Insights into Layout Patterns of Mobile User Interfaces by an Automatic Analysis of Android Apps

Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt
University of Stuttgart, Stuttgart, Germany
firstname.lastname@vis.uni-stuttgart.de

Robin Goldberg, Benjamin Schmidt, Hansjörg Schmauder
University of Stuttgart, Stuttgart, Germany
firstname.lastname@studi.informatik.uni-stuttgart.de

ABSTRACT

Mobile phones recently evolved into smartphones that provide a wide range of services. One aspect that differentiates smartphones from their predecessor is the app model. Users can easily install third party applications from central mobile application stores. In this paper we present a process to gain insights into mobile user interfaces on a large scale. Using the developed process we automatically disassemble and analyze the 400 most popular free Android applications. The results suggest that the complexity of the user interface differs between application categories. Further, we analyze interface layouts to determine the most frequent interface elements and identify combinations of interface widgets. The most common combination that consists of three nested elements covers 5.43% of all interface elements. It is more frequent than progress bars and checkboxes. The ten most frequent patterns together cover 21.13% of all interface elements. They are all more frequent than common widget including radio buttons and spinner. We argue that the combinations identified not only provide insights about current mobile interfaces, but also enable the development of new optimized widgets.

Author Keywords

mobile applications; user interface; design, pattern; widget; android; reverse engineering; apps

ACM Classification Keywords

H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

INTRODUCTION

Over the last decade, mobile phones became the most ubiquitous devices. Worldwide mobile phone subscriptions grew to almost 6 billion in 2011 [25]. Recently, mobile phones evolved from simple phones to sophisticated smartphones with various sensors, powerful processors, and run third-party applications. In particular, with the emergence of the iPhone, Android, and recently Windows Phone, smartphones became open for third-party developers. The market share of smartphones is dramatically increasing. According to Nielsen half of the mobile subscribers in the US own a smartphone [21].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EICS'13, June 24–27, 2013, London, United Kingdom.
Copyright 2013 ACM 978-1-4503-2138-9/13/06...$15.00.

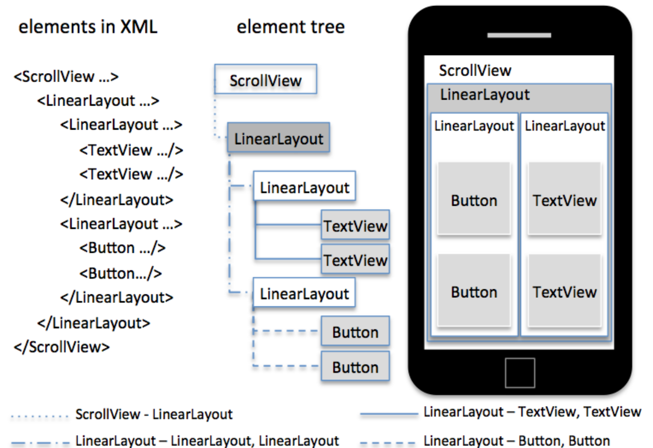

Figure 1. Four common UI element combinations extracted from 400 popular apps downloaded from Google Play. Combinations are describe as follows: *<parent element>–<child element1>,<child element2>*

Smartphone users are no longer limited to the applications provided by the phone's manufacturer. One of the main aspects that differentiate smartphones from their precursor is the app model. Users can easily install third-party applications from application stores. Development environments and centralized application stores are available for all major smartphone platforms. They enable developers to easily build and distribute mobile applications. Together, over a million applications are available for current smartphones in the most prominent stores, i.e., Google Play and Apple's App Store. The most popular apps have been installed several million times. In September 2012, Google reported "... *We've now crossed 25 billion downloads from Google Play* ..." [3].

Today, popular smartphone applications are among the most widely used applications in general. A number of disciplines try to learn about the nature of mobile applications from their own perspective. The general approach is to collect a large number of mobile applications and develop automatic means to analyze them. Software engineering researchers, for example, developed techniques that automatically find privacy leaks in Android applications [10]. They show that widely used applications leak private data. Obviously, it is crucial that applications are not only trustworthy but also usable by a diverse population. It is therefore important to investigate the nature of existing mobile applications from a user interface design perspective. Previous work developed techniques for model driven engineering [23], investigated formal methods for prototyping and simulating mobile and ubiquitous systems [29] and presented interactive tools for reverse en-

gineering UI code [28]. While previous work formalizes and generates UIs through model-driven design it is necessary to learn about currently used patterns to close the gap between formal approaches and commercial systems.

In this paper, we investigate a large number of popular Android applications. We automatically download and analyze 400 popular Android applications from Google Play, Google's official application store for Android devices. We decode the application to reconstruct their source. We retrieve and assess the layouts, user interface elements, and the features used. We determine common interface elements and identify patterns how they are combined and used. The contribution of this paper is threefold: (1) we describe the disassembling of Android application packages (APK) and how information can be retrieved from the decoded files, (2) we report the most common features and components used by popular mobile applications, and (3) we determine the most common user interface elements and identify common combinations (see Figure 1 for an example).

The paper is constructed as follows: first we discuss related work followed by an explanation of how to disassemble Android application packages. Then, we describe the data set used for our investigation. We report features and components extracted from the data set. Later, we discuss the analysis of user interface elements and the patterns retrieved. We address the limitations we come across. Finally, we conclude our findings and describe potential future work.

RELATED WORK

Mobile applications are currently distributed through market places, such as Apple's App Store and Google Play. Users access these marketplaces to download applications. Researchers continually investigate, collect data, and monitor users' application usage behavior to gain insights into how users interact with their mobile phones. For example, Cui and Roto investigated how people surf the mobile web [8]. They state that the duration of web sessions is short in general, but browser usage is longer if users are connected to WiFi. Böhmer et al. conduct a large-scale study and log detailed application usage information from mobile Android devices [7]. They report basic and contextual descriptive statics. Moeller et al. analyze the update behavior and security implications in the Google Play market [20]. They describe that users do not install an update even seven days after it is released. The usage of smartphone-based services is examined by observing 14 teenage users in [22]. It is reported that usage is highly mobile, location dependent, and serves multiple social purposes. Verkasalo [31] also shows that users use certain types of mobile services in certain contexts. He finds that users mostly use browsers and multimedia services when they are on the move, but play more games while they are at home. Balagtas et al. assess different user interface designs and input techniques for touch-screen phones [6].

