UI Obfuscation and Its Effects on Automated UI Analysis for Android Apps

Hao Zhou
The Hong Kong Polytechnic University
Hong Kong, China
cshaoz@comp.polyu.edu.hk

Ting Chen*
University of Electronic Science and Technology of China
Chengdu, China
brokendragon@uestc.edu.cn

Haoyu Wang
Beijing University of Posts and Telecommunications
Beijing, China
haoyuwang@bupt.edu.cn

Le Yu
The Hong Kong Polytechnic University
Hong Kong, China
cslyu@comp.polyu.edu.hk

Xiapu Luo*
The Hong Kong Polytechnic University
Hong Kong, China
csxluo@comp.polyu.edu.hk

Ting Wang
Pennsylvania State University
Pennsylvania, USA
inbox.ting@gmail.com

Wei Zhang
Nanjing University of Posts and Telecommunications
Nanjing, China
zhangw@njupt.edu.cn

ABSTRACT

The UI driven nature of Android apps has motivated the development of automated UI analysis for various purposes, such as app analysis, malicious app detection, and app testing. Although existing automated UI analysis methods have demonstrated their capability in dissecting apps' UI, little is known about their effectiveness in the face of app protection techniques, which have been adopted by more and more apps. In this paper, we take a first step to systematically investigate UI obfuscation for Android apps and its effects on automated UI analysis. In particular, we point out the weaknesses in existing automated UI analysis methods and design 9 UI obfuscation approaches. We implement these approaches in a new tool named UIObfuscator after tackling several technical challenges. Moreover, we feed 3 kinds of tools that rely on automated UI analysis with the apps protected by UIObfuscator, and find that their performances severely drop. This work reveals limitations of automated UI analysis and sheds light on app protection techniques.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

*The corresponding authors.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416642

ACM Reference Format:
Hao Zhou, Ting Chen*, Haoyu Wang, Le Yu, Xiapu Luo, Ting Wang, and Wei Zhang. 2020. UI Obfuscation and Its Effects on Automated UI Analysis for Android Apps. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416642

1 INTRODUCTION

Millions of Android apps are available in Google Play and third-party Android app markets. Due to the UI driven nature of Android apps, many automated UI analysis methods have been proposed [21, 24, 27, 37, 38, 41, 47–51, 59, 60, 64, 65] for various purposes, such as, UI centric app analysis [21, 47, 59, 60], UI based repackaged app detection [27, 41, 48, 49, 51, 64, 65], UI driven app testing [20, 37–39, 50], and many other applications that rely on the correct UI information extracted from apps [22, 23, 25, 26, 28, 30, 31, 43, 52, 62, 63, 67].

Existing automated UI analysis methods can be classified into 4 categories, including (1) static layout (or static view hierarchy) based methods; (2) static activity transition based methods; (3) runtime view hierarchy based methods; (4) runtime screenshot based methods. The methods in categories (1) and (2) conduct static analysis on apps to collect UI information, while the methods in categories (3) and (4) perform dynamic analysis to extract UI information. The static analysis based methods are more scalable than dynamic analysis based methods, but the latter can collect more accurate UI information than the former.

As more and more apps adopt protection mechanisms to impede app analysis [32, 34, 54, 57], little is known whether the existing automated UI analysis methods for apps are still effective. One possible reason may be that existing app protection mechanisms focus on protecting the bytecode of apps, such as, the obfuscation methods for raising the bar of understanding the bytecode [34, 54], and

they alone cannot always protect the UI information of apps, such as the UI information in the manifest file of apps. Such the blindspot gives malware (or repackaged app) makers the opportunity to leverage UI obfuscation, which manipulates the elements related to the UI of an app, to obstruct automated UI analysis methods.

To fill the gap, in this paper, we conduct the first systematic investigation on UI obfuscation mechanisms for Android apps and their effects on automated UI analysis methods through 3 steps. First, we identify the weaknesses of existing automated UI analysis methods by invalidating their implicit/explicit assumptions, and then propose nine basic UI obfuscation approaches exploiting those weaknesses (in §3). These basic approaches can be used together to strengthen the effectiveness of UI obfuscation.

