2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)

An Empirical Study of UI Implementations in Android Applications

Mian Wan, Negarsadat Abolhassani, Ali Alotaibi, William G. J. Halfond
Department of Computer Science
University of Southern California
Los Angeles, California, USA
Email: {mianwan, abolhass, aalotaib, halfond}@usc.edu

***Abstract*—Mobile app developers are able to design sophisticated user interfaces (UIs) that can improve a user's experience and contribute to an app's success. Developers invest in automated UI testing techniques, such as crawlers, to ensure that their app's UIs have a high level of quality. However, UI implementation mechanisms have changed significantly due to the availability of new APIs and mechanisms, such as fragments. In this paper, we study a large set of real-world apps to identify whether the mechanisms developers use to implement app UIs cause problems for those automated techniques. In addition, we examined the changes in these practices over time. Our results indicate that dynamic analyses face challenges in terms of completeness and changing development practices motivate the use of additional analyses, such as static analyses. We also discuss the implications of our results for current testing techniques and the design of new analyses.**

***Index Terms*—empirical study, mobile applications, user interface, static analysis, dynamic analysis**

I. Introduction

The user interfaces (UIs) of mobile apps serve as the focal point for users. These UIs generally provide the primary way users are able to interact with the apps and access their features and contents. To ensure that their apps work reliably and with high quality, developers employ a range of techniques for verifying the correct behaviors of their UIs and, more broadly, their apps. Many of these techniques rely on *app crawlers*, such as Monkey [1], Dynodroid [2], and PUMA [3], that automate interaction with an app's UIs. The crawlers employ a range of heuristics and algorithms to interact with elements in an app's UIs and explore different behaviors. During this exploration, the crawlers check each UI to ensure that certain invariants hold and that the app's behavior is correct.

A key assumption underpinning these verification techniques is that an app's UIs can be identified by a crawler. For apps with static UIs that are completely defined by layout XML files in the apps' apk files, this is generally true. However, even such simple UIs may pose problems for crawlers. In cases where an activity's UI can only be accessed if the crawler interacts with the app in a certain way or enters inputs that satisfy specific constraints, the UIs of some activities may never be fully interacted with. For example, a new user dialog that requires the same password to be entered twice may prevent a crawler from advancing unless it can figure out the constraint on the two inputs. Beyond such scenarios, elements in UIs can also be altered programmatically by calls to SDK APIs that create new UI elements and modify style attributes of those elements, such as color and size. Since these elements are created in code, they may be guarded by complex conditions that are beyond the capabilities of a crawler to satisfy. The implication of this is that the UI information obtained by crawlers may be incomplete and lead to unsoundness in the verification process.

At this time, our current knowledge about the effectiveness of crawlers is limited. Although crawlers have been extensively compared against each other [4], this has mostly been a relative comparison, with the techniques compared against each other using the metric of code coverage instead of the ground truths of layouts. As a result, we are able to determine which crawlers will perform well as compared to others, but not in an absolute sense and without any indication for how complete these techniques are. However, while it is conceptually clear that crawlers have limitations when UIs are built dynamically, based on code, it is not clear in practice how often such situations occur. Therefore, we lack an understanding of how incomplete the information generated by crawlers may be for real apps.

As an alternative to crawlers, developers may consider static analyses to more completely identify the elements in an app's UIs. However, static analyses face many potential challenges to be effective for this application. In particular, static analyses must account for complex lifecycles and semantics when analyzing invocations of APIs that create UI elements. Furthermore, static analyses are known for being imprecise and may identify many false positive UI elements that could not actually be created at runtime by the app. Most importantly, there do not exist any static analysis based tools that are able to accurately identify UI layouts. Therefore, crawlers remain an easier and more accessible approach since many are available for off the shelf usage.

In this paper, we report on our large-scale investigation of the mechanisms developers use to build UIs. The goal of this investigation is to determine if developers are building app UIs in ways that can cause problems for automated analyses, such as crawlers and static analyses. To carry out this investigation, we studied a large set of apps using a combination of crawling, lightweight static analysis, and manual analysis. Our study characterized the type of implementation mechanisms used for

2576-3148/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSME.2019.00016

UI construction and discussed their implications for the design of better crawlers and static analyses. We also investigated these practices over time, allowing us to ascertain if these trends were increasing or decreasing, thus giving an indication of which practices have become more popular in creating UIs, and which practices could be the most important to address moving forward. Overall, our results found that, since 2013, more complex mechanisms for UI definitions are being used by developers and that different aspects of how these mechanisms are used have significant implications with respect to the accuracy of current UI analysis techniques. The issue we identified can guide the design of future analyses and point our community towards areas that, if targeted by research, could lead to significant improvements in the quality of UI analyses and the verification of mobile apps.

The remainder of the paper is organized as follows. Section II provides the background information on Android UIs. In Section III, we motivate the list of research questions (RQs). Section IV presents the common steps that we followed to conduct the experiments. In Section V, we report and discuss the results of our experiments. Next, we discuss the threats to the validity of our experiments in Section VI. We then describe the related work in Section VII. At last, we conclude in Section VIII.

II. Background

In the Android platform, each UI window displayed on a screen is called an *activity*. A developer can customize a UI by implementing a subclass of the *Activity* class. Each activity will execute a series of lifecycle callbacks when it reaches a certain lifecycle state. For example, when an activity is created, it will first execute the *onCreate* callback to do some initialization. In an Android app, all the activities are registered in the *AndroidManifest.xml* file.

Each activity consists of a bunch of widgets, which are called *Views*. This is because every widget is a subclass of the *View* class. In most cases, a widget can respond to some user operations (e.g., click) by executing corresponding event handlers. There is another kind of views, *ViewGroups*, that are usually not visible and serve as the container of widgets. Both Views and ViewGroups determine the layout of an activity. There are two different ways to customize the layout of an activity. The first way is to use a layout XML file to represent the layout, and load it in the application code by calling certain APIs (e.g., *setContentView(int)*). A widget maps to an XML tag, and the attributes of a tag can define many features of the widget. For example, the *android:id* property assigns an ID to the widget. The other way is to instantiate a View class and insert the view object into the activity by calling some APIs (e.g., *ViewGroup.addView(View)*).

