ERICA: Interaction Mining Mobile Apps

Biplab Deka[1] **Zifeng Huang**[2] **Ranjitha Kumar**[2]
[1]Department of Electrical and Computer Engineering [2]Department of Computer Science
University of Illinois at Urbana-Champaign
{deka2,zhuang45,ranjitha}@illinois.edu

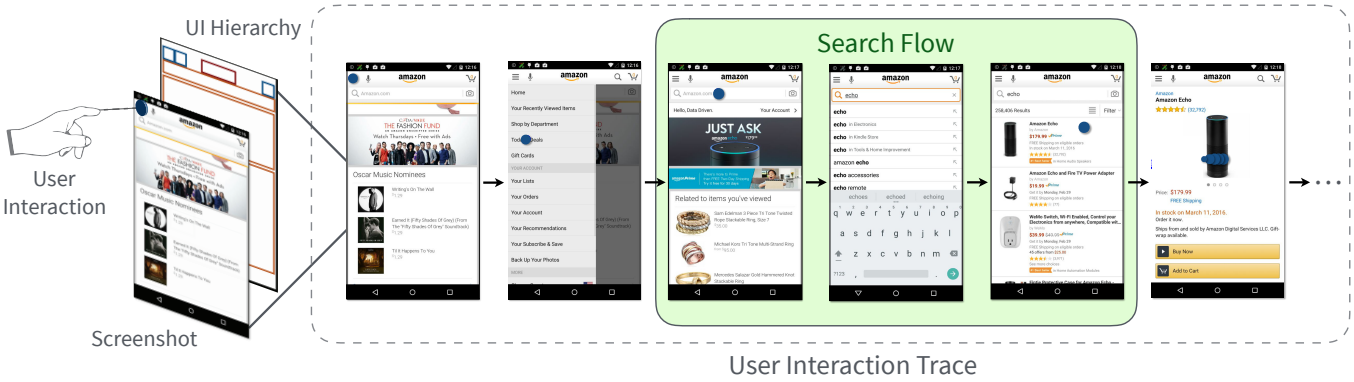


Figure 1: ERICA, a scalable system for *interaction mining* Android applications, captures *interaction traces* as users interact with an Android app. These traces contain snapshots of the app's UI over time (as screenshots and view hierarchies) as well as user interaction data. Here a *search flow* is highlighted within an interaction trace from the Amazon shopping app.

ABSTRACT

Design plays an important role in adoption of apps. App design, however, is a complex process with multiple design activities. To enable data-driven app design applications, we present *interaction mining* – capturing both static (UI layouts, visual details) and dynamic (user flows, motion details) components of an app's design. We present ERICA, a system that takes a scalable, human-computer approach to interaction mining existing Android apps without the need to modify them in any way. As users interact with apps through ERICA, it detects UI changes, seamlessly records multiple data-streams in the background, and unifies them into a user *interaction trace* (Figure 1). Using ERICA we collected interaction traces from over a thousand popular Android apps. Leveraging this trace data, we built machine learning classifiers to detect elements and layouts indicative of 23 common *user flows*. User flows are an important component of user experience (UX) design and consists of a sequence of UI states that represent semantically meaningful tasks such as *searching* or *composing*. With these classifiers, we identified and indexed more than 3000 flow examples, and released the largest online search engine of user flows in Android apps.

UIST 2016, October 16 - 19, 2016, Tokyo, Japan

ISBN 978-1-4503-4189-9/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2984511.2984581

ACM Classification Keywords

D.2.2 Software Engineering: Design Tools and Techniques

Author Keywords

Interaction mining; app design; design mining; user flows

INTRODUCTION

Design plays an important role in the adoption of apps [21]. App design, however, is a complex process comprised of multiple design activities: researchers, designers, and developers must all work together to identify user needs, create user flows (UX design), determine the proper layout of UI elements (UI design), and define their visual (visual design) and interactive (interaction design) properties [2]. To create durable and engaging applications, app builders must consider hundreds of solutions from a vast space of design possibilities, prototype the most promising ones, and evaluate their effectiveness heuristically and through user testing.

To help navigate this complex process, this paper introduces *interaction mining*: capturing and analyzing both static (UI layouts, visual details) and dynamic (user flows, motion details) components of an application's design. Mined at scale, the data produced by interaction mining enables tools that scaffold the app design process: finding examples for design inspiration, understanding successful patterns and trends, generating new designs, and evaluating alternatives.

This paper takes a human-computer approach to interaction mining, using people to understand and interact with UIs, and machines to capture the UI states they explore. This ap-

proach is manifest in ERICA[1], a system for interaction mining Android apps which demonstrates, for the first time, a scalable way to mine the dynamic components of digital design. ERICA provides a web-based interface through which users interact with apps installed on Android devices. As a user navigates an app's interface, ERICA detects UI changes; seamlessly records screenshots, view hierarchies [6], and user events; and combines them into a unified representation called an *interaction trace* (Figures 1 and 2). ERICA requires no modifications to an app's source code or binary, making interaction mining possible for any Android app.

We used ERICA to collect user interaction traces from more than one thousand popular apps from the Google Play Store. These traces contain more than 18,000 unique UI screens, 50,000 user interactions, and half a million interactive elements. Leveraging this trace data, we built machine learning classifiers to detect elements and layouts indicative of 23 common *user flows*. User flows are important components of UX design, comprising sequences of UI states that represent semantically meaningful tasks such as *searching* or *composing* [4]. With these classifiers, we identified and indexed more than 3000 flow examples, and released the largest online search engine of user flows in mobile apps: http://interactionmining.org.

This paper presents the principles of interaction mining, ERICA's design choices and implementation, and an analysis of the collected mobile interaction trace data. It illustrates how the data captured by ERICA enables automatic identification of user flows, and presents example flows found in several popular apps. Lastly, it sketches the space of applications interaction mining enables, with examples from UI Design, motion design, UI implementation, and usability testing.

