contents of popular apps with a monkey [6], analysing the data offline to extract rules, and pushing the rules to the mobile device for a rule-based inference. This would make the inference more automated (since rules do not need to be handcrafted) and powerful (with many more rules than a handcrafted system), but as lightweight as rule-based inference.

Another challenge that fully-automated techniques pose is that they do not provide perfect accuracy (e.g., existing entity extraction algorithms typically provide a recall of 80–90%). The OS must be prepared to deal with such false positives. If a term cannot be classified with high confidence as any of the target entities, it is best for the OS to return a null result.

Options for access control. To control access to behavioural analytics, one option is to treat the analytics database as a system resource and to adopt the same model current mobile OSes use to control access to device resources, such as sensors and network. However, the analytics database can potentially contain many different types of BDIs and one-size-fits-all access control policy may not be appropriate for all apps.

We consider various granularities of access control to the analytics library. Overhead and utility of these strategies will be decided by actual use cases. The first option is the all-or-nothing option, where an app can either get access to the entire analytics library or nothing, depending on how the user grants permission to the app. Another option is to manage the permissions separately per BDI category. For example, OpenTable and Yelp both produce BDIs of category RestaurantName, and an app with permission granted to access BDIs of category RestaurantName can access all such BDIs produced by any app on the device. Yet another option is finer-grained control within a BDI category. One could specify a "fine" and "coarse" subcategory for each BDI category, similar to the ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION permissions on Android. The drawback of fine-grained permissions is that the system may become harder to maintain and use, if the permission set grows large, leading to overprivilege issues [5].

While permissions are a necessary part of security, research has shown that permissions are not sufficient to completely govern access to sensitive data. Thus, entity-type permissions inherit the same problems of traditional permissions. For example, once an app gains access to behavioural analytics, it can leak that data to a malicious network server. Orthogonal techniques to control use of sensitive data are applicable here [1, 2, 4].

Data ownership. Do the extracted behavioural analytics belong to the user or to the app? It depends. An app like Amazon may be willing to share some data but not all its data with other apps. For example, Amazon lets users export their purchase history (and if this information is not shared today, it is mostly because there is currently no easy way for doing it). On the other hand, product reviews are probably a more proprietary type of data. The challenge is finding the right balance which will allow apps to protect their data and business, but also allow users to benefit from new functionality.

5 Conclusions

Users spend most of their time on mobile devices in apps, but the OS today has no visibility into the content users consume within them. We argue the OS is in a unique position to shoulder the responsibility of extracting behavioural analytics from a user's activity within her apps. We propose a behavioural analytics service (BAS) hosted entirely in the OS, which promises to work with zero developer effort and which, at least from a design point of view, does not compromise user privacy. The BAS allows for scalable sharing of behavioural analytics data among apps, makes it easy for developers to personalise their apps, and helps emerging digital assistants to become truly "personal".

An implementation of the BAS, Appstract, is in progress.

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269. ACM, 2014.

[2] M. Conti, E. Fernandes, J. Paupore, A. Prakash, and D. Simionato. Oasis: Operational access sandboxes for information security. In *Proc. of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 105–110. ACM, 2014.

[3] T. Danny. Flurry releases newest app use stats. http://tech.co/flurry-app-stats-2014-09.

[4] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 393–407. USENIX Association, October 2010.

[5] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638. ACM, 2011.

[6] 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 *Proc. of MobiSys*, pages 204–217. ACM, June 2014.

[7] J. Hoffart, M. A. Yosef, I. Bordino, H. Fürstenau, M. Pinkal, M. Spaniol, B. Taneva, S. Thater, and G. Weikum. Robust disambiguation of named entities in text. In *Proc. of EMNLP*, pages 782–792, 2011.

[8] Huge. Lean Personalization. http://www.hugeinc.com/ideas/report/lean-personalization, 2014.

[9] D. Nadeau and S. Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, Jan. 2007.

[10] S. Nath, F. X. Lin, L. Ravindranath, and J. Padhye. SmartAds: bringing contextual ads to mobile apps. In *Proc. of MobiSys*, pages 111–124, 2013.

[11] L. F. Rau. Extracting company names from text. In *Proc. of the 7th IEEE Conference on Artificial Intelligence Applications*, pages 29–32, 1991.

[12] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proc. of OSDI*, pages 107–120, 2012.

[13] S. Sarawagi. Information extraction. *Found. Trends databases*, 1(3):261–377, Mar. 2008.

[14] VentureBeat. Aggregate Knowledge raises $5M from Kleiner, on a roll. http://venturebeat.com/2006/12/10/aggregate-knowledge-raises-5m-from-kleiner-on-a-roll/, 2006.