Since Android 3.0, in order to support tablets, fragments were introduced to represent an arrangement of views that may be reused. Fragments share some features with both Activity classes and View classes. Like views, fragments can be declared as XML tags in layout XML files, and instantiated objects in the program. Similar to activities, a fragment has its own lifecycle callbacks to do data processing. One of them, *onCreateView*, initializes the details of UI elements, and returns the root view of the fragment layout. Even though the purpose of introducing fragments is reusing UIs, but a fragment may not necessarily be a UI element. If the return value of a fragment's *onCreateView* callback is null, then this fragment is not a UI element.

To help developers manipulate the views flexibly, Android SDK enables developers to specify an attribute of a widget both in the XML tag and in the code. As mentioned earlier, a developer can directly set a property value for a view tag in a layout XML file. In the code, since each widget is a View object, the first step is to retrieve the widget object. A common way to do that is to call *findViewById(int id)*, whose parameter corresponds to the value of *android:id* in the widget tag. This mapping is maintained by the special class *R.id*. Once the developer obtains the View object, they set the properties of this object by using different API calls. For example, *setBackgroundColor(int)* sets the background color of a widget, achieving the same effect of an *android:background* property.

In some cases, a developer may want to reuse a certain UI configuration across activities. Similar to CSS files, Android SDK provides *styles* and *themes* to reduce the complexity and repetition in the layout XML files. A style is a set of attributes that define the appearance of a view. A theme is a style that is applied to an activity or the whole app. If a theme is not specified for an activity, the app's theme will be used. Just like CSS styles, a style or a theme can inherit some attribute settings from a parent, and override some new settings by modifying the related property values. To apply a style, developers need to set the *style* attribute of a widget tag. Similarly, a theme is specified by setting the *android:theme* property of an <activity> tag or the <application> tag. Similar to attributes of views, developers can make API calls to set a style or a theme in code.

III. Research Questions

To start our study, we began with the most foundational research question: **RQ 1: How well does dynamic analysis work for UI identification?** In this RQ, we explored the question of whether crawlers are complete with respect to identifying all of the UI elements of an app. Although we know that in theory, there are situations in which a crawler would be incomplete, in this RQ, we are investigating if this happens in practice. To address this RQ, we employed two different methods. First, we compared the traversed activities against the activities defined in the app and then we compared UI elements that could be identified by a crawler against the apps' ground truth of layouts. Second, we used a static analysis to identify code patterns that could potentially be problematic for crawlers. Our results showed that crawlers were markedly incomplete. This motivated our subsequent RQs, which focused on identifying and understanding the programming practices that could pose problems for both crawlers and alternative solutions, such as static analyses.

Our next RQ focused on obtaining a high level overview of the kinds of UI related actions developers were performing in code: **RQ 2: How do developers use APIs to define UIs in Android apps?** In this RQ, we systematically classified all invocations in the code based on their types. For example, *void addView(android.view.View)* is used to add a view to a layout dynamically while *void setContentView(int)* is used for setting the activity content from a layout resource file. Besides defining the UI, many invocations may modify the look of existing UI elements. For example, developers may use *void setTextColor(int)* to change the text color of a TextView to a specific color. Exploring this RQ enables us to get a better understanding of the types of actions that might be missed by crawlers or that static analyses would need to handle to get a more thorough model of the possible UIs. The results of this RQ helped us to identify types of UI related actions that could cause problems for crawlers and static analyses.

One of the types of actions that caught our attention was the usage of fragments. Broadly, fragments allow developers to define UI snippets in an XML file and then include these snippets in the UIs of different activities. While a crawler would see the results of including fragments the same as any other dynamically defined UIs, they pose special challenges for static analyses. In particular, fragments, unlike views, have unique lifecycles callbacks similar to those of activities. Therefore, fragment usage may create new challenges because it introduces new semantics to the Android apps which requires additional analysis. So in this RQ, we examined the following question: **RQ 3: Do developers use fragments frequently in Android apps?** Our results found the use of fragments had consistently increased over time. Since many static analysis based techniques do not consider fragments, our next RQ tried to determine the impact of omitting fragments. Specifically, we asked: **RQ 4: How many views are defined in fragments, and activities respectively?** In this RQ, we counted the number of views that were being created in each activity versus the number created in fragments. Overall, our results found a non-trivial usage of fragments both in terms of frequency and in terms of the number of views they found, motivating the importance of handling these in a static analysis based approach.

Broadly, two types of views may be incorporated into an activity's UI, atomic views and adapter-based views. Atomic views (e.g., button) are the basic building blocks of a UI, whose layouts are predefined in their corresponding classes. These types of views may created and configured either in code or via the static XML based layout files. Detecting and analyzing these views is straightforward for both crawlers and static analysis based approaches, since for crawlers they appear in the UI and for static analyses, semantics can be attached to a single invocation call associated with an atomic view. Adapter-based views are views used to display a set of children views provided by an Adapter, which is the input to a *setAdapter* call. Adapter-based views do not know details about the contained children and a developer has to invoke *setAdapter* to render those views in the UIs. The *setAdapter* configuration can only happen via code. For crawlers, the more complex views do not pose a problem, since when they are displayed, they can be identified easily like any other views. However, for static analyses, they pose significant challenges because an analysis needs to first identify its corresponding adapter and then to track where and how each child view is defined and added to the adapter. For this reason, we were interested in understanding how prevalent these are and how difficult they might be for static analyses to handle. Therefore, we investigated the following question: **RQ 5: How do developers use views to customize the Android UIs?**

Another kind of action that caught our attention was the large amount of invocations used to set or modify the look of UI elements. Results showed that transparency, color, and size properties were defined frequently. For changing the transparency of views, only a few APIs are used and the arguments provided to them are mostly constants. On the other hand, more complicated expressions are often used for setting color and sizes. Identifying and interpreting these actions is important for static analyses since both of these properties directly affect the visual representation of the UI elements and are used by many higher-level analyses, such as checking UI equivalence and detecting GUI changes in evolving mobile apps [5]. However, interpreting these actions could also be very difficult for static analyses, depending on how the arguments provided to those APIs are defined. For example, if they are simple string or numeric constants, then the values of these can be trivially found by a static analysis. However, if they are defined inter-procedurally or via complex expressions, then it would be non-trivial to identify these values and, absent a likely sophisticated mechanism for handling these, a static analysis may generate a large amount of inaccurate style information. Therefore, we investigated the following question: **RQ 6: How do developers set the UI style properties of views?** Using a static analysis, we looked at the data flows and control flows of the arguments of the API invocations and classified the patterns used to define each of these.

