

**A Deep and Longitudinal Approach to Mining Mobile  
Applications**

by

**Khalid Ahmed Alharbi**

B.S., King Abdulaziz University, 2007

M.S., University of Colorado Boulder, 2012

A thesis submitted to the  
Faculty of the Graduate School of the  
University of Colorado in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy  
Department of Computer Science

2016

This thesis entitled:  
A Deep and Longitudinal Approach to Mining Mobile Applications  
written by Khalid Ahmed Alharbi  
has been approved for the Department of Computer Science

---

Prof. Tom Yeh

---

Prof. Kenneth Anderson

---

Prof. Shaun Kane

---

Prof. Danielle Szafir

Date \_\_\_\_\_

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Alharbi, Khalid Ahmed (Ph.D., Computer Science)

A Deep and Longitudinal Approach to Mining Mobile Applications

Thesis directed by Prof. Tom Yeh

Modern software platforms feature digital distribution channels called marketplaces, which have revolutionized the way applications are developed and delivered to users. As the number of applications continues to proliferate in marketplaces, the need to fully understand them is ever increasing. While researchers have recently started to observe the wealth of information in marketplaces, their efforts have been largely constrained to one view of analysis and a single snapshot in time. As a result, the increasing number of application updates published to marketplaces has largely gone unobserved. Such view misses the much larger opportunity of mining applications with both a deep and longitudinal views and utilizing it to create innovative systems.

This dissertation introduces a new approach to analyzing large, ever-evolving marketplaces, such as the official Android marketplace, by taking a deep and longitudinal perspectives. To make this approach feasible, I designed and developed a scalable platform called *Sieveable*. Sieveable provides efficient retrieval of hundreds of thousands of applications with the goal of enabling a deep and longitudinal analysis of the design and development of mobile applications.

I demonstrate how Sieveable enables different types of analyses in three main areas that would have been very difficult to perform otherwise. In user interface design, the release of official design libraries enabled new applications to narrow the gap with the most downloaded ones in adopting best design practices. In accessibility, results showed that accessibility is a problem for many applications including the most downloaded ones. In privacy, the most added permissions in each year are the ones often required by ad libraries, which raises privacy concerns. The findings of this work offer insights to marketplace owners, platform engineers, and developers. I argue that considering both a deep and a longitudinal views results in a more useful analysis to support the design and development of mobile applications.

## Dedication

This dissertation is dedicated to my parents,  
Nora and Ahmed,  
and my family,  
Zaynab, Jana, and Rami.

## Acknowledgements

This journey would not have been possible without the support of my family, advisor, and friends. I would like to express my gratitude to my parents and my family. They are my biggest supporters. Thank you to my parents, Nora and Ahmed, who always believed in me and encouraged me to pursue my dreams. I'm grateful to my wife Zaynab, who always provided me with support and kept me happy.

There are a number of people who made this work possible but at the top of this list is my Ph.D. advisor, Tom Yeh, who took me as his first Ph.D. student, taught me how to write a research paper, carefully reviewed my code and design choices, and helped me with organizing my talks. His keen advice helped me identify research directions and come up with new approaches to problems. He got me involved in the publication process early on, encouraged me to submit my early work as works-in-progress, and assigned me as a reviewer in top conferences in my area. I recall the difficulties I encountered in my first year and how harsh academia felt, but he taught me how to deal with it in a positive way. I'm extremely lucky and grateful that I worked with him.

I would like to thank the members of my committee: Danielle Szafir, Shaun Kane, Kenneth Anderson, and Qin Lv. I would also like to express my appreciation to all my colleagues at the Sikuli lab, Jackson Chen, without him the size of my dataset would be much smaller, and Sanghee Kim, who helped me with setting up the RAID and the infrastructure to run the early stage of my research. Finally, I would like to thank King Abdulaziz University, which helped fund my research through a generous scholarship program, and Amazon for their AWS Cloud Credits for Research program, which helped cover my computing costs.

## Contents

<b>Chapter</b>	
<b>1</b>	<b>Introduction . . . . . 1</b>
1.1	Overview . . . . . 4
<b>2</b>	<b>Digital Marketplaces . . . . . 5</b>
2.1	The Actions and the Changes . . . . . 5
2.2	The Key Dimensions . . . . . 8
2.2.1	Time Resolution . . . . . 9
2.2.2	Inclusion Criteria . . . . . 9
2.2.3	Depth . . . . . 10
2.3	Summary . . . . . 11
<b>3</b>	<b>Related Work . . . . . 12</b>
3.1	Android Applications . . . . . 12
3.1.1	User Interface Layout . . . . . 13
3.1.2	Reverse Engineering Android Applications . . . . . 14
3.2	Data Mining . . . . . 15
3.2.1	Web Mining . . . . . 15
3.2.2	Mining Mobile Applications . . . . . 16
3.2.3	Mining Software Repositories . . . . . 22
3.2.4	Mining User Interface Data . . . . . 23

3.3	Design Aesthetics . . . . .	25
<b>4</b>	<b>Mining User Interface Design Pattern Changes</b>	<b>27</b>
4.1	Introduction . . . . .	27
4.2	Approach . . . . .	29
4.2.1	Collect . . . . .	30
4.2.2	Decompile . . . . .	30
4.2.3	Extract . . . . .	30
4.2.4	Stats . . . . .	31
4.2.5	Diff . . . . .	31
4.3	System Implementation . . . . .	31
4.3.1	Apps Crawlers . . . . .	32
4.3.2	Feature Extractors . . . . .	33
4.3.3	Data Stores . . . . .	33
4.3.4	Database Client Drivers . . . . .	35
4.3.5	Transformation Trackers . . . . .	36
4.4	Design Pattern Changes . . . . .	36
4.4.1	Custom UI Components . . . . .	36
4.4.2	Home Screen Widgets . . . . .	38
4.4.3	Tab Layout with TabHost . . . . .	40
4.4.4	Fragment . . . . .	42
4.4.5	Horizontal Paging . . . . .	43
4.4.6	Action Bar with Tabs . . . . .	45
4.4.7	Up Navigation . . . . .	46
4.4.8	Navigation Drawers . . . . .	48
4.5	Summary . . . . .	49

<b>5</b>	<b>Sieveable: A Scalable Platform for Mining Mobile Applications</b>	<b>51</b>
5.1	Introduction . . . . .	51
5.2	System Overview . . . . .	53
5.2.1	Requirements . . . . .	54
5.2.2	Approach . . . . .	55
5.3	System Implementation . . . . .	60
5.3.1	App Crawlers . . . . .	60
5.3.2	Feature Extractors . . . . .	61
5.3.3	Data Indexers . . . . .	62
5.3.4	Data Store Servers . . . . .	63
5.3.5	DOM Matchers . . . . .	63
5.3.6	RESTful API . . . . .	64
5.4	Summary . . . . .	64
<b>6</b>	<b>Deep Search Queries</b>	<b>65</b>
6.1	Introduction . . . . .	65
6.2	Search Queries . . . . .	66
6.3	The Retrieval Task and Formulating Search Queries . . . . .	68
6.4	Design Queries . . . . .	69
6.5	Security Queries . . . . .	71
6.6	Program Analysis Queries . . . . .	72
6.7	Summary . . . . .	74
<b>7</b>	<b>Temporal Analysis of the Design and Development of Mobile Applications</b>	<b>75</b>
7.1	Visual Design Mining . . . . .	75
7.1.1	Material Design Components . . . . .	76
7.2	Accessibility Mining Analysis . . . . .	83
7.2.1	Accessibility Violation: Labeling Visual UI controls . . . . .	84



7.3	Privacy Mining Analysis . . . . .	87
7.3.1	Analyzing Permission Usage . . . . .	88
7.3.2	Analyzing Added Permissions . . . . .	89
7.4	Summary . . . . .	92
8	Conclusion and Future Work . . . . .	93
8.1	Conclusion . . . . .	93
8.2	Future Work . . . . .	94
8.3	Summary . . . . .	95
	<b>Bibliography</b> . . . . .	96

## Tables

### Table

2.1	Several popular digital marketplaces for different platforms. . . . .	6
3.1	Summary of studies that involve mining of mobile application (continued on next page). . . . .	19
4.1	List of features extracted from each app. . . . .	29
4.2	The number of apps by release date grouped by quarters. . . . .	32
4.3	The changes to the presented design patterns. The columns show the name of the design pattern, the number of apps that used this pattern in the dataset, the number of apps that used it for the first time in a recent release, the number of apps that used it but later switched to a different design pattern, and the number of apps that maintained using it in future releases. . . . .	50
5.1	Sieveable's main extracted features. . . . .	54
7.1	Summary statistics on the total number of permissions requested by apps over the years. . . . .	90

## Figures

### Figure

2.1	The actions that a creator or a user may trigger to change the state of a single artifact. The creator may trigger these actions once, while the user may trigger the actions multiple times. . . . .	6
2.2	A chain of actions that a creator (C) or a user (U) may trigger to change the state of an artifact (A). The actions triggered by the creator result in a change to the state of an artifact or in a totally new artifact. . . . .	6
4.1	Architecture of the analytic system. . . . .	32
4.2	The distribution of apps by versions. . . . .	34
4.3	The download distribution of the apps by rating stars. Colors show details about the rating. . . . .	35
4.4	Two versions of an app titled “Business Insider” before (left) and after (right) adding the resizing functionality to its home screen widget. . . . .	39
4.5	Two versions of an app titled “DDLIVE”. The version on the left uses TabHost and the version on the right uses a Fragment that contains a list of items. . . . .	41
4.6	Two versions of an app titled “Shopping List” before (left) and after (right) adopting the Fragment design pattern. . . . .	43
4.7	Example of an app titled “JEFIT Workout Exercise Trainer” before (left) and after (right) adopting the Horizontal Paging design pattern. . . . .	44

4.8	Two versions of an app titled “PicsArt” before (left) and after (right) adopting the Action Bar pattern with Tabs. . . . .	46
4.9	Two versions of an app titled “Buscape” before (left) and after (right) adopting the Up Navigation pattern. The Up navigation button is shown at the top left corner as a left-facing caret next to the app icon. . . . .	47
4.10	Example of an app titled “Carango - Car Management” that shows two versions before (left) and after (right) using the Navigation Drawer pattern. . . . .	49
5.1	The total number of app versions in the collected dataset. . . . .	56
5.2	The total number of apps (multiple versions) in the dataset grouped by download count and the calendar quarter in which they are released. Three download count groups are used: top (1M downloads or more), middle (100K or more and less than 1M downloads), bottom (less than 100K downloads). . . . .	56
5.3	The execution sequence for a query that includes multiple level search conditions. . .	59
5.4	Sieveable system architecture. . . . .	61
7.1	The Floating Action Button (FAB) shown at the bottom right corner of two apps, Gmail and Lifesum . . . . .	77
7.2	The adoption rate of the Floating Action Button (FAB) over the years. . . . .	79
7.3	The percentage of apps that started adopting the Floating Action Button (FAB) using four popular libraries by calendar quarter. . . . .	80
7.4	An example of an app with a closed navigation drawer (left) and an open navigation drawer (right). . . . .	80
7.5	The adoption rate of the Navigation Drawer over the years. . . . .	82

7.6	The percentage of apps that started adopting the navigation drawer grouped by three download count groups: most, middle, and least downloaded apps. The x-axis shows the calendar quarter of the release date in which the adoption is observed, and the y-axis shows the percentage of apps that adopted it. The percentage value is computed by dividing the number of adopted apps in each quarter over the total of apps released in each quarter. The time of major platform events is also marked in the chart. . . . .	83
7.7	The average of accessible image buttons in apps over the release years. The error bars represent the standard error of the mean. . . . .	86
7.8	The percentage of apps that have no accessible ImageButton elements grouped by download count and release year. . . . .	87
7.9	The total number of requested permissions. . . . .	89
7.10	The total number of added permissions in apps grouped by download count. . . . .	90
7.11	The five most added permissions in each year. The x-axis shows how many times the permission was added in each year. . . . .	91

## Chapter 1

### Introduction

Modern mobile platforms feature a distribution platform for applications called marketplaces (or app stores). App marketplaces are online software distribution stores for developers to publish their apps for free or sell them, and for users to discover, purchase, download, and update apps. The popularity of mobile devices and the advances in their operating systems have led to significant increase in the number of apps published in marketplaces. For example, as of October 2016, the number of apps in the Google Play Store is over 2.4 million apps [14], which makes it the largest digital marketplace. These apps have become a valuable data source to mine and extract insight from in both academia and industry.

In the recent years, there has been a noticeable amount of research activities on how to extract meaningful insights from apps data. The research in mining mobile apps has been dominated by three single views. First, researchers have mined the meta-data or listing details data of apps such as ratings and user reviews to perform sentiment analysis and help developers make informed decisions supported by data [59, 47, 80]. Others have created tools and commercial services to assist app developers and publishers in a better understanding of listing details data [13, 12, 11]. Second, researchers have mined user interface data such as styles and layouts of thousands of apps to gain insights into their design patterns [123, 31]. Third, researchers have also mined the source code of apps to learn about malicious behavior and protect users' privacy-sensitive data [146, 98, 32]. What is missing in prior research is an approach that takes a deep, holistic view of the apps encompassing these three views.

This dissertation pursues a more deep search approach that takes a holistic view of the apps over time and can potentially accomplish what is currently not possible in a single view approach. In user interface (UI) design analysis, UI components are often created or modified at runtime. When only analyzing the static layout files, this observation is missed because that behavior is defined in the app source code. This shortcoming can be solved by combining both the design and code views. In sentiment analysis of user reviews, it is often difficult to link opinions to specific app features. By incorporating the visual view and code view, one can potentially establish a causal relationship between a new feature (or a bug) and the onset of certain opinions. In security analysis, a function could be determined, through program analysis, to be triggering a sensitive operation, such as sending an SMS message or taking a photo. But it is often hard to judge if the sensitive operation is warranted from the program view alone. By also taking a view of the design, one may examine which button may be linked to this sensitive operation and whether the button’s label legitimizes such use (e.g., a button labeled “Send” for sending an SMS message). Indexing apps to support multi-view data mining of apps is always challenging because it requires an infrastructure for integrating multiple heterogeneous data sources.

Mobile applications are always changing and increasingly updated at high rates. Most prior work in this area focuses on a single snapshot approach that only tells us about the moment of the observation. This single snapshot approach implies that all app updates and changes are gone unobserved. Such view misses the benefits of observing the changes to the design and development of mobile apps in response to major events. We cannot overlook the importance of the changes happening to mobile apps that are producing larger patterns and interesting insights. To observe and extract knowledge from these changes, one needs to track historical app releases. Collecting and analyzing mobile apps over time is challenging and requires building a scalable infrastructure to support analyzing a large amount of data. It is also hard if not impossible to collect the data and observe all the changes that occurred to mobile apps. However, recognizing the dynamic nature of mobile apps and the value of the generated historical data necessitates collecting it even if we cannot observe all the changes.

Over a short period of time, mobile user interface design has evolved to enhance the overall user experience. We observe changes to UI design guidelines, tools, and patterns at different points of time. A once popular design pattern may begin to decline in popularity. A new design pattern may be introduced with a lot of hype and promises but never gets widely adopted. A little-known design pattern may all of a sudden gains high popularity. These phenomena would have been missed, had we considered a single snapshot of the app that only tells us about the current moment. In mobile security, an app may start showing normal behavior and perhaps later in a new version starts to exhibit malicious behavior. Considering a single snapshot of the app would make security analysis less effective. In estimating the accessibility of mobile apps, it is also hard to gain useful insights with great implications from a single snapshot approach. One may attempt to quantify the prevalence of accessibility problems in mobile apps using the single snapshot approach. Such attempt will only tell us about the moment of the observation and will fail to provide insights on how did we get to this observation. When analyzing the same task but over time, we can uncover critical insights that help us understand whether these observations represent a state of improvement or deterioration. Unfortunately, the single snapshot view of analysis only tells us about the “current moment” and that’s not enough in today’s ever-evolving mobile apps scene. A longitudinal perspective that counts for all changes over time can tell us more than that and open myriad opportunities for further research in multiple areas.

To this end, this dissertation makes two major contributions. First, it presents a novel approach to mining large software marketplaces such as the official Android marketplace by taking a deep and longitudinal perspectives, manifested in a scalable retrieval platform called *Sieveable*. Sieveable indexed more than four hundred thousand Android applications at an unprecedented level of depth (listing details, user interface, and code data). Second, it demonstrates the utility of the approach taken by conducting diverse types of analyses that would have been difficult to perform otherwise.



## 1.1 Overview

This thesis is divided into 8 chapters. Chapter 2 discusses the main concepts and elements in a digital marketplace that drive changes. It presents a conceptual framework for mining digital artifacts in marketplaces. Chapter 3 reviews prior related work in the area of mining the web, mobile applications, and software repositories. Chapter 4 presents a pilot experiment for mining user interface design pattern changes in a small-scale dataset of Android apps. The challenges encountered during this pilot experiment led to the design of a scalable platform for mining mobile applications over time called Sieveable. Chapter 5 introduces Sieveable, discusses the technical requirements of designing the retrieval platform, and presents the process of indexing apps at multiple levels. Chapter 6 presents several illustrative search queries across multiple levels that demonstrate Sieveable's capabilities, and how it enables different types of deep analyses of mobile apps. Chapter 7 applies the presented approach to problems in mining mobile app design, accessibility, and privacy. Finally, chapter 8 concludes with a discussion of the main contributions of this thesis and discusses the future directions this work may take.

## **Chapter 2**

### **Digital Marketplaces**

To facilitate understanding the problem domain, it is necessary to briefly discuss the main elements in a digital marketplace and the key events that drive changes. In this chapter, I draw inspirations from the research in temporal data mining [29, 30] to better understand the phenomena of marketplaces. I define the main concepts and terminologies in the context of digital marketplaces and how various elements interact in this ecosystem generating multiple events. The goal is to highlight the broader impact and applications of the work presented in this dissertation.

#### **2.1 The Actions and the Changes**

Modern software platforms and technologies feature an online digital distribution platform called marketplace or store that allows users to discover and use digital artifacts made by third-party creators. This platform created a thriving community of creators and users, which led to an exponential growth over a relatively short period of time. In recent years, an increasing number of marketplaces enabled direct access of digital content or artifacts (e.g., software, multimedia files) to millions of users. Marketplaces often take the form of an online store for a specific platform that allows creators to sell or distribute their products to users. Table 2.1 shows a list of popular marketplaces for different platforms and the size of each marketplace [24, 26, 15].

Marketplace	Platform	Size
Pinshape.com	3D-printable design files	70,000+ makers and designers
AWS marketplace	Software and Cloud Services on Amazon Web Services	2,200+ software systems and services
Chrome Web Store	Extensions and apps for the Chrome web browser	18,000+ apps and thousands of extensions
Windows Store	Universal Apps on Microsoft Windows	669,000+ apps
iOS App Store	Mobile Apps on iOS	2,000,000+ apps
Google Play	Mobile apps on Android	2,200,000+ apps
Amazon Appstore	Mobile apps on Android	600,000+ apps

Table 2.1: Several popular digital marketplaces for different platforms.

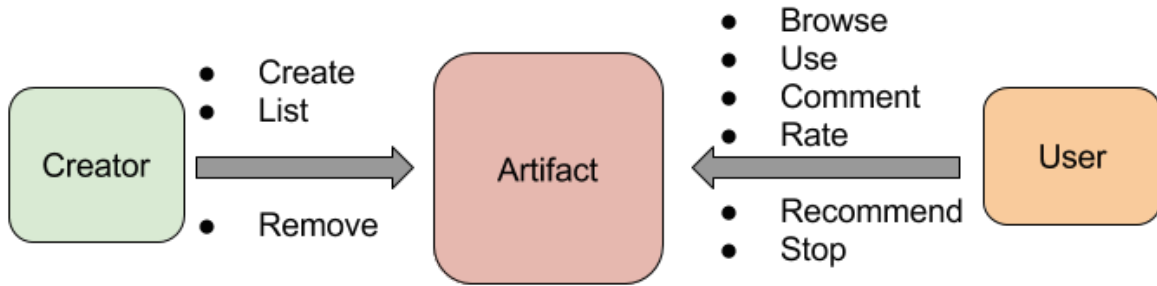


Figure 2.1: The actions that a creator or a user may trigger to change the state of a single artifact.

The creator may trigger these actions once, while the user may trigger the actions multiple times.

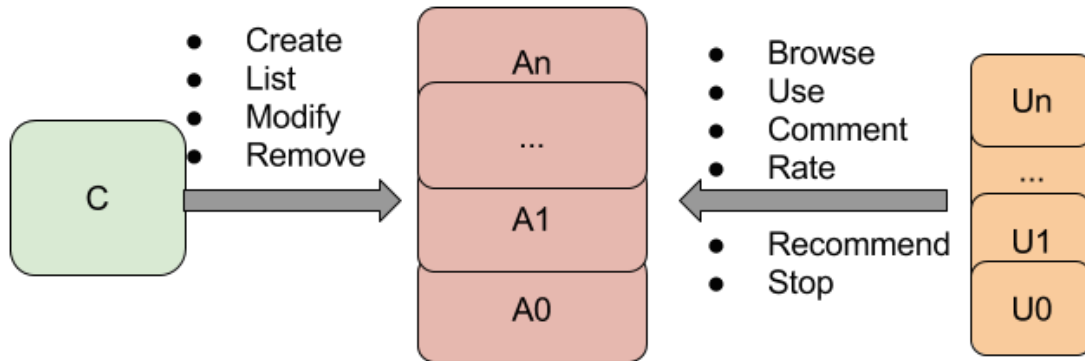


Figure 2.2: A chain of actions that a creator (C) or a user (U) may trigger to change the state of an artifact (A). The actions triggered by the creator result in a change to the state of an artifact or in a totally new artifact.

The popularity of marketplaces as a means of distributing and sharing digital mediums in various platforms led to a growing community. I define the key elements in this community as follows:

- (1) Creator (C): The person who is involved in the design, implementation, testing, and maintaining a digital artifact. A marketplace has a tuple of creators:  $C = \{C_0, C_1, \dots, C_n\}$
- (2) User (U): The person who interacts with the digital artifact produced by the creator (C). In a marketplace, we will have a tuple of users:  $U = \{U_0, U_1, \dots, U_n\}$
- (3) Artifact (A): Is a digital object made by a creator (C) to perform a task that benefits the user (U). In a marketplace, we will have a list of artifacts:  $A = \{A_0, A_1, \dots, A_n\}$

In addition, each of these elements is associated with custom actions that result in important events that describe or change the state of an artifact (depicted in figure 2.1). To understand the actions that may change the state of an artifact in a marketplace, let's consider a single artifact published at a marketplace by a single creator and used by multiple users. This artifact can be a simple calculator app released in a mobile marketplace at some point in time, downloaded by multiple users, but never received an update. This app represents the simplest form of the single snapshot view, where the time of observation is not considered important. Installing this app now will almost match the exact same view taken in its first and only version with the exception of a few properties (e.g., reviews and ratings). Thus, observing an artifact using the single snapshot approach will almost match the observation at the time of its inception. While marketplaces contain a considerable amount of artifacts that correspond to the single snapshot view, a significant amount of artifacts are ever changing. Hence, we need a different model to describe them.

The nature of the marketplace is described as a dynamic sequence of events that results in complex changes to a collection of artifacts. For instance, consider an app that was published in a marketplace by a single creator, used by multiple users, reacted to multiple events, and evolved over time receiving multiple changes. Figure 2.2 attempts to capture the factors and events that may contribute to the dynamic nature of the artifacts in the marketplace. In this case, the creator's

actions will often result in either a new artifact or a change to the state of the existing artifact, while the user's actions only result in a change to the state of the artifact (e.g., seen, used, recommended, etc.). The common between all of these actions is the fact that they are associated with a series of time data that indicates when each action occurred. A creator  $C$  may observe the actions of a tuple of users:  $\{U_0, U_1, \dots, U_n\}$  or react to external events which led to the changes or creation of new artifacts  $\{A_0, A_1, \dots, A_n\}$  at different times  $T = \{T_0, T_1, \dots, T_n\}$ . These time events are key to capturing and understanding the dynamic nature of marketplaces. There are various challenges involved in capturing and analyzing the dynamic nature of the marketplace. Perhaps the most difficult one is indicating which actions led to a change in the artifact. This stems from the uncertainty in the events that led to a change in the state of the artifact, which is often caused by the lack of data that convincingly interpret the event. The work and contributions presented in this dissertation aim to address the lack of data issue and motivate the creation of data-driven systems to answer complex analytical questions.