Another strand of work explored users' behavior while using and interacting with mobile applications. Henze et al., for example, assess the touch performance on mobile applications [13]. They derive a compensation function that shifts the users' touches to reduce the number of errors. Further, they investigate the typing behavior using a virtual keyboard on mobile phones [14] and conclude that visualizing the touched positions using a simple dot decreases the error rate of the Android keyboard. Leiva et al. [17] investigate mobile application interruptions caused by intentional switching back and forth between applications and unintended interruptions caused by incoming phone calls. They report that these interruptions rarely happen. But when they do, they may introduce a significant overhead.

Furthermore, various projects have investigated dynamic analysis of mobile applications. Szydlowski et al. discuss challenges for dynamic analysis of iOS applications [30]. The challenges are mainly driven from graphical user interfaces. Lim and Bentley use *AppEco*, an artificial life model of mobile application ecosystems, to simulate the Apple's iOS app ecosystem [18]. Researchers have presented methodologies for automatically analyzing applications to find possible security problems and user interface bugs. Gilbert et al. propose a security validation system that analyzes applications and generates reports of potential security and privacy violation [11]. Permissions requested by Android applications are used to detect potentially malicious applications [9]. Di Cerbo et al. present an approach to detect malware Android applications. The approach relies on the comparison of the Android security permissions of each application with a set of reference models for applications that manage sensitive data. *Andromaly* is another framework for detecting malware on Android mobile devices [27]. It collects various features and events from the phone and classifies them using machine learning anomaly detectors. Mahmood et al. describe an analysis technique for automated security testing of Android apps [19]. The technique generates a large number of test cases for fuzzing an app and testing its security. Hu and Neamtiu introduce an approach to verify graphical user interface (GUI) bugs in Android applications [16]. It automatically generates test cases and feeds the application random events to generate trace files and analyze them. *AndroidRipper* is an automated technique that test Android apps via their GUI [5]. An app's GUI is explored with the aim of exercising it in a structured manner. Zhang et al. also present technique to find invalid thread access errors in multithread GUI applications [32].

In contrast to previous work and instead of accessing certain users' interaction or application usage, we analyze Android applications' source code to obtain insight into common components, features, and user interface elements used. To achieve this goal, we downloaded 400 of the most popular Android application packages (APKs) from the marketplace and analyzed them to extract valuable information.

DISASSEMBLING ANDROID APPLICATIONS

At the conception of this paper, the Android ecosystem is the most popular smartphone platform. Like other mobile platforms, the Android system is centered on the concept of apps that usually focus on a set of specific services. Android phones come with a number of pre-installed apps, including the phone app used to make and receive phone calls and the web browser app for surfing the web. Android users can install additional apps from Android marketplaces such as Google Play, Google's application store for the Android platform. Android apps are developed using the Android Soft-

ware Development Kit (SDK). It provides a frame for developing apps and ensures that a certain structure and metadata is provided by the developer. An app is then packaged into an Android application package (APK) file. APKs are file archives used to install applications on the Android system.

Structure of Android applications

Android applications are typically written in the programming language Java using the Android SDK. The Android SDK compiles the source code along with all data and resource files, into an APK file, which is an archive file with an .apk suffix. All of the code and resources in a single APK file is considered as one application.

The essential building blocks of an Android app are the application components. While there are different types of applications components, users only interact with so called activities, directly. The Android developer guidelines [1] recommend that each activity represents a single screen with a user interface. An application often consists of multiple activities but typically, one activity of an app is specified as the "main" activity, which is presented to the user when the application is launched. Having more main activities in an app allows users to enter the app from different starting points (e.g., launch the Skype app from the home screen or from the address book). An activity can start other activities to perform different actions. Only one activity can be in the foreground and thus, users can only interact with one activity at a time.

An activity's user interface is structured by a hierarchy of views. Each view controls a rectangular space within the activity's window and can respond to user input. For example, a view can be a button that initiates an action when the user touches it. The most common way to define a layout (according to the developer guidelines) is to use XML layout files. These XML layout files offer a human-readable structure for the UI's layout, similar to HTML. However, it is also possible to define a layout programmatically.

Android developers can use resources to separate graphical objects and texts from the source code. Most importantly, resources include the XML layout files that describe the user interface, images, and texts. All resources are organized in files and folders that are located in the '*res*' folder of the application project. An app can contain resources for different languages, screen resolutions, and screen sizes. The resources enable to provide a single APK that supports a range of devices and is localized to different regions. Resources are referenced in the source code and the layout files.

An app's important metadata is specified in the '*manifest*' file. This file specifies all activities and, in particular, defines an app's main activities. Having no main activity means that the app is started without any user interfaces (e.g, a service runs in the background). Furthermore, it declares which security permissions the app requests (e.g., Internet access, activation of the vibrator, or retrieval of location information).

Decoding Android application packages

To analyze Android apps from third parties, we extract the content from APKs and convert the included files to a human-readable form. To decode an APK, we use the *apktool* [2], an open-source tool for reverse engineering binary APK files. This tool decodes APK files almost to the original Android application project. The *apktool* reconstructs the complete resource folders including all layout files, pictures, animations, and string files. Furthermore, it provides the source code of the app in the intermediate '*Smali*' format [4]. Smali is an assembler language that is equivalent to the byte code for Android's Java Virtual Machine. The files resulted from unpacking an APK are used for analysis of its source code and to obtain insights into features and components used. Figure 2 shows the steps required to unpack an AKP file.

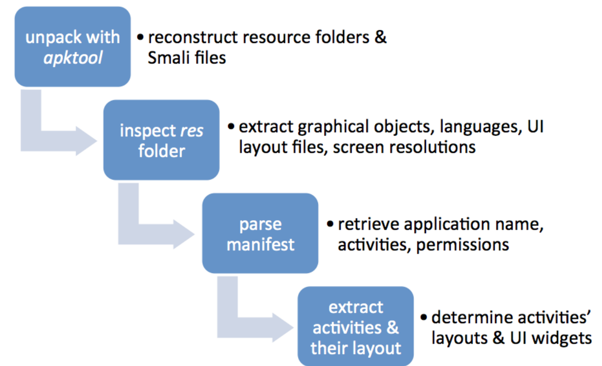

Figure 2. Steps to decode Android application packages

Inspecting decoded application packages

After decoding an APK file, we then analyze the app's files and folders. Next, we analyze the metadata specified in the app's manifest. Finally, we determine the interface layouts included. In the following we describe the information that we retrieve in each step.