IV. Data Collection

In this section, we discuss the experiments that are designed to address each of the research questions mentioned in Section III. The general steps are as follows. First, we downloaded a large pool of real-world Android apps from the Google Play app store. To enable analysis of the UI implementation practices over time, we downloaded apps that were published at different times in the Google Play Store. Specifically, we looked at three time ranges: late 2013, early 2017, and early 2019. To extract the implementation of those apps, we leveraged Soot [6] to analyze the apk file of each app. For the RQ1 requiring dynamic analysis, we employed Monkey [1] to interact with the apps, and dumped the UI layouts using UI Automator [7]. In the manual analysis, we used jadx [8] to analyze the decompiled Java source code of apk files. We discuss details about common steps for the data collection in the remaining part of this section. The

dedicated approach description to each RQ will be presented in Section V.

A. Select Subject Applications

We had three criteria for selecting the subject applications. They (1) are from different categories, to make sure our results can be generalized to a broad set of apps; (2) can be successfully analyzed by the tools we used, to ensure we can gather complete information from an app's apk; (3) were published at different points in time, to allow us to observe developer trends over time. To meet the above criteria, we downloaded free apps from the Google Play Store. We collected three groups of apps, one of apps that were published in late 2013, another from apps published in early 2017, and the last one from apps published in early 2019. Within each group, we downloaded 9,000 apps and then randomly selected a subset of 600 apps from each group for analyses in our study. From each subset, we only excluded an app whose bytecode or resource files could not be analyzed. Overall, this gave our study a subject pool comprised of 1,671 apps and representing over 23 app store categories.

B. Identify Developer Code

Since we are studying developers' practices, we wanted our results to account for only developer written code. However, the subject apps contain both library code and developer code. Furthermore, some parts of an app may be obfuscated, which makes it more difficult to distinguish the two types of code. The research community has developed several techniques for addressing aspects of this problem (e.g., [9], [10]). However, these techniques focus on a precise identification of library code, not developer code. Therefore, these techniques could mistakenly classify unidentified library code as developer code. Our preference was to have a more precise identification of developer code, i.e., we would prefer to miss some developer code rather than erroneously include library code in our analysis. Our reason for this preference was that library code is generally quite large and may contain a broad range of implementation practices that are not necessarily representative of regular developer code.

To consider developer code only, our insight was that many developer defined classes' (e.g., the main activity class) names and library classes are generally not completely obfuscated. The package names of those classes can be used as keywords to locate the corresponding library projects. Based on this insight, we identified developer code as follows. First, we manually checked nearly 180 top ranked apps, and summarized a list of library package names. Then, we extracted an app's package name and the main activity class package name from the *AndroidManifest.xml* file. Based on these two package names, we then created a string pattern that was sufficient to uniquely represent the developer classes' packages. For example, for an app whose main activity class name is "com.thepuzzle.activity_start", the summarized pattern is "com.thepuzzle", because "com" is very common. For classes whose name matched the summarized pattern and did not match any entry in the library package list, we considered them as developer classes.

C. Create UI Related API List

In order to study the developers' practices, it is essential to create a complete list of UI related APIs in the Android SDK. To do that, we first summarized the relevant packages (e.g., "android.widget") and method name keywords (e.g., "color"). Then we read through a list of the methods defined in each class under relevant packages in *android.jar* file in Android SDK. If a method's name matched the keywords, then we added it to our API list. Later, we asked two of the authors to go over the Android SDK documentation to confirm each method was correctly identified and belonged to the right category. At the same time, they examined the documentation for any missing methods. In total, we identified 272 API calls from three categories in the list. The three categories were defined as APIs that are used for: (1) setting UI's look (e.g., setTextColor()), (2) inflating XML layouts (e.g., setContentView()), and (3) modifying UI in code (e.g., addView()).

V. Experiments, Results, and Discussions

A. RQ 1: How well does dynamic analysis work for UI identification?

Approach: To address this research question, we investigated the amount of incompleteness of dynamic analysis at different levels of granularity: (1) activity coverage, (2) view coverage, and (3) certain invocation statement coverage. For the first two metrics, we designed a methodology to show the coverage differences between dynamic analysis and ground truth. For the third metric, we used another methodology to statically determine how straightforward it might be for crawlers to trigger the execution of certain invocations of UI related APIs.

In the first methodology, we directly evaluated the completeness of a representative crawling technique against the ground truth for a set of apps. To define the ground truth at the activity level was relatively straightforward. We could obtain a list of all the activities from the *AndroidManifest.xml* file of an apk file. The definition of the ground truth at the view level was more challenging and required extensive manual analysis. Therefore, for a small set of eight apps, we manually analyzed each app to develop the view ground truth. One of the authors interacted with the app, systematically exploring all UIs, clicking on all buttons and settings, and manually inspecting all code and resource files. We then ran Monkey [1] on each of a larger set of 71 apps for about 30 minutes to represent the results we would expect when using a crawler. We chose Monkey because of its widespread popularity, results in the research community that consistently show it among the top performing crawlers [4], and because it could robustly run against all apps. We used only eight apps in the view coverage study because of the long amount of time needed for each app. Additionally, we filtered out apps for which Monkey got stuck at certain screens, for example those requiring a login.

We then compared the ground truth against the Monkey results and calculated the activity coverage and view coverage. For calculating view coverage, we only considered activities that Monkey could successfully interact with.

In the second method, we wanted to determine if UI elements were created in ways that would require the crawler to interact with the app in a certain way in order for them to display. To calculate this number, we implemented a static analysis that found the invocations in the app's bytecode that modified the UI by creating a new view or changing the style properties of a view and then determined if these invocations were control dependent on another node. Informally, a statement is control dependent on another node if its execution depends on the conditional execution (i.e., true or false) of that statement. So our analysis was determining how many of the UI related API invocations in the app were dependent on a switch or if statement. Essentially, this gave us an approximation for the amount of UI related behaviors would only be triggered if some conditions were met during execution (e.g.,, satisfied by the crawler). Our analysis for this method was based on the classic intra-procedural control dependency analysis defined by Ferrante, Ottenstein, and Warren [11].