## 2.2 The Key Dimensions

Although Chapter 3 provides a detailed review of previous related work, it is worth noting here that the vast majority of prior work in this area falls under the approach of studying the first model I referred to as the “single snapshot” model. While prior work in analyzing the content of marketplaces has yielded valuable results, it is often limited to the “current moment” of the finding. The lack of investigating the dynamic and ephemeral nature of the artifacts in the marketplace leaves us unable to answer critical questions. For instance, we may not be able to answer questions like: How can we answer questions like: How did we get to this finding? Does this finding imply better outcome compare to the past? Can we predict if a different outcome is most likely to happen in the future?. By viewing the digital marketplace as a dynamic, ever-evolving system, it opens up a new space for designing and conducting novel analyses to gain insights about this system, insights that are not possible with a single-snapshot view. In this section, I describe the design space for such analysis methods for dynamic digital marketplaces. To the best of my knowledge,

this represents the first attempt to systematically map out this space. The major dimensions of this space are time resolution, inclusion criteria, and depth.

### **2.2.1 Time Resolution**

Given the large amount of artifacts in marketplaces, a large number of observations may be made at different time points. The first step in analyzing time series data is the selection of a specific frequency time span. The precision of selecting a measurement of time is highly specific to the data analysis task. This comes from the fact that different measurements of time-frequency result in different observations. In some cases, taking a longer measurement such as at each year or quarter will reveal patterns about slow changes. This approach could be useful in observing slow changes and reporting on trends over a relatively longer period of time. For example, when analyzing a list of artifacts in a marketplace, we can ask questions after taking a measurement at each year with respect to a specific design or development property. This could enable us to ask questions like what is the most popular design property in a given year? How popular is it in a different year? What was the increase in its popularity in another year? In some cases, adjusting the measurement to a shorter period of time (e.g., monthly, weekly) may be considered a convenient way to sharpen the observations. This short measurement is particularly useful in observing fast changes to digital artifacts in a marketplace. For instance, we can ask questions like: what is the most changing design or development property over a given period? what are the top artifacts that made changes to this property? To conclude, it is important to remember that the precise measurement of time may affect the outcome of the observations significantly.

### **2.2.2 Inclusion Criteria**

This dimension is concerned with the selection of a subset of artifacts from a marketplace for analysis. This problem has been studied extensively in statistics but it is worth to briefly discuss it here in the context of digital marketplaces. One might begin by taking the approach of collecting the entire artifacts in the marketplace (i.e. conducting a census). While it is obvious that this

approach is expensive and requires lots of resources, there are reasons that this may not be the best approach. The first reason is that access to the entire population is only possible to collect by the owners of the marketplace. The second reason is that the population of artifacts changes constantly in response to different actions and events, which means this approach will never result in a perfect measure of the population. The more reasonable approach, however, is to collect a sample of the entire population of artifacts. Sampling is easy to obtain and does not exhaust a lot of resources. However, sampling is often subject to error and bias. For example, suppose that we attempt to estimate the existence of a certain design property in the artifacts in a marketplace. If we collect a sample based on the top 10 artifacts from each category in a marketplace, then this sample would suffer from a selection bias because it is not a representative of the population. We cannot make a valid conclusion about a collection of artifacts (i.e. statistical inference) when our sample suffers from bias. Thus, it is important to conduct exploratory data analysis on the sample at hand to estimate the validity of the chosen sampling method. There are a few sampling methods to consider and ensure that our sample is a representative of the population. The commonly applied method is to randomly select artifacts from the marketplace such that each case is likely to be selected. For example, one might start by taking a random sample of artifacts in a marketplace based on a randomly selected search terms. If our sample does not suffer from sampling bias, we can generalize the observation to the entire population of artifacts. The chosen inclusion criteria of our sample must result in a representative sample of the population in order for the conclusion to be more reasonably valid.

### **2.2.3 Depth**

This dimension is concerned with the internal and external parts an artifact is composed of. A digital artifact consists of intrinsic and extrinsic parts that make up the artifact as a whole object. These properties are essential to understanding the changes to the state of the artifact. For example, consider a single digital artifact in a marketplace that has three part types: a) Listing information on the marketplace that promotes it and describes its purpose. b) Data that defines its

visual appearance and how a user may interact with. c) Data that describe its functional behavior. An important characteristic of the changes in a digital marketplace is that a change at one of these levels may trigger another change at a different level. For example, a developer may add changes to the functionality of the artifact. This change may require changes to the data of the visual appearance to make it visible to users. This change may also result in another change at the listing information level to promote the addition to users. Therefore, we need to consider all the parts that compose the artifact in order to understand the changes that occur to them over time and gain useful insights.

## **2.3 Summary**

In this chapter, I formally defined the digital marketplace, its key elements, and how changes take place. I discussed the limitations and problems that may arise when considering a single snapshot approach. I described how artifacts can be analyzed using a deep and longitudinal approaches.



## **Chapter 3**

### **Related Work**

This chapter provides a brief background on Android applications and an overview of the technologies related to reverse engineering Android applications. It also discusses the relevant work in the area of mining the web, mobile applications, and software repositories to motivate the discussion on the work presented in this dissertation.

#### **3.1 Android Applications**

Android applications are often written using the Android software kit (SDK) in the Java programming language. In addition, parts of the application code can be written in native languages such as C and C++ using the Native Development Kit (NDK). Android provides a rich framework with various APIs for third party developers and a set of tools to build and compile the application source code along with its resources into an Android package file (APK). The APK file is an archive file in ZIP format with a .apk file extension. An application developer generates a certificate to digitally sign the application before releasing it on the marketplace for Android devices. When publishing the app to the Google Play store, the official marketplace, the developer provides information for the store listing to describe and promote the application. Users can search for applications in the marketplace and install the APK file to their devices. Applications in the marketplace are uniquely identified using a package name field. Android uses the Java package naming conventions to uniquely identify applications by their package names. In general, developers use a package name that begins with their reversed Internet domain name (e.g., com.airbnb.android).

Developers can update the APK file with a newer version and change the store listing details page at any time. The Google Play store only offers the most recent version of the app and does not keep previously submitted versions. However, some unofficial or third-party marketplaces maintain an archive for all versions of particular apps. In addition to package names, Android requires applications to define version information, so it can be used when releasing or upgrading the APK file. Android uses two version values. a) version name, which is a string value that represents the application release version and is visible to users (e.g., 1.2.0), and b) version code, which is an integer value that is not visible to the users but used by marketplaces including the Google Play store to check for version updates.

### **3.1.1 User Interface Layout**

The user interface (UI) of Android applications is built using two basic UI elements, View and ViewGroup elements. Views represent a single UI component (e.g., input elements such as text view, button, etc.), while ViewGroups are the invisible containers that group View elements (e.g., LinearLayout, RelativeLayout, etc.). UI elements are usually defined in static XML layout files. The app content (e.g., text, images, animations, etc.) is usually stored in XML files within special directories and embedded using special XML tags and attributes. The Android framework parses the layout XML files into a Document Object Model (DOM) tree, performs pre-processing on the tree at build time, and inflates the screen with the visual rendering of the layout. The XML layout files provide a complete specification for the Android framework engine to render the user interface. The UI of Android applications can be constructed entirely at runtime; however, for most applications the UI is implemented in static XML layout files, and Java code is used to add content or enhance interactive UI elements. For example, to implement a navigation drawer, the developer would need to define the drawer layout, initialize the drawer list, and handle navigation click events (see listing 1).

```

$ main_layout.xml
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <FrameLayout
        android:id="@+id/content_frame"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
    <ListView android:id="@+id/main_drawer"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#111"/>
</android.support.v4.widget.DrawerLayout>

$ MainActivity.java
public class MainActivity extends Activity {
    private String[] mMenuItems;
    private DrawerLayout mDrawerLayout;
    private ListView mDrawerList;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mMenuItems = getResources().getStringArray(R.array.menu_items);
        mDrawerLayout = (DrawerLayout) findViewById(R.id.main_layout);
        mDrawerList = (ListView) findViewById(R.id.main_drawer);

        mDrawerList.setAdapter(new ArrayAdapter<String>(this,
            R.layout.drawer_list_item, mMenuItems));

        mDrawerList.setOnItemClickListener(new DrawerItemClickListener());
    }
}

```

Listing 1: A snippet of an Android drawer menu defined in an XML file and initialized in Java code.

### 3.1.2 Reverse Engineering Android Applications

According to the taxonomy of Chikofsky and Cross [48], reverse engineering is defined as “the process of analyzing a subject system to identify the system’s components and their interrelation-

ships and create representation of the system in another form or at a higher level of abstraction.” The analysis presented in this thesis is performed on third-party, closed-source, binary Android applications. In order to expose the app’s structure, we make use of a reverse engineering tool called *apktool* [10] to unpack APK files and decode their resources to their nearly original formats. Running *apktool* on an APK file results in a directory tree that makes up the app. The tool disassembles the bytecode into *smali* files, which uses an assembler-like syntax based on Jasmin [20], a largely used assembly format for the Java Virtual Machine (JVM).

## 3.2 Data Mining

Data mining refers to the process of extracting previously unknown information from data and discovering new patterns in the data [140]. Data mining techniques have been applied to find structural patterns in large-scale data from various domains such as business intelligence, fraud detection, and surveillance. This section describes the most relevant data mining application domains to the work presented in this thesis: Web mining, mining mobile applications, and mining software repositories.

### 3.2.1 Web Mining

Web mining is an area of research that involves the use of data mining techniques to automate the discovery and extraction of information from the World Wide Web [55]. The use of data mining techniques has proven to be a powerful approach for extracting knowledge and detecting new patterns from large collections on the Web. The web mining research can be classified into three main categories: content mining, structure mining, and usage mining [100, 81].

Web content mining involves the use of various techniques due to the different kinds of content on the web (e.g., text, images, and videos). Research in mining text-based content such as news articles often represents the unstructured text documents as a bag of words or vector representation, n-gram, phrase based, or term-based representation [102]. The applications of mining text content include text classification, clustering, and patterns discovery. The study of mining text-based

content over time is called Temporal Text Mining (TTM). It is concerned with discovering patterns in temporal documents and has many applications such as events tracking [63], summarizing [105], and detecting [70]. The Web is regarded as the largest database of images and videos. Advances in computer vision and image processing techniques have enabled data mining researchers to use the extracted features to discover new patterns. [138, 141].

Structure mining research focuses on extracting information from the underlying hyperlinks graph structure of the Web itself. Applications of web structure mining include Web pages ranking, categorization, and community discovery. A number of algorithms have been proposed to model the graph structure of hyperlinks and rate Web pages based on quality or importance factors. Examples include the Hyperlink-Induced Topic Search (HITS) algorithm [78], PageRank [40], and Clever [44].

Web usage mining involves collecting and analyzing server and browser logs data that result from users interacting with Web pages. Researchers have used different kinds of data in Web usage mining [129]. User's registration data are used to provide demographic information about the users interacting with Web pages. Click-stream data is a sequential series of page clicks and used to provide insights into the path the user takes when navigating through Web pages. The applications of mining usage data include modeling user profiles, collaborative filtering, recommender systems, and discovering navigation patterns.

### **3.2.2 Mining Mobile Applications**

Mining mobile apps started to be of interest to researchers ([53, 54, 99, 49, 58, 146, 144, 114, 107, 112, 61, 47, 96, 90, 116, 95, 125, 143, 34, 46, 94, 134, 113], see Table 3.1). Harman et al. [64] conducted one of the earliest studies on mining and analyzing app store data. They mined app meta-data (listing details data) of over 32,000 apps in the Blackberry app store to find patterns such as a correlation between app rating and download count. Wei et al. [139] developed ProfileDroid, a monitoring and profiling system for analyzing the behavior of Android apps. The system statically analyzes APK files (bytecode only) and dynamically collects log data for system events and network

traffic. They uncovered behavioral characteristics among apps such as most of the network traffic is not encrypted and travel to third-party servers. Fu et al. [59] conducted a sentiment analysis on the reviews of over 170,000 mobile apps to identify the common reasons why users like or hate an app. They collected listing details data and analyzed the reviews at multiple granularities. They found that the lack of attractive design was the largest cause of negative reviews. However, only listing details data were considered and no design and code data were analyzed. Viennot et al. [136] developed the largest scale crawler for Android apps, crawling over 1,000,000 apps and analyzing a subset of their listing details and source code to identify library usages and detect dangerous security practices. Despite the immense scale, they stored the listing details and source code files as plain-text data and used a full-text search engine to query them. This greatly restricted the applicability of their approach to keyword search and left structured data search unsupported (e.g., user interface layout data). Shirazi et al. [123] collected 400 Android apps to analyze the common design patterns of these apps. They estimated the complexity of each app’s design by counting the number of activities, layout files, and images. They computed descriptive statistics such as the most frequent interface elements and the most common combination of widgets. Minelli and Lanza introduced SAMOA [107], an analytic platform for mobile apps with the goal to understand the complexity of mobile apps and third-party API usage. The platform was among the first attempts to analyze apps at depth (source code structure and listing details) over time. However, the platform does not utilize user interface data and used a small dataset of 20 open source apps.

In order to deal with the recent increase in the number of mobile apps, researchers have studied them for various data mining and machine learning applications. Chen et al [46] presented SimApp, a framework for detecting apps similarity based on listing details data using a kernel-based learning algorithm. Lin et al. [91] proposed a framework to improve app recommendation by incorporating version histories with standard recommendation techniques. Zhu et al. [147] proposed a ranking fraud detection system based on historical app ranking data to detect fraudulent means that could boost the app’s rating in the marketplace. AppJoy [142] is a personalized recommender system for Android apps. The system computes a “usage score” based on the actual user’s usage and

uses that as an input for a collaborative filtering algorithm to make personalized recommendations.

Finally, I note two recent research efforts that have investigated interesting changes in mobile apps over time. First, McIlroy et al. [104] studied the update frequencies of mobile apps. They tracked 10,713 apps over a period of 47 days. They found that 14% of the studied apps are updated on a bi-weekly basis and around 45% of the top 100 most updated apps do not include a rationale for the update in the "What's new" section of the listing details data. Second, Martin et al. [103] tracked the releases of 26,339 apps over a year to find the releases with the most impact in ratings. They found that releases that describe bug fixes or new features in their release note (the "What's New" section) tend to have a more positive impact in ratings.

Table 3.1: Summary of studies that involve mining of mobile application (continued on next page).

Domain	Study/System	Method/Purpose	Depth	Scale
Privacy	TaintDroid by Enck et al. [53]	An information flow tracking system that provides realtime monitoring of privacy sensitive data leaked by applications.	Code	30
	PEDAL by Liu et al. [95]	A system that enables users to control inherited permission to ad libraries, so they can grant a permission to the app to function but not the ad component within the app itself.	Code	60,000
	Seneviratne et al. [125]	Extract listing detail features for a list of user installed apps to predict user's gender.	Listing details	4,167
	Lin et al. [90]	A static code analysis approach to analyzing requested permissions and determining which ones are needed for the app's core functionality and the ones needed for sharing sensitive information with ad libraries. They leveraged crowdsourcing to collect privacy preferences and identified four user privacy profiles.	Code and Listing details	108,246



	SUSI by Rasthofer et al. [116]	An automated machine learning approach for classifying sources of sensitive data (e.g., location) and sinks of potential channels through which data may leak to an adversary (e.g., a network connection) from Android API methods.	Code	11,000
Security	DNADroid [49]	A tool for computing similarities between apps to detect illegitimate app clones in multiple marketplaces.	Listing details and code	75,000
	Pandita et al. [112]	Use NLP techniques to detect whether the app description indicates the app needs to use a particular permission.	Description, API usage, and Manifest file.	581
	Gorla et al [61]	Analyze and cluster apps by their descriptions and API usages to detect outliers.	Description and API usage	22,500
	The ded decompiler by Enck, et al. [54]	A decompiler to recover source code from the binary file. They used it to perform security analysis and discovered suspicious behavior that links to misuse of personal information by the app and ad libraries.	Code	1,100
	Zhou et al.[146]	A classification of Android malwares into 49 different families.	Code	1,260

Software Engineering	Chen et al. [47]	Analyze app user reviews to extract valuable information for developers.	User re-views	4
	SAMOA [107]	An analytic Platform for mobile apps to analyze the structure of source code over time.	Listing details and code.	20
	Yang et al. [143]	Control-flow analysis system for Android. It generates a callback control-flow graph that can be used for analysis applications such as software and GUI testing.	Code	20
	PerfChecker by Liu et al. [96]	A static code analyzer to detect performance bugs	Listing details and code	29
Machine Learning	Baeza-Yates et al. [34]	A model for predicting the next installed app the user is going to use. The goal is to improve the usage of home-screen/launcher applications.	Log usage data	70,000
	Chen et al [46]	A framework for detecting similar apps using an online kernel algorithm.	Listing details	21,624
	Liu et al.[94]	App recommender system that incorporates both apps' functionalities and users' privacy preferences.	User re-views	6,157
	Yin et al. [144]	A recommender system for recommending new apps to replace already installed ones.	Description	5,661

Data Mining	Park et al. [113]	An information retrieval approach that leverages user reviews with app description for mobile app retrieval.	Listing De-tails	43,041
	Frank et al.[58].	Cluster apps by permissions to find patterns in high-reputation and low-reputation apps.	Listing De-tails	188,389
	Ma et al.[99]	Mining context logs of mobile app users to identify similar patterns.	context logs	logs from 443 users
	Petsas et al. [114].	A study in apps' popularity trends and revenue strategies across four third-party Android marketplaces.	Listing and code.	300,000+

### 3.2.3 Mining Software Repositories

Software repositories such as version control, bug tracking, and team communication systems have received a lot of attention in recent years. Mining software repositories (MSR) is an emerging research area concerned with finding patterns in large software repositories. It has seen a wide range of topics including detecting code redundancy [75, 42], finding relevant code examples [69, 124, 101, 131], bug-introducing changes [77], bug fixes [76, 111], mining software changes [148, 118], identifying hard to change code (code decay [52]), and finding common idioms [56].

Maintaining large collections of open source software repositories and indexing them has recently become an active research area (e.g., [35, 41, 88, 130]). For instance, Boa [51] is a domain specific language for mining software repositories on a large-scale infrastructure with the goal of reducing the efforts of writing analyses tasks and reproducing practical mining experiments.

### 3.2.4 Mining User Interface Data

User Interface design is often considered hard and challenging. Designers use various methods and techniques during the creative design process including: a) The use of paper prototypes [109, 79]. b) The use of storyboarding and wireframing [109, 137]. c) The use of reusable design solutions (e.g., templates, design patterns) [73, 60, 122]. d) The use of interactive sketching tools [85, 89, 110, 126]. e) The use of inspirational curated examples [115, 67, 87, 122, 106]. Several lines of research were developed around these methods and largely used small controlled studies to evaluate their effectiveness.

Prior research has studied the use of examples to draw inspirations and aid designers in the design process. The applications of using design examples include exploring alternative designs, seeking creativity, and getting an inspiration [115, 67, 87, 122, 45, 106]. Herring et al. [67] studied how designers use examples and the difficulties they faced when searching, sharing, and using design examples. Miller et al. [106] studied how professional designers find design examples. They found that designers had several difficulties to find what they are looking for and turning that into a search query. This forces designers to use search terms that are not directly related to what they are looking for. For instance, a search query for the term “large navbar” usually returns results to question-and-answer websites or tutorials that describe the implementation details of “navbars”. These findings suggested the need for better tools and retrieval systems to support designers in retrieving example.

Attempting to address this pressing need, data-driven approaches have been applied to identifying design examples. HCI researchers have realized that in order to increase the usefulness of design examples to designers, a new generation of design based search engines needs to be developed. This manifested from the limitations of existing search engines that are designed to deal with text rather than visual structure.

Several tools have been developed to aid designers in finding and using a curation of design examples. The Adaptive Ideas tool [87] is a browser extension that enables designers to view

design examples of a manually curated corpus of 250 web design examples. It allows designers to borrow design elements from multiple design examples while designing their websites. D.tour [122] is a search tool for finding web design examples. The tool consists of a curated database of 300 web pages. Users can search for similar design examples or use textual design terms. WebCrystal [45] is a tool that adds visualization and text to explain how design examples are constructed allowing novice web designers to use them in their own web pages. Bricolage [84] is an algorithm for retargeting content between web pages. It creates a mapping between the visual elements of web pages allowing designers to transfer the style from one page to another page. Bricolage trains a model on a corpus of design mapping of 50 web pages collected from crowdsourced workers to automatically transfer the design between web pages. Webzeitgeist [83] is a retrieval system for mining design data. It provides a custom JSON-like design query language (DQL) that can be used to find design examples by the visual appearance and the DOM structure of web pages.

The HCI literature has also examples on the use of crowds to complement the data-driven approaches in finding more relevant design examples [84, 128], understand aesthetic preferences [121], and design demographics [120]. Reinecke et al. [121] collected 450 web pages and obtained subjective ratings from 548 volunteers about the visual complexity and colorfulness of these pages. They were able to develop computational models that accurately measure the perceived visual complexity and colorfulness of website screenshots. Extending this work, Reinecke and Gajos [120] collected 2.4 million ratings from 40,000 diverse participants for 430 websites to identify the demographical factors that influence visual preferences. They developed a computational model that predicts users perception of visual aesthetics for specific demographic group.

The web search and data mining literature had a different goal when dealing with design data, removing design data to improve the accuracy of information retrieval systems. Design data is often considered unclean data that pollutes the content; thus, research has been conducted to detect and remove them to improve page ranking, indexing, and data mining algorithms [36, 60, 43]. Researchers have also developed algorithms to extract web page templates and understand their evolution. These algorithms primarily target web browsers (e.g., template caching), search engines

(e.g., improve indexing), and data mining applications (e.g., analytic tools). For instance, Gibson et al. [60] extracted templates from web pages and measured their prevalence. They found that templates are counted for 40-50% of the size of web pages. They further studied changes to the number of templates for 183 websites over a period of 8 years and showed that it is growing by 6-8%.

### 3.3 Design Aesthetics

Early on, researchers realized the need to formally develop design guidelines to improve the overall appealing of graphical user interfaces (GUIs) [62, 68]. Platform owners and organizations also developed their own design guidelines with recommendations to solve problems like usability and inconsistency in GUIs and promote a good visual design appealing [28, 25, 19, 16]. While design guidelines are sometimes useful to designers, it is now acknowledged that design guidelines are vague, conflicting, and difficult to apply [38, 50, 97, 72]. In addition, there is no common agreement on what constitutes valid design guidelines [117]. Research efforts have moved to studying how existing web design examples influence users in many ways: visual aesthetics preferences [119, 121, 120], perceived trustworthiness [37, 92], usability [62, 133, 82, 135], quality [66], and satisfaction [132].

Researchers have conducted a series of controlled laboratory experiments with users and professional experts to understand the qualitative factors that influence users when judging the appearance of a website [132, 86, 66]. Lindgaard et al. [93] conducted three lab studies to find how quickly users assess the attractiveness of website design and found that participants were able to judge the design within 50 milliseconds. Hartmann et al [65] conducted a small controlled lab study to evaluate the attractiveness of websites by applying the adaptive decision-making theory. They collected subjective ratings from 43 participants for only 3 websites and found that participants background influenced their rating of website design quality. Researchers have also applied different techniques to measure web design aesthetics. For instance, Ivory et al. [72] used a quantitative method to compute 11 web page attributes metrics (e.g., number of fonts, images, and words) for

1,898 web pages from 163 websites. The web pages were manually obtained from a list of websites awarded an award for excellence in web design. They achieved an accuracy of predicting 65% of the award judgment ratings. Ivory et al. [71] expanded upon this work by adding more 146 attribute metrics, increasing the number of samples to 5,300 web pages, achieving a higher accuracy of 94%, and creating a profile of good and bad website design.

Advances in computer vision algorithms enabled researchers to apply pixel-based methods to analyze the design of websites and develop computational models to predict users judgments. Zheng et al. [145] computed low-level image statistics to predict users' judgments for 30 web pages. They computed layout structures and evaluated them with human ratings collected from a study with 22 participants. Reinecke et al. [121] collected 450 web pages and obtained subjective ratings from 548 volunteers for the visual complexity and colorfulness of these web pages. They were able to develop computational models that accurately measure the perceived visual complexity and colorfulness of website screenshots. Finally and most recently, Miniukovich and De Angeli [108] developed a tool for GUI aesthetics evaluation based on eight metrics of GUI aesthetics.

## Chapter 4

### Mining User Interface Design Pattern Changes

Mobile user interface (UI) design patterns have been widely used across different mobile platforms. UI design patterns have evolved and changed significantly as new trends emerge and fade at different times. This chapter presents a pilot experiment for mining design pattern changes in Android apps. Over a period of 18 months, I tracked 24,436 apps and collected their versions. In total, the sample consists of 56,349 unique app versions, more than 5 million source files, and more than 25 million UI elements. The work presented here is heavily dependent on custom scripts to extract features to support the differential analyses regarding design pattern changes.

This work is the result of collaboration with Tom Yeh ([31]).