BACKGROUND AND RELATED WORK

Our work builds upon previous work in *design mining*. First introduced by Kumar et al. in Webzeitgiest [22], design mining was developed to learn about web design practices at scale. Their system captured and combined the visual and structural representations of webpages to compute design features that were used to power a number of data-driven design tools. Our work generalizes this approach to dynamic applications, capturing not just static layouts, but also dynamic components of design (such as animations) and how multiple UI designs are connected via user interactions.

In the mobile domain, several authors have used static approaches to mine mobile app UIs. These methods extract UI layouts from app packages without running the app, and have been used to study design pattern changes over time [15], identify frequently-used UI components [28], and understand characteristics of highly-rated apps [31]. The main limitation of static methods is that they fail to capture any dynamic data, and so cannot help build design tools for the dynamic components of an app's design.

Dynamic approaches for mining mobile apps, conversely, capture data at runtime [23]. Dynamic mining methods can be difficult to implement due to the challenges associated with driving an app's graphical UI to reach new states [29], but dynamic approaches have nonetheless been deployed for testing [24], detecting security vulnerabilities [27, 17], and modeling web applications [25]. The main limitation of these approaches, however, is that they fail to explore app UIs in a manner that mirrors human usage. This makes it difficult to extract meaningful user flows from the captured data — an important goal for interaction mining. ERICA overcomes this limitation by using humans to drive app exploration.

[1]ERICA is an acronym for **E**nabling **R**ealtime **I**nteraction **C**apture for **A**ndroid apps.

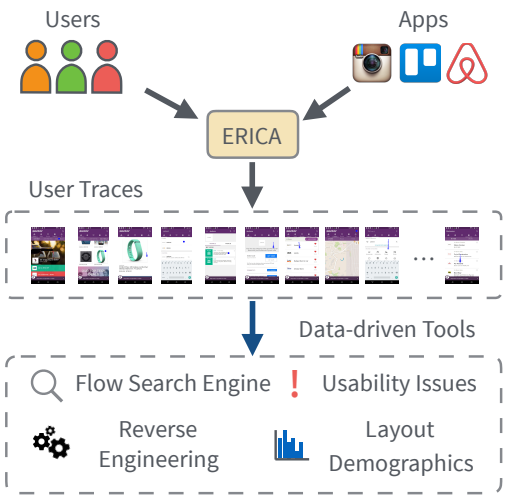


Figure 2: As users interact with apps through ERICA, it records user interaction traces in the background. Mining interaction traces from apps in the wild enables data-driven design applications such as building a search engine for user flows.

INTERACTION MINING

The aim of interaction mining is to build a design repository and set of data structures to support design tools that scaffold the entire mobile app design process. Accordingly, an interaction mining system must possess the following capabilities:

Capture of multiple design components. An app's design consists of multiple components: how users move from one UI to another to complete tasks, how UIs are laid out, the visual details of the UIs and their elements, and how each UI responds to user interactions. Supporting data-driven design tools for app design requires capturing each of these components. This capture is challenging because it requires collecting, combining, and correlating multiple types of data, including both visual and structural representations of UIs and their elements, as well as user interaction details.

Coverage of important UI states. In order for the design data produced by interaction mining to help designers understand the experiences an app affords, it must capture UI states that are frequently used by humans. Discovering and navigating to each of these states in an automated way is difficult, and requires an understanding of how to interact with different UIs. Additionally, states must be visited in a natural, semantic order to be able to capture meaningful user flows.

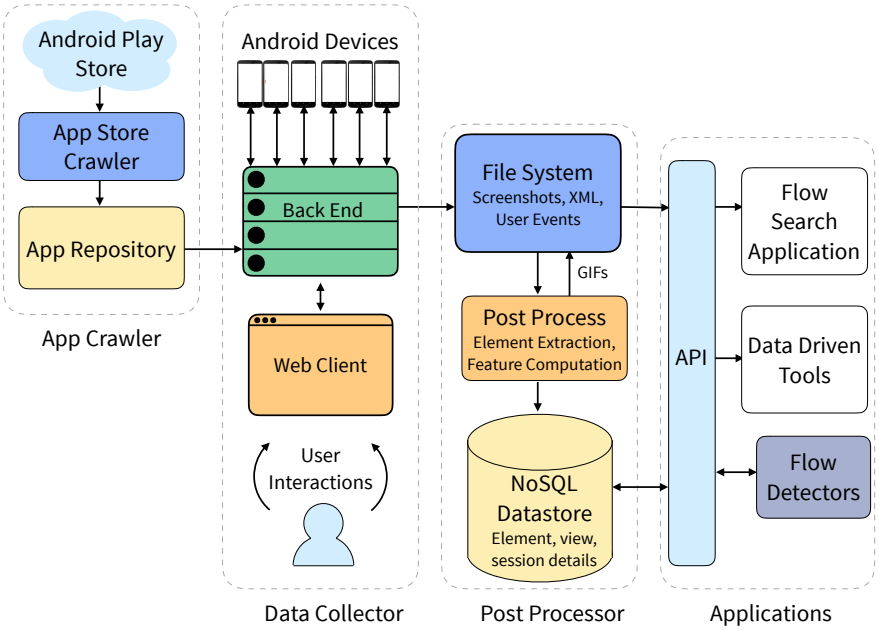


Figure 3: ERICA's Architecture. Apps from the Google Play Store are run on physical Android devices. Users interact with these apps through a web interface that allows the system to capture their interactions as well as the state of the app's UI. Post processing scripts compute structural properties of elements and UIs and store them in a NoSQL datastore. Client applications access data through a web API.