Analysis of files & folders structures

To determine information about resources, we first parse the APK's *res* folder to inspect the names of the files and folders. An Android application can have multiple sets of resources, for example, to support multiple languages. Each resource set is customized for a different device configuration. The Android OS automatically chooses the resources that best match the device. To create alternative resources, specific suffixes are used in the files' and folders' name. Suffixes can be languages indicated by region codes (e.g., DE for German, FR for French), screen sizes (e.g., small, normal, large, xlarge), and screen orientation (e.g., port for portrait mode, land for landscape mode). Analysis of the *res* folder reveals the following information:

Graphical Objects: Static images are located in the *res/drawable/* folder and animations are located in the *res/anim/* folder. We can determine the number of images and animations as well as their formats.

Languages: The *res/value–<suffix>/* folders contain the texts used by the application. We use the names of the folders to determine the number of languages supported in an app.

User Interface layouts: The XML files that describe the layout of the user interface are located in *res/layout–<suffix>/*.

By parsing this folder, we can estimate how many user interfaces the application has.

Screen resolutions: We examine the suffixes of the *res* folder to identify devices an app specifically addresses (with a specific screen size or a screen pixel density). If no specific suffix is used, it can be assumed that the application's layouts support all screen sizes. However, it is also possible to optionally specify screen sizes an app supports in the *manifest* file.

Analysis of the Metadata

We determine an app's metadata by analyzing its *manifest* file. The manifest includes the application name and describes the components of the application. We extract the permission(s) an app requires. In addition, the manifest also contains the app's minimum and the maximum application programming interface (API) level. This API level is equivalent to a specific version of the Android platform. All activities of the app are also declared in the manifest. We determine the number of activities as well as the main activities that can be identified through the *android.intent.action.MAIN* attribute.

User Interface Layouts and Elements

To determine which activities use which layouts, we combine activities declared in the manifest file with the activity's corresponding Smali file. For each activity, we parse the Smali file to find the call of the *SetContentView()* method. This method includes an ID for the layout file that is used to render the user interface on the screen. With this ID, it is possible to determine the respective XML layout file. Thus, it is possible to parse the layout files and extract UI elements used in the layouts. The elements in the layout are either Android standard elements such as a <TextView>— a widget for displaying texts—or custom elements implemented by developers.

DATA SET

To analyze typical Android applications we downloaded APKs from the Google Play market. We implemented a script that downloads APKs using the android-market-api[1]. The script connects to the market's server using an existing Google account that is linked to an Android device. The script can download all free apps that are not protected. Using the API, we downloaded and stored the APKs of the top 400 highest ranked apps from the Google Play market on August 20th, 2012.

We configured the query to the Google Play server to receive free applications ordered by descending popularity. We also store additional information provided by the market while downloading the APKs such as the application name, its category, users' average rating, and its popularity rank. The used Android Market API requires an Android device ID in order to download apps from the store. We used a HTC Wildfire's device ID to download apps and did not explicitly specify any locale information. However, the device's locale used was German. Also the SIM card installed on the device and the IP address of the server were both from Germany. Among the 400 apps downloaded, some were clearly associated with locales outside Germany, e.g., "Domino's Pizza USA" or "FOX News". Therefore, we assume that our approach did not distort the popularity order due to our language or locale configuration.

[1]The android-market-api is an open-source API for the Android Market. It is not affiliated with Google: `http://code.google.com/p/android-market-api/` accessed 17.12.2012

Category	N	Rating	Activities	Layouts	Images
Tools	58	4.37	14.00	29.64	34.47
Communication	37	4.29	36.65	88.16	65.22
Entertainment	34	4.17	21.41	49.68	23.56
Efficiency	34	4.38	25.32	65.38	60.29
Social Networks	34	4.11	44.44	118.88	74.71
Music & Audio	31	4.19	24.97	66.35	59.77
Photography	21	4.34	24.57	60.76	61.81
Shopping	19	4.03	36.89	106.53	56.89
Books & References	16	4.25	18.94	54.81	50.50
Travel & Locales	15	4.21	40.73	131.47	73.73
Lifestyle	14	4.30	37.57	89.43	41.43
Health & Fitness	13	4.28	50.77	93.92	55.38
Media & Videos	12	4.34	22.92	53.08	37.75
Personalization	11	4.44	15.45	55.82	29.55
News & Magazines	11	4.10	24.73	66.09	40.09
Finances	10	4.26	61.10	118.40	44.20
Office	8	4.12	25.25	99.38	69.38
Weather	8	4.24	18.13	36.25	166.50
Sports	7	3.98	40.00	136.29	46.86
Software & Demos	4	4.28	1.25	1.75	0.00
Learning	3	3.72	39.33	88.00	78.67

Table 1. The distribution of the apps downloaded and analyzed within the different application categories. The last three columns show the average number of activities, layouts, and images in the respective category.

We downloaded APKS from 21 different categories. Categories with the highest numbers of apps are "Tools" (14.5 %) and "Communication" (9.2 %). Table 1 shows the number of applications in each category. We intentionally did not download games. The average rating of the applications was 4.25 (Median = 4.36, SD = 0.45). 80 % of the applications had a rating of four or higher. It is obvious that popular apps are likely to be highly rated. However, there were few applications with low ratings, i.e., the app "More for me" from the "Shopping" category with a 1.88 rating and a rank of 131. The three categories with the best average rating were "Personalization", "Efficiency", and "Tools" (ratings from 4.37 to 4.44). Lowest average ratings were found in the categories "Shopping", "Sports", and "Learning" (3.72 to 4.03).

STATISTICS OF POPULAR ANDROID APPLICATIONS

After downloading the APKs, we decoded them using the aforementioned process. This resulted in 778,071 files organized in 47,706 folders.