#### 4.1 Introduction

UI design patterns are general, reusable solutions to common design problems. Take, for example, the problem of organizing menu items on a small screen of a mobile device. A number of design patterns have emerged as useful solutions to this problem. For instance, Android design guidelines feature a number of design patterns such as the use of “Navigation Tabs” pattern [2] and the “Navigation Drawer” pattern [23]. Design patterns are valuable to both third party app developers and UI framework engineers. UI framework engineers invest a substantial amount of time and effort in building new design patterns for their platforms to enable third-party app developers to enhance the UIs of their applications. Third-party developers make difficult decisions when choosing a particular design pattern to communicate their ideas in a way that pleases their users.



A good design pattern often enjoys wide adoption by developers and acceptance by users. An interface following good design patterns is familiar to users, consistent with users' expectations, and easy to learn. A bad design pattern would have the opposite effects.

Design patterns change, evolve, or in some cases, die out over time. A design pattern once popular may begin to lose popularity to a better alternative. A once obsolete pattern may experience a resurgence in adoption. A new design pattern may be introduced with a lot of hype and promises but never go on to wide adoption. A little-known design pattern may all of a sudden explode in popularity. These phenomena may have important HCI implications. But there has not been a large-scale comprehensive study on changes in design patterns.

The design guidelines of mobile applications have changed significantly with new design patterns, UI elements, and styles to improve the overall visual design of mobile apps. UI framework engineers add new widgets for complex UIs, implement new APIs for them, deprecate previous widgets and UI related APIs, and add new visual design patterns to help developers build beautiful applications. How do we know how many apps have made the switch to a particular design pattern and maintained the usage across future releases? It is hard to quantify the adoption rate of these design patterns using existing approaches.

This chapter presents a data-driven approach to studying design pattern changes on a large scale. We applied this approach to the Android framework and present our findings here. Our approach consists of five steps:

- (1) Collect a large number of apps. Continue to download their subsequent updates.
- (2) Decompile each app into code that can be analyzed to understand the app's UI design (e.g., XML) and actual programmed behaviors (e.g., `void onClick()`).
- (3) Extract a comprehensive set of features about each app from the app's listing details web page, user interface layout, and actual code.
- (4) Stats: Compute statistics (e.g., distribution, min., max, mean, outliers) with respect to a feature of interest.

<b>Listing Details Features</b>	package name, title, description, reviews, store URL, category, price, date published, version name, version code, target system version, ratings count, rating, content rating, creator, creator URL, install size, downloads count, permissions, what's new.
<b>Appearance Features</b>	layout directories, layout files, view group containers, view elements, relationships, drawable resources, UI text resources.
<b>Behavioral Features</b>	app framework invocations, manifest (AndroidManifest.xml), third-party libraries.

Table 4.1: List of features extracted from each app.

- (5) Diff: Compare two versions of an app and compute their differences. Identify common and unusual change patterns.

Our approach provides new insight into design pattern changes in Android apps. For example, there are five common design patterns for navigation: Tab Layout, Fragments, Horizontal Paging, Up Navigation, and Navigation Drawers. Which of these are more common? Have any apps made a switch between two pattern versions? What are the most common switches? How prevalent are they? What apps use an unusually large number of custom widgets? Are there apps switching from custom widgets to built-in widgets? These are just a sample of questions our approach was able to address.

We present our approach in detail, describe a system that supports our application of this approach to the Android framework, and finally present eight analyses of design pattern changes to demonstrate the usefulness of our approach.

## 4.2 Approach

Our approach consists of five steps: collect, decompile, extract, stats, and diff. We choose the Android platform as an application and explain how we carry out each step to analyze design pattern changes.

### 4.2.1 Collect

The first step is to collect a large sample of apps and extract the visual structure of their user interfaces. We download apps from the official marketplace, Google Play store, and crawl their listing details web pages. Moreover, in order to analyze changes, we need to continue to monitor these apps for updates and download them.

### 4.2.2 Decompile

The second step is to decompile the user interface program to expose its “code” portion so that the actual programmed behaviors can be subject to analysis. This step involves running an “unpack” tool to open each app’s Android Application Package (APK) file to obtain a set of design layout files and a “disassembler” tool to obtain the app’s byte code.

### 4.2.3 Extract

The third step is to compute a rich set of features to describe each app at three levels. First, at the listing details level, we find descriptive information about a GUI from where the GUI may be listed, promoted, or reviewed. Second, at the appearance level, we gather data that defines the look and feel, content, and structure of a GUI. Third, at the behavioral level, we examine the decompiled code to gain insight into the actual programmed behaviors of a GUI. We mine the Google Play store to obtain a comprehensive set of listing features, such as the title, price, ratings, install size, and what is new. We parse the manifest file, the layout files, and the string definition files to extract appearance features such as the use of custom components and the relationships between components. Finally, we apply program analysis to the byte code to extract the behavioral features such as the use of GUI related APIs that dynamically change the GUI. Table 5.1 gives a comprehensive list of the features we considered on the three levels.

#### 4.2.4 Stats

After extracting a rich set of features about the apps in the corpus, the fourth step is to conduct statistical analysis about them. The most common analyses would be to compute the average, min, max, and histogram for quantitative features. For categorical features, counting and distribution often yield useful insights. For text features, one can compute the most frequent words. Complex analyses are possible at this step, such as correlations, clustering, detection of outliers, and sentiment analysis. In the application of design pattern changes, we carry out a range of statistical analyses like “what’s the percentage of apps using a given design pattern?” and “what’s the percentage of apps that switch to a different design pattern?” Although computing descriptive statistics on UI design pattern changes seems simple, it was not even possible before our approach.

#### 4.2.5 Diff

The final step is to compare two subsequent versions of an app and identify the aspects that have been updated. Usually an update contains changes to certain aspects of an app’s GUI design. Sometimes changes are obvious, such as design overhaul. Sometimes changes are subtle, such as rewording the caption of a button. We pay attention to changes that occur in the listing, the appearance, and the behaviors. For instance, an app may add a new screen to support a new feature. This change can be reflected at all levels, including description in the “what’s new” section promoting the new feature, an extra layout file defining the new screen, and an extra function calls in the app’s code to implement the new features. Moreover, we consider differences at the group level, by asking, for example, what are the most common design patterns dropped in a collection of apps?

### 4.3 System Implementation

On a small scale, the five steps in our approach are easy to carry out. But they quickly become challenging when the scale goes up to a level of tens of millions of layout and program files.

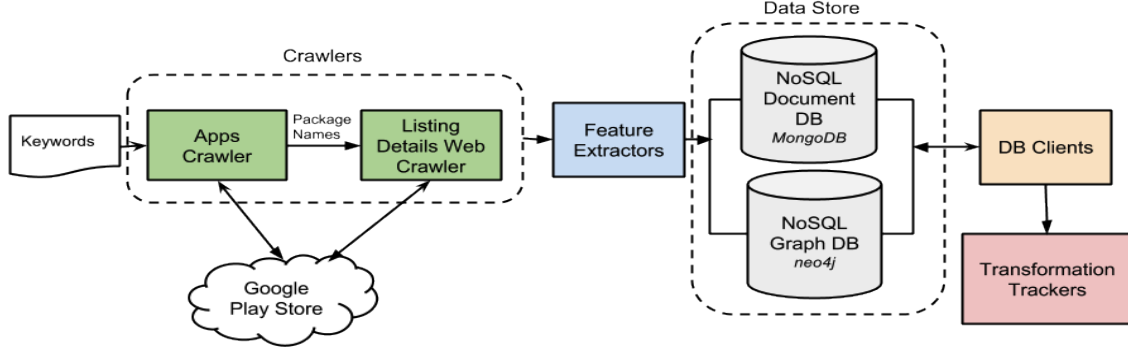


Figure 4.1: Architecture of the analytic system.

2010 Q2	2010 Q3	2010 Q4	2011 Q1	2011 Q2	2011 Q3	2011 Q4	2012 Q1	2012 Q2	2012 Q3	2012 Q4	2013 Q1	2013 Q2	2013 Q3	2013 Q4	2014 Q1	2014 Q2
2	5	11	14	35	33	65	127	206	406	758	1,600	3,697	14,969	17,326	12,718	4,377

Table 4.2: The number of apps by release date grouped by quarters.

There is no off-the-shelf, ready-to-use system to support large-scale differential analyses to answer our questions regarding design pattern changes. Therefore, we designed and implemented a system dedicated to supporting our analyses. The technical detail of this system is presented here.

The system consists of five components (see Figure 4.1): Apps crawlers, feature extractors, data stores, client drivers, and transformation (changes) trackers. The apps crawler downloads free apps from the Google Play Store and saves them to the file system. Feature extractors are a rich tool chain that decodes apps, extracts features, and stores them in the data stores. The client drivers handle all interaction with the data stores. The change trackers are a rich tool chain that tracks and computes statistics on changes to the extracted features.

#### 4.3.1 Apps Crawlers

Our system utilizes two custom crawlers we built: the apps crawler and the listing details web crawler. The apps crawler maintains a list of keywords, crawls the Google Play Store, and returns a list of package names for free apps. We used Google-Play-Crawler<sup>1</sup>, an unofficial open-source

<sup>1</sup> <http://github.com/Akdeniz/google-play-crawler>

Java API for the Google Play Store, to retrieve package names and download APK files. Prior releases of apps are not available on the Google Play Store. Thus, the apps crawler obtains the version code value for each retrieved package name and queries the data store to check if it exists. If it does not already exist in the repository, the crawler downloads and saves the APK file in the data store.

Once the APK file is downloaded, the apps crawler notifies the listing details web crawler to download the most recent app listing details web page. The listing details web crawler is written in Ruby. It generates a URL using the package name and downloads both the HTML page and the resources of the listing details web page. Finally, another process runs to decode the APK file using Apktool [10], an open-source reverse-engineering tool. When the APK file is decoded, we get a directory tree of app files that make up the app.

#### **4.3.2 Feature Extractors**

Once an APK file and its listing details web page have been downloaded, a set of feature extractors is run. The feature extractors comprise three main tools: Listing details extractor, UI extractor, and source code extractor. The listing details extractor parses the listing details HTML file, extracts the features, and stores them in the document data store. The UI extractor traverses the unpacked APK file, extracts UI features from layout files and resources, and stores them in the graph data store. The source code extractor computes features from the smali files, a human readable assembly like language for the disassembled byte code. It uses string pattern matching techniques (e.g., grep, regular expressions) to extract features available in the app's byte code and stores them in the document data store.

#### **4.3.3 Data Stores**

Our data store is built on top of two database technologies: a NoSQL document-oriented database for storing apps listing detail features, code features, and APK files. The second database is a graph database for storing the XML elements of layout files. The document-oriented database

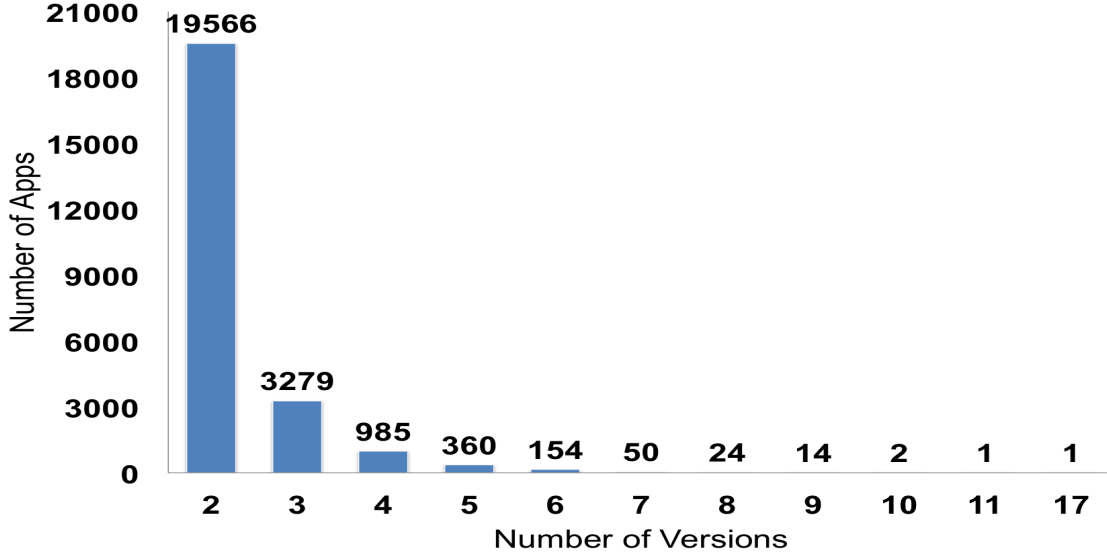


Figure 4.2: The distribution of apps by versions.

is a MongoDB instance, an open-source document-oriented database scalable to accommodate large and complex data. Our MongoDB instance comprises five collections: listing details, the AndroidManifest features, code features, and two GridFS collections to store the binary APK files and additional metadata. The MongoDB instance contains collections for over 56,349 APK files for 24,436 unique apps with multiple releases. The total size of the database is 501 GB hosted on a 24-core server with 47 GB RAM running Ubuntu 12.04. Over the observation period of our analysis (from January 2013 to June 2014), we collected a minimum number of versions per app in two versions in our dataset, which represents 80.1% of the dataset (Figure 4.2). Table 4.2 shows the number of apps by release date, and Figure 4.3 shows the download count distribution by their ratings.

The graph database is a Neo4j server. Graph databases are well suited to model hierarchical connected data like UI structure. Neo4j uses the property-graph model, which allows storing XML elements with their attributes (key-value-pairs) in graph nodes. In android, developers can reuse multiple layouts using the “<include>” tag to embed layouts to the current element in the hierarchy. Our UI feature extractors attach the included layout to the current UI tree element,

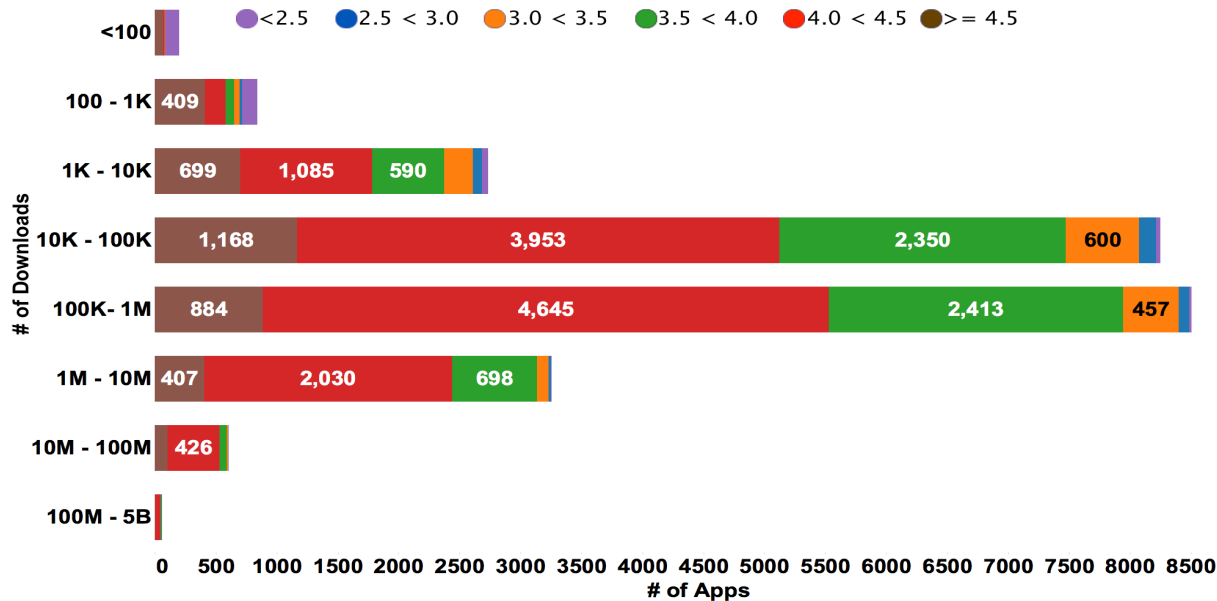


Figure 4.3: The download distribution of the apps by rating stars. Colors show details about the rating.

connecting layouts to one app tree. Queries like “find a specific ViewGroup element with two specific children” run faster than the traditional many JOIN queries in Relational DBs. In our database model, each UI element (e.g., a view like ImageView or a view group like LinearLayout) corresponds to a graph node connected with one relationship (e.g., has view, has view group). The number of graph nodes in our database is 29,733,591 nodes.

#### 4.3.4 Database Client Drivers

A set of MongoDB drivers is running to query and retrieve data from the database. The complexity of the executed queries varies from finding fields in multiple collections to running multiple-pass mapReduce tasks. The graph database clients execute transaction queries on the graph database using Cypher language, a graph query language for querying Neo4j graphs. These clients output the results as CSV files to be processed by the transformation trackers.



### 4.3.5 Transformation Trackers

To interpret the results generated by the client drivers, another set of tools are running to compute statistics on the generated results. These are data analysis tools written in Python using the Pandas library, an open-source library for high-performance data analysis. The data store client drivers output the results as CSV files, which are passed as input to the transformation trackers. The transformation trackers load each CSV file into a Data Frame, a tabular data structure with labeled axes. The Data Frame is indexed by an array of tuples where each tuple is a unique value of the app's package name and version code values. This makes it easier to track changes and perform sophisticated data analysis on multiple versions.

## 4.4 Design Pattern Changes

The system we developed has enabled us to conduct many analyses on design pattern changes. Here we chose eight of the most illustrative ones to present: custom UI components, home screen widgets, as well as various ways of navigating: Tab Layouts, Fragments, Horizontal Paging, Action Bar with Tabs, Up Navigation, and Navigation Drawers. For each analysis, we discuss the motivation, method, results, and implications.

### 4.4.1 Custom UI Components

#### 4.4.1.1 Motivation

A GUI framework typically provides a rich set of standard UI widget classes with the goal to ease the GUI development effort. Developers often can meet their needs using these widget classes. If not, they may need to define custom UI widget classes. Some custom components are provided by third-party libraries to provide custom widgets or backward-compatibility of the standard components. Others are developer-customized components. How prevalent is the use of custom widgets? We are interested in this question because extensive uses of custom widgets may be a sign that the standard UI widget classes are inadequate.

#### 4.4.1.2 Method

A custom component is declared in layout files using the full qualified class name of the component’s class file which starts with a package name, such as

```
<com.android.notepad.MyEditText id="@+id/note" />
```

rather than using the name of one of the default widget classes such as EditText, TextView, and Button. Thus, to find evidence of use of custom UI components in an app, we queried the graph database for apps with views that start with a package name and counted the number of declarations of this form in all of the app’s layout files.

#### 4.4.1.3 Results

We found 15,808 (64.7%) apps used at least one custom component, which is more than half of the apps in our corpus. Among them, 818 (5.2%) apps initially did not use any custom component and began using it after an update. 410 (2.6%) apps did the opposite, reverting to use only standard components in subsequent updates. We looked at the listing details of these apps to find if the developers described any information related to this change in the listing details section “What’s New”. We searched for UI related keywords and discovered apps that report enhancements to tablets and large screens. For example, one app stated, “App is optimized for tablets and many UI enhancements” and another app stated “Improved layout and graphics for larger screen”. We also looked for unusual change patterns. For example, we were interested in whether there were apps that had introduced a large number of custom components after an update. We found 58 apps added more than 500 custom components after updates. We identified 234 unique custom components in these apps by the first three parts of the custom component name (e.g., com.airbnb.android). The top three added custom libraries are: android.support.v4 (18.8%), com.facebook.widget (11.5%), and com.actionbarsherlock (5.1%). Interestingly, we observed a high concentration of these apps in the Finance category. To dig further, we examined the “what’s new” section of these apps and found mentions of “tablets” and “large screens” that coincide with this

sudden increase in the use of custom components. For example, one app stated “Accept payment on Tablets” and another one stated “Resolved login crashes on multiple Tablets.”

#### 4.4.1.4 Discussion

Even though most of the standard Android UI components provide built-in support for automatic scaling to fit content on large screens, getting these components to work properly may require a tremendous amount of effort by developers. We speculate that the 818 (5.2%) apps we found that had begun to use custom components may be to achieve better large screen support that is not offered by standard components. We found there were more apps adding custom components than those removing custom components. This suggests an increased level of reliance on custom components over our sample of Android apps. One could interpret this as a sign that default components are gradually becoming inadequate.

### 4.4.2 Home Screen Widgets

#### 4.4.2.1 Motivation

Home screen widgets are shown on the user’s home screen to provide immediate access to useful information (e.g., today’s weather) or control of commonly used functionalities (skipping to the next song). By choosing which app’s home screen widget to display, a user may implicitly indicate her preference for the app. Since there is limited area on the home screen, the size of a home screen widget matters. An app widget may provide a set of predefined dimensions (e.g., 1x1 or 2x2). Some apps may allow the users to adjust or stretch the dimensions of its home screen widgets freely.

#### 4.4.2.2 Method

To find apps that use widgets, we queried the document database for apps that declare widgets in the apps `AndroidManifest.xml` file. In this file, the `<receiver>` element has a child element named `<meta-data>` that has an attribute named `android:name` whose value is set to

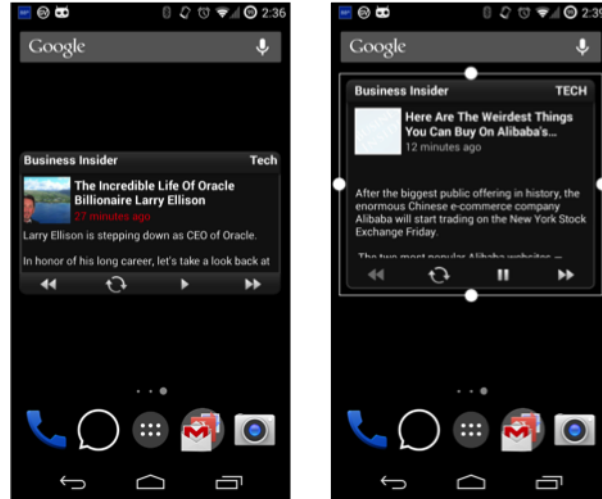


Figure 4.4: Two versions of an app titled “Business Insider” before (left) and after (right) adding the resizing functionality to its home screen widget.

“*android.appwidget.provider*”. To support resizable widgets, Android developers need to declare the resize mode to be horizontal, vertical, or both by setting the value of the attribute *resizeMode* of the `<appwidget-provider>` element located at *res/xml/*. To find resizable widgets, we queried the UI graph database to find a widget layout file whose root element is `<appwidget-provider>` with the attribute *resizeMode*.

#### 4.4.2.3 Results

We found 2,639 apps (10.8%) support home widgets. Among them, 255 (9.6%) added home widgets for the first time, and 80 (3%) later dropped home widgets altogether. 1,118 (42%) apps allow home widgets to be resizable. Among these, 295 (26.4%) were not resizable before and became resizable only in the most recent release. 24 (2.1%) apps had resizable home screen widgets but later on disabled the resizing functionality. In all cases, we did not find any information in the listing (i.e., description and what’s new sections) mentioning changes in home widget support. Figure 4.4 shows an example of a home screen widget of an app before and after making it resizable.

#### 4.4.2.4 Discussion

We found more apps adding support for home screen widgets than those dropping the support (255 vs. 80). This suggests an increased adoption of home screen design pattern. Among those already providing home screen widgets, a similar trend of increased adoption of the “resizable” pattern is observed. This suggests apps are increasingly competing for the limited home screen space. Apps that did not provide home screen widgets would lose out because users are unable to place them on their home screen and may use them less frequently as a result.

#### 4.4.3 Tab Layout with TabHost

##### 4.4.3.1 Motivation

Tab layouts are used to hold a set of tab labels to allow users to navigate between different content. One way to implement this design pattern is through the TabHost API. We chose this API for two reasons. First, this API was included in the initial release of the Android GUI framework (level 1). Second, it was later deprecated (level 11) in favor of new navigation patterns, such as the Fragment and the Action Bar patterns. We were interested in whether most or only a small percentage of apps had made this transition.

##### 4.4.3.2 Method

In order to create a tabbed UI, developers need to use a ViewGroup element named TabHost and a View element named TabWidget. We queried the UI database for a GroupView named TabHost with a child View named TabWidget.