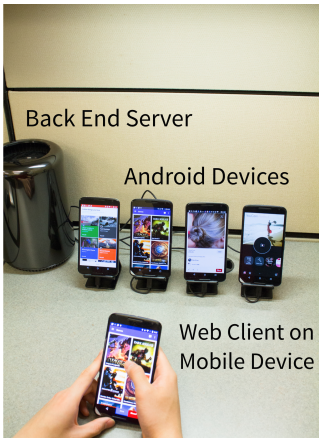


Figure 4: ERICA's data collection infrastructure. Here ERICA's web client is being used in a browser on a phone in fullscreen mode to interact with an app running on one of the other Android devices connected to the server (second from the left). ERICA collects information about the app in real-time while the user is interacting with it.

Scalability. The utility of interaction mining hinges on the scale of the corpus it can build. Larger repositories improve both diversity of results in example-based applications as well as the accuracy of machine learning models trained on the mined data. Mining strategies that require access to an app's source code or the cooperation of an app's developer, therefore, are less useful than those that take a black-box approach to working with the millions of extant mobile apps.

ERICA: INTERACTION MINING FOR ANDROID APPS

ERICA is a system that enables interaction mining for Android apps in a black-box manner. It combines the strengths of humans and machines, using humans to drive app interactions and computers to capture design and interaction data.

Design Choices

Human-Powered Exploration

To generate meaningful interactions for different UIs, ERICA leverages humans to interact with an app while it captures their interactions and the resultant UIs. ERICA employs a human-powered approach over an automated one for several reasons.

First, human interactions are more likely to produce realistic user flows. The space of possible interaction sequences in apps is large, and only a small fraction of such sequences represent semantically-meaningful flows.

Second, many apps require user input (e.g., usernames, passwords, queries, content, etc.) before meaningful interaction can occur. Automatically creating valid user data across a large number of apps is a challenging problem.

Third, for captured UI data to be meaningful, a new UI state should only be captured after the UI has completely updated in response to a user interaction. Automatically detecting the completion of UI updates is another challenging problem, since updates can happen asynchronously in response to external events (for example, when data from a remote server is received). Humans, however, are capable of detecting the completion of UI updates with relative ease.

Web-Based Interaction Interface

ERICA users interact with Android apps through a web-based interface, as opposed to directly accessing the devices on which the apps run. There are two reasons for this design decision. First, having a web-based interface makes it possible to crowdsource data collection over the Internet. Users do not need to use a specific device or install any software on their devices. Second, the servers that power ERICA have much greater compute capability than a phone or tablet, allowing interaction data to be captured in the background without negatively affecting user experience.

Although ERICA users interact with apps through a web interface, ERICA still offers a near-native experience for users. ERICA is powered by a responsive web front-end which, when used in fullscreen mode on a mobile browser, mirrors the screen of the device on which the app is running.

Physical Devices

ERICA runs Android apps on physical phones and tablets connected to the ERICA server. Physical devices offer more predictable UI performance and support a wider variety of features (such as OpenGL 2.0) across a larger number of apps, compared to the best Android emulators available today.

Implementation

ERICA comprises four components: an app crawler, a data collector, a post-processor, and a web API (Figure 3). The app crawler obtains APK files for Android apps from the Google Play Store. The data collector captures the state of an app's UI as users interact with it. The post-processor combines the captured data streams into a single unified representation – an *interaction trace*. It also computes visual and textual features for UI elements and stores them in a NoSQL datastore for subsequent querying. Client applications access ERICA's data through the web API.

App Crawler

The app crawler uses the Google Play Store API to download Android APK files for apps, along with metadata such as app rating, number of downloads, and app category.

Data Collector

The data collector allows users to interact with Android apps through a web interface and captures three kinds of data in real-time: screenshots (visual representation of UIs), view hierarchies (structural representation of UIs) and user interactions. The collector comprises three components: a set of Android devices, a back-end server, and a web client. Each Android device is a physical phone connected to the back-end server via USB, and each one runs a modified version of Android OS.

The back-end manages the devices and installs Android apps on them. It sends the device's UI as compressed JPEG images to the web client using a WebSocket connection. It also relays user inputs from the web client to the device. Finally, it saves the captured data to the local file system.

The web client displays images of the app's UI in a browser and allows users to interact with it. It has a responsive layout and can be used on devices of varying form factors, including desktops and mobile devices. On a mobile device, with the browser in full-screen mode, it offers a near-native app experience (Figure 4). Both the back-end server and the web client are implemented on top of the `OpenSTF` framework [12]. The `minitouch` library allows the web client to support all common multitouch gestures when used on touchscreen devices, and two-finger pinch, rotate, and zoom on regular PCs [10].

Implementing the data collector required overcoming two technical challenges:

Capturing UI Data With Low Latency. Capturing UI data in an app without any perceptible user delay is nontrivial. To capture UI changes in response to user interactions, ERICA requests the structured representation of an app's UI *every time* a user interacts with it. Android's default `UiAutomator` service, which is commonly used to request view hierarchies from Android devices, is poorly-suited for this purpose as it takes multiple seconds to return the data [7]. Instead, we modified Android OS to implement a custom service that responds within about a hundred milliseconds. This service was built by augmenting the Android classes that define UIs and UI elements with custom methods that dump properties of their instances at runtime. The service returns a hierarchical representation of the UI in XML format and has properties (such as position and size) for all the elements on the screen. The service also allows capturing additional properties of UI elements that are not available through the `UiAutomator` service, such as font family and size information for text elements.

Capturing screenshots of a UI multiple times a second enables a smooth user experience when interacting with apps through the web front-end. It also enables high-fidelity viewing of the motion details of the app (animations and transitions) from the captured data. ERICA uses the `minicap` library [9] to enable the capture of more than ten screenshots per second, configured to trigger whenever pixels on the device screen change.

Inferring Gestures. When a user interacts with an app's UI through the web client, the client captures user input events (such as `TouchUp`, `TouchMove` and `TouchDown`) and their coordinates. These events are relayed to the Android device via the server. Android translates these low-level input events to gestures (such as clicks, scrolls, longtaps etc.), and, if applicable, dispatches them to the appropriate element on the UI [8].