Results: We found that the crawler was able to achieve 49% activity coverage on average per app and 93% view coverage on average per activity. More specifically, for 63/71 apps, the crawler missed at least one activity and for 6/8, the crawler missed at least one view. In terms of the control dependency of UI related APIs, we found that in the 2013 app set 39% of the calls were conditionally guarded; in the 2017 set this number was 42%; and in the 2019 set, this number was 41%.

Discussion: Overall, our results confirm that apps are built in ways that can cause problems for crawlers. More broadly, these results indicate that crawlers face challenges in terms of completeness, which has significant implications for the soundness of verification techniques built on top of them. This finding of incompleteness motivates further investigations into the practices used to develop UIs to identify the potential challenges that alternative techniques, such as static analyses, may face in analyzing UI related code in apps.

B. RQ 2: How do developers use APIs to define UIs in Android apps?

Approach: To answer this research question, we investigated the types of APIs invoked by developers to define the UIs of an app. To do this, we developed a static analysis to analyze each app's bytecode and count invocations to the UI related APIs. We compared the signature of the target method of each invocation in the app's code against the list of API calls described in Section IV.

We divided API calls into three categories based on their usage and impact. The first category is API calls used for setting the content of the UI (e.g., *android.view.View inflate(int,android.view.ViewGroup)*). The second category contains API calls used to manipulate the UI by adding, removing or replacing UI elements (e.g., *void removeViewAt(int)*). This category is divided into two subcategories, (1) views related API calls, and (2) fragment related API calls. The third category comprises of API calls that modify the look of UI elements. We divided this third category into five subcategories based on the impact of each API call on the UI. These five subcategories are setting the (1) transparency, (2) position, (3) color, (4) size, and (5) general look. For example, we considered *setMargin(float, float)* as an API call for setting the size of views. For general look settings, we considered views' constructors, *setTheme* and *setStyle*, which are able to set different UI properties from different categories described above. For the third category, we only consider invocations on View and Window classes since these two are responsible for creating the look of a UI.

To compare the number of invocations over time, we used two metrics. One metric is the average number of API calls used in each activity, and the second one is the average number of API calls used in each app. For the first and third categories, we used the first metric, as their usage in apps is prevalent. However, for the second category, we leveraged the second metric because (1) their usage in apps is optional, and (2) some API calls in this category (e.g., adapter-based API calls) are defined for specific purposes. For computing these metrics, we excluded the apps with no activities like service apps and apps whose UIs are defined in native methods. In total, we excluded 15, 50, and 73 apps from 2013, 2017 and 2019 apps, respectively.

Results: The number of API calls related to transparency increased from almost 5 calls per activity in 2013 to 10 and 11 calls in 2017 and 2019 respectively. The same observation is valid for the other look related API calls. For color property, the number of related API calls increased from 3 in 2013 to 5 in 2017 and in 2019. Size-related API calls are also used around three times in each activity while the number was less than two in 2013. The API usage for setting position and general attributes of views has not changed significantly. The usage of adapters in apps for both views and fragments was 8 in 2013, 14 in 2017, and 19 in 2019. The total number of fragments and views instantiated in code was 17 in 2013, and 40 in 2017 and 2019 per app.

Discussion: Table I indicates that the contribution of code to set the look of the UI in each activity is increasing over time. Therefore, analyzing XML files for understanding the nature of UIs is not enough. Also, as the number of XML inflating calls (e.g., *setContentView()*) per activity is increasing, determining the possible looks of an app is getting more complicated since the use of these APIs indicates that a UI can change to another layout based on a set of conditions. In such situations, crawlers might miss the UI elements and their arrangement in layouts. Table II shows an increasing trend of using the code for building UI in apps over time by modifying the views and fragments both with the help of adapters or only API calls. The dramatic increase in the average number of views and fragments instantiated in code per app implicates that crawlers will miss a considerable amount of views if they only account for the layout XML files. However, this aspect also adds complications to static analyses, as these analyses

TABLE I: Distribution of average API calls building UI per activity over time

Year	Transparency	Position	Color	Size	General	XML Inflators
2013	4.83	0.08	3.42	1.73	0.77	2.19
2017	10	0.30	5.46	3.10	1.56	3.35
2019	10.94	0.42	4.84	2.49	1.67	3.11

TABLE II: Distribution of average code usage for modifying UI per app over time

Year	Fragment Related Adapters	Fragment Related Calls	View Related Adapters	View Related Calls	View Instantiation	Fragment Instantiation
2013	0.29	2.13	7.65	27.78	13.30	4.64
2017	0.63	3.45	13.97	49.67	23.47	17.35
2019	0.98	5.58	18.10	59.29	22.67	18.55

must consider code to identify the views used in a layout and their properties both from XML files and code.

C. RQ 3: Do developers use fragments frequently in Android apps?

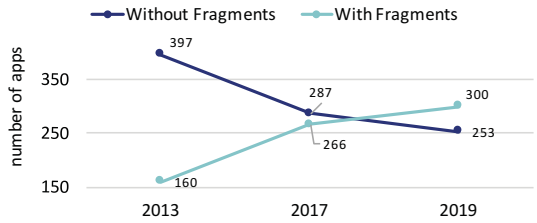


Fig. 1: Distribution over time of the number of apps that use fragments.

Approach: To answer this research question, we analyzed each application to determine how many fragments it used. To do this, we built a static analysis to inspect all of the application classes defined in each app by the developer's code and determine how many were subclasses of the Fragment class. Our analysis could determine if a particular instance was a subclass by transitively following each class' parent class to see if the root class was the Fragment class. Our analysis then counted the number of apps that defined at least one Fragment class and compared them against the number of apps without any Fragment classes. It is possible that some fragments may not create any views. For example, this may happen when a developer wants provide a feature to an activity without being interrupted by activity re-rendering caused by UI configuration changes. We were interested if these situations might affect our results, so we also calculated the subset of fragments that did not create any view. I.e., they would not have any impact on the user interface. Our analysis identified these by looking at the return value of the *onCreateView()* callback in each Fragment class found and determined if it returned a view object or a non-view value (i.e., null). We counted those that returned a non-view value as *non-UI fragments*.