Languages

We assessed the applications' resources that can be used to internationalize them to determine which languages an app explicitly supports. For all inspected apps, English is the default language. Including regional variation (e.g., "en_us" and "en_gb") we found a total of 235 other languages. On average, an app supports 12.74 languages (SD=16.42). 47 apps support only the default English and 56 support an additional language. More than half of the apps (216) support five or more languages. Figure 3 shows the 12 most frequently supported languages (without the default English). The most common languages besides English are Chinese (63.8 %),

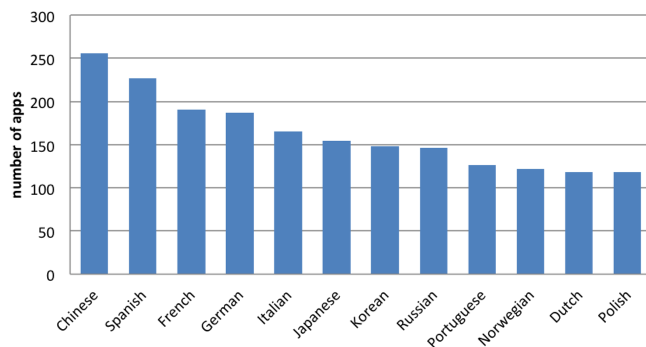

Figure 3. The 12 most frequently supported languages (besides English).

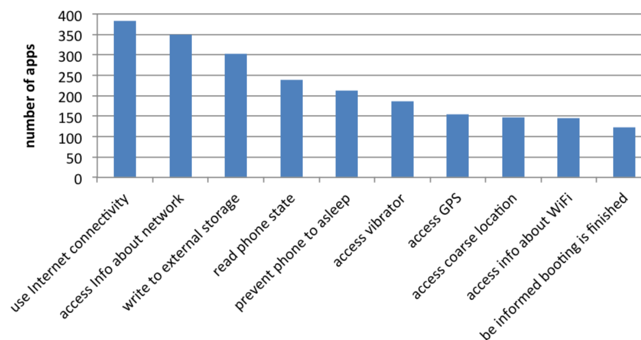

Figure 4. The most frequent Android standard permissions.

Spanish (56.6 %), and French (47.6 %). In total, 30 different languages are supported by more than 50 apps (12.5 %).

Supported screen

Parsing the resource files' suffixes reveals which applications support devices with different pixel densities and screen sizes. For the pixel density (dpi) there are four suffixes: ldpi (low dpi), mdpi (medium), hdpi (high dpi), and xhdpi (extra high dpi). The analysis shows that only 173 apps explicitly support all four variants and 26 apps do not specifically address any density type. 93% of the applications support hdpi, 75% mdpi, 70% lpdi, and 50% xdpi. Four suffixes are used for the screen size: small (low-density QVGA screen), normal (medium-density HVGA screen), large (a medium-density VGA screen), xlarge (medium-density HVGA screen). Nine apps support all four screen sizes explicitly and 215 apps do not specify any screen size. The screen size "large" is the most common size supported by 170 apps.

It should be mentioned that a screen size or screen density suffix does not imply that the resources are only for screens of that size or density. If resources with suffixes that match the current device configuration are not provided, the system may use whichever resources match best. This can be a reason that most apps do not explicitly provide specific resources.

App's launchers (main activities)

As previously mentioned, apps can have more than one main activity. Parsing the manifest files shows that 10 apps have no main activity, 300 apps have one main activity, and 90 apps that have more than one main activity. The *Kayak* app has the highest number of main activities (64).

Analysis of the permissions

We investigated the metadata provided by each application by analyzing their manifest files. In total, we extracted 355 different permissions (M=11.2 permissions per app, SD=7.83). In particular, we looked at Android standard permissions the applications require. From the 355 permissions, 121 are Android standard permission (M=9.6 permissions per app, SD=6.6). Figure 4 shows the ten most common Android standard permissions. The three most frequently used permissions are Internet access (8.7% of all extracted permissions), the permission to determine if network access is available (7.9%), and the permission to store data on the mobile phone's external storage (6.8%).

Tactile Feedback. Most Android devices are equipped with a vibration motor that is used to provide tactile feedback. A permission is required to activate the vibration motor. The results show that 47.25% of the applications use vibration motor to provide tactile feedback.

Location Information. We were also interested in the use of contextual information such as the user's location. In total, 190 applications can access the device's location. 154 apps use the fine location information provided by the GPS sensor and 147 accessed coarse location details (e.g., determined via the phone network's Cell-ID or visible WiFi networks). 111 applications can access both, the fine location and the coarse location of the device.

Connectivity. 96.25% of the applications use the Internet access permission. Further analysis reveals that 10.25% of the applications use the Bluetooth connectivity, 8.25% have the permission to send SMS, and 2.5% use near field communication (NFC). It should be mention that not all mobile phones support the NFC technology.

Number of User Interfaces

To examine whether the user interface of applications from various categories differ, we conducted a statistical analysis of the most frequent categories in our data set. Because of the small sample size for some categories we focus on the ten most frequent categories ($N>=15$). We use the number of activities, the number of layouts, and the number of images as indicators for the complexity of the user interface. Table 1 provides an overview of the number of activities, layouts, and images for each category. To determine if categories are different we did an analysis of variance (ANOVA). As we were doubtful that the variances are equivalent, a Games-Howell post hoc test is used for the pairwise comparison.

Activities

After extracting the number of activities for each app, we assessed if the average number of activities differ between the ten most frequent categories. Levene's test indicates that the assumption of homogeneity is violated $F(9,289)=3.89$, $p<.001$. An ANOVA test reveals a significant difference between the categories, $F(9,289)=5.14$, $p<.001$. Games-Howell post hoc test shows six pairwise significant differences between the categories. Applications in the category Tool (M=14.00, SD=17.34) have fewer activities than applications in the categories Social Networks (M=44.44, SD=27.64,