Accurately inferring high-level gestures from the low-level input events captured by the web client is a challenging task. Instead, ERICA captures gestures directly from Android OS by modifying Android to output the type and target element of a gesture every time one is detected.

Post-Processor

The post-processor reconciles the three streams of captured data and combines them to form a unified representation called *an interaction trace*. It parses the structural representation of UI screens to identify individual UI elements, crops the elements from the screenshot, and saves them to the file system. Element properties (such as size, location, and contained text) are also saved to a NoSQL datastore for subsequent querying. Multiple screenshots are combined to form animated GIFs that show how a UI responds to different user interactions. A web API provides access to the data stored in the file system and in the NoSQL datastore.

COLLECTING USER INTERACTION TRACES

We used ERICA to collect user interaction traces from more than a thousand Android apps, building a corpus by crawling the top 100 apps from 30 different categories on the Google Play Store. We recruited 28 participants for data collection through posters and email lists, briefed them about the study, and gave them a short tutorial on ERICA's web interface. Each participant was free to use each app totally unsupervised and without any time limit. To protect participants' privacy, we instructed them to input fake personal data as necessary.

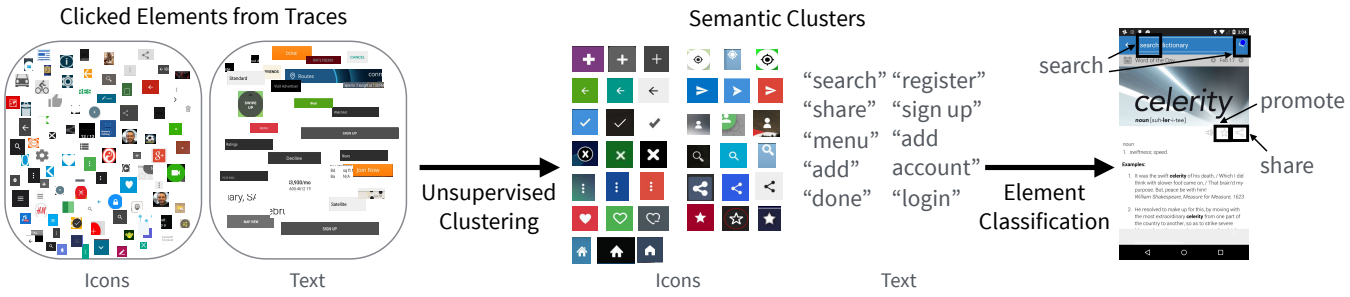


Figure 5: Unsupervised clustering of interactive elements produced clusters that helped identify the most common types of semantically-meaningful elements in the ERICA dataset. We used this information to build visual and text-based classifiers to detect these elements in UIs, and leverage them to identify semantically meaningful flows in user interaction traces.

The participants completed 1186 sessions for 1150 apps. After rejecting sessions with less than 5 interactions, we were left with 1065 sessions for 1011 apps, consisting of more than 18,000 unique UI screens, 52,000 gestures, 500,000 interactive elements, and 6.7 million total elements. On average, sessions lasted 4 minutes (min 30 seconds, max 21 minutes) and had 51 interactions over 18 unique UI screens. Users employed 5 Android activities in each app on average, and performed 6.2 interactions in each activity (activities are components of Android apps that allow users to perform a specific task [5]).

EXTRACTING SEMANTIC USER FLOWS AT SCALE

In this section, we present an automated method for identifying examples of user flows in interaction traces. User flows are an important component of UX design, comprising sequences of UIs and associated user interactions that define a semantically-meaningful task (such as *searching* or *sharing*).

UX designers often search for example flows from existing apps when seeking out inspiration for new projects [20]. This need has given rise to several popular online repositories of flows [14, 13, 11]. These repositories, however, are manually curated by designers who must identify desirable flows and manually capture associated screenshots, making them difficult to scale to large numbers of apps [1]. Our automated flow identification places minimal burden on users, capturing UI data during normal app usage and identifying flows as a post-process in a scalable way.

Identifying User Flows

We developed two methods to identify user flows in interaction traces: an element-based method and a layout-based one. These methods are based on two insights. First, apps are generally designed with visual, textual, or structural cues in their elements (or layouts) that help users identify important semantic tasks. For example, a magnifying glass icon usually indicates a search flow, and the words *sign in* usually indicate a login flow.

Second, a user must interact with an indicative element in order for it to be part of a corresponding flow. For example, if we know that a user clicked the magnifying glass icon on a particular UI screen, we expect that screen to be part of a search flow. Based on these insights, we combine the user interaction and UI data in an interaction trace to identify meaningful flows.

Element-Based Flow Detection

To identify elements indicative of common flows in the ERICA dataset, we clustered the elements users interacted with based on their visual and textual features. We then build classifiers to automatically identify similar elements in new traces.

Clustering Interactive Elements. We clustered both text elements and icons. We identified text elements using the text string associated with them in the structured representation of the UI. Non-text elements smaller than 200×200 pixels and with aspect ratios between 0.5 and 2 where chosen as candidate icons.

To cluster icons, we converted them to grayscale and scaled the images to 50×50 pixels. We trained an autoencoder [16] with 2500 inputs and two hidden layers (with 500 and 200 neurons, respectively) to learn a 20-dimensional feature vector, and then performed k-means clustering in this space. We chose 100 clusters, since the mean-squared error plateaued at that number. Icons from some of the largest clusters are presented in Figure 5. We observed that many of the clusters correspond to icons with semantic meaning such as *sharing*, *searching*, or *liking*.

We computed the frequency of words and phrases across all text elements in our dataset. Some of the text elements with the highest frequency are presented in Figure 5. As with icons, these elements also contain semantic meaning related to flows.