Second, we develop UIObfuscator, a new tool for automatically obfuscating apps without source code by using our UI obfuscation approaches. It is non-trivial to develop UIObfuscator due to 2 challenges. **C1 (invisibility)**: the effect of UI obfuscation should be transparent to app users. That is, it should be hard for users to distinguish original apps from obfuscated ones. **C2 (non-intrusiveness)**: the operation of UI obfuscation should neither obstruct interactions between obfuscated apps and app users nor introduce obvious overhead. We address the challenges by carefully designing and implementing each UI obfuscation approach (in §4-§7).

Third, we evaluate the impact of UI obfuscation on automated UI analysis methods used in 3 kinds of major applications, including, UI centric app analysis [21, 47, 59, 60], UI based repackaged app detection [27, 41, 48, 49, 51, 64, 65], and UI driven app testing [20, 37–39, 50]. More precisely, we apply UIObfuscator to randomly selected apps, and then feed obfuscated apps to the representative tools in each kind of applications. By comparing their performance on original apps and obfuscated apps, we observe that UI obfuscation can significantly decrease the performance of these tools. Moreover, we evaluate the overhead introduced by UIObfuscator and find that it incurs at most 15 milliseconds delay to launching obfuscated apps and produces a slight size expansion. UIObfuscator and the apps involved in the evaluation are available at https://github.com/moonZHH/UIObfuscator.

It is worth noting that the insights learnt from the experiments are also applicable to other applications that rely on the correct UI information extracted from apps [22, 23, 25, 26, 28, 30, 31, 43, 63, 67] because they use the same UI analysis methods as the studies examined in this paper. Moreover, our UI obfuscation methods can help developers protect their apps from being inspected by adversaries through testing (e.g., anti-fuzzing [35]), and inform app analysts the limitations of existing automated UI analysis methods.

In summary, we make the following major contributions:

- To the best of our knowledge, it is the *first* systematic investigation on UI obfuscation and its effects on automated UI analysis for Android apps. We not only point out the common weaknesses for existing automated UI analysis methods but also propose 9 basic UI obfuscation approaches exploiting these weaknesses.
- We design and develop UIObfuscator, a novel tool that implements the 9 UI obfuscation approaches and can automatically obfuscate the UI of Android apps without source code.
- We extensively evaluate the impact of UI obfuscation on representative automated UI analysis methods for 3 kinds of applications. The results show that proposed UI obfuscation approaches can impede UI centric app analysis, thwart UI based repackaged app detection, and obstruct UI driven app testing. This study sheds light on the design of robust automated UI analysis methods.

2 BACKGROUND

This section introduces the necessary background. In particular, §2.1, §2.2, and §2.3 are relevant to our UI obfuscation approaches that exploit dynamic resource loading (in §4.2), app method patching (in §5.4), and overlay injection (in §6.2), respectively.

2.1 Asset Management in Android Apps

APK: An APK is a compressed file, including one or more dex files containing the app's bytecode, a unique manifest file, and multiple asset (or resource) files, such as bitmaps and layout files [13].

Layout: Layout files determine basic view hierarchies of the app. When asset files are packaged into an APK, each layout file is compressed and assigned with a unique resource identifier.

AssetManager: Android apps can use 2 classes (i.e., `Resources` and `AssetManager`) to manage their layout files. `AssetManager` provides access to all resource files including layout files, and the activities of an app share a common `AssetManager` instance. `Resources` relies on `AssetManager` to query layout files.

2.2 Method Execution in Android Runtime

Android Runtime: Before Android 5.0, DVM is the default runtime, which uses the interpreter to execute the Dalvik bytecode of an app, which is compiled from Java source code. Afterwards, it is replaced by ART, and the Dalvik bytecode of an app will be conditionally transformed to native instructions that can be directly executed.

ArtMethod: In ART, a Java method is represented by an `ArtMethod` object, and the object's `dex_code_item_offset_` field refers to the `CodeItem` structure that stores the bytecode of this method. If the execution of an `ArtMethod` object is handled by the interpreter, the `entry_point_from_interpreter_` field (for Android 5.0, 5.1, 6.0) or the `entry_point_from_quick_compiled_code_` field (for Android versions since Android 7.0) of the `ArtMethod` object holds the entry address of the interpreter.