Results: Figure 1 shows the usage of fragments over time. The results show that out of 557 apps each year, 160 apps used fragments in 2013, 266 used fragments in 2017, and 300 apps used fragments in 2019. Our results also indicated that UI fragments dominated the fragment usage, averaging between 92% and 97% in each year.

Discussion: The results showed that fragments have become more popular over time and that the vast majority of these cause changes in the UI of an activity. The increasing usage of fragments poses significant challenges for Android UI analyses, in particular, static analysis based approaches. The layout of a fragment is defined via a unique set of lifecycle callbacks. To know the layout information, it is necessary to analyze the code of the lifecycle callbacks and understand their semantics. Of special note, the binding relationship between fragments and activities can be many to many, since a fragment can be shared across different activities. Beyond identifying the possible bindings, it is also essential to identify which fragment is introduced in which activity. For example, finding the mapping in cases of using a Collection in the fragment related adapter makes it difficult to analyze. All of these aspects would require sophisticated analyses to be accurate.

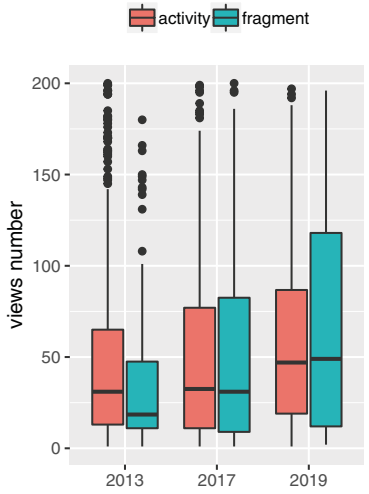


Fig. 2: The average number of views in fragments and activities.

D. RQ 4: How many views are defined in fragments, and activities respectively?

Approach: To address this research question, we developed an analysis to determine how many views, on average, would be missed if fragments were not accounted for by an analysis. Since we do not have a path sensitive analysis that is able to compute exact UI layouts, our methodology computes an estimation of this number. To do this, we first determined how many views on average were created in each activity and fragment. Then, because multiple fragments may be included in an activity, we determined the average number of fragments included in each activity. Our analysis computed averages for each of these numbers on a per-app basis and then used the product of those averages to compute a final estimation.

To find the average number of views in an app's fragments and activities, we developed a static analysis to examine the bytecode of the methods of an app. For each app, the analysis counted the number of views that were defined in each activity and fragment of an app. This analysis considered views defined in both code and XML layout files. To count the number of views defined in the XML layout files, the analysis extracted the layout resource ID from the API calls that included XML content (e.g., *inflate(int)*) in fragment or activity classes. Specifically, the analysis identified calls to such APIs that appeared in the body of *onCreateView()* and *onCreate()*, since these are the methods responsible for initializing fragments and activities. Then for each XML file identified in this way, the analysis used an XML parser to count the number of views defined in the file. To count the number of views defined by code, the analysis scanned the method bodies of all of the Fragment and Activity classes and counted the number of views created in each. Specifically, the analysis identified view creation by identifying instantiations of objects of type View. For instantiations in loops, our static analysis assumed that the loop would be unraveled once, which means our calculated number is likely a lower bound on the average view numbers for a fragment or activity. Next, our analysis computed the average number of fragments included in each activity of the app. To do this, the analysis identified invocations to APIs that modify fragments in an activity and computed how many of these appeared, on average, in the activities of the app. Our analysis considered two ways of modifying fragments: (1) calls to the add, replace or remove methods of the FragmentTransaction class, and (2) calls to the *getItem()* method of the PagerAdapter class. Since the latter can add an indeterminate number of fragments in a single call, we conservatively assumed that these calls only added one fragment. This means that our estimation of the average number of fragments per activity is a lower bound. Finally, for each app, we estimated the average number of views defined in fragments of an activity by multiplying the average number of views in a fragment by the average fragment number in an activity. The resulting number served as our estimate of how many views, on average, could be missed by analyses that did not not account for fragments.

Results: Our calculations indicate that, on average, for each activity 13 views in 2013, 14 views in 2017, and 11 views in 2019 may be missed by analyses that do not account for fragments. We illustrate some of the intermediate values that enabled us to reach these results. Specifically, Figure 2 shows the distribution of views defined in UI Fragment classes and Activity classes. The median number of views defined in activities was around 25 in 2013 and in 2017, and it reached to over 40 in 2019. The median number of views defined in fragments increased from less than 20 views in 2013 to 30 in 2017, and then to 40 in 2019. We also show results regarding the number of fragments added per activity. In Table II, the column labeled Fragment Related Calls shows the number of fragments added, on average, by the FragmentTransaction class and the column labeled Fragment Related Adapters shows the number of adapters that can create at least one fragment. Together these two columns show the lower bound of total number of fragment manipulations, on average, per activity.

Discussions: Figure 2 shows an increasing trend of defining more views in majority of fragments while the trend is more stable for activities. This shows an increasing chance of missing views in more recent apps in terms of total views per app. However, as the number of activities has increased dramatically in 2019, the number of missing views per activity decreased. The results from this RQ and previous one emphasize on the importance of fragments and its impact on the UI. We observed activities in which developers define an empty layout only and populate it using fragments. Although, crawlers can extract information about fragments, they might miss triggering an event responsible for manipulating a fragment in an activity. Therefore, missing fragments in crawlers means missing many views in majority of cases. Static analysis techniques also have complications for analyzing fragments in code similar to the previous RQ.

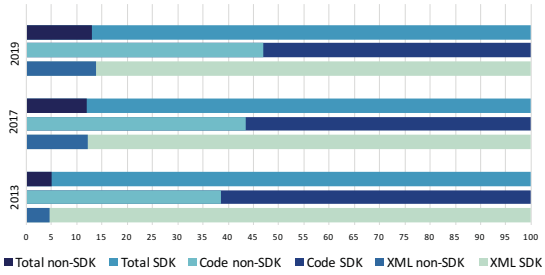


Fig. 3: Distribution of the usage of different view types over time.

E. RQ 5: How do developers use views to customize the Android UIs?

Approach: To address this research question, we analyzed the apps to identify the number of views that represented standard views versus developer–customized views. Since customized views can pose additional challenges for static

analyses, depending on whether these views can themselves have children, we also looked at the prevalence of this programming practice. To address both aspects, we developed a static analysis that examined the bytecode of the apps and classified views based on their types and then based on the views' ability to have children. This analysis considered views created in code and in XML based layout files.