##### 4.4.3.3 Results

We found 3,809 (15.6%) apps used TabHost in their UIs. Among them, 666 (17.5%) apps used TabHost for the first time and 413 (10.8%) apps stopped using it and shifted to other types of navigations. We further inspected their listing details to see if it includes reasons that describe the

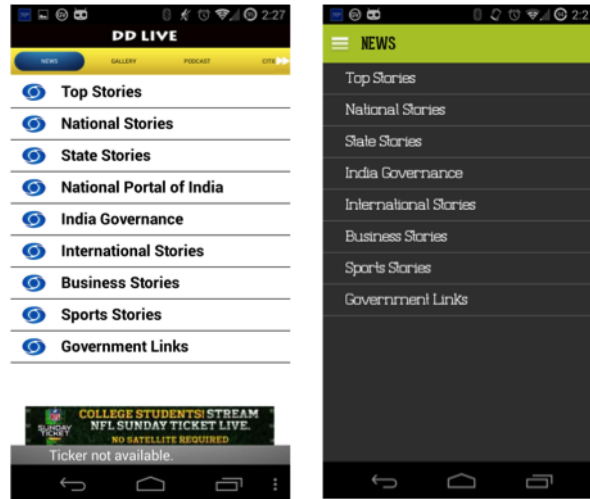


Figure 4.5: Two versions of an app titled “DDLIVE”. The version on the left uses TabHost and the version on the right uses a Fragment that contains a list of items.

migration. We found 106 (25.58%) of these apps provided explanations in their listings. DDLIVE is an example of an app that has undergone this change. In the “what’s new” section of its listing on Google Play, the phrase “UI Changes for better experience” can be found. Figure 4.5 shows two screenshots of the app before and after the migration from the TabHost pattern to the Fragment pattern.

#### 4.4.3.4 Discussion

The migration rate of this design pattern is very slow. Even though the TabHost pattern has been deprecated for at least three years, some new apps continued to use it. Existing apps using them slowly dropped the use of the TabHost pattern but not at a very fast rate. This raises the question of why deprecated APIs and design patterns enabled by them are so “sticky” among Android apps.

#### 4.4.4 Fragment

##### 4.4.4.1 Motivation

Fragments are used to create a multi-pane view and a responsive UI that works on a variety of screen sizes and devices. It is a relatively new design pattern that was introduced in API level 11 to allow developers to add multiple independent UI components and reuse them in different parts of the UI with its own lifecycle. We were interested in how widely this design pattern was adopted.

##### 4.4.4.2 Method

Fragments can be statically added to the layout files or dynamically added in the source code. Thus, our analysis needs to look at both the UI and code. In the UI, we query the UI database for apps with the `<fragment>` element and obtain the value of the `android:name` attribute that references a class in the source code. In the code, we search for the class to see if it extends the `Fragment` class or any of its subclasses.

##### 4.4.4.3 Results

We found 3,963 (16.2%) apps used the Fragments pattern. Among them, 1,814 (45.8%) used `Fragment` for the first time and only 139 (3.5%) apps stopped using it in recent releases. Figure 4.6 shows an example of a shopping list app before and after using `Fragment` pattern.

##### 4.4.4.4 Discussion

It had been four years since the `Fragment` pattern was introduced. We initially anticipated to observe a broad adoption of the `Fragment` pattern by apps to take advantage of the benefit this pattern offers. We instead found a lower than expected adoption rate. Only 3,963 (16.2%) of the apps use it. Among them, almost half were recent adoption during our observation period. We speculate that the `Fragment` pattern is a lot more complicated to implement. For example, StackOverflow, a popular online community for developers to ask and answer questions about

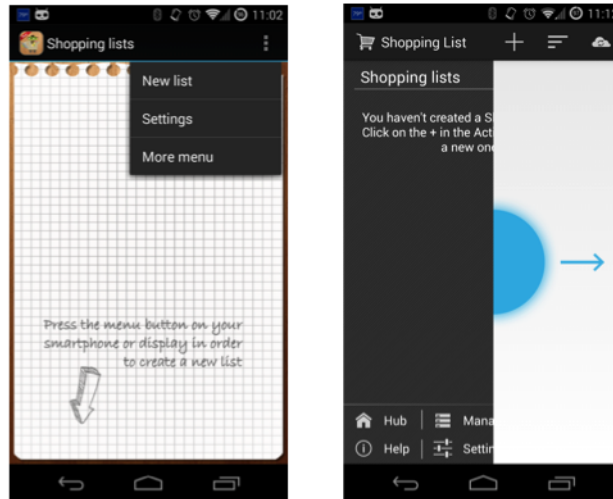


Figure 4.6: Two versions of an app titled “Shopping List” before (left) and after (right) adopting the Fragment design pattern.

coding, has over 16,100 questions tagged with “android-fragments”. Some developers may rather stick to what they are familiar with rather than risking implementing the switch to the Fragment pattern incorrectly.

#### 4.4.5 Horizontal Paging

##### 4.4.5.1 Motivation

Horizontal paging is a navigation pattern that allows users to navigate between screens using left and right swipes. We explore this design pattern to find how wide this type of interaction is and how preferable it is over multiple releases.

##### 4.4.5.2 Method

To implement this pattern, the most common approach is to use a *ViewPager* view group, which is a container that can hold multiple child view elements, each of which represents a distinct screen. Child views can be populated using *PagerAdapter*s in the source code. To find apps with this pattern, we queried the UI graph database for apps that use the *ViewPager* element as a



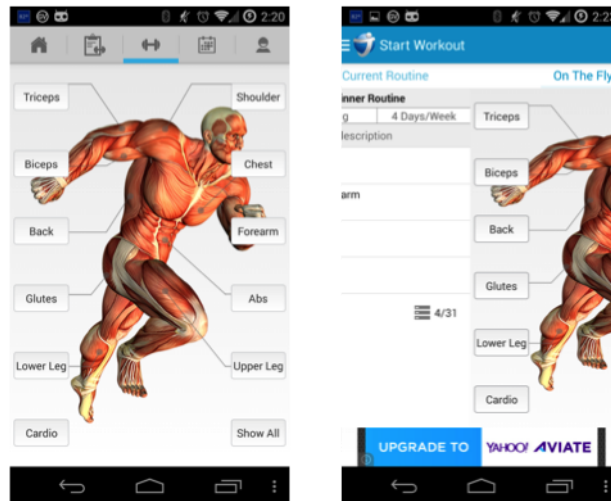


Figure 4.7: Example of an app titled “JEFIT Workout Exercise Trainer” before (left) and after (right) adopting the Horizontal Paging design pattern.

*ViewGroup* container and make use of any *PagerAdapters* subclasses in the source code.

#### 4.4.5.3 Results

We found 4,524 (18.51%) apps used the Horizontal Paging pattern. Among them, 941 (20.8%) added this pattern for the first time. 149 (3.3%) apps dropped this pattern. Figure 4.7 shows an example of an app before and after using the horizontal paging design pattern.

#### 4.4.5.4 Discussion

We observed a lower first-time adoption rate of the Horizontal Paging pattern than that of the Fragment pattern. This suggests that Horizontal Paging pattern has a longer history than the Fragment pattern. Apps still continue to migrate to these two patterns but more of them chose the more general Fragment pattern.

### 4.4.6 Action Bar with Tabs

#### 4.4.6.1 Motivation

Action Bar provides several functions to an app including the screen title, action buttons, and a navigation view with two modes of navigations: navigation tabs or drop-down lists. It was first introduced in Android 3.0 (API level 11) but also available for lower API levels through additional support libraries. We explore the use of the Action Bar with navigation tabs, and how apps have maintained the use of this pattern in their recent versions.

#### 4.4.6.2 Method

While there are multiple ways to create an Action Bar with navigation tabs, they all require implementing the *ActionBar.TabListener* interface that provides the required callbacks to respond to user's actions. To explore the use of Action Bar with tabs, our analysis needs to look at the code. We search the source code for apps that implement the *TabListener* interface.

#### 4.4.6.3 Results

We found 8,483 (34.7%) apps used the Action Bar with Tabs. Among them, 2,729 (32.2%) were first-time adopters of this pattern. 330 (3.9%) stopped using it. Figure 4.8 shows an example of an app before and after using the Action Bar with Tabs.

#### 4.4.6.4 Discussion

The Action bar with navigation tabs design pattern is the most commonly used new navigation design pattern in our dataset of apps. 34.7% of the apps used it. We suspect the reason is that this pattern has been considered mainstream over the past couple years. An app would look out of date without adopting this pattern. Moreover, the Action Bar API is one of the easier APIs for developers to implement and add additional functionalities such as title, icon, and view controls. Yet, 330 still decided to remove the Action Bar pattern.

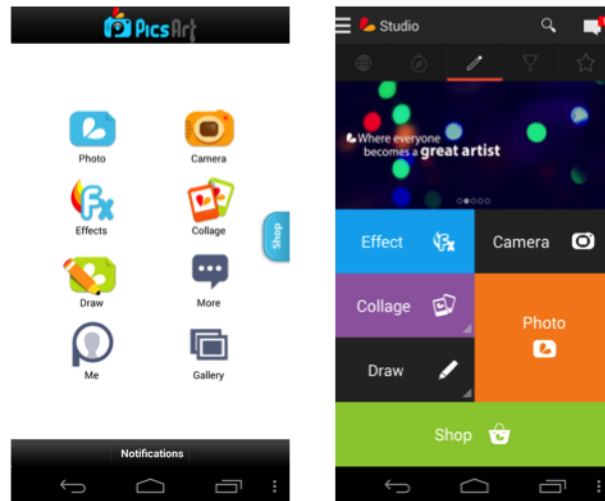


Figure 4.8: Two versions of an app titled “PicsArt” before (left) and after (right) adopting the Action Bar pattern with Tabs.

#### 4.4.7 Up Navigation

##### 4.4.7.1 Motivation

Apps can use the app icon as an Up button to support navigation between screens based on their hierarchal relationships. We study the use of this pattern because it provides a new way of navigation based on the app’s hierarchy rather than navigating through the history of visited screens, a feature already provided by the physical Back button. Unlike the physical Back button, the Up button always ensures that the user navigates through the parent of the current screen and does not exit the app while navigation. The Up button usually appears as a left-facing caret to the left of the app icon in the *Action Bar*. When a user taps on the Up button, the app navigates to the parent screen in the activity hierarchy.

##### 4.4.7.2 Method

In order to use Up buttons, developers need to call the *setDisplayHomeAsUpEnabled()* of the *ActionBar* class. To find this navigation mechanism, our analysis searches the app’s code for the fully qualified name of this method.

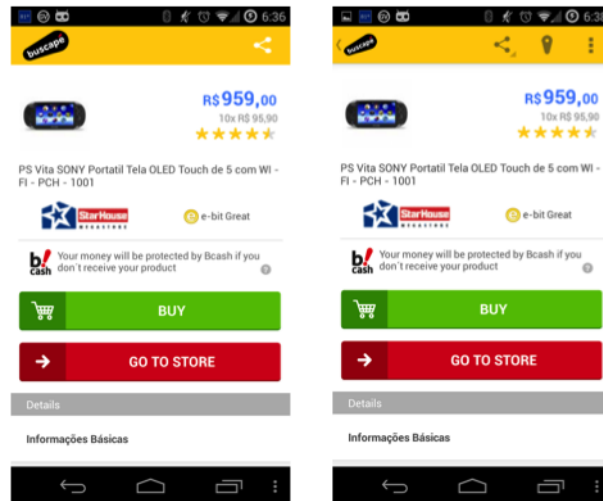


Figure 4.9: Two versions of an app titled “Buscape” before (left) and after (right) adopting the Up Navigation pattern. The Up navigation button is shown at the top left corner as a left-facing caret next to the app icon.

#### 4.4.7.3 Results

We found 4,431 apps (18%) used this navigation pattern. Among them, 1,257 (28.4%) added it for the first time and only 98 (2.2%) of the app’s stopped using it. Figure 4.9 shows an example of an app before and after using it.

#### 4.4.7.4 Discussion

The Up navigation pattern is less common than navigating with a physical back button. Only 18% of the apps adopt this pattern. The reason behind the Up navigation pattern’s low adoption rate could be that apps have relied on the back button for navigation since the initial release of Android, which is already implemented by default. The Up navigation button was common in apps that have several sibling activities in the hierarchy (e.g., email clients, shopping apps), which may not be the case for most apps.

## 4.4.8 Navigation Drawers

### 4.4.8.1 Motivation

Navigation Drawer is a hidden panel that displays the app’s main navigation menu. It helps users quickly navigate through the structure of the app. When the user taps on the top-left app’s icon or swipes from the left to the right of the screen, the Navigation Drawer expands to cover part of the screen and shows a list of items.

### 4.4.8.2 Method

In order to use the Navigation Drawer, developers need to create a layout with a *DrawerLayout* as the root *ViewGroup* element with two children elements. The first child is the Layout element that represents the content when the drawer is hidden and the second is the actual drawer that slides in the Navigation Drawer panel. We queried the graph database to find apps that have layouts that starts with a *DrawerLayout* element as a root element with two child views.

### 4.4.8.3 Results

We found 1,183 apps used the Navigation Drawer. Among them, 771 (65.2%) added it for the first time and only 37 (3.1%) of the apps stopped using this pattern. Figure 4.10 shows an example of an app before and after using this pattern.

### 4.4.8.4 Discussion

Navigation drawer is relatively less common than other navigation patterns. We speculate the reason is two-folds. First, implementing this pattern requires a high degree of customization by developers to fit their needs. Second, an app may not have a deep hierarchical navigation structure to make use of this pattern. We also found that the majority of the apps using the Navigation Drawer were recent adopters (771 out of 1183), which suggests Navigation Drawer a relatively new design pattern.

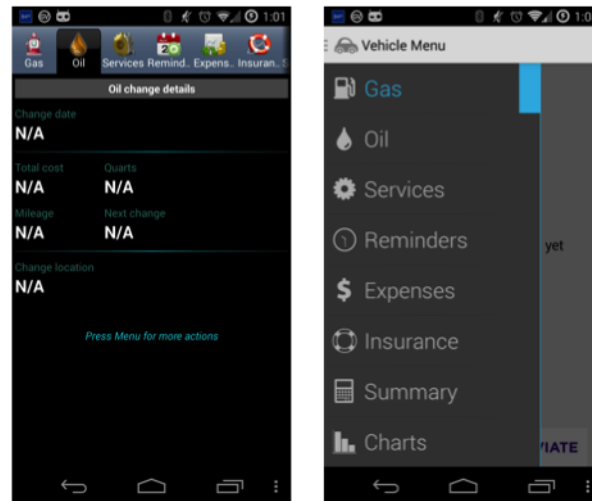


Figure 4.10: Example of an app titled “Carango - Car Management” that shows two versions before (left) and after (right) using the Navigation Drawer pattern.

## 4.5 Summary

This chapter presented a large-scale data-mining approach to analyzing design pattern changes. I discussed its implementation and illustrated a subset of analyses regarding the changes in the use of design patterns (see table 4.3). This work demonstrated the value of using data-driven approaches to discovering design pattern changes on a large-scale.

<b>Design Pattern</b>	<b>Usage</b>	<b>First Time</b>	<b>Changes</b>	<b>No Changes</b>
Custom UI Components	15,808	818	410	15,398
Home Screen Widgets	2,639	255	80	2,559
Resizable Home Screen Widgets	1,118	295	24	1,094
Tab Layout with TabHost	3,809	666	413	3,396
Fragment	3,963	1,814	139	3,824
Horizontal Paging	4,524	941	149	4,375
Action Bar with Tabs	8,483	2,729	330	8,153
Up Navigation	4,431	1,257	98	4,333
Navigation Drawers	1,183	771	37	11,46

Table 4.3: The changes to the presented design patterns. The columns show the name of the design pattern, the number of apps that used this pattern in the dataset, the number of apps that used it for the first time in a recent release, the number of apps that used it but later switched to a different design pattern, and the number of apps that maintained using it in future releases.

## Chapter 5

### Sieveable: A Scalable Platform for Mining Mobile Applications

As the number of mobile apps continues to proliferate in marketplaces, the need to study them at large-scale has begun to receive increased attention. Most of the prior work in this area is limited to a single view of the data and lacks a holistic view of the apps. In addition, data-driven systems are largely constrained by the availability of samples and efficient indexing. This chapter presents Sieveable, a multi-view search engine for Android apps. Sieveable enables deep searching and filtering across multiple levels: (a) Listing Details, (b) User Interface structure, (c) Manifest structure, and (d) API calls. Sieveable crawled and indexed more than 450,000 apps. It provides a query by example language to specify deep search queries. I discuss the key challenges in designing and developing a scalable retrieval system to enable the deep and longitudinal approach presented in this dissertation.

#### 5.1 Introduction

Mobile software platforms feature a distribution platform for applications called marketplaces (or app stores). App marketplaces have become the largest platform for distributing mobile applications, where users can search for apps, browse through different categories, rate or review apps, and install free and paid apps. The popularity of mobile devices and the advances in their operating systems have led to a significant increase in the number of apps published in marketplaces. For example, as of April 2016, the number of apps in the Google Play Store has exceeded two million apps [14]. These apps have become a valuable data source to mine and extract insight from in both



academia and industry.

In the recent years, there has been a noticeable amount of research activities on how to extract meaningful insights from apps data. The research in mining mobile apps has been dominated by three single views. First, researchers have mined the listing details data (metadata) of apps such as ratings and user reviews to perform sentiment analysis and help developers make informed decisions supported by data [59, 47, 80]. Others have created tools and commercial services to assist app developers and publishers in a better understanding of listing details data [13, 12, 11]. Second, researchers have mined user interface (UI) design data such as styles and layouts of thousands of apps to gain insights into their design patterns [123] and my work in chapter 4. Third researchers have also mined the source code of apps to learn about malicious behavior and protect users' privacy-sensitive data [146, 98, 32]. What is missing in prior research is an approach that takes both a holistic view (design and development) and temporal view (multiple versions) of the apps. There are multiple benefits for considering a holistic view that involves both design and development views in mobile app analysis.

App design analysis focuses on UI related data (e.g., layout and style files) while development analysis focuses on code and configuration data (e.g., Manifest files). In design analysis, UI components are often created or modified at runtime. When only analyzing the static layout files, this observation is missed because that behavior is defined in the app source code. This shortcoming can be solved by combining both the design and development views. In sentiment analysis of user reviews, it is often difficult to link opinions to specific features. By incorporating the design view and development view, one can potentially establish a causal relationship between a new feature (or a bug) and the onset of certain opinions. In security analysis, a function could be determined, through program analysis, to be triggering a sensitive operation, such as sending an SMS message or taking a photo. But it is often hard to judge if the sensitive operation is warranted from the program view alone. By also taking a view of the design, one may examine which button may be linked to this sensitive operation and whether the button's label legitimizes such use: for example, "Send" for sending an SMS message. Furthermore, the increasing number of app updates published

to marketplaces has largely gone unobserved. By only considering a single version of the app, one cannot form a deep understanding of the emerging trends in mobile design and development. Supporting multi-view temporal analysis of apps is always challenging because it requires a scalable infrastructure for integrating multiple heterogeneous data sources. The store listing details data is document-oriented where every app shares a number of common fields such as the title, ratings, and category, which can be mined from app marketplaces. Design data is hierarchical; the layout files specify the relationship between various UI components in a tree structure. Program data is highly structural as it involves a large number of names of classes, methods, and variables. Mining apps in multiple views would require an integrated infrastructure that supports document-oriented, hierarchical, and structural data, and provides an easy interface to store, index, mine and analyze such data.

In this chapter, I pursue a more general analysis approach that takes a holistic view of the apps over time and can potentially accomplish what is currently not possible in a single-view, single-version approach. I present **Sieveable**, a novel retrieval platform for multi-view data mining of apps on a large scale to address the needs mentioned above. I discuss a new query language with a declarative, “example-based” syntax. Using this language, analysts could quickly retrieve a sample from a large corpus of apps that meet certain criteria with respect to the listing view, design view, and program view in an integrated manner. Prior to Sieveable, each of these analyses would take days of effort in writing custom scripts and running them on a cluster of servers (my work in chapter 4). With Sieveable, each analysis task can be expressed as a search query and results can be obtained in a matter of minutes. The chapter continues as follows. An overview of our system, detail description of its implementation to enable a new, multi-view analytical queries.

## 5.2 System Overview

In this section, we describe the key requirements of our system and its core components at a high level.

<b>Listing Details Features</b>	package name, title, description, reviews, store URL, category, price, date published, version name, version code, target system version, ratings count, rating, content rating, creator, creator URL, install size, downloads count, permissions, what's new.
<b>User Interface Features</b>	layout directories, XML layout file DOM structure, drawable resources, string resources.
<b>Manifest Features</b>	Manifest File XML DOM structure, which includes elements such as activities, permissions, services, etc.
<b>Code Features</b>	Framework API invocations, dependency libraries.

Table 5.1: Sieveable's main extracted features.

### 5.2.1 Requirements

We identify four main requirements that drive the technical development of our system:

- **Generalizable:** Given the large and diverse ecosystem of the Android platform, our search engine must be general enough to meet diverse user search goals. One may use the system to search for user interface design examples, API usage examples, or security permissions that protect sensitive resources. In order to meet various search goals, Sieveable captures powerful app's features and uses an example-based search.
- **Scalable:** Given the large volume of apps updated frequently, it is essential to design a scalable search engine. The system must be designed to be highly scalable to index billions of files. We address this goal by building a distributed search engine that can be scaled horizontally when an index becomes too large to fit a single machine.
- **Deep:** The search must be comprehensive and deep, taking into account features intrinsic to the app, such as code and UI data, as well as extrinsic features that describe the app, such as the marketplace listing details information.

- **Extensible:** The system must be designed to be modular and extensible to easily extend its search capabilities. To achieve this goal, Sieveable uses a modular plug-in architecture where each search level (e.g., UI search, code search) is a separate module that can be incorporated into the search system. By delegating search tasks into modules, a developer may add a new plug-in to extend the search system. For example, a developer may create a search plug-in for finding open source Android apps hosted on GitHub.

### 5.2.2 Approach

We present a novel approach that supports searching apps across multiple levels. Our approach consists of four main steps: data collection, features extraction, features indexing, and a language specification for performing search queries.

#### 5.2.2.1 Data Collection

We download the Android Application Package (APK) file for apps from the official marketplace, Google Play Store. For each APK file we download and add to the dataset, a web crawler is run to obtain its listing details web page. Next, we parsed the listing details HTML page to extract store listing values. In order to expose the app's UI and code structure, we run a reverse-engineering tool to decompile it, which results in a directory tree of app files that make up the app. With more than one year effort, we managed to collect 452,775 apps with multiple versions. Note that sometimes the crawler was down, so it may have missed a number of updates. Figure 5.1 shows the number of app versions in our dataset, and figure 5.2 shows the total apps by release date and download count.

#### 5.2.2.2 Features Extraction

Once an app is downloaded and decoded, we run a set of tools to extract features at four levels: a) Listing details level: it includes information defined in the app's marketplace listing web page. b) User interface level: it includes all layout related files. c) Manifest level: it includes

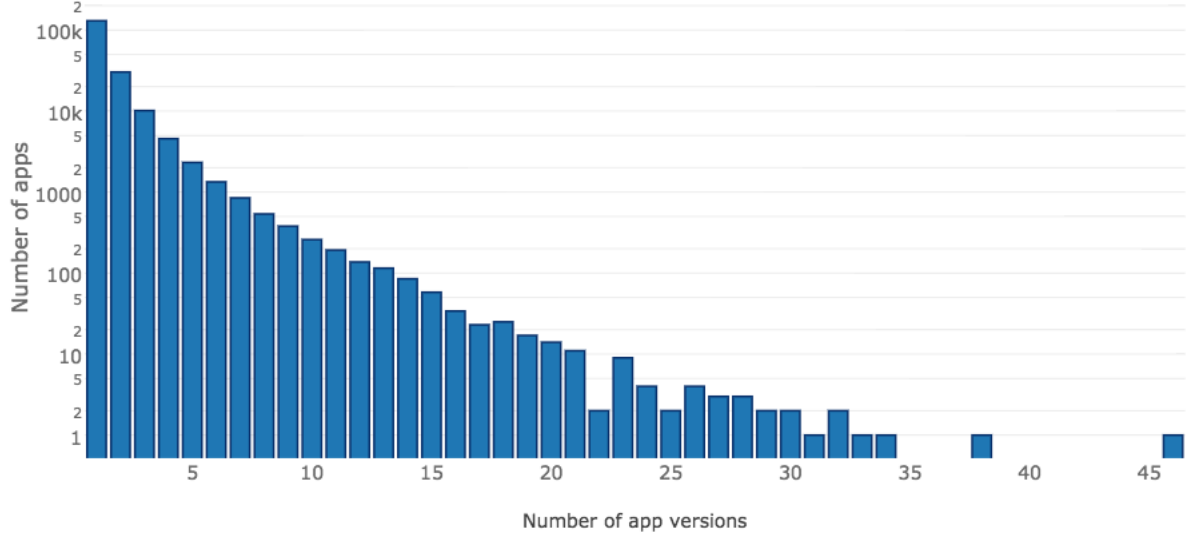


Figure 5.1: The total number of app versions in the collected dataset.