Method Invocation: Besides using the bytecode `invoke-virtual` or the compiled code `bl`, an app can employ Java reflection or native reflection to call an instance method, which is always invoked with respect to an object (i.e., the receiver object) [10]. In particular, the Java/native reflection approach invokes the target function through calling the Android runtime method `ArtMethod::Invoke`. Before executing the method invocation, ART verifies whether the Java/native reflection is valid by calling the `VerifyObjectIsClass` method, which examines whether the receiver of the method invocation is an instance of the class that defines the callee method.

2.3 Window Organization in Android

View: View component is a basic building block for the UI of apps. By default, each view instance occupies a rectangular area on the device screen and is responsible for responding user events.

View Hierarchy: A view hierarchy (or a layout) is a tree structure, where each node represents a view component that composes the UI of apps and the edge indicates the parent-child relationship among view components presented in a window.

Window: Each window instance holds the view hierarchy of a UI component, e.g., the activity, the dialog, or the menu. One important property of a window is *z-order*, which denotes the z-axis position where the view hierarchy will be rendered on the device screen. A window with a larger z-order value will usually conceal the window with the smaller z-order value completely or partially.

Window Type: Android provides 3 types of windows: (1) application window, which contains the view hierarchy of each activity; (2) system window, which refers to the UI of the system input method, the system status bar, or the system keyguard. A system window can be created by normal apps as long as the `SYSTEM_ALERT_WINDOW` permission has been granted; (3) sub-window, which is a special type of windows (e.g., the window of a dialog) affiliated to the application window or the system window.

Window Flag: Android defines a set of flags to control the window behaviors. Among them, 4 flags are important to our study:

* `FLAG_NOT_FOCUSABLE`: If set, the window will not intercept the key or other button events (e.g, clicking the button), and another window behind it will consume the user event.
* `FLAG_NOT_TOUCH_MODAL`: If set, the pointer events (e.g., touching the device screen) happened outside of the window will be sent to another window behind it.
* `FLAG_WATCH_OUTSIDE_TOUCH`: If set, a special notification (i.e., `MotionEvent.ACTION_OUTSIDE`) will be sent to the window to inform the touch conducted outside of the window.
* `FLAG_SECURE`: If set, the content of the window will not appear in the screenshot captured by the common app or grabbed by the shell command, `screencap`.

3 OVERVIEW OF OUR UI OBFUSCATION METHODS FOR APPS

We first point out the common weaknesses of the existing automated UI analysis methods (in §3.1), and then introduce the basic ideas of our UI obfuscation approaches exploiting the weaknesses. The technical details of these approaches are presented in §4-§7.

3.1 Weaknesses in Automated UI Analysis Methods

W1: Static layout based methods parse the layout files to get static view hierarchies of the app. However, such static view hierarchies can be easily manipulated.

W2: Static activity transition based methods construct the activity transition graph (ATG) of the app. They locate activity transition related APIs (e.g., `Activity.startActivity`) to determine transition relationships among app activities. However, since this process relies on static bytecode analysis, it will be hindered by dynamic language features.

W3: Runtime view hierarchy based methods usually leverage a UI testing tool from Google, UIAutomator [18], to dynamically retrieve the app's view hierarchies. However, we find that `UIAutomator` can only capture *the view hierarchy of the topmost focused window*. That is, it cannot obtain layouts of windows that are partially or completely covered by the others. These methods also suffer from **W1** because changing static view hierarchies may lead to changes in runtime view hierarchies.

Table 1: Weaknesses exploited by UI obfuscation methods.

Idx	MLF	SLF	IPA	ESC	RFC	PAM	UVH	MOW	PAS
W1	✓	✓	✗	✗	✗	✗	✗	✗	✗
W2	✗	✗	✓	✓	✓	✓	✗	✗	✗
W3	✓	✗	✗	✗	✗	✗	✓	✓	✗
W4	✗	✗	✗	✗	✗	✗	✗	✗	✓

W4: Runtime screenshot based methods usually employ the shell command `screencap` provided by Android or the screen mirroring/-casting tools [7, 15, 16] to dynamically capture the app's snapshots instead of getting the app's view hierarchies. However, these tools usually fail to retrieve the visual content of windows protected by the window flag, `FLAG_SECURE`.