Classifying Elements. Based on these clusters, we developed classifiers to automatically identify similar interactive elements. For icons, we trained neural network-based classifiers with an architecture similar to the autoencoder's (2500 inputs, two hidden layers with 200 and 100 neurons). For text elements, we built simple binary keyword-identification classifiers for each flow. Some flows (such as *search*) have both a icon-based and a text-based classifier. Examples of flows detected in our dataset with these classifiers are shown in Figure 6.

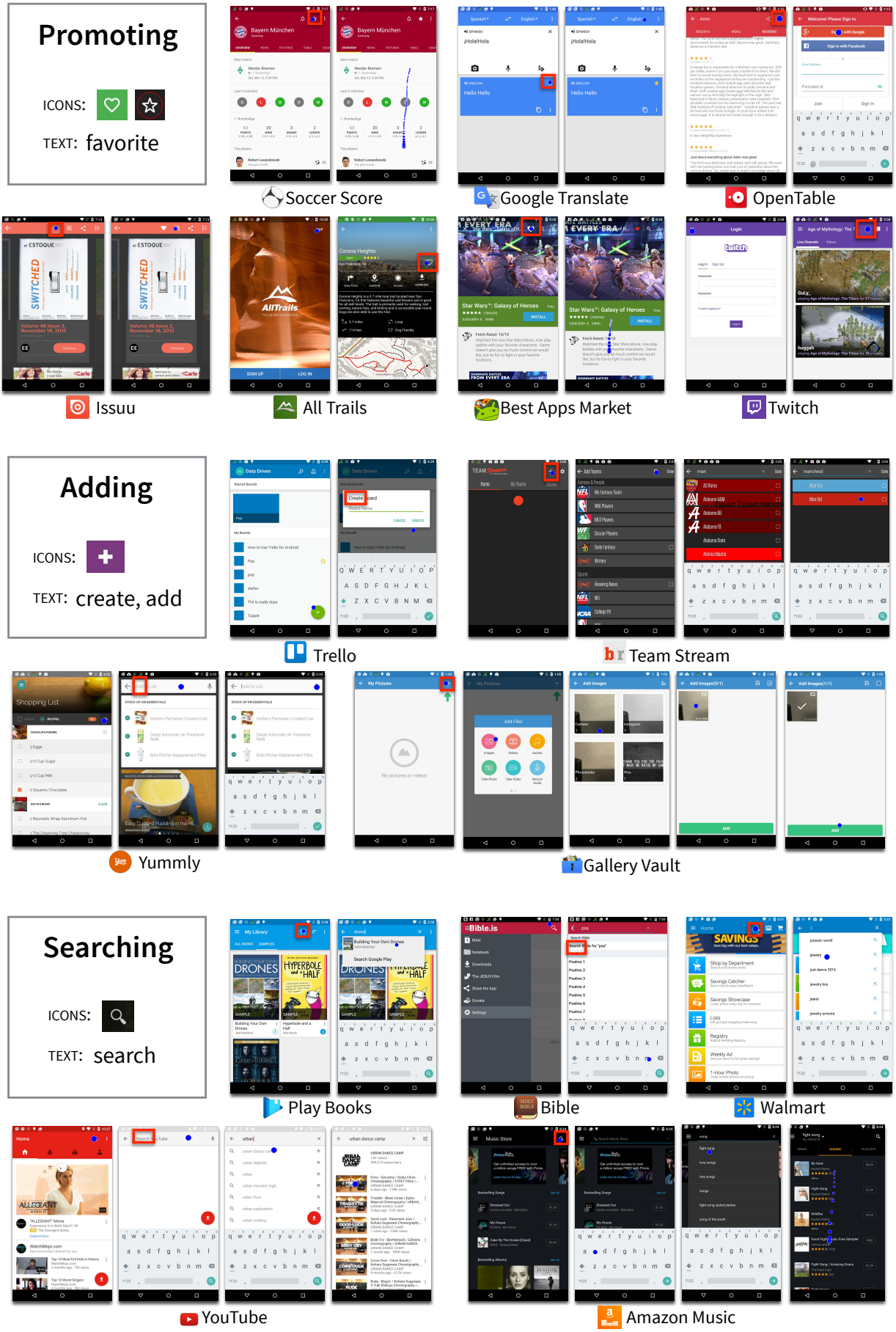


Figure 6: Examples of three different types of flows detected in our dataset by using element-based detectors. Visual markers and textual keywords that were used to identify each type of flow are also shown.