Our analysis to classify views based on types worked as follows. First, the analysis scanned the bytecode of the app and checked the type of new instances of objects in the code to determine if they were a subclass of the *View* class. Once a view object was identified, it was classified as either a standard view defined by the SDK or a custom view. To determine if a view was a standard view, the analysis looked at the package name to determine if it was from the Android SDK (i.e., the package name starts with "android"). The analysis classified the view as non-standard if its package name was not part of the Android SDK. Our analysis also extracted and classified all views defined in XML based layouts by analyzing XML files referenced in invocations to API calls such as *setContentView()* or *inflate()*. The analysis used an XML parser to extract the names of views defined in layouts and leveraged the same approach to identify views in XML. The only difference was some SDK views are used without their package name in layout files which can be identified based on the same characteristic.

Our analysis also identified views that could have children. In general, any view that extends AdapterView is able to have content that could be a set of views (i.e., have children). Our analysis also determined whether an identified view extends AdapterView, ViewPager, or RecyclerView, and if so counted it as a view that could have children. Note that this represents a lower bound on the number of classes that could have children, as we noticed many customized adapter-based views that did not extend any of the above mentioned adapter-based classes, but still provided some sort of capability to have children views.

Results: Figure 3 shows the results of our analysis by type. In this graph, we show the breakdown of standard (SDK) views versus non-standard (non-SDK) views for each year. We further break down the results based on whether the view was defined in the code or via an XML based layout file. Each bar in the figure, shows the proportion of views in total, code or XML based on their types. Overall, in 2013 customized views accounted for 5% of all defined views, while in 2017 and 2019 they accounted for 13% of all views. In code, non-standard views accounted for 38%, 43%, 46% in 2013, 2017, and 2019. For our investigation into the second aspect, views with children, the column labeled View Related Adapters in Table II shows the number of such views over time. The table reports the number of *setAdapter()* which would be considered as the total number of adapter-based views which can be used.

Discussion: We found that there is an increasing trend in using non SDK views in applications, specifically in defining views in code and also adapter-based views are becoming more prevalent among developers. Both customized views and adapter-based views create complications for analyzing UIs. First, customized views define new semantics and can have unique appearances when rendered. Absent an analysis that can understand rendering rules, such views may require developer intervention to define their semantics. Second, we observed that views that have children often define these children inter-procedurally and track children using collections. Inter-procedural analyses and analyses that target collections are known for introducing complexity and imprecision in static analyses.

F. RQ 6: How do developers set the UI style properties of views?

Approach: In this RQ, we investigated how arguments to style related APIs were set in code. To do this, we developed a static intra-procedural data-flow analysis that evaluated each of the style related arguments (e.g., width and height) to APIs that set color and size properties. The analysis built definition-use (DU) chains for each argument then analyzed the DU chains to determine how the definitions were created. For example, if the value was the result of a mathematical expression or defined using a parameter provided to the method enclosing the API invocation.

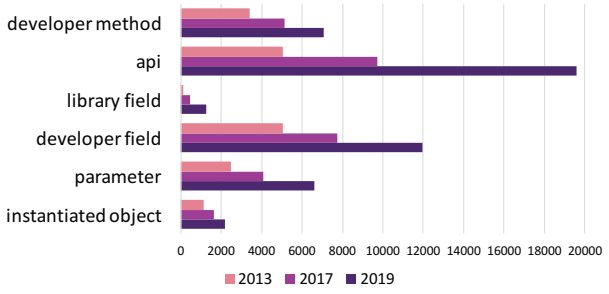
The analysis examined the call site of each API invocation that was related to color and size. For each of the relevant arguments, the analysis calculated its backwards DU chains until no new uses or definitions could be found in the method enclosing the call site (i.e., the analysis reached a fixed point). The analysis then classified the argument based on the operations that were used to create each of the definitions in the DU chain. The broad classifications used by our analysis were: instantiated objects (e.g., create by *new LayoutParams()*), parameter place holders (e.g., define by the i^{th} parameter of a method), developer defined fields, library fields, API calls (i.e., the invoked methods are defined by library code), constants (e.g., "red"), and developer defined method calls (i.e., the invoked methods are defined by a developer). For a given argument, our analysis counted all of the different ways used to define it.

Results: For the color related APIs, the frequencies of calls using constant parameters were 9,535 (1.2/activity) in 2013, 12,979 (1.4/activity) in 2017, and 17,832 (1.4/activity). The frequencies of color related API calls using variable parameters were 11,675 (1.5/activity) in 2013, 20,782 (2.3/activity) in 2017, and 32,962 (2.5/activity) in 2019. Compared to all color related API calls, invocations using constant parameters accounted for 45% in 2013, 38% in 2017, and 35% in 2019, while invocations using variable parameters accounted for 55% in 2013, 62% in 2017, and 65% in 2019.

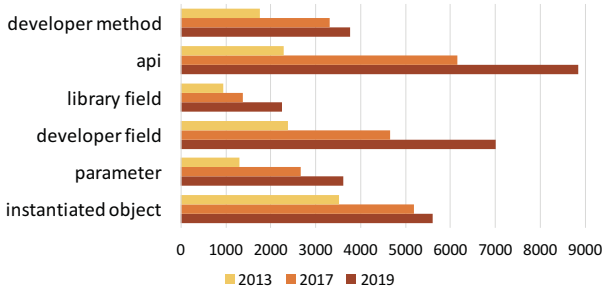
For the size related APIs, the frequencies of API calls using constant parameters were 3,266 (0.4/activity) in 2013, 3,484 (0.4/activity) in 2017, and 4,775 (0.4/activity) in 2019. The frequencies of size related API calls using variable parameters were 9,084 (1.2/activity) in 2013, 17,074 (1.9/activity) in 2017 and 21,475 (1.6/activity) in 2019. Compared to all size related API calls, invocations using constant parameters accounted

for 26% in 2013, 17% in 2017, and 18% in 2019, while invocations using variable parameters accounted for 74% in 2013, 83% in 2017, and 82% in 2019.

For those invocations using variables as parameters, Figure 4 shows the frequency of different categories of expression components for color and size parameter expressions. As a whole, the top frequent expression components for color parameters were APIs (53%), developer fields (38%), developer methods (24%), and parameter placeholders (20%). For size parameters, the top popular expression components were APIs (36%), instantiated objects (30%), developer methods (29%), developer methods (19%), and parameter placeholders (16%).