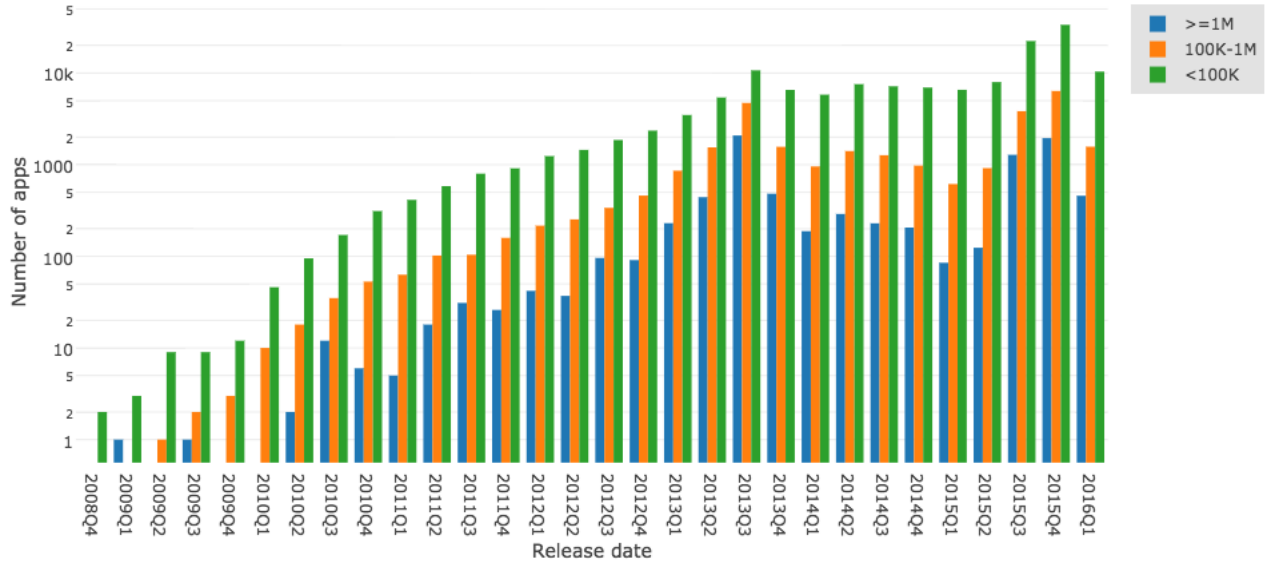


Figure 5.2: The total number of apps (multiple versions) in the dataset grouped by download count and the calendar quarter in which they are released. Three download count groups are used: top (1M downloads or more), middle (100K or more and less than 1M downloads), bottom (less than 100K downloads).

additional internal information that describes the app. d) Source code level: it includes all API calls invoked by the app. All extracted features are listed in Table 5.1.

### 5.2.2.3 Features Indexing

When app features are extracted, we store and index them in scalable data stores to support efficient execution of search queries. Sieveable comprises four main data collections: APK files, listing details, user interface, Manifest, and source code. 1) APK files: The binary APK files are saved on disk across multiple locations and their paths are stored in a document-oriented database along with their name and version values to facilitate quick retrieval. 2) Listing details features are stored in a document-oriented database. We use text indices on specific fields (description, what's new section, reviews, and title). 3) User interface: we use a structural index that keeps track of all DOM elements' relationships (parent/child and ancestor/descendant). 4) Manifest features: we index DOM element names and their attribute values. 5) Code: we use a text index to support search of invoked API classes and methods.

### 5.2.2.4 Query Language Specification

**Query Syntax:** Sieveable uses an SQL-like declarative query syntax. The syntax is composed of three main clauses:

- *MATCH* The app to match.
- *WHERE* The search condition.
- *RETURN* The results to return.

Example:

```
MATCH app
```

```
WHERE
```

```
<LinearLayout>
```

```

        <Button/>

    </LinearLayout>

RETURN app

```

The *MATCH* clause defines the apps to match the given search conditions. The *WHERE* clause defines specific search conditions. A search condition in its simplest form is an example of a single listing details field, UI element, manifest element, or an API call. Multiple search conditions can be combined together allowing for a deep search across multiple levels. The *RETURN* clause defines the subset of fields to include in the results. Multiple levels search conditions can be added to the *WHERE* clause. For example, a search query for apps developed by *Google*, have a *LinearLayout* with a child *Button*, use the *SEND\_SMS* permission, and call the *takePicture* API call, will look like:

```

MATCH app(latest=true)
WHERE
    <developer>Google Inc.</developer>
    <LinearLayout>
        <Button/>
    </LinearLayout>
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <code class="android.hardware.Camera" method="takePicture"/>
RETURN app

```

Listing 2: Sieveable query example.

**Query Parser** When a query is submitted to Sieveable, it parses the search condition parts to determine their query levels (listing, UI, manifest, and code). Sieveable maintains a set of dictionaries to lookup and extract search condition parts by their levels. In particular, it maintains three dictionaries: a predefined set of listing details fields, and a set of manifest XML elements, and a single XML element named code for code related queries. If an XML element in the query condition is not defined in any of the dictionaries, we consider it as a UI search element. For example, Sieveable interprets the query in Listing 2 as follows. It parses the *WHERE* clause part and groups the query parts by their levels. This results in four query parts: a) listing details query

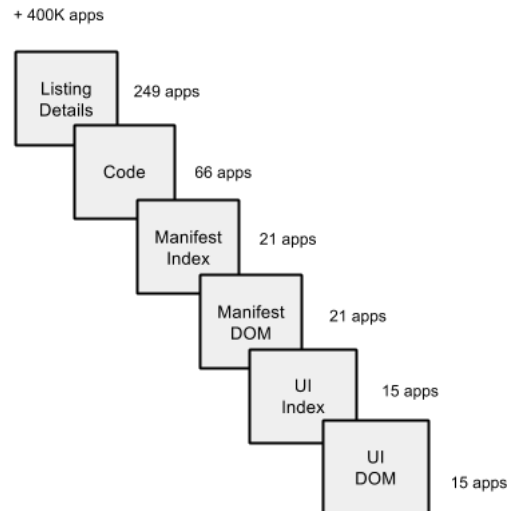


Figure 5.3: The execution sequence for a query that includes multiple level search conditions.

for the `<developer>` tag, b) manifest query for the `<uses-permissions>` tag, c) code query for the `<code>` tag, and UI query for the remaining elements (`<LinearLayout>` and its child). Sieveable sends these query parts to the query executor.

**Query Executor** The query executor creates a plan for executing the received query parts. The plan is an order of steps to query each index and data store (listing, UI, manifest, and code). Query parts are executed by collection finder plug-ins (e.g., find by UI Index, find by UI DOM, find by code, etc.). Collection finders return a set of app ids that matched the given query. Sieveable computes the intersection of app ids to avoid scanning entire collections. This also ensures that the final query results only include apps' Ids that met all query part conditions.

Figure 5.3 shows the execution sequence of the query in Listing 2 and the number of apps scanned by each collection. The entire dataset contains over 400,000 apps. The listing details query part is executed by the *findByListing* module, which returns 249 apps developed by Google and filtered by the latest version of each app. Next, the *findByCode* module executes the code query part only on those 249 apps and returns 66 apps. The *findByManifestIndex* module searches the Manifest index for the Manifest query part only on those 66 apps and returns 21 apps. The Manifest query part is sent to the *findByManifestDOM* module to match the DOM tree for those



21 apps. Since the query part contains no hierarchical structure, the DOM matcher returns the 21 apps found by the index. Next the UI query part is sent to the UI index to search for apps with `LinearLayout` and child `Button`. The *findByUIIndex* searches the UI index for those 21 apps and finds that 15 of them have that UI structure. The UI query part is sent to the *findByUIDOM* module to match the DOM tree for those 15 apps which returns the final query results (15 apps). Finally, Sieveable returns any field defined in the return clause for the matched app ids. The return clause in the submitted query includes only a single app field, so Sieveable will return an array of objects where each object contains key-value pairs for the app id, package name, and version code. Below is part of the final query result:

```
[{ "app": {
  "id": "com.google.android.talk-21224130",
  "packageName": "com.google.android.talk",
  "version": "21224130"
}
```

```
}]
```

## 5.3 System Implementation

In this section, we discuss the technical details of designing and implementing our system. The system consists of six main components: app crawlers, feature extractors, data indexers, data store servers, DOM matchers, and a restful API (see Figure 5.4).

### 5.3.1 App Crawlers

Sieveable features two crawlers that collect our dataset of apps: apps crawler and listing details web page crawler. The app crawler is responsible for downloading APK files from the Google Play store. We feed a dictionary of popular Android search keywords into the crawler. Google has no official API for downloading APK files. We used an unofficial API<sup>1</sup> to collect APK files. When an APK file is downloaded we run the Android Asset Packaging Tool (aapt) on the

---

<sup>1</sup> <http://github.com/Akdeniz/google-play-crawler>

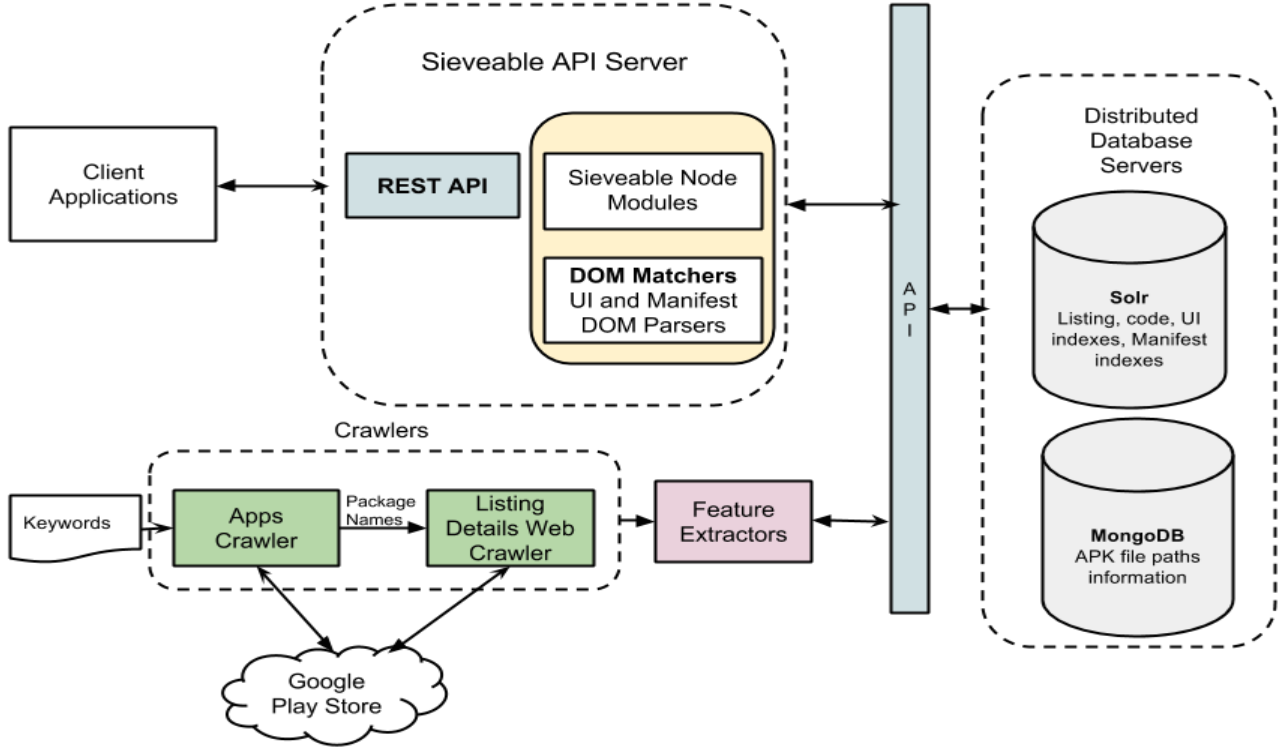


Figure 5.4: Sieveable system architecture.

file to obtain its package name and version code. We use a combination of the package name and version code as a unique identifier for each app. APK files are stored in the file system while their Ids and disk paths are stored in a MongoDB collection for fast retrieval. Once an APK file is downloaded, we run a web crawler to fetch the most recent listing details web page of the app. We parse the listing details fields and save them in another MongoDB collection.

### 5.3.2 Feature Extractors

We built a set of command line tools to extract features from the APK files. First, we use apktool [10], an open-source tool for reverse engineering binary Android applications. Second, we run a custom UI parser that parses layout files and resolves any references to external resources (e.g., `android:text="@string/variable"`) or embedded layouts (using `<include/>` and `<merge/>` tags). The parser produces a single XML file that contains the entire app UI tree, including all files. This is especially important when performing DOM matching queries since it eliminates the

need to load multiple layout files into memory when performing such queries. Below is a snippet from an XML file generated for the YouTube app.

```
<App name="com.google.android.youtube" version_code="5021">
  <Directory directory_name="layout">
    <File file_name="activity_feed_item.xml">
      <RelativeLayout>
        <ImageView android:id="@id/channel_avatar"/>
      </RelativeLayout>
      ...
    </File>
  </Directory>
</App>
```

Third, we extract all API calls from the smali files, a human readable assembly-like language for the disassembled byte code. We save the extracted API calls in one text file per app.

### 5.3.3 Data Indexers

Sieveable indexes the extracted features in Solr collections for fast data access. A collection is a complete logical index. We use five Solr logical indexes to index our dataset: listing index, UI tag index, UI structural index, manifest tag index, and code index.

**Indexing Listing Details:** It contains all listing detail fields. Four fields are indexed as generic text fields (description, title, “what’s new”, and reviews) to support text based search.

**Indexing UI Data:** The extracted UI files are indexed in two Solr indices: 1) *Tag Index*: We extract all tag names and attributes and store them in one index document. This index holds all tag names and their attributes. 2) *Structural Index*: a suffix tree based index that stores the XML tree in a suffix array format [127]. This index holds values that indicate the parent-child relationship of nodes in the XML tree. (e.g., `LinerLayout->Button`). We parse the single XML

UI tree file we extracted for each app and generate two text documents. The first text document contains the tag and attribute names for all UI elements (e.g., `EditText(android:layout_width="fill_parent")`). We add this document to Solr and use it as the tag index. The second generated document describes the UI tree in a suffix-tree format where each line corresponds to a root-to-leaf path for all XML elements (e.g., `RelativeLayout->LinearLayout->ImageButton`). We add this document to Solr and use it as the structural index.

**Indexing Manifest Data:** The extracted manifest files are indexed by their tag names and attribute values. Unlike UI queries, Manifest queries are less structural (e.g., find "`android.permission.CAMERA`"). Therefore, we use the same UI tag index method to add manifest files to the Solr index.

**Indexing Code Files,** we add the extracted invoked API calls from the smali source code files to Solr.

#### 5.3.4 Data Store Servers

Sieveable uses two main NoSQL database servers: a) MongoDB [22]: is a document-oriented NoSQL database. It contains a collection for storing information about the downloaded APK files. b) Apache Solr [8]: is a document-oriented NoSQL Search Platform. We use Solr to store distributed collections for listing detail fields, UI tags and attributes, UI structural index, Manifest attributes, and code files. Due to the large size of our data sets (over 8TB), we found that a single machine is not sufficient to store this large data sets. Thus, we shard and replicate the collections across a cluster of multiple nodes to improve the system performance and availability. We use an external Zookeeper [9] cluster of five nodes to synchronize Solr's configuration and elect leaders.

#### 5.3.5 DOM Matchers

Sieveable UI and Manifest search queries are written in "example-based" syntax. We implement a custom DOM matchers module to navigate the XML DOM tree and select DOM elements with a particular DOM structure. The DOM matchers perform a jQuery like DOM manipulation.

While loading DOM files for complex queries is often considered expensive, the use of hierarchical indices reduces the number of false candidates significantly. However, parsing the DOM for a large number of files has an inevitable memory and system resources overhead. Therefore, Sieveable uses a configurable default limit of 50 results per a given search query and return an inalterable cursor to the client to receive the entire query results.

### **5.3.6 RESTful API**

Client applications can access Sieveable through a RESTful API. The API provides an HTTP GET request mechanism allowing client applications to submit queries and receive the result as a JSON document. For an easy query access, Sieveable has both a command line and web client applications that interact with the RESTful API.

## **5.4 Summary**

This chapter presented Sieveable, a multi-view search engine for apps on a large-scale. It discussed the importance of searching and mining apps across multiple views. For the first time, Sieveable can help users execute deep search queries (illustrated in Chapter 7) and apply data-driven approaches with a holistic view of apps at large-scale that could yield valuable results. Sieveable has been deployed into a distributed computing infrastructure. For more information on using Sieveable including source code access, please visit: <http://github.com/sikuli/sieveable>.

## Chapter 6

### Deep Search Queries

#### 6.1 Introduction

Sieveable enables users to perform queries across multiple levels to meet diverse search goals. It can find apps with specific listing fields, UI structures, manifest attributes, and API calls. Sieveable can produce results beneficial to several stakeholder groups: a) User interface design researchers can use Sieveable to find apps with specific design structures. b) Accessibility researchers can use it to find answers to accessibility questions. c) Privacy and security researchers can use it to find apps that request an unusual number of permissions or use vulnerable APIs. d) Program analysts can use Sieveable to reduce the time to find examples of interest to apply defect analysis. e) Data mining researchers can mine applications efficiently and build machine-learning applications at large scale without incurring the overhead of collecting and indexing large datasets. In a prior work (discussed in chapter 4), these types of analyses would require custom scripts to be written and executed on a cluster, which would require several days of work. In contrast, Sieveable is able to reduce the problem to a simple search query, which takes an analyst only seconds to specify and minutes to run.

In this chapter, I discuss Sieveable’s capabilities, how queries can be formulated, and present several illustrative search queries to show that Sieveable enables users to conduct many types of analyses.

## 6.2 Search Queries

In the previous chapter, I briefly described the syntax of the search queries. In this section, I describe the syntax in more detail and highlight important search features. Sieveable uses an SQL-like declarative query syntax. The syntax is composed of three main clauses:

- *MATCH* The app to match.
- *WHERE* The search condition.
- *RETURN* The results to return.

The *MATCH* clause defines the apps to match the given search conditions. One can retrieve all app versions by simply writing:

```
MATCH app
```

Conditional statements can be added to the match clause to limit the search to an app version. For example, below is the syntax of the match clause to restrict the search to the latest app version.

```
MATCH app(latest=true)
```

Or to narrow the search to the latest version of a specific app:

```
MATCH app(package=com.google.android.music, latest=true)
```

The *WHERE* clause defines specific search conditions. A search condition in its simplest form is an example of a single listing details field, UI element, manifest element, or an API call. Multiple search conditions can be combined together allowing for a deep search across multiple levels. The syntax used here is an example of UI, manifest elements, or listing details fields. For code queries, Sieveable uses a specific XML element in the where clause to recognize code queries. For example, the where clause for a code query will be specified as follows:

```
<code class="fully-qualified-class-name" method="method-name">
```

Multi-level search conditions can be added to the *WHERE* clause. Sieveable also supports multiple character wildcard searches in the *WHERE* clause. This is in particular useful when the value of a specific XML element or attribute is unknown. For example, when searching for custom UI components in a specific app: `<com.whatsapp.*>`. Another example is when searching for an unknown XML element, we can use the underscore character as the tag name:

```
<TabHost>
    <_></_>
    <_></_>
</TabHost>
```

The *RETURN* clause defines the subset of fields to include in the results. Projection fields can be added to the *RETURN* clause to indicate which fields to include in the search result. For example, below is a query to retrieve the download count for all apps in the dataset.

```
MATCH app
WHERE
<downloads>(*)</downloads>
RETURN app, l$1
```

Note that, we specified “(\*)” as the text value of the downloads tag because we do not know what the actual value is, and we want to retrieve that value and include it with the results. The above return clause contains the app info object (package name, version name, and version code), and the listing details projection field *l\$1* which represents the retrieved download count. Listing 3 shows a query that combines all these features together.

In the next sections, I describe how Sieveable can be used in different scenarios to answer analysis questions in three areas: UI design, security, and program analysis. The goal is to highlight the utility of Sieveable and how it simplifies the analysis task to a simple search query.



```

MATCH app(latest=true)
WHERE
  <store-category>Lifestyle</store-category>
  <downloads>*</downloads>
  <Button android:text="*">/>
  <uses-permission android:name="(android.permission.*)">/>
  <code class="android.hardware.Camera" method="takePicture">/>
RETURN app, l$1 AS downloads, u$1 AS button_label, m$1 AS permissions

```

Listing 3: Sieveable query to find apps in the *Lifestyle* category, have a *Button* with a text label, use one or more system permissions, and call the `takePicture` API call. The query result will include the app info object (package name, version code, and version name), the download count of the apps, the text labels of all the buttons, and the list of system permissions they required.

### 6.3 The Retrieval Task and Formulating Search Queries

Sieveable inherits the limitations that exist in information retrieval (IR) systems. It is a well-known limitation of any IR system that the user has to know ahead of time what she is looking for and translates that into a search query. Query formulation is an essential part of any IR system; unfortunately, it is not a transparent process to an analyst or a researcher whose search goal is to find examples of visual design or development concepts. The retrieval task is a complex task that consists of identifying an example of interest, translating that into a search query, submitting the query into an IR system, and examining the results for validation [74]. Generally, domain expertise plays an important factor that could result in a successful retrieval task. The process of translating a high-level design or development concept into a search query is a complex, manual process. For instance, in UI design analysis, this task can be initiated when an analyst observes an interesting visual design in an app. The hierarchal structure of this visual design can be revealed by dynamically inspecting the UI of the running app [7] or by statically reverse engineering the app [10]. Once the hierarchal structure is revealed (i.e., which combination of views are used), it can be used as an example UI syntax in the Sieveable’s search query. Similarly, in code analysis, an analyst may identify a problem associated with a specific API by reading the documentation or posts on question-and-answer websites. The next step is to articulate the problem and translate it into a search query in Sieveable:

```
MATCH app
```

```
WHERE
```

```
<code class="the_fully_qualified_class_name" method="method_name" />
```

```
RETURN app
```

It is important to emphasize here that Sieveable does not attempt to address the difficult task of turning a high-level design concept into a search query. It builds on the assumption that a user has the knowledge and expertise to overcome this inherent limitation, so she can focus on turning the more complex analysis task into a search query problem.

## 6.4 Design Queries

### 6.4.0.1 UI Screen

In mobile app design, designers and developers tend to create a set of UI screens that serve a common purpose. We can use Sieveable to find common screens and search for design alternatives. We can also aggregate the results and find trends in implementing one design over the other.

**Sign In:** Many apps allow users to sign into their accounts to enable them to use certain features. Some apps use the sign in screen as the first screen that welcomes new users to their applications. Sign in screens usually take the form of a single sign in button. To find such examples, we can write a query in Sieveable to find apps that use a Button with the phrase “Sign In”. We can also include the number of downloads in the result fields to reflect on how popular they are:

```
MATCH app
```

```
WHERE
```

```
<Button android:text="Sign In"/>
```

```
<downloads>(*)</downloads>
```

```
RETURN app, $1
```

We can also use Sieveable to find alternative ways of designing a sign in screen. For example, we can also search for apps that use a TextView as a sign in button and obtain their download count

to compare it with the previous results:

```
MATCH app

WHERE

    <TextView android:text="Sign In"/>

    <downloads>(*)</downloads>

RETURN app, $1
```

#### 6.4.0.2 Design Interactions

Mobile apps feature unique interactions allowing users to navigate within the app using various touch gestures and UI controls. We can use Sieveable to explore real-world examples of apps that use a combination of gestures and UI controls and identify common patterns among them.

**Pull Down to Refresh:** In mobile app design, there is a common UI interaction gesture called “Pull down to refresh” [39]. It is used for in-app content updates, allowing users to see new content by scrolling a view vertically. Some apps use this mechanism to push content updates to the UI view when it is requested by the user as an alternative to auto-updates, which may consume critical mobile device resources such as battery and network. In Android, this interaction gesture is not built into the standard UI widgets, so developers need to use additional library APIs to implement this gesture. One commonly used library is the Android Support Library, which includes a layout view called `SwipeRefreshLayout` that enables developers to implement this gesture. In addition to declaring this view in the UI layout file, developers need to register event listeners in the code to respond to the gesture and refresh the content. We are also interested in finding the top three categories of the apps that use this interaction mechanism. To find such examples, we can use Sieveable to search for apps by three levels search criteria (UI, code, and listing details).

```
MATCH app
```

WHERE

```
<android.support.v4.widget.SwipeRefreshLayout>
    <ListView/>
</android.support.v4.widget.SwipeRefreshLayout>
<code class="android.support.v4.widget.SwipeRefreshLayout" method="setOnRefreshListener" />
<store-category>(*)</store-category>

RETURN app, $1
```

## 6.5 Security Queries

The popularity of mobile apps resulted in an increase in the number of exploited vulnerabilities. Security researchers and mobile apps security analysts use sophisticated techniques to detect software vulnerabilities. However, their techniques are largely constrained by the availability of samples of apps that are vulnerable to attacks. Sieveable can be used to perform code search queries for apps that are vulnerable to attacks as a result of weak API implementations. The search query can be combined with multiple search criteria to restrict the search to specific API versions.