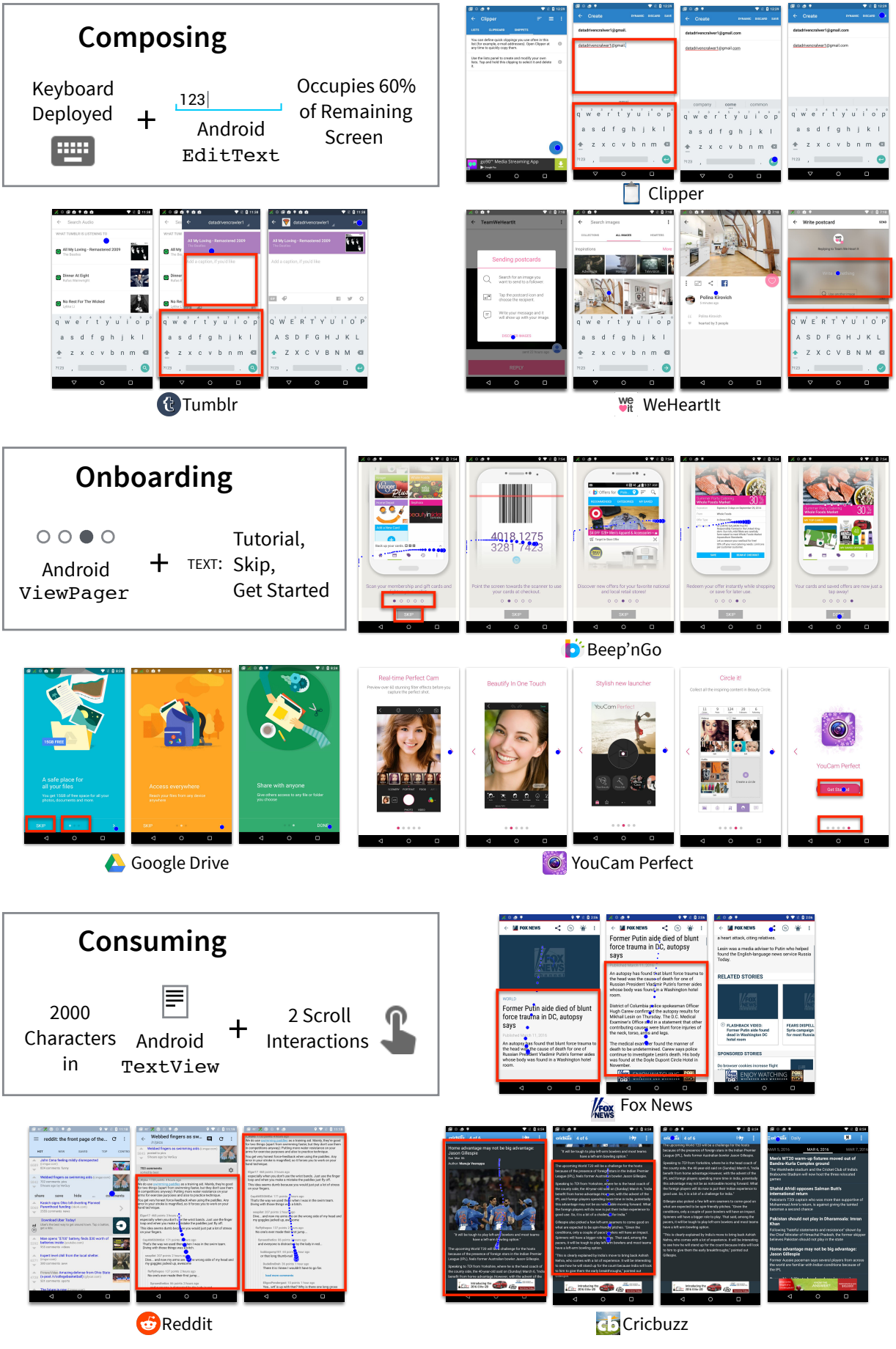


Figure 7: Examples of three different types of flows detected in our dataset by using layout or Android component-based features. Patterns that were used to identify each type of flow are also shown.

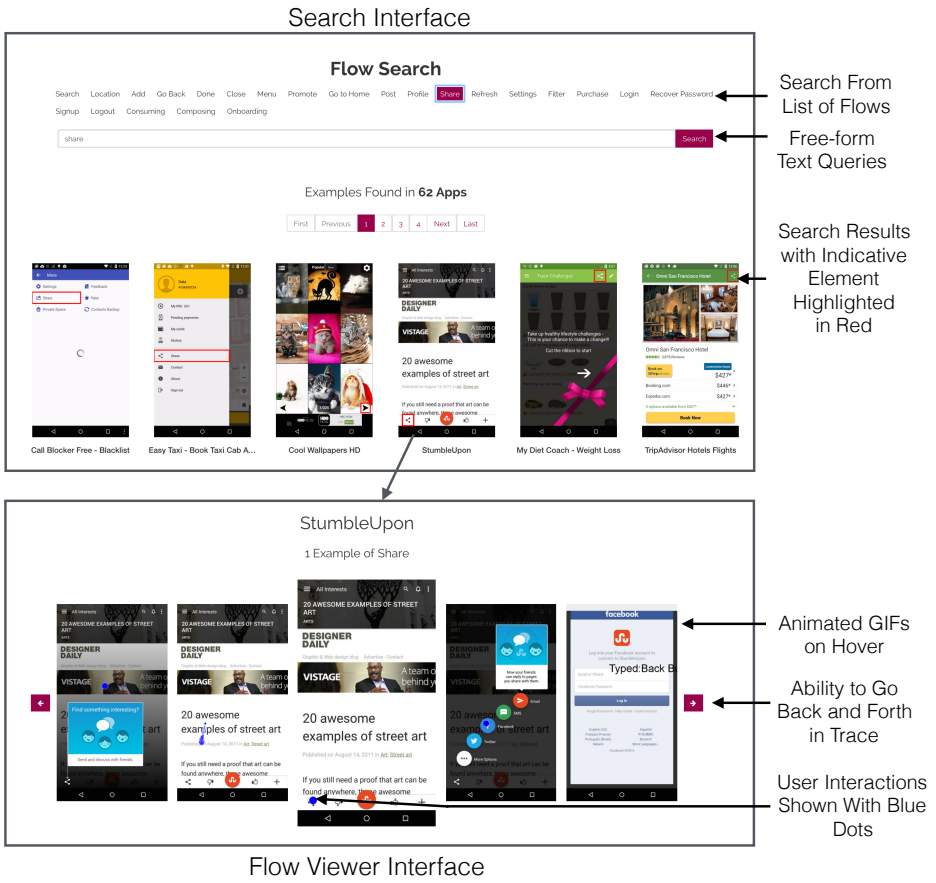


Figure 8: The search interface for finding and visualizing example flows found in the ERICA dataset. Users can search for one of the 23 pre-identified flows or perform free-form text based queries. Each search result shows one UI screen from an app with the indicative element for the flow highlighted in red. Clicking on one of the results takes users to a flow viewer interface that shows the screen from the results page in the context of the nearby screens in the interaction trace. Hovering on an image brings up an animated GIF showing how the UI responded to the user interaction shown on it.