(a) Color parameter



(b) Size parameter

Fig. 4: Different component distribution in parameter expressions.

Discussion: In general, the frequency of the color and size calls using variable parameters has been growing from 2013 to 2019 and represents a sizable majority all such definitions of style related APIs. This indicates that a technique for estimating the parameter values of API calls is necessary for the accuracy of a static analysis.

As for expression components, APIs are the top components in the expressions. This implies the semantics of API calls must be accounted for when statically analyzing the color and size values. Developer fields also play an important role in defining the values. This fact shows that a field analysis is required to estimate the values. Developer methods, and parameter placeholders are quite popular for both color and size parameters. The prevalence of these two components shows an inter-procedural analysis is indispensable to accurately approximate the values. It is worthy noting that size parameters use instantiated objects commonly to derive the value. This indicates that size parameters are usually objects that consist of fields mapping to size properties (e.g., width). To know a specific size property value, an intra-procedural analysis is required to analyze how that field's value is defined in the constructors. If the value originates from a formal parameter, then an inter-procedural analysis is need to provide the runtime value of that parameter.

VI. Threats to Validity

External Validity: A potential threat is that our study did not consider paid apps, which may require different design of experiments. Therefore, our conclusions may not hold for paid apps. However, it is worth noting that our results are representative, since 94% of all Android applications in Google Play Store are free apps [12].

To ensure that the applications in our study are representative of real-world applications, we randomly selected 557 applications for each of three app set from the Google Play Store. These applications were from over 23 categories and had various functionalities.

The dynamic analysis tool we used is Monkey, in which we did not apply heuristics of inputs for our large scale study. Therefore, the results in RQ1 may not apply to some advanced dynamic analysis tools that are optimized to some apps. To alleviate this threat, we excluded those apps that have certain UIs (e.g., login UI) that Monkey cannot bypass. Note that Monkey is confirmed to achieve higher average code coverage and trigger more failures than many existing dynamic analysis tools [4].

Internal Validity: Even though we designed a mechanism to make sure the UI related API list is complete, it is still possible that API list is not thorough. This would only impact the results of RQ 2 and 6. To mitigate this, we employed keyword matching and manual analysis to create the API list. Even if the list is incomplete, it does not prevent us to draw the conclusion that developers frequently use UI related APIs to define UIs in Android apps.

VII. Related Work

The closest related work is done by Lamba and colleagues [13]. They revealed the API call usage patterns among a large set of open-source Android apps. They focused on collecting which API calls are invoked together within a user-defined method. However, our work targets how developers define UIs in Android apps. Our work takes not only the UI related API calls in the program, but also the static layout XML files into account. For some UI properties (e.g., size) of a view, which are set via API calls, our work also studies how developers define those values.

There are also some studies about the layouts or elements of Android UIs. Shirazi and colleagues [14] carried out an empirical study on 400 Android apps to discover the commonly used UI element combinations. To investigate that, they analyzed the layout XML files which are referred via *setContentView()* API call. Taba and colleagues [15] identified commonly used

elements from layout XML files and summarized those elements' features. Our work differs in the following aspects: (1) our work studies how developers define both the UI layouts and UI element properties; (2) For the UI layout aspect, our work studies the types of fragments and views that are used in Android apps; (3) Our work considers UI elements created in the program as well.

Moran and colleagues [5] developed an automated tool GCAT to identify and abstract GUI changes for different commits of mobile apps. Their technique identified the changes in UI among different commits of an app by comparing the captured screenshots generated by a crawler. However, our work exploits a combination of static analysis, dynamic analysis and manual work to study how developers define UI elements and their properties in a large pool of mobile apps over time.

Chen and colleagues [16] found that fragments are exploited in 91% of 217 Android apps, and developed a technique to statically analyze the transition between fragments and activities. Our work focuses on investigating the program practices of developers, and performs a more comprehensive study about fragments. In addition, our work also studies how a developer defines the views and their UI properties as time goes.

Many studies have been conducted with respect to Android APIs. McDonnell and colleagues [17] studied the API evolution and adaptation in Android application. Li and colleagues [18] surveyed the evolution and the potential impact of inaccessible APIs in Android SDK. Wang and colleagues [19] analyzed the API usage obstacles for iOS and Android platforms from the developers' Q&A website posts. Linares-Vásquez and colleagues [20] investigated how Android API changes trigger discussions on the StackOverflow website. None of the aforementioned works gives us insights about how developers define UI layouts and properties in Android apps.

Many researchers have performed empirical studies on Android apps for different topics, such as energy consumption [21], energy impact of refactoring [22] and obfuscation [23], [24], design pattern changes [25], software aging manifestations [26], local database usage [27], configuration errors [28], the robustness of Inter-component Communication [29]. However, none of them is targeting developers' UI definitions in Android apps.

There are also many works to model the GUIs of mobile apps. One way to do that is using dynamic analysis [2], [30]–[38]. Another way to generate GUI model is via static analysis [39]–[43]. Hybrid approaches [44], [45] are also proposed to construct GUI model. All of these works are targeting build GUI models for mobile apps, and do not investigate developer practices of defining UI elements and their properties.

VIII. Conclusions

In this paper, we presented the results of an extensive empirical study we conducted on the mechanisms developers use to build UIs. For this study, we investigated over 1,600 apps from Google Play Store, which were collected in 2013, 2017, and 2019, by applying a combination of crawling, lightweight static analysis, and manual analysis. Our study discovered interesting trends and observations. First, we found that dynamic analysis techniques, such as crawlers, do face challenges in terms of completeness when analyzing real-world apps. Second, we noticed that the contribution of code to define UIs of mobile apps are increasing over time. Third, we learned that fragments are becoming more popular over time and playing an important role in building UIs. Fourth, our study showed that developers are using more non-SDK views over time. At last, we found that the way of setting color and size properties is quite complex. To sum up, our study provides useful insights that can guide the design of future static and dynamic UI analyses, and point the research community towards areas of improving the quality of UI analyses and verification techniques of mobile apps.

IX. Acknowledgments

This work was supported by the following grants: NSF-1619455, NSF-1528163, and ONR-14-17-1-2896.