**WebView:** A webView is a UI component that displays web pages. This component is widely used in mobile apps to display web pages or display banner ads. While the webView is a great feature that helps developers deliver rich content to their users, it can also become the most dangerous component in an app. For example, developers can enable JavaScript in a webView and expose the Java object's methods to the JavaScript interface. This allows untrusted content viewed in the webView to use reflection to access all public and inherited methods like *Class.forName("java.lang.Runtime")*. An attacker may send a link to the user of a vulnerable app and gain access to all device resources or wipe the entire device content. This vulnerability was fixed in Android 4.2 but the fix is not effective unless the app targets the API level 17 or higher [27]. Sieveable can be used to find apps that are possible candidates for this vulnerability. We can search for apps that set the Manifest attribute *targetSdkVersion* to a value less than 17. Below is an example query.

```

MATCH app

WHERE

<code class="android.webkit.WebView" method="addJavascriptInterface" />

<uses-sdk android:targetSdkVersion="12" />

RETURN app

```

**Permissions:** The security permissions system is the core component of Android security. Apps have no access to any sensitive operations, hardware resources, or user's data without explicitly asking for permissions. These permissions are granted at install time by the user. Apps may also declare custom permissions to protect access to certain app features. To distinguish between custom app permissions and system permissions, the name of system permissions start with "android.permission.\*" Malicious apps tend to request an unusual number of permissions with more frequently on SMS-related permissions [146]. We can use Sieveable to find apps that request a large number of system permissions. For example, we can use the following query to find apps that request 11 or more system permissions and at least an SMS related permission.

```

MATCH app

WHERE

<uses-permission android:name= "android.permission.*" __min="11" />

<uses-permission android:name= "*_SMS" />

RETURN app

```

## 6.6 Program Analysis Queries

Static program analysis tools are used to examine the source code statically to detect vulnerabilities and possible runtime errors. Perhaps the most common approach applied in static analysis is searching plain text source files for lines matching a string. Opening a large number of files, scanning them for matches and combining the results with other parsing tools (e.g., UI and Manifest) is a frustrating process. In addition, finding a sample of apps that match a given crite-

tion (e.g., permissions, UI structure) along with the source code text search is a challenging search task. Static analysis researchers are still constrained by the lack of a large-scale comprehensive search systems to work with. Sieveable’s complete view of the app helps researchers find samples by multiple search criteria and focus their efforts on designing more rigorous static analysis tools. This is in particular valuable for static analysis tools because it reduces the overhead of obtaining samples and increases the sample size.

**Overprivilege Analysis:** In Android, application developers may require a number of permissions in the AndroidManifest file without actually using permission protected API calls. Such apps are considered overprivileged. Researchers have developed tools that perform overprivilege analysis on apps’ source code [57, 33]. These tools work on the decompiled code by constructing a call graph over the entire app and performing traversal to identify API calls that are unreachable. Their tools could be scaled by using our search system. Sieveable can be used to find apps that request a particular permission and declare permission protected API calls. Static tools can be used to analyze the results to determine whether these API calls are unreachable. For example, below is a query that searches for apps that request the *Camera* permission and declare the `android.hardware.Camera.open()` API method.

```
MATCH app
WHERE
  <code class="android.hardware.Camera" method="open"/>
  <uses-permission android:name="android.permission.CAMERA"/>
RETURN app
```

**Bug Fixes:** Mobile apps are updated regularly to improve stability and fix bugs. Updates that include bug fixes are especially interesting to bug finding tool builders. Android allows developers to list the log of changes for the recent app update in a listing details section named “What’s New”. This information might be valuable to a program analyst who is interested in understanding various attempts to fix bugs and evaluating them. We can use Sieveable to find apps with potential

bug fixes to common developer errors. For example, Android uses the Service API for running long operations in the background. However, the Service API might be confusing to new developers since the Service is not necessarily running in a background thread. Instead, it runs in the app's main thread by default which causes the app to crash in "Application Not Responding" (ANR) error. We can use Sieveable to get a set of apps that might be candidates for bug fixes related to the use of background services. The query below searches for apps that use the Service API and mention the phrase "bug fixes" in the "What's new" listing details field.

```
MATCH app
WHERE
  <code class="android.app.Service" method="onStartCommand"/>
  <whats-new>bug fixes</whats-new>
RETURN app
```

## 6.7 Summary

This chapter described Sieveable search queries and how a complex analysis task can be expressed in a simple search query. I explained how one could use Sieveable to answer analysis questions in three areas: UI design, security, and program analysis. I presented a set of illustrative query examples that spanned across multiple levels and met diverse search goals. These queries illustrate the powerful approach Sieveable uses, which can open up a wide opportunity to apply an in-depth analysis of mobile apps across multiple levels to investigate emergent trends in mobile apps development.

## Chapter 7

### Temporal Analysis of the Design and Development of Mobile Applications

The trends in designing and developing mobile applications are always changing and evolving over time. Platform owners regularly introduce new features, remove previous ones, and change their best practices and guidelines. Since all of these changes are observed at some points in time, it is necessary to apply a temporal (time series) based analysis to capture interesting events and draw more meaningful statistical insights. Temporal analysis enables us to ask new questions and understand the changes in the design and development of mobile applications. The lack of effective retrieval systems and resources that provide access to the temporal nature of applications have limited the questions we might ask and led to studies that focus on a single snapshot in time. Sieveable is what gives us access to the temporal nature of mobile applications and helps us to answer diverse analysis questions. In this chapter, I present some illustrative temporal analysis examples to understand how apps are evolved over time in three areas: UI design, accessibility, and privacy<sup>1</sup>.

#### 7.1 Visual Design Mining

Mobile UI design guidelines are constantly updated to provide a better and consistent user experience across the platform. For instance, in June 2014, Google introduced material design, a design specification language that makes use of elements from print design, responsive transitions, and depth effects [21]. Since the concepts were announced, several third-party implementations

---

<sup>1</sup> The raw data, source code, additional analyses, and interactive data visualizations are available at <https://github.com/sieveable/sieveable-mining>



have been built and used by developers. A year after the announcement, Google introduced an official library called the Android Design Support Library, which allows developers to implement a number of material design components that are backward compatible with old devices. Now, it has been over two years of efforts invested in creating concepts and tools to promote the use of this design language. We can use Sieveable to understand how these concepts are adopted over time and ask questions like: What is the state of adopting material design patterns? Did these concepts get adopted immediately by developers or at a slow pace? Are there any factors that may have contributed to a sudden increase or decrease in adoption?

### **7.1.1 Material Design Components**

In this section, I present a set of analyses on the adoption rate of two major material design components: the floating action button and the navigation drawer. The floating action button is arguably the most recognizable material design component and represents the primary action in an app. The navigation drawer is perhaps the first design pattern that was introduced before material design and later was added to the design language. To identify the possible implementation options and formulate search queries, I manually inspected the UI structure of a number of applications that implemented material design concepts. This was done using the UI hierarchy viewer tool [7], which can inspect the app's UI layout dynamically. It is important, however, to note that these are not a complete, comprehensive list of implementation alternatives due to the inherent limitation and challenges of formulating search queries, which I discussed in the previous chapter.

### 7.1.1.1 Floating Action Button

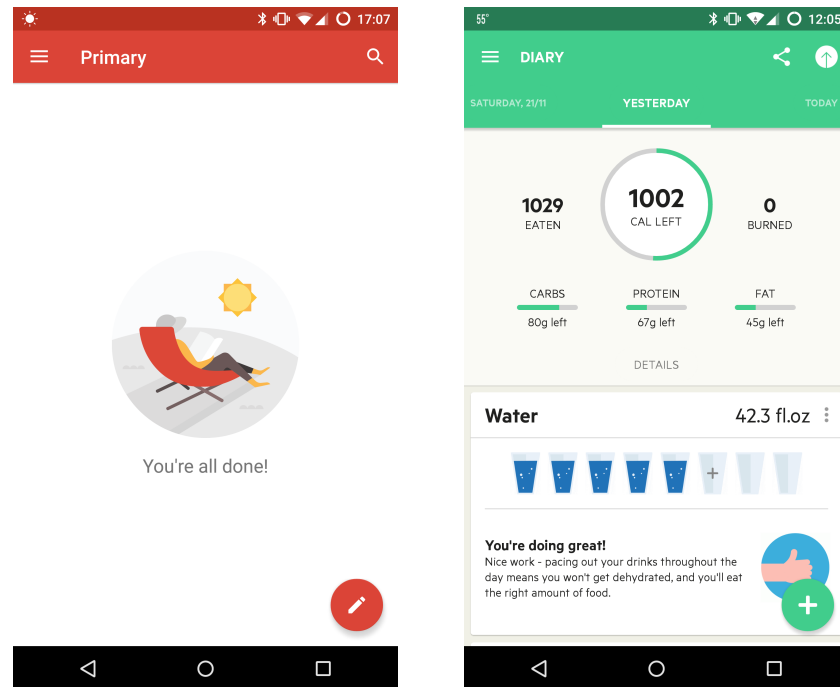


Figure 7.1: The Floating Action Button (FAB) shown at the bottom right corner of two apps, Gmail and Lifesum

The Floating Action Button (FAB) is a circled button floating above the UI to promote the primary action in the app (Figure 7.1). While developers can create a custom FAB with material design style, there are a few libraries that simplify adding this widget. To find apps that have this UI component, one will write custom scripts to parse a large number of UI layout files and count for all possible implementation options. This task is achieved by writing a simple search query in Sieveable. We can use Sieveable to evaluate implementation alternatives and help developers make an informed decision of using a specific library over others. With Sieveable, one can examine the popularity of a specific UI library within any time window. This can also provide insightful feedback to UI framework engineers on how well their efforts are received by developers. We can find the trends of adopting the FAB as a design pattern over time using different alternatives. At the time of writing, there are four common alternative ways to implement the FAB in Android:

the official design support library [4], and three other additional third-party libraries [6, 18, 17]. I use the following Sieveable search queries to retrieve a set of apps that implement the FAB using the previously mentioned libraries:

```
MATCH app
WHERE
    <android.support.design.widget.FloatingActionButton />
    <date-published>(*)</date-published>
    <rating>(*)</rating>
    <downloads>(*)</downloads>
RETURN app, l$1 AS rDate, l$2 as rating, l$3 as downloads

MATCH app
WHERE
    <com.getbase.floatingactionbutton.FloatingActionButton />
    <date-published>(*)</date-published>
    <rating>(*)</rating>
    <downloads>(*)</downloads>
RETURN app, l$1 AS rDate, l$2 as rating, l$3 as downloads

MATCH app
WHERE
    <com.melnykov.fab.FloatingActionButton />
    <date-published>(*)</date-published>
    <rating>(*)</rating>
    <downloads>(*)</downloads>
RETURN app, l$1 AS rDate, l$2 as rating, l$3 as downloads

MATCH app
WHERE
    <com.software.shell.fab.ActionButton />
    <date-published>(*)</date-published>
    <rating>(*)</rating>
    <downloads>(*)</downloads>
RETURN app, l$1 AS rDate, l$2 as rating, l$3 as downloads
```

When we combine all the results together and aggregate by the release year in which the first adoption is observed, we can get a sense of the adoption rate of the FAB over the last years as shown in figure 7.2. This shows that the adoption rate of this design pattern is growing slowly since it was first introduced in June 2014. We can also analyze the adoption rate of the different implementation or library options over time. In Figure 7.3, we can see that apps started adopted this pattern before the official release. Once the official library was introduced by Google in May 2015

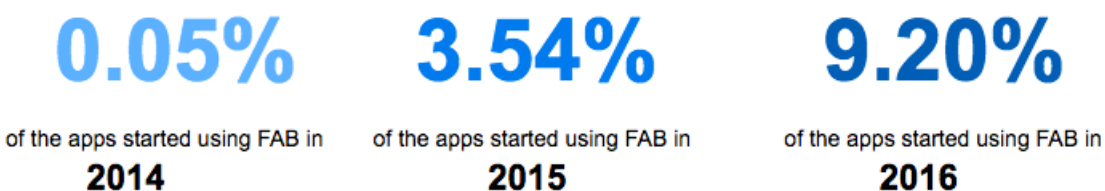


Figure 7.2: The adoption rate of the Floating Action Button (FAB) over the years.

[5], the adoption rate started to increase noticeably. This shows that a small number of developers tend to use third-party libraries to implement a new design that is not yet officially supported. The larger number of developers, however, are much quicker to use a consistent implementation by an official library. Finally, I analyzed the group of apps that started adding FAB earlier using a third-party library before the existence of the official library. I found that the majority of these apps have a low number of downloads (89.81% with 100,000 download times or less compared to 10.19% with more than 100,000 download times).

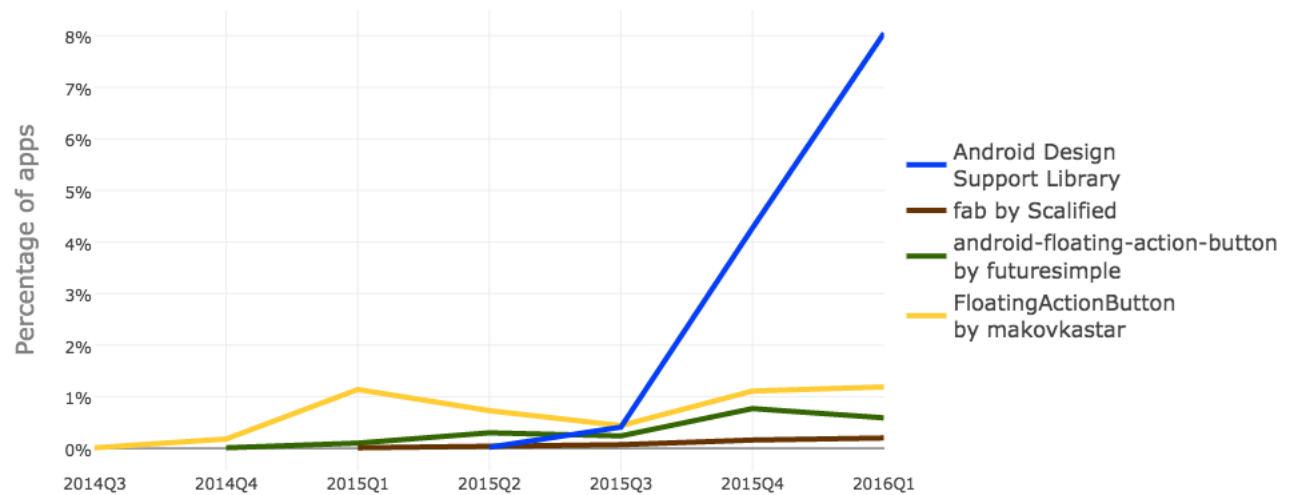


Figure 7.3: The percentage of apps that started adopting the Floating Action Button (FAB) using four popular libraries by calendar quarter.

#### 7.1.1.2 Navigation Drawer

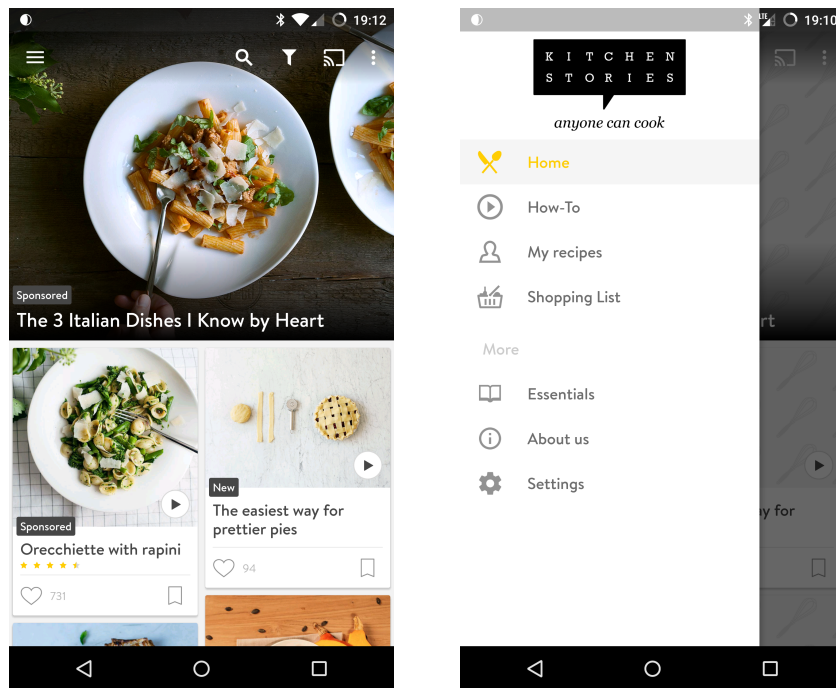


Figure 7.4: An example of an app with a closed navigation drawer (left) and an open navigation drawer (right).

The navigation drawer is a hidden menu panel that can be revealed by tapping on the app icon menu (also known as the hamburger icon), which causes the panel to slide from left to right (figure 7.4). The navigation drawer has existed before the concepts of material design were introduced and was later added to the list of material design patterns with minor styling recommendation. In order to create a navigation drawer, a developer needs to add a root element to the layout file with two children. The root element must be a specific view called *android.support.v4.widget.DrawerLayout*, which holds two children. The first child can be any view group element to hold the content of the app. The second child is a view that contains the content of the navigation drawer, which can be populated statically or dynamically with the drawer's list of items (e.g., *ListView* or *android.support.design.widget.NavigationView*). To find apps with a navigation drawer, I use the following Sieveable search queries:

```
MATCH app
WHERE
  <android.support.v4.widget.DrawerLayout>
    <android.support.design.widget.NavigationView />
  </android.support.v4.widget.DrawerLayout>
  <date-published>(*)</date-published>
  <rating>(*)</rating>
  <downloads>(*)</downloads>
RETURN app, l$1 AS rDate, l$2 as rating, l$3 as downloads

MATCH app
WHERE
  <android.support.v4.widget.DrawerLayout>
    <_>
    <_>
  </android.support.v4.widget.DrawerLayout>
  <date-published>(*)</date-published>
  <rating>(*)</rating>
  <downloads>(*)</downloads>
RETURN app, l$1 AS rDate, l$2 as rating, l$3 as downloads
```

These two search queries returned 20,165 unique apps in total. To find the adoption rate of this design pattern over the years, I remove the duplicate apps from the results, keep the first version in which the adoption is observed, and group the remaining results by year. Below, figure 7.5 shows the adoption rate of the navigation drawer over the years from 2013 to 2016. We can see an increase

over the years but at a relatively slow rate (6.95% is the average annual adoption increase). We can



Figure 7.5: The adoption rate of the Navigation Drawer over the years.

group the results by the release date quarterly and download count to see the trends in adopting this design pattern among most and less downloaded apps. To better approximate the popularity of Android apps, I group the apps by download count into three main groups: a) most downloaded apps for apps with one million or more download times, b) middle downloaded apps for apps with 100,000 and fewer than 1,000,000 download counts, and c) least download apps for apps with fewer than 100,000 download counts. Figure 7.6 shows that the adoption rate of the navigation drawer is more prevalent in apps with the most download count than others with the exception of the first two quarters of 2015. The analysis shows how the adoption of this design is changing after multiple major platform events. We can see that this design is more adopted in top downloaded apps. It is also noticeable that it experienced a period of low popularity at some points in time but later gained a resurgence of interest after the release of an official library.

Analyzing apps over time enabled us to capture a correlation between design changes and major platform events. UI design patterns may become more or less popular over time, but they may react positively to a related platform event. This phenomenon would have been missed had we considered a single version of the app. Our finding suggests that the release of an official library (the design support library in this case) caused the adoption rate of the navigation drawer for apps that are less popular to soar. We observed that apps with fewer download counts maintained an adoption rate below 10% before the release of the official library, and that adoption doubled to over 20% after the release of the library. This event allowed developers of the less popular apps to reduce the gap with top apps in design adoption.

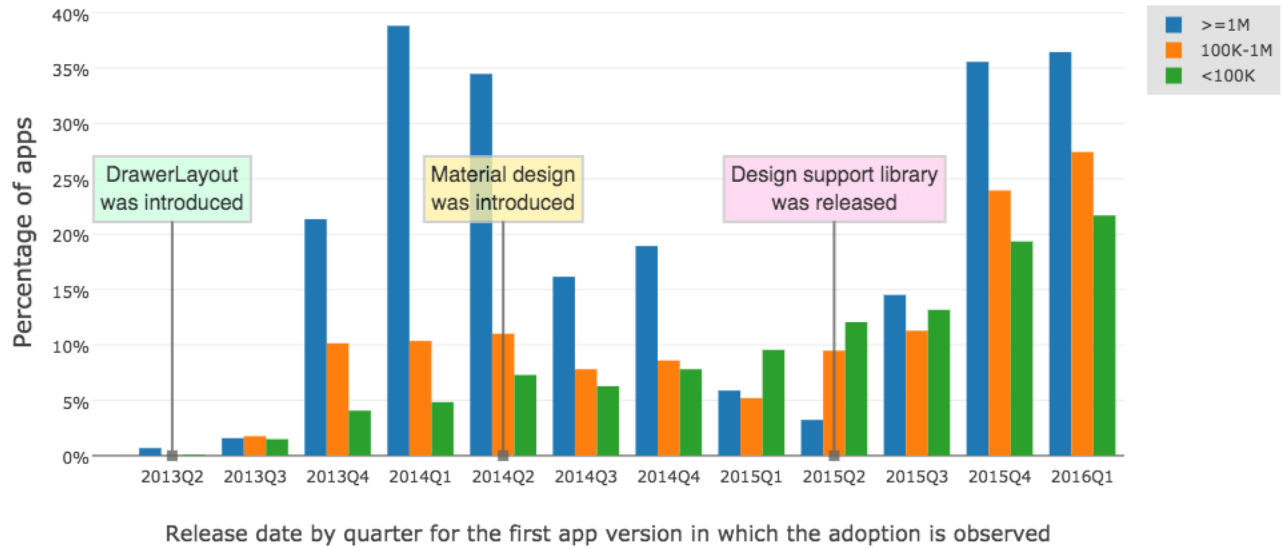


Figure 7.6: The percentage of apps that started adopting the navigation drawer grouped by three download count groups: most, middle, and least downloaded apps. The x-axis shows the calendar quarter of the release date in which the adoption is observed, and the y-axis shows the percentage of apps that adopted it. The percentage value is computed by dividing the number of adopted apps in each quarter over the total of apps released in each quarter. The time of major platform events is also marked in the chart.

## 7.2 Accessibility Mining Analysis

The users of mobile devices are diverse with different physical capabilities including people with disabilities. Some users may not be able to touch or see the screen, distinguish between its colors, hear a sound from the app, or speak back to the app. Apps that are inaccessible to users with disabilities may exclude them from using important services and social activities. Android features a set of accessibility APIs that help developers build accessible apps for users who have special needs. Many research and development efforts are invested in addressing various accessibility issues and enhance existing system-wide accessibility services. Platform owners provide accessibility services, APIs, and guidelines to help developers improve the accessibility of their apps and increase their audiences. But what is the impact of these efforts? Are these guidelines effective? Over the past



few years since these APIs were released, have developers (collectively as a community) been doing a better or worse job in making their apps more accessible? If there's an improvement, has the improvement been steady, accelerating, or plateauing? As a research community, should we be satisfied with our progress or should we do more? Data collection and analysis of apps for the presence of accessibility problems can help us answer these questions and bring more attention to this area.

Advancement in accessibility tools have wide range of benefits and applications to all users including users with no special needs. For instance, some of the technologies that we use on a daily basis started as an accessibility tools and evolved to benefit all users (e.g., speech recognition and word completion). Even a user with no disability may experience accessibility restrictions in certain environments, i.e., when someone is not able to touch the device while driving, or distinguish between colors in a sunny environment. Therefore, an app that is more accessible can benefit all users. In the following section, I demonstrate how Sieveable can be used to answer some of the questions described above and quantify the prevalence of accessibility violations over time.

### **7.2.1 Accessibility Violation: Labeling Visual UI controls**

Modern mobile platform features accessibility guidelines that serve as a living document that is always updated with changes to the platform. In android, the accessibility guidelines [1] provide developers with a checklist of accessibility requirements to help developers build more accessible applications [3]. The first item of this list is labeling visual user interface controls that do not have visible text. This is a common accessibility violation in UI elements such as `ImageButton` and `ImageView`. In many applications, UI controls use visual images to indicate the usage (e.g., a camera icon on an `ImageButton` to take a picture). A UI control with no descriptive text (also known as alternate text) is simply inaccessible to users with visual impairments. It is worth noting that not all `ImageView` elements should be labeled since some are used for aesthetic purposes or visual spacing. However, it is particularly important to label all `ImageButton` elements since they meant to represent an action in the app. Thus, analyzing apps for the presence of labels in

ImageButton elements tend to produce more accurate results than other UI elements.