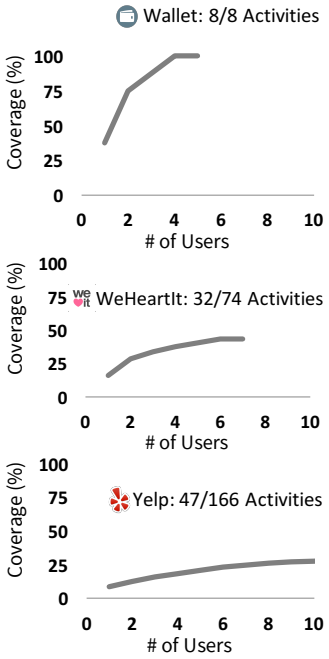


Figure 9: Coverage of activities in the app *wallet* (top), *weheartit* (middle), and *yelp* (bottom) with respect to the number of users whose interaction traces were combined. Coverage does not increase significantly after 5–8 users since there is significant overlap between activities that different users visit.

Layout-Based Flow Detection

We can also detect flows by identifying screens with specific UI layouts (or Android components) on which specific gestures are performed. We developed detectors for three such patterns. Figure 7 shows examples of these flows in our dataset. *Consuming* flows were identified by searching for UIs with more than 2000 characters of text where users have also scrolled at least twice. *Composing* flows were detected by searching for UIs with a deployed keyboard and with text input boxes that covered more than 60% of the remaining screen space. *Onboarding* flows were found by searching for the Android `ViewPager` component (generally seen as a sequence of dots on the screen) along with the words *get started*, *skip*, or *tutorial*.

Overall, we used element and layout based flow detection methods to identify 23 flow types that are popular in existing flow repositories and occur frequently in our dataset. 6.5% of the elements users interacted with in our dataset were indicative of one of these flows. On average, each app contained 3 such flow examples.

Flow Search Interface

Our pipeline for automated user flow identification allowed us to build — in two months — a repository with flows from 7× more apps than UX Archive, the most popular existing repository which has been in operation for four years [14]. We developed a web interface for searching and visualizing the user flows found in our repository (Figure 8), and made it available online at `http://interactionmining.org`.

The interface offers two ways to search for flow examples. Users can select the name of a flow from a list, and see search results from apps showing the relevant UI screen with the indicative element highlighted in red. Users can also use text-based queries to search for flows. These free-form queries return results based on the text contained in UI elements as well as the elements' `classname` and `id`, since developers frequently use descriptive names for these fields.

Clicking on a search result takes the user to the flow viewer interface. It shows the UI screen from the results page with the two previous and subsequent screens from the interaction trace, as well as the interactions the user performed on these

screens. It does not explicitly show the beginning or end of an example flow, but rather lets the user go back and forth in the trace to view more UI screens as desired. In addition to viewing a flow as a sequence of screenshots, users can also view animated GIFs of the UIs responding to the shown user interactions by hovering over the screenshots.

DISCUSSION AND FUTURE WORK

This paper demonstrates — for the first time — the possibility of mining user interactions and dynamic design data from mobile apps. Our system, ERICA, uses a web-based interface to allow crowdsourced data collection over the Internet. One important avenue for future work is to mine a more substantial portion of the available mobile apps, perhaps using crowd workers on platforms such as Amazon Mechanical Turk.

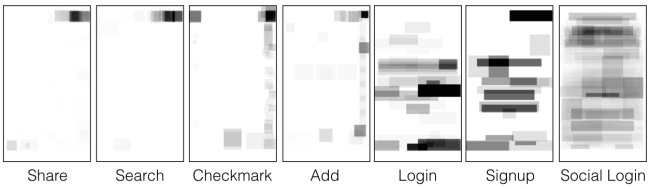
Another way to improve the scale of ERICA's repository would be to increase the coverage of UI states within each app. One way this might be accomplished is by combining multiple user traces. Figure 9 illustrates how coverage increased for three apps of varying complexity (measured by the number of Android activities found in their APK files) as multiple traces were combined. We observe that, like in heuristic evaluation, 5-8 users appear to provide optimal coverage [26]. For truly complex apps, coverage may remain low even after aggregating many user traces, since many UI states exist that humans do not visit during regular usage. Future work could explore augmenting ERICA with automated exploration strategies to visit these states.

While this paper focused on an application in UX design, interaction mining can be useful for building data-driven design tools targeted at other app design activities. The data produced by interaction mining can help designers understand UI layout patterns and motion details in existing apps. For example, Figure 10a shows heatmaps of common semantic element placements in UI layouts. Figure 10b shows animation curves of sliding drawer menus, inferred using motion detection techniques on the changing images of the UI.

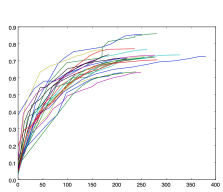
The UI data captured by interaction mining can also support learning probabilistic generative models of mobile designs [30]. Such models could enable automated mobile UI generation — useful for building personalized interfaces and retargeting UIs across form factors. The UI data produced by ERICA even has enough information about elements and layouts that it can be used to reverse engineer the source code of existing app UIs. Figure 10d shows an example of reverse engineering a UI from a popular app and rendering it on an emulator in a different form factor.

Another important application area for interaction mining is usability testing. Interaction mining can help designers discover usability bugs (such as users mistaking a UI screen to be scrollable in Figure 10c). Interaction data collected from a sufficiently large number of users could also enable summative usability testing for mobile apps without the need for source code modifications [19].

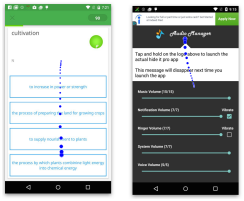
Outside of design, semantic understanding of apps enabled by interaction mining could improve the current metadata-based approaches to indexing and searching in online app stores.