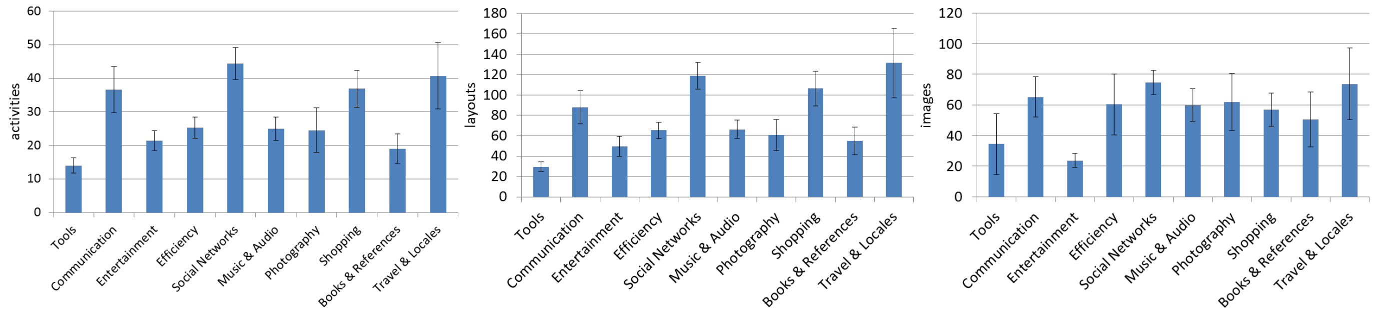

Figure 5. The average number of activities (left), layouts (center), and images (right) for the ten categories with the most frequent applications in our data set. Error bars show standard error.

p<.001) and Shopping (M=36.89, SD=24.07, p<.05). Entertainment apps (M=21.41, SD=17.34) have significantly less activities than Social Networks applications (p<.01). Furthermore, applications in the Social Networks category have significantly more activities than Music & Audio apps (M=24.97, SD=19.41, p<.05) and Books & References apps (M=18.94, SD=17.83, p<.01).

Layouts

We further investigated if the average number of layouts per application statistically differ between the categories. Again, Levene's test indicates that the assumption of homogeneity is violated F(9,289)=4.44, p<.001. An ANOVA reveals a significant difference between the categories, F(9,289)=6.87, p<.001. Games-Howell post hoc test reveals eight pairwise significant differences between the categories. Applications in the category Tool (M=29.64, SD=35.52) have less layouts than applications in the categories Communication (M=88.16, SD=98.72, p<0.05), Efficiency (M=65.38, SD=45.99, p<.01), Social Networks (M=118.88, SD=76.21, p<.001), Music & Audio (M=66.35, SD=51.13, p<.05), and Shopping (M=106.53, SD=74.39, p<.01). Applications in the Entertainment category (M=49.68, SD=57.74) have fewer layouts compared to applications in the category Social Networks (p<.01). Efficiency applications have fewer layouts than Social Network applications (p<.05).

Images

Further, we compared the average number of images per application. Levene's test shows that the assumption of homogeneity is not violated F(9,289)=0.64, p=.77. An ANOVA test revealed no significant difference between the categories, F(9,289)=1.03, p=.413. We therefore, refrained from conducting a post hoc analysis.

Correlations

Looking at the charts shows in Figure 5 suggests that there might be a correlation between the number of activities, layouts, and images of an application. Therefore, we further investigate the correlation between the number of activities, the number of layouts, and the number of images. The Pearson correlation reveals that there are significant pairwise correlations between all three parameters. There is a strong correlation between the number of activities and the number of layouts (r=0.79, p<.0001). Furthermore, there is a correlation between the number of activities and the number of images (r=0.29, p<.0001) and between the number of layouts and the number of images (r=0.39, p<.0001). While it is not surprising that an application with a larger number of activities has a larger number of layouts, the strong correlation suggest a common pattern.

Discussion

Of the Android applications we analyzed, we found that 88.25% support more languages in addition to English. Furthermore, we determined that a diverse range of languages is supported and the majority supports five or more languages. The results suggest that popular Android applications are diverse in terms of supported languages. It can be assumed that the chance to become popular is much higher if an application supports languages in addition to English.

We analyzed the applications' number of activities, layout files, and images. It is shown that applications from different categories use significantly different number of activities and layout files. We show that tools and as well as applications from the categories Entertainment, Efficiency, Music & Audio, Photography, and Books & References have fewer views and interface layouts than applications from the categories: Communication, Social Networks, Shopping, and Travel. The strong linear correlation between the number of activities and the number of layout files suggests a linear factor. Further, it is not common that an app provides more than one entry point. Only 20% of the apps have more than one main activity. With 96.25%, the overwhelming majority of the applications analyzed require Internet access and almost half of the applications (47.50%) access location information. While there are different reasons why an application requires Internet access (e.g., to display advertisements), the very high number of applications that require it still suggest that the majority of the applications rely on dynamic content. In particular, if ads are considered dynamic content. Notably, almost half of the applications (47.25%) can provide tactile feedback through the phones' vibration motor. Furthermore, the apps explicitly support various devices based on screen pixel densities rather than screen sizes.

Applications from the categories differ in terms of interface complexity. Tools, for example, have distinctly fewer views and layouts compared to social networks. Tools, as the name of the category suggests, address specific use cases. A typical example is the application "Spirit Level Plus" that enables use of the device as a spirit level. However, the few numbers of activities and layouts of other categories (i.e., Entertainment, Efficiency, Music & Audio, Photography, and Books &

Layout	Apps	Percent	Total
LinearLayout	390	66.95	51780
RelativeLayout	365	24.20	18716
FrameLayout	307	7.82	6048
ScrollView	332	2.35	3733
ListView	325	1.77	2814
TableLayout	167	0.92	710
AbsoluteLayout	35	0.12	89

Table 2. The seven standard layout containers. The columns show the name of the layout container, the number of applications that use the layout, the percent of the total number of layout containers, and the total number of times they are used.

Widget	Apps	Percent	Total
TextView	383	35.50	56467
ImageView	380	15.59	24794
Button	355	9.37	14912
View	271	4.35	6917
EditText	318	2.91	4628
ImageButton	294	2.71	4308
ProgressBar	300	1.67	2662
CheckBox	285	1.54	2443
RadioButton	176	0.76	1213
Spinner	178	0.48	759

Table 3. The ten most frequently used widgets used by the applications. The columns show the name of the widget, the number of applications that use the widget, the percent of the total number of widgets, and the total number of times they are used.