References

[1] G. LLC, "UI/Application Exerciser Monkey." [Online]. Available: https://developer.android.com/studio/test/monkey

[2] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An Input Generation System for Android Apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234.

[3] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 204–217.

[4] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet? (E)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2015, pp. 429–440.

[5] K. Moran, C. Watson, J. Hoskins, G. Purnell, and D. Poshyvanyk, "Detecting and Summarizing GUI Changes in Evolving Mobile Apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 543–553.

[6] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java Bytecode Optimization Framework," in *CASCON First Decade High Impact Papers*, ser. CASCON '10. Riverton, NJ, USA: IBM Corp., 2010, pp. 214–224.

[7] G. LLC, "UI Automator." [Online]. Available: https://developer.android.com/training/testing/ui-automator

[8] skylot, "jadx - Dex to Java decompiler." [Online]. Available: https://github.com/skylot/jadx

[9] Z. Ma, H. Wang, Y. Guo, and X. Chen, "LibRadar: Fast and Accurate Detection of Third-Party Libraries in Android Apps," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, May 2016, pp. 653–656.

[10] M. Backes, S. Bugiel, and E. Derr, "Reliable Third-Party Library Detection in Android and Its Security Applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 356–367.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[12] Statista, "Distribution of free and paid Android apps in the Google Play Store from 3rd quarter 2017 to 1st quarter 2018." [Online]. Available: https://www.statista.com/statistics/266211/distribution-of-free-and-paid-android-apps/

[13] Y. Lamba, M. Khattar, and A. Sureka, "Pravaaha: Mining Android Applications for Discovering API Call Usage Patterns and Trends," in *Proceedings of the 8th India Software Engineering Conference*, ser. ISEC '15. New York, NY, USA: ACM, 2015, pp. 10–19.

[14] A. Sahami Shirazi, N. Henze, A. Schmidt, R. Goldberg, B. Schmidt, and H. 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*, ser. EICS '13. New York, NY, USA: ACM, 2013, pp. 275–284.

[15] S. E. S. Taba, I. Keivanloo, Y. Zou, and S. Wang, "An exploratory study on the usage of common interface elements in android applications," *Journal of Systems and Software*, vol. 131, pp. 491 – 504, 2017.

[16] J. Chen, G. Han, S. Guo, and W. Diao, "FragDroid: Automated User Interface Interaction with Activity and Fragment Analysis in Android Applications," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 398–409.

[17] T. McDonnell, B. Ray, and M. Kim, "An Empirical Study of API Stability and Adoption in the Android Ecosystem," in *2013 IEEE International Conference on Software Maintenance*, Sept 2013, pp. 70–79.

[18] L. Li, T. F. Bissyandé, Y. L. Traon, and J. Klein, "Accessing Inaccessible Android APIs: An Empirical Study," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 411–422.

[19] W. Wang and M. W. Godfrey, "Detecting API Usage Obstacles: A Study of iOS and Android Developer Questions," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 61–64.

[20] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 83–94.

[21] D. Li, S. Hao, J. Gui, and W. G. J. Halfond, "An Empirical Study of the Energy Consumption of Android Applications," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, Sept 2014, pp. 121–130.

[22] C. Sahin, L. Pollock, and J. Clause, "How Do Code Refactorings Affect Energy Usage?" in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14. New York, NY, USA: ACM, 2014, pp. 36:1–36:10.

[23] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause, "How Does Code Obfuscation Impact Energy Usage?" in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 131–140.

[24] C. Sahin, M. Wan, P. Tornquist, R. McKenna, Z. Pearson, W. G. J. Halfond, and J. Clause, "How does code obfuscation impact energy usage?" *Journal of Software: Evolution and Process*, 2016.

[25] K. Alharbi and T. 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*, ser. MobileHCI '15. New York, NY, USA: ACM, 2015, pp. 515–524.

[26] Y. Qiao, Z. Zheng, and F. Qin, "An Empirical Study of Software Aging Manifestations in Android," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct 2016, pp. 84–90.

[27] Y. Lyu, J. Gui, M. Wan, and W. G. J. Halfond, "An Empirical Study of Local Database Usage in Android Applications," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2017, pp. 444–455.

[28] A. K. Jha, S. Lee, and W. J. Lee, "Developer Mistakes in Writing Android Manifests: An Empirical Study of Configuration Errors," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, May 2017, pp. 25–36.

[29] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An Empirical Study of the Robustness of Inter-component Communication in Android," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, June 2012, pp. 1–12.

[30] T. Takala, M. Katara, and J. Harty, "Experiences of System-Level Model-Based GUI Testing of an Android Application," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, March 2011, pp. 377–386.

[31] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A GUI Crawling-Based Technique for Android Mobile Application Testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, March 2011, pp. 252–261.

[32] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI Ripping for Automated Testing of Android Applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261.

[33] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, Sept 2015.

[34] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, Stochastic Model-based GUI Testing of Android Apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 245–256.

[35] W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 623–640.

[36] Y. Cao, G. Wu, W. Chen, and J. Wei, "CrawlDroid: Effective Model-based GUI Testing of Android Apps," in *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, ser. Internetware '18. New York, NY, USA: ACM, 2018, pp. 19:1–19:6.

[37] Y.-M. Baek and D.-H. Bae, "Automated Model-based Android GUI Testing Using Multi-level GUI Comparison Criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 238–249.

[38] P. Wang, B. Liang, W. You, J. Li, and W. Shi, "Automatic Android GUI Traversal with High Coverage," in *Communication Systems and Network Technologies (CSNT), 2014 Fourth International Conference on*, April 2014, pp. 1161–1166.

[39] T. Azim and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 641–660.

[40] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static Window Transition Graphs for Android," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 658–668.

[41] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, and A. Rountev, "Static Window Transition Graphs for Android," *International Journal of Automated Software Engineering*, vol. 25, no. 4, pp. 833–873, Dec. 2018.

[42] A. Huang, M. Pan, T. Zhang, and X. Li, "Static Extraction of IFML Models for Android Apps," in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '18. New York, NY, USA: ACM, 2018, pp. 53–54.

[43] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing Combinatorics in GUI Testing of Android Applications," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 559–570.

[44] W. Yang, M. R. Prasad, and T. Xie, *A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 250–265.

[45] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 93–104.