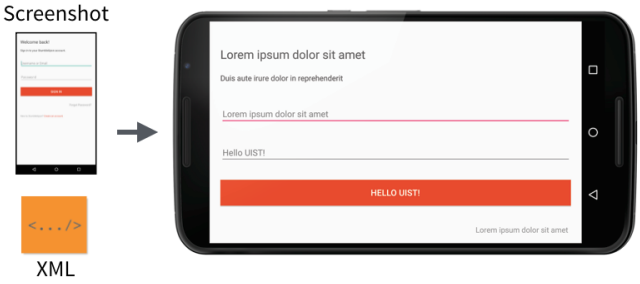
(a) Heatmap showing the locations of different kinds of semantic elements in UIs in the ERICA dataset.



(b) Animation curves for all the sliding drawer menus in the ERICA dataset.



(c) Usability bugs. Users tried to scroll on UIs that did not support scrolling.



(d) The representations of UIs captured by ERICA can be used to generate Android source code that recreates the UI. We reverse engineered a UI from the app *StumbleUpon* and generated the Android source code which is then rendered on an emulator with a different form factor.

Figure 10: Examples of data-driven applications enabled by ERICA in UI design, motion design, usability testing, and UI implementation.

For example, learning a similarity metric over apps based on the types of user flows they expose could enable more accurate labeling and clustering [18]. In addition, improved semantic understanding of mobile apps could enable automated identification of useful target states for deep-linking.

With Google's recent foray into web-based Android app streaming, mobile app usage through a web interface may become mainstream [3]. In such a world, an approach like ERICA could enable interaction mining at truly massive scale.

ACKNOWLEDGMENTS

We thank Erik Luo, Sujay Khandekar, Abhishek Harish, Stefanus Hinardi, Jinda Han, and Kedan Li for their assistance implementing different parts of this work.

REFERENCES

1. Review: UX Archive., 2013. https://uxmag.com/articles/review-ux-archive.

2. UI, UX: Who Does What? A Designer's Guide to the Tech Industry, 2014. http://www.fastcodesign.com/3032719/ui-ux-who-does-what-a-designers-guide-to-the-tech-industry.
3. New Ways to Find (and Stream) App Content in Google Search, 2015. https://search.googleblog.com/2015/11/new-ways-to-find-and-stream-app-content.html.
4. Tools for Mobile UX Design: Task Flows., 2015. http://www.uxmatters.com/mt/archives/2015/03/tools-for-mobile-ux-design-task-flows.php.
5. Android Activities, 2016. https://developer.android.com/guide/components/activities.html.
6. Android Hierarchy Viewer, 2016. https://developer.android.com/studio/profile/hierarchy-viewer.html.
7. Android UI Automator Framework, 2016. http://developer.android.com/tools/help/uiautomator/index.html.
8. Detecting Common Gestures, 2016. https://developer.android.com/training/gestures/detector.html.
9. Minicap: Stream Real-time Screen Capture Data Out of Android Devices, 2016. https://github.com/openstf/minicap.
10. Minitouch: Minimal Multitouch Event Producer for Android, 2016. https://github.com/openstf/minitouch.
11. Mobile Patterns, 2016. http://www.mobile-patterns.com/.
12. OpenSTF: Smartphone Test Farm, 2016. https://openstf.io/.
13. Pttrns, 2016. http://www.pttrns.com/.
14. UX Archive, 2016. http://www.uxarchive.com/.
15. Alharbi, K., and Yeh, T. Collect, decompile, extract, stats, and diff: Mining design pattern changes in Android apps. In *Proc. MobileHCI* (2015).
16. Bengio, Y. Learning deep architectures for ai. *Foundations and Trends in Machine Learning 2*, 1 (2009).
17. Bhoraskar, R., Han, S., Jeon, J., Azim, T., Chen, S., Jung, J., Nath, S., Wang, R., and Wetherall, D. Brahmastra: Driving apps to test the security of third-party components. In *Proc. SEC* (2014).
18. Chang, S., Dai, P., Hong, L., Sheng, C., Zhang, T., and Chi, E. H. Appgrouper: Knowledge-based interactive clustering tool for app search results. In *Proc. IUI* (2016).
19. Dray, S., and Siegel, D. Remote possibilities?: international usability testing at a distance. *Interactions 11*, 2 (2004).
20. Eckert, C., Stacey, M., and Earl, C. References to past designs. *Studying Designers 5* (2005), 3–21.
21. Gualtieri, M. Best practices in user experience (UX) design. 2009.
22. Kumar, R., Satyanarayan, A., Torres, C., Lim, M., Ahmad, S., Klemmer, S. R., and Talton, J. O. Webzeitgeist: Design mining the web. In *Proc. CHI* (2013).
23. Li, K., Xu, Z., and Chen, X. A platform for searching UI component of Android application. In *Proc. ICDH* (2014).
24. Machiry, A., Tahiliani, R., and Naik, M. Dynodroid: An input generation system for Android apps. In *Proc. FSE* (2013).
25. Mesbah, A., van Deursen, A., and Lenselink, S. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web 6*, 1 (Mar. 2012), 3:1–3:30.
26. Nielsen, J. Finding usability problems through heuristic evaluation. In *Proc. CHI* (1992), 373–380.
27. Rastogi, V., Chen, Y., and Enck, W. Appsplayground: Automatic security analysis of smartphone applications. In *Proc. CODASPY* (2013).
28. Sahami Shirazi, A., Henze, N., Schmidt, A., Goldberg, R., Schmidt, B., and Schmauder, H. Insights into layout patterns of mobile user interfaces by an automatic analysis of Android apps. In *Proc. EICS* (2013).
29. Szydlowski, M., Egele, M., Kruegel, C., and Vigna, G. Challenges for dynamic analysis of iOS applications. In *Open Problems in Network Security*. Springer, 2012, 65–77.
30. Talton, J., Yang, L., Kumar, R., Lim, M., Goodman, N., and Měch, R. Learning design patterns with bayesian grammar induction. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, ACM (2012).
31. Tian, Y., Nagappan, M., Lo, D., and Hassan, A. E. What are the Characteristics of High-rated Apps? A Case Study on Free Android Applications. In *Proc. ICSME* (2015).