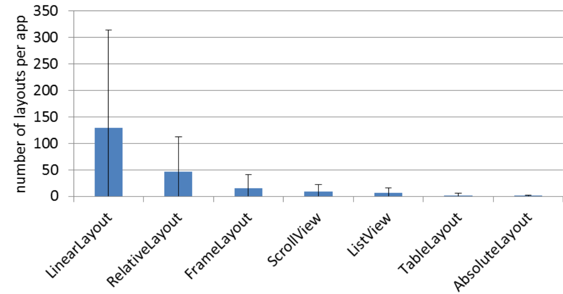

Figure 6. The average number of layouts per application for the seven standard Android layout container. Error bars show standard deviation

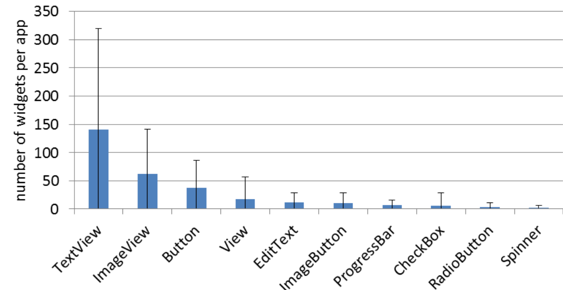

Figure 7. The average number of widgets per application for the most frequent widgets in our data set. Error bars show standard deviation

References) suggest that they also address specific use cases.

USER INTERFACE ELEMENTS & PATTERNS

We are further interested in common UI elements and potential design patterns used in the applications. The user interfaces of Android applications are typically defined in XML layout files. These files help to define the interface elements and their structures. In the following, we briefly describe Android user interfaces. We determine which interface elements are most frequently used in Android applications. By analyzing the hierarchy of interface elements, we identify common interface elements combinations.

Android User Interface Layouts

The user interface of an Android application consists of a set of activities. Each activity represents a single screen with an interface. Each of these interfaces is a composition of widgets such text boxes, checkboxes, and buttons. These widgets are embedded in layout containers that define the visual structure of the interface. By nesting layout containers in other layout containers, the developer creates a tree of UI elements. While it is also possible to define the user interface directly in the source code, the Android developer guidelines recommend declaring the trees of UI elements in XML layout files.

The Android API provides a number of different layout containers that structure the arrangement of the embedded elements. In addition, developers can define their own widgets and layout containers. The five most common layout containers are briefly described in the following. The *LinearLayout* arranges its elements in a single column or a single row. The *RelativeLayout* allows for relative positioning of its elements in relation to each other or the parent. The *FrameLayout* blocks out an area on the screen to display a single item. The *TableLayout* is similar to the *LinearLayout* but can arrange its elements in rows and columns. The *AbsolutLayout* enables specification of the exact locations of it elements and, hence, it is less flexible and harder to maintain.

While the layout containers provide the structure of the interface, the user only interacts with the embedded widgets. Android provides typical widgets that are also used in Desktop applications and the Web. Typical examples are the TextView (a text label), ImageView (an image), the Button, EditText (to enter text), and the ProgressBar (visual indicator of progress). One can also implement and use customized elements.

User Interface Elements

In total, we retrieved 29,086 XML layout files from the 400 Android applications. We analyzed the XML layout files to determine the most frequent layout containers and widgets. In total, the layout files contain 77,343 Android standard layout containers and 159,072 widgets. Thus, there are about twice as many widgets than layout containers.

Table 2 shows all standard layout containers in our data set. The LinearLayout accounts for 66.95% of all layout containers and is used by 390 applications. The RelativeLayout accounts for 24.20% of all layout containers and is used by 365 applications. The FrameLayout and the ScrollView are used by the majority of the applications (307 and 332 applications) but account for only 7.82% and 2.35% of all layout containers. The TableLayout is used by 167 applications and the AbsoluteLayout is used by 35 applications. Both account for less than one percent of the total number of standard layout containers. Figure 6 shows the average frequency of a layout is used by an application.

We also extracted the Android standard widgets used in the applications. From the 159,072 widgets retrieved, *TextView* is by far the most common element (35.5%) followed by *Im-*

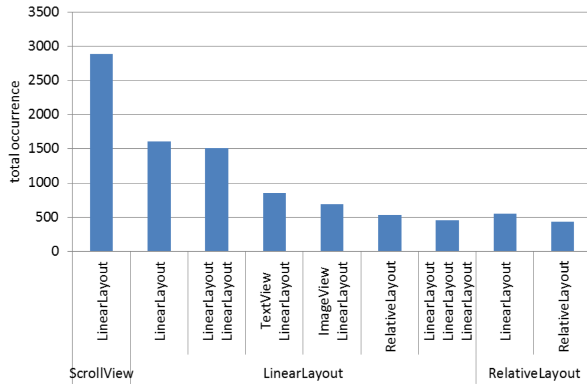

Figure 8. The most common layout patterns.

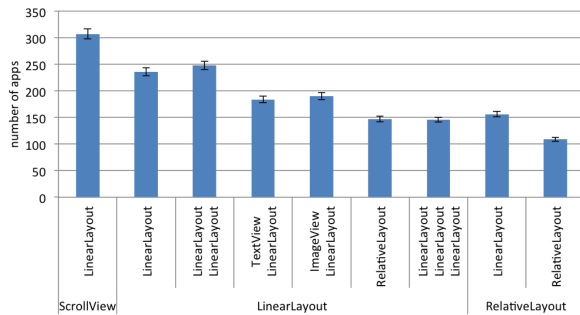

Figure 9. The use of common layout patterns by the apps.

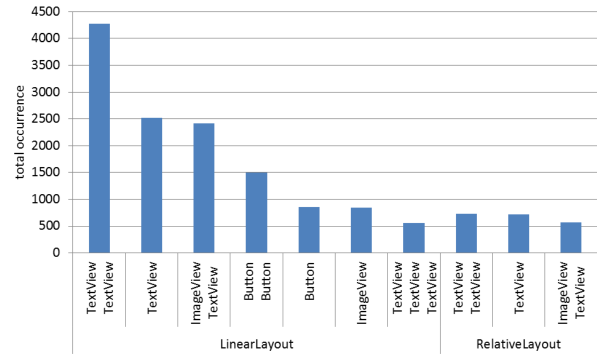

Figure 10. The most common widget patterns.

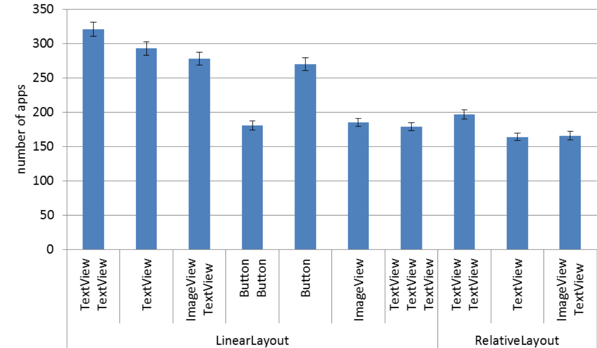

Figure 11. The use of most common widget patterns used by the apps.

ageView (15.6%), and *Button* (9.4%). Table 3 shows the ten most frequent widgets. Together, these frequently used widgets account for 74.87% of the total number of the extracted widgets and are used by more than half of the 400 applications. Figure 7 further depicts how often on average a widget is used by an application.