One can use Sieveable to investigate the extent of this bug and whether developers pay attention to it over the years. Below are the search queries I use in this analysis:

```
<!-- Find all apps that have an ImageButton -->
MATCH app
WHERE
  <ImageButton />
  <date-published>(*)</date-published>
  <downloads>(*)</downloads>
RETURN app, l$1 AS rDate, l$2 as downloads

<!-- Find all apps that set the contentDescription statically in layout files-->
MATCH app
WHERE
  <ImageButton android:contentDescription="*" />
  <date-published>(*)</date-published>
  <downloads>(*)</downloads>
RETURN app, l$1 AS rDate, l$2 as downloads

<!-- Find all apps that set the contentDescription dynamically in source code files-->
MATCH app
WHERE
  <code class="ImageButton" method="setContentDescription" />
  <date-published>(*)</date-published>
  <downloads>(*)</downloads>
RETURN app, l$1 AS rDate, l$2 as downloads
```

The first query is for finding apps that have an image button, while the last two queries are for finding apps that added a text label to an image button statically in layout files or dynamically in the source code. We can analyze the results of these search queries and compute the ratio of labeled image buttons in each app. We compute the percentage of accessible image buttons in each app as follows:  $coverage = \frac{image\_buttons\_with\_labels}{total\_image\_buttons}$ . Then, we compute the weighted mean for all apps in each year, the standard deviation as a measure of variability, and the standard error of the mean as follows:

$$\bar{x} = \frac{\sum_{i=1}^N w_i x_i}{\sum_{i=1}^N w_i} \quad , \quad SD_{\bar{x}} = \sqrt{\frac{\sum_{i=1}^n w_i (x_i - \bar{x}^*)^2}{\frac{N-1}{N} \sum_{i=1}^n w_i}} \quad , and \quad SE_{\bar{x}} = \frac{SD_{\bar{x}}}{\sqrt{N}}$$

where  $w_i$  is the number of image buttons,  $x_i$  is the coverage, and  $N$  is the total number of observations (apps with image buttons). Figure 7.7 shows the weighted mean of accessible image

button elements with text labels in all apps released from 2010 to 2016. We observed, on average, around 5% increase of accessible image buttons from 2012 to 2014 but that improvement growth rate reached a plateau at a lower level in 2015 and 2016.

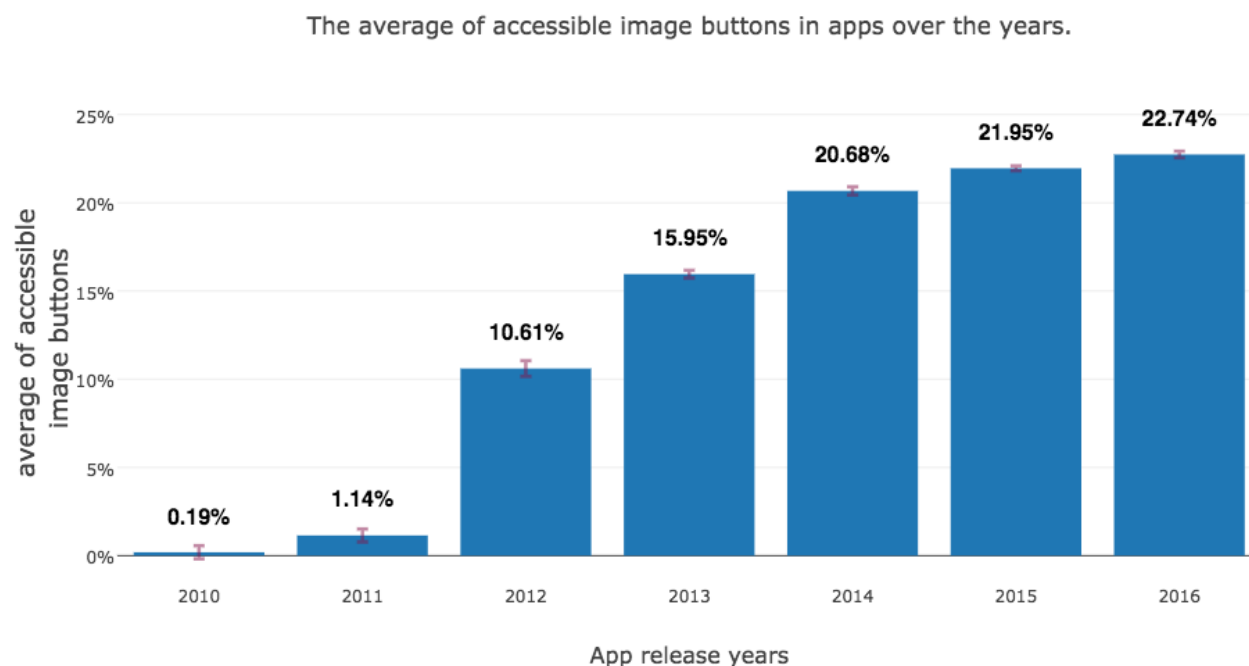


Figure 7.7: The average of accessible image buttons in apps over the release years. The error bars represent the standard error of the mean.

We can also sample the apps with no accessible image buttons (i.e. an app that has no label in all of its image buttons) and group them by download count. Sieveable enables us to count for cases where text labels are added dynamically in the source code. Thus, the results of the previous queries are more accurate and can be used to quantify apps that do not label image buttons at all. Figure 7.8 shows the percentage of apps in three download count groups (top, middle, and bottom) that had no text labels in all the image buttons they used. This result shows that this accessibility bug is common across all apps of different download times. It was even more common in the top downloaded apps in the years from 2011 to 2014. However, it dropped in the following two years to become relatively common in all three download count groups.

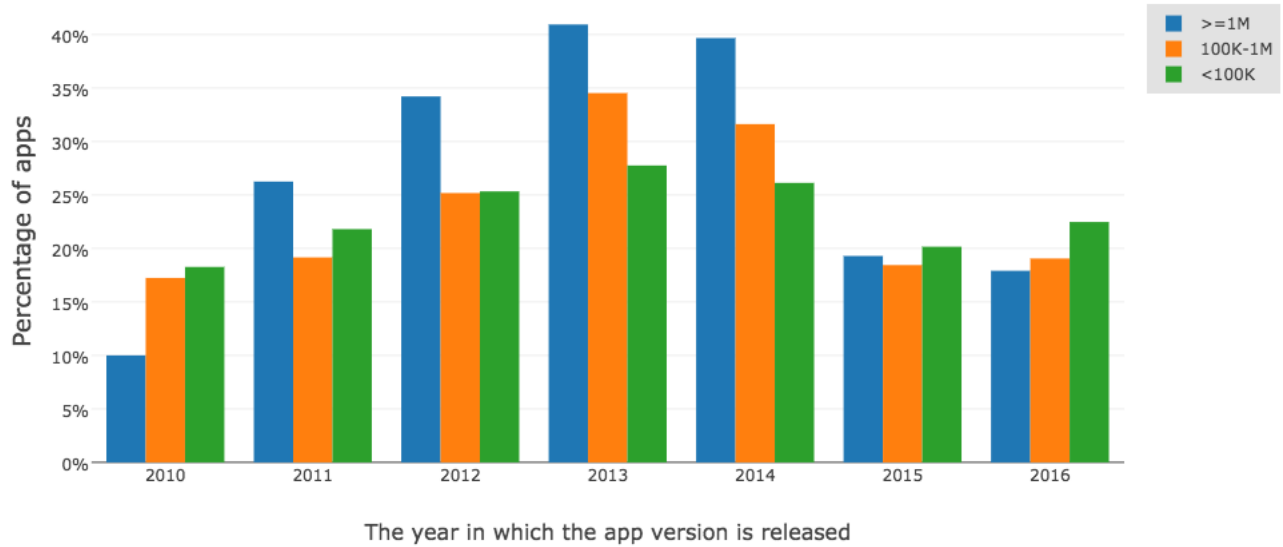


Figure 7.8: The percentage of apps that have no accessible ImageButton elements grouped by download count and release year.

The results of this analysis show a mixed view of accessibility improvements in Android apps over the years. While we have seen a small yet noticeable improvement in the years before 2015 but that growth rate plateaued in 2016. These findings suggest that accessibility is still a problem for all apps including the most downloaded apps and bring to light the prevalence of this accessibility problem across all apps regardless of their popularity levels. The results presented here highlight the potential of using Sieveable in future work to estimate the accessibility of mobile apps over time and identify areas where accessibility violations are more severe.

### 7.3 Privacy Mining Analysis

Android uses a permission system to protect privacy- and security-sensitive resources. Android has recently changed its permission's model from install-time (pre 6.0) to a runtime permission system (6.0 and above). In the install-time model, users are prompted with a list of permissions an app requires when they try to install the app. In the runtime model, instead of showing permissions at install-time, Android gives users the choice of granting or denying a permission while running the app. Many research efforts have studied several privacy and security problems in the

permission’s model of Android (discussed in detail in chapter 3). However, previous studies have been limited in their understanding of permission changes over time. It is valuable to ask how permission usage changes in Android apps. Do apps tend to request fewer or more permissions over time? Are developers becoming privacy-conscious or non-privacy-conscious over the years? Did they ask for more permissions to use in new features or included libraries? In this section, through an analysis of permission usage changes, I demonstrate how easy it is to use Sieveable to perform

these analyses. The analysis presented here uses the following Sieveable query:

```
MATCH app
WHERE
  <uses-permission android:name="(android.permission.*)" />
  <date-published>(*)</date-published>
  <rating>(*)</rating>
  <downloads>(*)</downloads>
RETURN app, l$1 AS rDate, l$2 as rating, l$3 as downloads, m$1 as permissions
```

This query returns the system permissions, release date, rating, and download count for all apps in the dataset. Since the goal of this analysis is to study permission changes over time, I removed all apps that had only one version in the dataset. This narrowed the sample of apps to 50,618 unique apps. The analysis of the search results is presented in two subsections: added and dropped permissions.

### 7.3.1 Analyzing Permission Usage

The average number of requested permissions is 7.33 permissions (standard deviation is 5.30). Figure 7.9 shows the distribution of the total requested permissions in apps. It shows that the majority of apps requested two permissions in total. These apps appeared to be less popular (79.12% with less than 100,000 downloads). This raises the question what are these two permissions that these apps depend on? To answer this question, I aggregated the list permissions for these apps and found that the top three permissions they requested to be: *android.permission.INTERNET*, *android.permission.ACCESS\_NETWORK\_STATE*, *android.permission.WRITE\_EXTERNAL\_STORAGE*. The first two permissions are found to be among the most frequently used permissions in ad libraries [95]. In addition, I analyzed the apps that had an unusual total of requested permissions

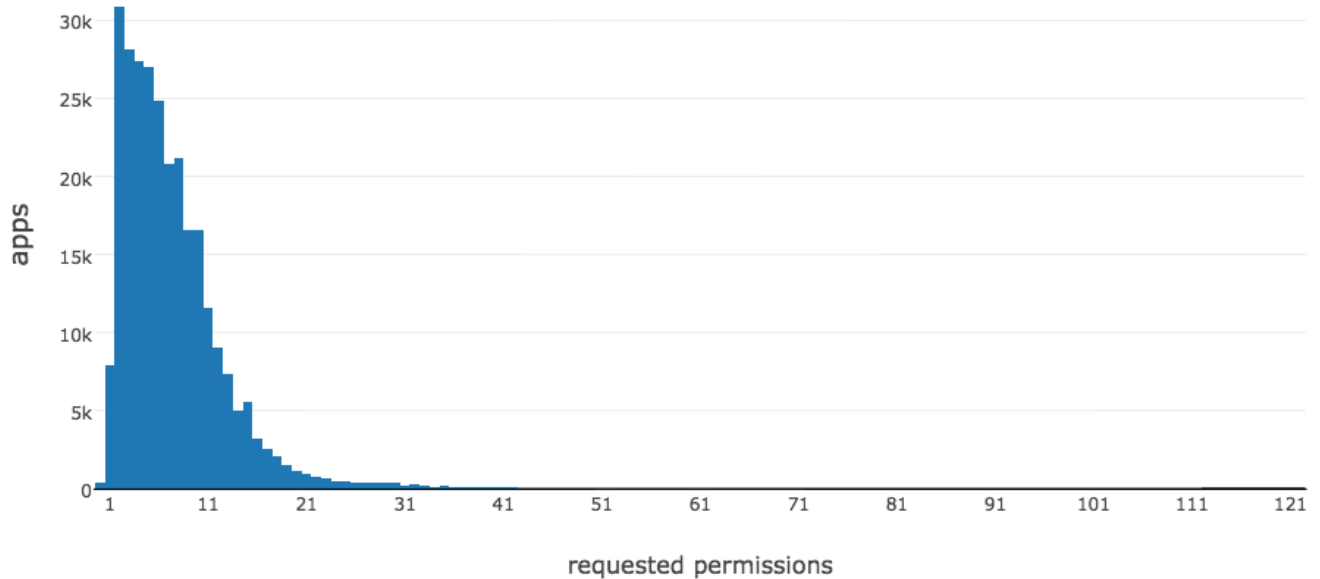


Figure 7.9: The total number of requested permissions.

(a sum total that falls within the most 10 requested permission counts). I found that 90% of these apps have a download count of 100,000 or less and only one app (the T-Mobile official app) that has requested 86 permissions and had 10 million downloads. The app with the highest requested permission (122 permissions) is called Webkey (package name is com.webkey). This app is listed under the tools category, had 100,000 downloads, 4.44 star-rating, and claims that it provides a remote control of the device from any browser. Finally, I discovered that the average number of requested permission keeps increasing over the years (see table 7.1). This suggest that developers tend to add more permissions over the years which make users more susceptible to privacy and security problems over time.

### 7.3.2 Analyzing Added Permissions

This analysis seeks to discover the trends in adding permissions throughout the apps' release history. In the entire dataset, I found that 13,976 apps (27.61%) asked for more permissions in their updates. On average, they added 2.28 permissions (standard deviation is 2.57).

Year	Average	Standard Deviation	Total apps
2010	2.82	2.79	547
2011	3.62	2.85	2,718
2012	4.95	4.17	7,518
2013	6.56	4.77	50,190
2014	6.71	4.99	55,242
2015	7.72	5.35	120,840
2016	8.72	5.97	40,435

Table 7.1: Summary statistics on the total number of permissions requested by apps over the years.

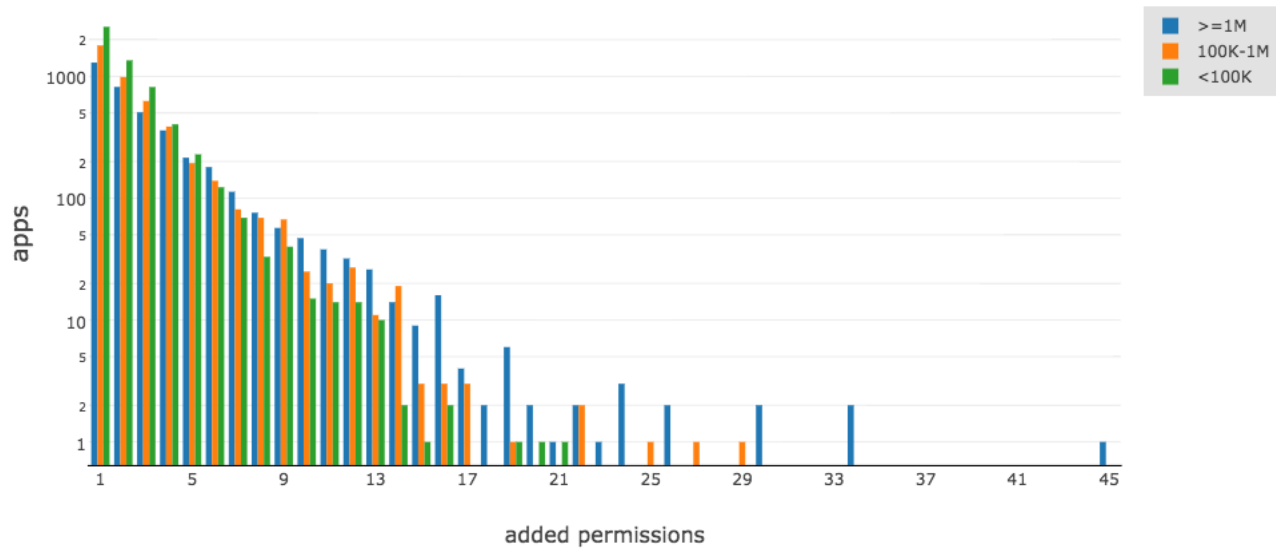


Figure 7.10: The total number of added permissions in apps grouped by download count.

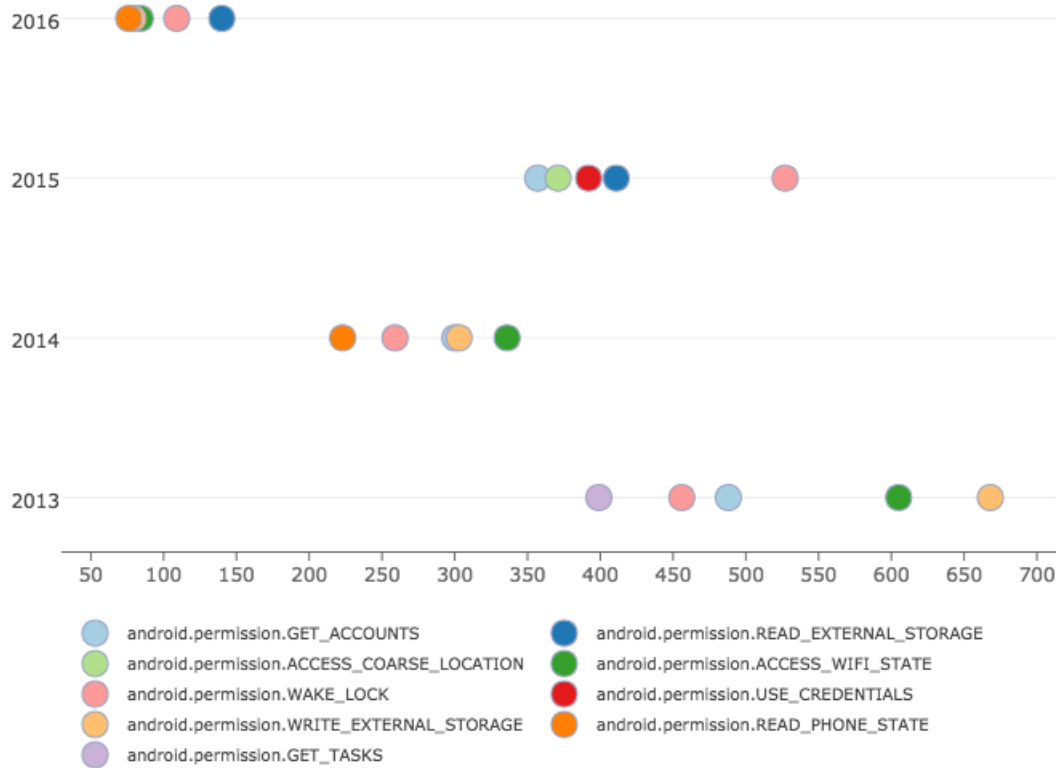


Figure 7.11: The five most added permissions in each year. The x-axis shows how many times the permission was added in each year.

Figure 7.10 shows the number of added permissions for apps by three download count groups (top, middle, bottom). It shows that the addition of one to five permissions seemed to be normal and relatively similar among the three download groups. But as the number of added permissions increases to 15 or more, all the observations appear as outliers.

Adding these permissions may have happened as a result of a framework upgrade, library adoption, or added features throughout the time of releasing a new app version. To explore the trends in adding permissions over time, I obtain a list of the five most added permissions in each year ( see figure 7.11). This graph shows that the most added permissions are always among the most frequently used permissions in ad libraries [95] The *android.permission.GET\_TASKS* appeared as the fifth most added permission in 2013, which is used to get the user's most recent tasks. Despite that APIs that depend on this permission could leak personal information, it can be granted



without the user’s approval at install time. It has been later deprecated (in Android 5.0), however, to protect user’s privacy. The permission named *android.permission.USE\_CREDENTIALS* appeared as the third most added permissions in 2015. This permission along with the *android.permission.GET\_ACCOUNTS* usually get added automatically during the build process when the developer adds the Google Play services library to the dependencies list in the app’s *gradle.build* file. It was later deprecated in API 23 (Android 6.0), so it was not seen among the five most added permissions in 2016. These findings suggest that apps tend to add more permissions related to either dependency or ad libraries rather than core app features.

## 7.4 Summary

This chapter has demonstrated how Sieveable simplified the task of analyzing apps over time at an unprecedented levels of analyses. I presented a number of analyses in three main areas: design, accessibility, and privacy. In design, the release of official design helper libraries enabled apps that had few downloads to reduce the gap with the top downloaded apps in adopting best design practices. In accessibility, all apps including the most downloaded ones had significant accessibility problems. In privacy, I uncovered that apps with few downloads tend to request either a small number of permissions, which are common in ad libraries, or a rather large and unusual number of permissions compared to the apps with high download counts. Furthermore, the most added permissions in each year are the ones often required by ad libraries, which raises privacy concerns. These analyses provided insights on how apps changed at an unprecedented level than previous research, both in terms of the depth of analysis (i.e. listing details, design, and code data) and the time studied (over multiple app versions).

## Chapter 8

### Conclusion and Future Work

#### 8.1 Conclusion

This dissertation introduced a deep and longitudinal approach to mining mobile applications. I used the term “deep” to refer to the deep and structural indexing of apps across multiple levels, and the term “longitudinal” to refer to observing them over multiple points of time. The ultimate contribution of this dissertation research is the advancement of understanding mobile apps at deep and longitudinal levels. The work proposed here sought to gain insights into mobile apps by enabling the analysis at unprecedented depth over time. Specifically, this dissertation aimed to answer the following two research questions:

- (1) How can we enable deep searching and mining of mobile apps over time?
- (2) What are the benefits of the deep and longitudinal approach to mining mobile apps?

In answering the first question, a large dataset has been collected and led to the development and deployment of a large-scale retrieval platform called Sieveable (discussed in chapter 5). In answering the second question, I used Sieveable to present several types of analyses that would have been difficult to perform otherwise (discussed in chapter 7). Some highlights of my findings include: 1) In user interface design, the release of official design libraries enabled new applications to reduce the gap with the most downloaded ones in adopting best design practices. 2) In accessibility, results showed that accessibility is a problem for many applications including the most downloaded ones.

3) In privacy, the most added permissions in each year are the ones often required by ad libraries, which raises privacy concerns.

## 8.2 Future Work

There are a number of promising future directions for the work presented in this dissertation.

### Leveraging Dynamic Analysis

A key inherent limitation of Sieveable is the fact that an analyst needs to know what she is looking for and translates that into a search query. This limitation can be overcome by leveraging dynamic analysis tools. The dynamic or runtime-based analysis involves running the app and analyzing its visual design or behavior. This can be applied by creating a dynamic analysis platform that uses a collection of virtual mobile devices and paid crowds that are capable of translating visual design examples into search queries. However, not all visual design concepts can be translated into search queries that can be submitted into a static-analysis-based retrieval system. For instance, UI screen transitions and animations are hard to capture because they involve complex dynamic, event-based changes. Exploring this area and the possibility of combining the strengths of both static and dynamic analysis could result in novel systems and new kind of applications.

### Systematic Evaluation Studies

I demonstrated the benefits of the deep and longitudinal approach by conducting a diverse number of analyses in three domains: visual design, mobile accessibility, and privacy. This exploration highlights the potential in going beyond that and applying comprehensive studies to identifying key problems in these areas. For instance, accessibility assessment involves many complex considerations for multiple situations to deem an app accessible. A systematic investigation that applies advanced static and dynamic analysis techniques could provide us with insights to address serious problems in this area.

## **Ranking Algorithms**

The number of apps has increased dramatically in mobile marketplaces. As a result, finding a new and well-designed app is becoming an exercise in frustration. Most if not all existing search ranking algorithms do not take into account apps' internal data when ranking search results. Search systems miss a great opportunity in creating novel ranking algorithms based on apps' data. Employing new ranking factors such as accessibility, design, privacy, etc. could lead to significant benefits to users.

## **Personalized Recommender Systems**

The existing recommender systems in marketplaces do not recommend apps based on user's preferences. It will be useful to recommend an app to a user if it matches her preferences. Users have different visual aesthetic or privacy preferences. One promising example of personalized recommender systems is a visual design recommender system. By extracting visual design features of each app and learning the preferences of each user, we can recommend apps that match users' design preferences.

## **8.3 Summary**

This dissertation introduced an approach to mining large and ever-changing marketplaces by taking a deep and longitudinal perspectives. I presented several analyses that encompassed this approach and uncovered new findings. I believe that additional efforts in this research area could result in innovative systems that could empower mobile developers and users. I hope this work inspires others to apply an approach with a deep and longitudinal perspectives to new problems.