In addition to the standard Android elements, we found 4,022 custom layout and widget elements used in layout files. These layout containers and widgets are often custom buttons or layouts, but also complete gallery views or a date-time picker.

User Interface Patterns

After extracting all widgets and layouts, we further analyzed the layout files. We determined how elements are combined together and used to obtain potential user interface design patterns. We inspected the combination of widget elements as well as combination of elements and their layout types.

To achieve this, we assessed layouts and their embedded elements. The elements in a layout file are hierarchically structured. Therefore, we parsed the hierarchy of the layout files to retrieve parents and their child elements. This means that for an element, we had its parent element as well as its siblings (if any exist). This allowed us to extract available combinations of elements. Then, we counted how often the combinations are used in order to find the common patterns.

In total we identified 22,870 unique combinations of elements. 75.8% of these combinations are used only once. The patterns can be classified in two different types. One type of pattern consists of a layout container that contains another layout elements as well as widgets. The most common pattern is a ScrollView that contains a LinearLayout. This combination can present more content in a linear fashion then the screen can show at once. Figure 8 shows the nine most frequent layout patterns. We also checked how many apps used these patterns. The most frequent pattern is used in 307 applications. Interestingly, the second most frequent pattern, which is a linear layout nested in a linear layout, is used in fewer applications than the third pattern, i.e., two linear layouts nested in a linear layout (236 vs. 248 applications). The use of the frequent patterns is shown in Figure 9. The applications in the Learning category use the patterns on average, more often than other categories.

The second type of pattern consists of layout elements that only contaion widget elements. Figure 10 presents the ten most frequent combinations. Having two TextViews in a LinearLayout is the most frequent pattern used in the applications. The second frequent pattern is a LinearLayout that has one TextView, and the third one is an ImageView together with a TextView in a LinearLayout. We also observed the use of ButtonViews in different patterns. The use of patterns by the applications reveals similar trend for the top three patterns. However, the pattern which consists of two buttons nested in a linear layout (forth most frequent pattern) is used less than the pattern which has single button nested in a linear layout (fifth frequent pattern). Figure 11 shows the use of patterns in the applications. The applications in the Social Network category use this type of pattern on average more often than other categories.

Implications

We analyzed the interface layouts of the 400 most popular Android applications. We determined which interface wid-

gets and layout containers are most frequently used. We found that the two widgets TextView and ImageView that are just used to show labels and images account for over 50% of all widgets. While it is rather obvious that widgets that display information are more common than interactive widgets, the proportion is still surprising. We found, for example, 47 times more TextViews than RadioButtons. Some interactive standard widgets, such as the ToggleButton and the SeekBar, can even be considered as esoteric. The ToggleButton accounts for only 0.37% and the Seekbar accounts for 0.25%. The most frequent interactive widgets (Button and EditText) suggest that the layouts of the analyzed applications are mainly used by entering text and pressing buttons.

We further identify frequent patterns of interface elements. The patterns retrieved are commonly used across all apps. We found that the most frequent patterns are more common than a number of standard widgets. Altogether, 21.13% of all widgets and layout containers in our data set are part of at least one of the 10 most frequent patterns. Since 77.28% of all ScrollViews contain a LinearLayout, replacing this combination by a layout container that combines the two would be the fifth most frequent layout container. The second most frequent layout pattern combines a LinearLayout with another LinearLayout. Interestingly, one of the layout containers in this combination is useless[2].

The most common widget pattern in our data set consists of a LinearLayout that includes two TextViews and covers 5.43% of all interface elements. This pattern alone is almost as frequent as check boxes, radio buttons, toggle buttons, and seek bars together. Development of a new widget that substitutes this pattern would rank it as the sixth most frequent widget. Thus, identification of frequent patterns in general will enable development of new optimized widgets.

LIMITATIONS

We rely on the ranking of the Google Play market to select popular applications. The parameters that influence the ranking and the algorithm itself are unknown. However, the same ranking is used if users browse Google Play. Therefore, the 400 selected applications might not be the 400 most widely used applications, but we assume that there is a very strong overlap. In addition, to the best of the authors' knowledge, 400 is the highest number of mobile interfaces that have been systematically analyzed. While selecting and downloading applications, we had to use a specific device ID and a specific locale. Developers can restrict applications to certain devices, locales, and configurations. In addition, we only downloaded free applications. We assume, however, that popular applications are widely available across devices and locales.

We only analyze UIs created through XML layout files. User interfaces and further enhancements can be developed programmatically beyond the XML files and can thus also be dynamically be created. However, the Android developer guidelines recommend using XML layout files to define the user interface. Accordingly, all of the analyzed apps use this approach. We therefore assume that this approach is so common that the analysis enabled general conclusions.

[2] If a developer adds a LinearLayout as the only child of another LinearLayout the Android SDK suggest that "This LinearLayout layout or its LinearLayout parent is useless"

CONCLUSION & FUTURE WORK

In this paper we show how to disassemble Android APK files to retrieve information about mobile interfaces. We downloaded and analyzed 400 apps from Google Play, Google's application store for Android devices. We disassembled the application packages to reconstruct the content. We found that 88.25 % of the apps support more than one language and 93% of the app explicitly support screens with high pixel density. We showed that tools as well as apps from different categories vary in the number of views and interface layouts. This can be used as indicators for the complexity of apps' use interfaces.

By analyzing the apps' interface layouts, we determined the interface widgets and layout containers used most often. We further identified frequent combinations of interface elements. Aggregatively, the ten most frequent combinations cover 21.13% of all interface elements. They are all more frequent than some common widgets, including radio buttons and Spinner. If the four most common widget combinations would be considered as separate widgets, they would all be among the ten most frequent widgets. The most common combination that consists of three nested elements covers 5.43% of all interface elements. It is more frequent than progress bars and checkboxes. This means, a new widget that substitutes this combination could replace all occurrences of the combinations. In this case, this new widget would be the sixth most frequent widget. The identification of frequent UI elements combinations provide the possibilities of optimizing widgets and introducing new widgets.