## Bibliography

- [1] Accessibility — Android Developers. <https://developer.android.com/design/patterns/accessibility.html>. Accessed: 2016-10-24.
- [2] Action Bar - Adding Navigation Tabs. <http://developer.android.com/guide/topics/ui/actionbar.html#Tabs>. Accessed: 2015-01-15.
- [3] Android Accessibility Checklist. <https://developer.android.com/guide/topics/ui/accessibility/checklist.html>. Accessed: 2016-10-19.
- [4] Android Design Support Library. <https://android.googlesource.com/platform/frameworks/support/+/master>. Accessed: 2016-10-19.
- [5] Android Design Support Library. <http://android-developers.blogspot.com/2015/05/android-design-support-library.html>. Accessed: 2016-10-21.
- [6] android-floating-action-button. <https://github.com/futuresimple/android-floating-action-button>. Accessed: 2016-10-19.
- [7] Android Hierarchy Viewer. <https://developer.android.com/studio/profile/hierarchy-viewer.html>. Accessed: 2016-11-22.
- [8] Apache Solr. <http://lucene.apache.org/solr>.
- [9] Apache Zookeeper. <https://zookeeper.apache.org>.
- [10] Apktool - A tool for reverse engineering Android apk files. <https://github.com/iBotPeaches/Apktool>. Accessed: 2015-10-29.
- [11] App Annie - The App Analytics and App Data Industry Standard. <https://www.appannie.com/>. Accessed: 2015-11-01.
- [12] App Store Analytics - App Store Sentiment Analysis - Applause Analytics. <http://www.applause.com/mobile-analytics>. Accessed: 2015-11-01.
- [13] App Store Analytics for iOS and Android developers. <https://appfigures.com/>. Accessed: 2015-11-01.
- [14] AppBrain - Number of Android applications. <http://www.appbrain.com/stats/number-of-android-apps>. Accessed: 2016-10-12.

- [15] Chrome Web Store. [https://en.wikipedia.org/wiki/Chrome\\_Web\\_Store](https://en.wikipedia.org/wiki/Chrome_Web_Store). Accessed: 2016-10-19.
- [16] Design for windows runtime apps windows app development. <https://dev.windows.com/en-us/design>. Accessed: 2015-04-23.
- [17] fab. <https://github.com/Scalified/fab>. Accessed: 2016-10-19.
- [18] FloatingActionButton. <https://github.com/makovkastar/FloatingActionButton>. Accessed: 2016-10-19.
- [19] ios human interface guidelines: Designing for ios. <https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/>. Accessed: 2015-04-23.
- [20] Jasmin - an assembler for the Java Virtual Machine. <http://jasmin.sourceforge.net>. Accessed: 2015-11-01.
- [21] Material design - Google design guidelines. <https://www.google.com/design/spec/material-design/introduction.html>. Accessed: 2016-04-01.
- [22] MongoDB. <https://www.mongodb.org>.
- [23] Navigation Drawer. <https://developer.android.com/design/patterns/navigation-drawer.html>. Accessed: 2015-01-15.
- [24] Number of apps available in leading app stores 2016. <https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores>. Accessed: 2016-10-19.
- [25] Patterns — android developers. <https://developer.android.com/design/patterns/index.html>. Accessed: 2015-04-23.
- [26] Pinshape. <https://Pinshape.com>. Accessed: 2016-10-19.
- [27] Security Enhancements in Jelly Bean. <http://goo.gl/ODjFYE>. Accessed: 2015-07-10.
- [28] Web design and applications - w3c. <http://www.w3.org/standards/webdesign>. Accessed: 2015-04-23.
- [29] Eytan Adar. Temporal-Informatics of the WWW. PhD thesis, Citeseer, 2009.
- [30] Charu C Aggarwal. Data mining: the textbook. Springer, 2015.
- [31] Khalid Alharbi and Tom Yeh. Collect, decompile, extract, stats, and diff: Mining design pattern changes in android apps. In Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services, pages 515–524. ACM, 2015.
- [32] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Outeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

- [33] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In CCS, pages 217–228, 2012.
- [34] Ricardo Baeza-Yates, Di Jiang, Fabrizio Silvestri, and Beverly Harrison. Predicting the next app that you are going to use. In Proceedings of the Eighth ACM International Conference on Web Search and Data Mining, WSDM '15, pages 285–294, New York, NY, USA, 2015. ACM.
- [35] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pages 681–682. ACM, 2006.
- [36] Ziv Bar-Yossef and Sridhar Rajagopalan. Template detection via data mining and its applications. In Proceedings of the 11th international conference on World Wide Web, pages 580–591. ACM, 2002.
- [37] Andrea Basso, David Goldberg, Steven Greenspan, and David Weimer. First impressions: Emotional and cognitive factors underlying judgments of trust e-commerce. In Proceedings of the 3rd ACM Conference on Electronic Commerce, EC '01, pages 137–143, New York, NY, USA, 2001. ACM.
- [38] José A Borges, Israel Morales, and Néstor J Rodríguez. Guidelines for designing usable world wide web pages. In Conference Companion on Human Factors in Computing Systems, pages 277–278. ACM, 1996.
- [39] L. Brichter. User interface mechanics, August 5 2010. US Patent App. 12/756,574.
- [40] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In Seventh International World-Wide Web Conference (WWW 1998), 1998.
- [41] Matthieu Caneill and Stefano Zacchiroli. Debsources: Live and historical views on macro-level software evolution. In Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, pages 28:1–28:10, New York, NY, USA, 2014. ACM.
- [42] Antonio Carzaniga, Andrea Mattavelli, and Mauro Pezzè. Measuring software redundancy. In Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pages 156–166, Piscataway, NJ, USA, 2015. IEEE Press.
- [43] Deepayan Chakrabarti, Ravi Kumar, and Kunal Punera. Page-level template detection via isotonic smoothing. In Proceedings of the 16th international conference on World Wide Web, pages 61–70. ACM, 2007.
- [44] Soumen Chakrabarti, Byron E Dom, S Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, Andrew Tomkins, David Gibson, and Jon Kleinberg. Mining the web’s link structure. Computer, 32(8):60–67, 1999.
- [45] Kerry Shih-Ping Chang and Brad A Myers. Webcrystal: understanding and reusing examples in web authoring. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 3205–3214. ACM, 2012.

- [46] Ning Chen, Steven C.H. Hoi, Shaohua Li, and Xiaokui Xiao. Simapp: A framework for detecting similar mobile applications by online kernel learning. In Proceedings of the Eighth ACM International Conference on Web Search and Data Mining, WSDM '15, pages 305–314, New York, NY, USA, 2015. ACM.
- [47] Ning Chen, Jialiu Lin, Steven CH Hoi, Xiaokui Xiao, and Boshen Zhang. Ar-miner: mining informative reviews for developers from mobile app marketplace. In Proceedings of the 36th International Conference on Software Engineering, pages 767–778. ACM, 2014.
- [48] Elliot J Chikofsky, James H Cross, et al. Reverse engineering and design recovery: A taxonomy. Software, IEEE, 7(1):13–17, 1990.
- [49] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In 17th European Symposium on Research in Computer Security (ESORICS), pages 37–54. Springer, 2012.
- [50] Flavio De Souza and Nigel Bevan. The use of guidelines in menu interface design: Evaluation of a draft standard. In INTERACT, pages 435–440. Citeseer, 1990.
- [51] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 422–431, Piscataway, NJ, USA, 2013. IEEE Press.
- [52] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. Software Engineering, IEEE Transactions on, 27(1):1–12, Jan 2001.
- [53] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones. In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.
- [54] William Enck, Damien Ocateau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In Proceedings of the 20th USENIX Conference on Security, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.
- [55] Oren Etzioni. The world-wide web: quagmire or gold mine? Communications of the ACM, 39(11):65–68, 1996.
- [56] Ethan Fast, Daniel Steffee, Lucy Wang, Joel R Brandt, and Michael S Bernstein. Emergent, crowd-scale programming practice in the ide. In Proceedings of the 32nd annual ACM conference on Human factors in computing systems, pages 2491–2500. ACM, 2014.
- [57] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In CCS, pages 627–638. ACM, 2011.
- [58] M. Frank, Ben Dong, A.P. Felt, and D. Song. Mining permission request patterns from android and facebook applications. In Data Mining (ICDM), 2012 IEEE 12th International Conference on, pages 870–875, Dec 2012.



- [59] Bin Fu, Jialiu Lin, Lei Li, Christos Faloutsos, Jason Hong, and Norman Sadeh. Why people hate your app: Making sense of user feedback in a mobile app store. In Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 1276–1284. ACM, 2013.
- [60] David Gibson, Kunal Punera, and Andrew Tomkins. The volume and evolution of web page templates. In Special interest tracks and posters of the 14th international conference on World Wide Web, pages 830–839. ACM, 2005.
- [61] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In Proceedings of the 36th International Conference on Software Engineering, pages 1025–1035. ACM, 2014.
- [62] John D. Gould and Clayton Lewis. Designing for usability—key principles and what designers think. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '83, pages 50–53, New York, NY, USA, 1983. ACM.
- [63] Viet Ha-Thuc, Yelena Mejova, Christopher Harris, and Padmini Srinivasan. A relevance-based topic model for news event tracking. In Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval, pages 764–765. ACM, 2009.
- [64] M. Harman, Yue Jia, and Yuanyuan Zhang. App store mining and analysis: Msr for app stores. In Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on, pages 108–111, June 2012.
- [65] Jan Hartmann, Antonella De Angeli, and Alistair Sutcliffe. Framing the user experience: Information biases on website quality judgement. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '08, pages 855–864, New York, NY, USA, 2008. ACM.
- [66] Jan Hartmann, Alistair Sutcliffe, and Antonella De Angeli. Towards a theory of user judgment of aesthetics and user interface quality. ACM Transactions on Computer-Human Interaction (TOCHI), 15(4):15, 2008.
- [67] Scarlett R Herring, Chia-Chen Chang, Jesse Krantzler, and Brian P Bailey. Getting inspired!: understanding how and why examples are used in creative design practice. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 87–96. ACM, 2009.
- [68] Thomas T Hewett and Charles T Meadow. On designing for usability: an application of four key principles. ACM SIGCHI Bulletin, 17(4):247–252, 1986.
- [69] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In Proceedings of the 27th International Conference on Software Engineering, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.
- [70] Guangyan Huang, Jing He, Yanchun Zhang, Wanlei Zhou, Hai Liu, Peng Zhang, Zhiming Ding, Yue You, and Jian Cao. Mining streams of short text for analysis of world-wide event evolutions. World Wide Web, pages 1–17, 2014.
- [71] Melody Y. Ivory and Marti A. Hearst. Statistical Profiles of Highly-rated Web Sites. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '02, pages 367–374, New York, NY, USA, 2002. ACM.

- [72] Melody Y Ivory, Rashmi R Sinha, and Marti A Hearst. Empirically validated web page design metrics. In Proceedings of the SIGCHI conference on Human factors in computing systems, pages 53–60. ACM, 2001.
- [73] Charles Jacobs, Wilmot Li, Evan Schrier, David Barger, and David Salesin. Adaptive grid-based document layout. In ACM transactions on graphics (TOG), volume 22, pages 838–847. ACM, 2003.
- [74] Karen Sparck Jones. Readings in information retrieval. Morgan Kaufmann, 1997.
- [75] David Kawrykow and Martin P. Robillard. Improving api usage through automatic detection of redundant code. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, pages 111–122, Washington, DC, USA, 2009. IEEE Computer Society.
- [76] Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. Memories of bug fixes. In Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pages 35–45, New York, NY, USA, 2006. ACM.
- [77] Sunghun Kim, T. Zimmermann, Kai Pan, and E.J. Whitehead. Automatic identification of bug-introducing changes. In Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on, pages 81–90, Sept 2006.
- [78] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. J. ACM, 46(5):604–632, September 1999.
- [79] Scott R Klemmer, Mark W Newman, Ryan Farrell, Mark Bilezikjian, and James A Landay. The designers' outpost: a tangible interface for collaborative web site. In Proceedings of the 14th annual ACM symposium on User interface software and technology, pages 1–10. ACM, 2001.
- [80] Deguang Kong, Lei Cen, and Hongxia Jin. Autoreb: Automatically understanding the review-to-behavior fidelity in android applications. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 530–541. ACM, 2015.
- [81] Raymond Kosala and Hendrik Blockeel. Web mining research: A survey. ACM Sigkdd Explorations Newsletter, 2(1):1–15, 2000.
- [82] Steve Krug. Don't Make Me Think: A Common Sense Approach to Web Usability. 2nd edition edition, August 2005.
- [83] Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R Klemmer, and Jerry O Talton. Webzeitgeist: design mining the web. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 3083–3092. ACM, 2013.
- [84] Ranjitha Kumar, Jerry O Talton, Salman Ahmad, and Scott R Klemmer. Bricolage: example-based retargeting for web design. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 2197–2206. ACM, 2011.
- [85] James A Landay and Brad A Myers. Interactive sketching for the early stages of user interface design. In Proceedings of the SIGCHI conference on Human factors in computing systems, pages 43–50. ACM Press/Addison-Wesley Publishing Co., 1995.

- [86] Talia Lavie and Noam Tractinsky. Assessing dimensions of perceived visual aesthetics of web sites. International journal of human-computer studies, 60(3):269–298, 2004.
- [87] Brian Lee, Savil Srivastava, Ranjitha Kumar, Ronen Brafman, and Scott R Klemmer. Designing with interactive example galleries. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 2257–2266. ACM, 2010.
- [88] Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, and Sunghun Kim. Instant code clone search. In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pages 167–176, New York, NY, USA, 2010. ACM.
- [89] James Lin, Michael Thomsen, and James A Landay. A visual language for sketching large and complex interactive designs. In Proceedings of the SIGCHI conference on Human factors in computing systems, pages 307–314. ACM, 2002.
- [90] Jialiu Lin, Bin Liu, Norman Sadeh, and Jason I. Hong. Modeling users’ mobile app privacy preferences: Restoring usability in a sea of permission settings. In Symposium On Usable Privacy and Security (SOUPS 2014), pages 199–212, Menlo Park, CA, 2014. USENIX Association.
- [91] Jovian Lin, Kazunari Sugiyama, Min-Yen Kan, and Tat-Seng Chua. New and improved: modeling versions to improve app recommendation. In Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval, pages 647–656. ACM, 2014.
- [92] Gitte Lindgaard, Cathy Dudek, Devjani Sen, Livia Sumegi, and Patrick Noonan. An exploration of relations between visual appeal, trustworthiness and perceived usability of homepages. ACM Transactions on Computer-Human Interaction (TOCHI), 18(1):1, 2011.
- [93] Gitte Lindgaard, Gary Fernandes, Cathy Dudek, and Judith Brown. Attention web designers: You have 50 milliseconds to make a good first impression! Behaviour & information technology, 25(2):115–126, 2006.
- [94] Bin Liu, Deguang Kong, Lei Cen, Neil Zhenqiang Gong, Hongxia Jin, and Hui Xiong. Personalized mobile app recommendation: Reconciling app functionality and user privacy preference. In Proceedings of the Eighth ACM International Conference on Web Search and Data Mining, WSDM '15, pages 315–324, New York, NY, USA, 2015. ACM.
- [95] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '15, pages 89–103, New York, NY, USA, 2015. ACM.
- [96] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 1013–1024, New York, NY, USA, 2014. ACM.
- [97] Jonas Löwgren and Tommy Nordqvist. Knowledge-based evaluation as design support for graphical user interfaces. In Proceedings of the SIGCHI conference on Human factors in computing systems, pages 181–188. ACM, 1992.

- [98] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In Proceedings of the 2012 ACM conference on Computer and communications security, pages 229–240. ACM, 2012.
- [99] Haiping Ma, Huanhuan Cao, Qiang Yang, Enhong Chen, and Jilei Tian. A habit mining approach for discovering similar mobile users. In Proceedings of the 21st International Conference on World Wide Web, WWW '12, pages 231–240, New York, NY, USA, 2012. ACM.
- [100] Sanjay Kumar Madria, Sourav S Bhowmick, Wee Keong Ng, and Ee-Peng Lim. Research issues in web data mining. Springer, 1999.
- [101] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM.
- [102] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. Introduction to information retrieval, volume 1. Cambridge university press Cambridge, 2008.
- [103] William Martin, Federica Sarro, and Mark Harman. Causal impact analysis for app releases in google play. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pages 435–446, New York, NY, USA, 2016. ACM.
- [104] Stuart McIlroy, Nasir Ali, and Ahmed E. Hassan. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. Empirical Software Engineering, 21(3):1346–1370, 2016.
- [105] Qiaozhu Mei and ChengXiang Zhai. Discovering evolutionary theme patterns from text: An exploration of temporal text mining. In Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, KDD '05, pages 198–207, New York, NY, USA, 2005. ACM.
- [106] Scarlett R Miller and Brian P Bailey. Searching for inspiration: An in-depth look at designers example finding practices. In ASME 2014 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, pages V007T07A035–V007T07A035. American Society of Mechanical Engineers, 2014.
- [107] R. Minelli and M. Lanza. Software analytics for mobile applications—insights amp; lessons learned. In Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, pages 144–153, March 2013.
- [108] Aliaksei Miniukovich and Antonella De Angeli. Computation of interface aesthetics. In Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI '15, pages 1163–1172, New York, NY, USA, 2015. ACM.
- [109] Mark W Newman and James A Landay. Sitemaps, storyboards, and specifications: a sketch of web site design practice. In Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques, pages 263–274. ACM, 2000.

- [110] Mark W. Newman, James Lin, Jason I. Hong, and James A. Landay. Denim: An informal web site design tool inspired by observations of practice. Hum.-Comput. Interact., 18(3):259–324, September 2003.
- [111] H. Osman, M. Lungu, and O. Nierstrasz. Mining frequent bug-fix code changes. In Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on, pages 343–347, Feb 2014.
- [112] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: Towards automating risk assessment of mobile applications. In Proceedings of the 22Nd USENIX Conference on Security, SEC’13, pages 527–542, Berkeley, CA, USA, 2013. USENIX Association.
- [113] Dae Hoon Park, Mengwen Liu, ChengXiang Zhai, and Haohong Wang. Leveraging user reviews to improve accuracy for mobile app retrieval. In Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR ’15, pages 533–542, New York, NY, USA, 2015. ACM.
- [114] Thanasis Petsas, Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos, and Thomas Karagiannis. Rise of the planet of the apps: A systematic study of the mobile app ecosystem. In Proceedings of the 2013 Conference on Internet Measurement Conference, IMC ’13, pages 277–290, New York, NY, USA, 2013. ACM.
- [115] A Terry Purcell and John S Gero. Effects of examples on the results of a design activity. Knowledge-Based Systems, 5(1):82–91, 1992.
- [116] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In 2014 Network and Distributed System Security Symposium (NDSS), 2014.
- [117] Julie Ratner, Eric M Grose, and Chris Forsythe. Characterization and assessment of html style guides. In Conference Companion on Human Factors in Computing Systems, pages 115–116. ACM, 1996.
- [118] Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. The uniqueness of changes: Characteristics and applications. In Proceedings of the 12th Working Conference on Mining Software Repositories, MSR ’15, pages 34–44, Piscataway, NJ, USA, 2015. IEEE Press.
- [119] Katharina Reinecke and Abraham Bernstein. Improving performance, perceived usability, and aesthetics with culturally adaptive user interfaces. ACM Transactions on Computer-Human Interaction (TOCHI), 18(2):8, 2011.
- [120] Katharina Reinecke and Krzysztof Z Gajos. Quantifying visual preferences around the world. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 11–20. ACM, 2014.
- [121] Katharina Reinecke, Tom Yeh, Luke Miratrix, Rahmatri Mardiko, Yuechen Zhao, Jenny Liu, and Krzysztof Z Gajos. Predicting users’ first impressions of website aesthetics with a quantification of perceived visual complexity and colorfulness. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 2049–2058. ACM, 2013.

- [122] Daniel Ritchie, Ankita Arvind Kejriwal, and Scott R Klemmer. d. tour: Style-based exploration of design example galleries. In Proceedings of the 24th annual ACM symposium on User interface software and technology, pages 165–174. ACM, 2011.
- [123] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjrg Schmauder. Insights into Layout Patterns of Mobile User Interfaces by an Automatic Analysis of Android Apps. In Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '13, pages 275–284, New York, NY, USA, 2013. ACM.
- [124] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. In Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06, pages 413–430, New York, NY, USA, 2006. ACM.
- [125] Suranga Seneviratne, Aruna Seneviratne, Prasant Mohapatra, and Anirban Mahanti. Your installed apps reveal your gender and more! ACM SIGMOBILE Mobile Computing and Communications Review, 18(3):55–61, 2015.
- [126] Tevfik Metin Sezgin, Thomas Stahovich, and Randall Davis. Sketch based interfaces: early processing for sketch understanding. In ACM SIGGRAPH 2006 Courses, page 22. ACM, 2006.
- [127] Dennis Shasha, Jason TL Wang, Huiyuan Shan, and Kaizhong Zhang. Atreegrep: Approximate searching in unordered trees. In SSDBM, pages 89–98, 2002.
- [128] Nikita Spirin, Motahhare Eslami, Jie Ding, Pooja Jain, Brian Bailey, and Karrie Karahalios. Searching for design examples with crowdsourcing. In Proceedings of the companion publication of the 23rd international conference on World wide web companion, pages 381–382. International World Wide Web Conferences Steering Committee, 2014.
- [129] Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan. Web usage mining: Discovery and applications of usage patterns from web data. ACM SIGKDD Explorations Newsletter, 1(2):12–23, 2000.
- [130] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. Solving the search for source code. ACM Trans. Softw. Eng. Methodol., 23(3):26:1–26:45, June 2014.
- [131] J. Stylos and B.A. Myers. Mica: A web-search tool for finding api components and examples. In Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on, pages 195–202, Sept 2006.
- [132] Hock-Hai Teo, Lih-Bin Oh, Chunhui Liu, and Kwok-Kee Wei. An empirical study of the effects of interactivity on web user attitude. International Journal of Human-Computer Studies, 58(3):281–305, 2003.
- [133] Noam Tractinsky. Aesthetics and apparent usability: empirically assessing cultural and methodological issues. In Proceedings of the ACM SIGCHI Conference on Human factors in computing systems, pages 115–122. ACM, 1997.

- [134] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, volume 1, pages 403–414, May 2015.
- [135] Paul Van Schaik and Jonathan Ling. Modelling user experience with web sites: Usability, hedonic value, beauty and goodness. Interacting with Computers, 20(3):419–432, 2008.
- [136] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In The 2014 ACM international conference on Measurement and modeling of computer systems, pages 221–233. ACM, 2014.
- [137] Shahtab Wahid, D Scott McCrickard, Joseph DeGol, Nina Elias, and Steve Harrison. Don’t drop it!: pick it up and storyboard. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 1571–1580. ACM, 2011.
- [138] Dayong Wang, Steven C.H. Hoi, and Ying He. Mining weakly labeled web facial images for search-based face annotation. In Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR ’11, pages 535–544, New York, NY, USA, 2011. ACM.
- [139] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Profiledroid: Multi-layer profiling of android applications. In Proceedings of the 18th Annual International Conference on Mobile Computing and Networking, Mobicom ’12, pages 137–148, New York, NY, USA, 2012. ACM.
- [140] Ian H. Witten, Eibe Frank, and Mark A. Hall. Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.
- [141] Pengcheng Wu, Steven Chu-Hong Hoi, Peilin Zhao, and Ying He. Mining social images with distance metric learning for automated image tagging. In Proceedings of the Fourth ACM International Conference on Web Search and Data Mining, WSDM ’11, pages 197–206, New York, NY, USA, 2011. ACM.
- [142] Bo Yan and Guanling Chen. Appjoy: personalized mobile application discovery. In Proceedings of the 9th international conference on Mobile systems, applications, and services, pages 113–126. ACM, 2011.
- [143] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE ’15, pages 89–99, Piscataway, NJ, USA, 2015. IEEE Press.
- [144] Peifeng Yin, Ping Luo, Wang-Chien Lee, and Min Wang. App recommendation: A contest between satisfaction and temptation. In Proceedings of the Sixth ACM International Conference on Web Search and Data Mining, WSDM ’13, pages 395–404, New York, NY, USA, 2013. ACM.
- [145] Xianjun Sam Zheng, Ishani Chakraborty, James Jeng-Weei Lin, and Robert Rauschenberger. Correlating low-level image statistics with users-rapid aesthetic and affective judgments of web pages. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 1–10. ACM, 2009.

- [146] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In IEEE Symposium on Security and Privacy, pages 95–109, 2012.
- [147] Hengshu Zhu, Hui Xiong, Yong Ge, and Enhong Chen. Ranking fraud detection for mobile apps: A holistic view. In Proceedings of the 22nd ACM international conference on Conference on information & knowledge management, pages 619–628. ACM, 2013.
- [148] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. Software Engineering, IEEE Transactions on, 31(6):429–445, June 2005.