Determining how frequently different interface elements are used can motivate further research. We only investigated the parent and siblings of an element to find possible parents. However, it would be interesting to take into account other information such as the child elements to retrieve other type of patterns. Researchers might consider focusing on common interface elements. Furthermore, the patterns identified motivate the development of new widgets that ease the interface development and improve the usability. We conducted a static analysis of the user interface. Executing the downloaded application in an emulator would enable to also observe the applications' dynamic behavior. A user's input can be simulated to combine the static analysis with an analysis of the visual appearance. The static analysis combined with an analysis of UI interaction paths [33], observations of actual user behavior collected on a large scale [7, 12, 24, 26], and consideration of biomechanics [15] could ultimately result in a holistic understanding of mobile interaction. In general, the presented approach suggests a new method to understand user interface designs that may help to implement new development tools for designing more successful and more usable applications.

Acknowledgment: This work was funded by the German Research Foundation within the SimTech Cluster of Excellence (EXC 310/1).

REFERENCES

1. *Android Developer Guideline*, accessed 17.12.2012. **http://developer.android.com/guide/practices/ui_guidelines/index.html**.
2. *apktool Software*, accessed 17.12.2012. **http://code.google.com/p/android-apktool/**.
3. *Google Blog*, accessed 17.12.2012. **http://officialandroid.blogspot.de/2012/09/google-play-hits-25-billion-downloads.html**.
4. *smali - An assembler/disassembler for Android's dex format*, accessed 17.12.2012. **http://code.google.com/p/smali/**.
5. Amalfitano, D., Fasolino, A., Tramontana, P., De Carmine, S., and Memon, A. Using gui ripping for automated testing of android applications. In *Proc. ASE* (2012).
6. Balagtas-Fernandez, F., Forrai, J., and Hussmann, H. Evaluation of user interface design and input methods for applications on mobile touch screen devices. In *Proc. Interact* (2009).
7. Böhmer, M., Hecht, B., Schöning, J., Krüger, A., and Bauer, G. Falling asleep with angry birds, facebook and kindle: a large scale study on mobile application usage. In *Proc. MobileHCI* (2011).
8. Cui, Y., and Roto, V. How people use the web on mobile devices. In *Proc. WWW* (2008).
9. Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *Proc. CCS* (2009).
10. Gibler, C., Crussell, J., Erickson, J., and Chen, H. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. *Trust and Trustworthy Computing* (2012).
11. Gilbert, P., Chun, B., Cox, L., and Jung, J. Vision: automated security validation of mobile apps at app markets. In *Proc. MCS* (2011).
12. Henze, N., Pielot, M., Poppinga, B., Schinke, T., and Boll, S. My app is an experiment: Experience from user studies in mobile app stores. *IJMHCI* (2011).
13. Henze, N., Rukzio, E., and Boll, S. 100,000,000 taps: analysis and improvement of touch performance in the large. In *Proc. MobileHCI* (2011).
14. Henze, N., Rukzio, E., and Boll, S. Observational and experimental investigation of typing behaviour using virtual keyboards on mobile devices. In *Proc. CHI* (2012).
15. Hrabia, C.-E., Wolf, K., and Wilhelm, M. Whole hand modeling using 8 wearable sensors: biomechanics for hand pose prediction. In *Proc. AH* (2013).
16. Hu, C., and Neamtiu, I. A gui bug finding framework for android applications. In *Proc. SAC* (2011).
17. Leiva, L., Böhmer, M., Gehring, S., et al. Back to the app: The costs of mobile application interruptions. In *Proc. MobileHCI* (2012).
18. Lim, S. L., and Bentley, P. J. How to be a successful app developer: lessons from the simulation of an app ecosystem. In *Proc. GECCO* (2012).
19. Mahmood, R., Esfahani, N., Kacem, T., Mirzaei, N., Malek, S., and Stavrou, A. A whitebox approach for automated security testing of android applications on the cloud. In *Proc. AST* (2012).
20. Möller, A., Diewald, S., Roalter, L., Michahelles, F., and Kranz, M. Update behavior in app markets and security implications: A case study in google play. In *Proc. MobileHCI* (2012).
21. Nielsen. *Smartphones Account for Half of all Mobile Phones, Dominate New Phone Purchases in the US*, accessed 17.12.2012. **http://blog.nielsen.com/nielsenwire/online_mobile/smartphones-account-for-half-of-all-mobile-phones-dominate-new-phone-purchases-in-the-us/**.
22. Rahmati, A., and Zhong, L. Studying smartphone usage: Lessons from a four-month field study. *IEEE Transactions on Mobile Computing* (2012).
23. Raneburger, D., Popp, R., and Vanderdonckt, J. An automated layout approach for model-driven wimp-ui generation. In *Proc. EICS* (2012).
24. Sahami Shirazi, A., Rohs, M., Schleicher, R., Kratz, S., Müller, A., and Schmidt, A. Real-time nonverbal opinion sharing through mobile phones during sports events. In *Proc. CHI* (2011).
25. Saylor, M. *The Mobile Wave: How Mobile Intelligence Will Change Everything*. Vanguard, 2012.
26. Schleicher, R., Sahami Shirazi, A., Rohs, M., Kratz, S., and Schmidt, A. Worldcupinion experiences with an android app for real-time opinion sharing during soccer world cup games. *IJMHCI* (2011).
27. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., and Weiss, Y. andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems* (2012).
28. Silva, C. E. Reverse engineering of gwt applications. In *Proc. EICS* (2012).
29. Silva, J. L., Campos, J., and Harrison, M. Formal analysis of ubiquitous computing environments through the apex framework. In *Proc. EICS* (2012).
30. Szydlowski, M., Egele, M., Kruegel, C., and Vigna, G. Challenges for dynamic analysis of ios applications. *Open Problems in Network Security* (2012).
31. Verkasalo, H. Contextual patterns in mobile service usage. *Personal and Ubiquitous Computing 13*, 5 (2009).
32. Zhang, S., Lü, H., and Ernst, M. D. Finding errors in multithreaded gui applications. In *Proc. ISSTA* (2012).
33. Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., and Zou, W. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proc. SPSM* (2012).