Remark. The 1st and 2nd categories of methods are more scalable because they can directly process the APK files without the need of running the apps. Although the 3rd and 4th categories of methods can collect more accurate UI information and thus may be more resilient to UI obfuscation, they need to execute the apps and thus take much longer time to process each app.

3.2 Basic UI Obfuscation Approaches

Exploiting the above weaknesses, we design 9 basic UI obfuscation approaches as shown in Table 1.

(1) Modifying Layout File (MLF): Invisible view components are added to the app's layout files to modify the view hierarchies.

(2) Substituting Layout File (SLF): Fake layout files are inserted into the APK while original layout files will be extracted from the APK and loaded at runtime to restore view hierarchies of the app.

(3) Injecting Proxy Activity (IPA): Additional proxy activities are injected to modify the app's ATG by intercepting the direct transition relationship between the app activities.

(4) Encoding String Constant (ESC): String constants, especially those indicating the class names of app activities, are encoded to set additional obstacles for ATG builders.

(5) Rewriting Function Call (RFC): Function calls that involve activity transition related APIs are rewritten through the Java reflection to impede the process of building the ATG.

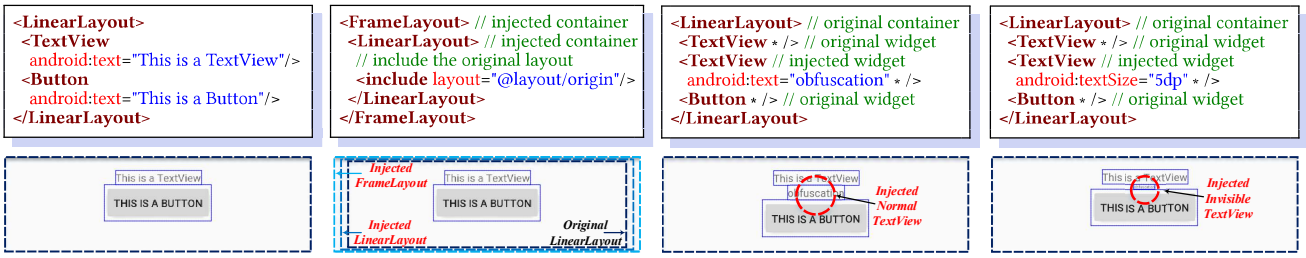
(6) Patching App Method (PAM): To hide method invocations related to constructing the ATG, app methods that contain activity transition related APIs will be first extracted from the APK and then loaded and executed at runtime to finish the original operations.

(7) Updating View Hierarchy (UVH): Instead of directly modifying layout files, view components will be created and inserted by bytecode to dynamically update the app's view hierarchies.

(8) Misusing Overlay Window (MOW): When an app activity is going to be rendered on the device screen, an overlay window is launched to seize the window focus to prevent runtime view hierarchies of the app from being captured by `UIAutomator`.

(9) Preventing App Screenshot (PAS): `FLAG_SECURE` is enabled in each activity of the app to prohibit its visual content from being presented in the screenshot.

We elaborate more on the design and implementation of MLF and SLF in §4, IPA, ESC, RFC, and PAM in §5, UVH, and MOW in §6, PAS in §7, respectively. Note that different UI obfuscation approaches can be used together to strengthen the effectiveness of UI obfuscation. For example, using SLF, PAM, MOW and PAS together can hinder all existing automated UI analysis methods.



(a) The original layout. (b) Inject view containers. (c) Inject normal widgets. (d) Inject invisible widgets.

Figure 1: Modify layout file.

4 MANIPULATING STATIC LAYOUT

This section presents 2 basic UI obfuscation approaches, namely modifying layout files (in §4.1) and substituting layout files (in §4.2), in order to exploit **W1**.

4.1 Modifying Layout File (MLF)

Design: We insert additional view components to the app's original layout files. Consequently, from the viewpoint of static layout based methods, the static view hierarchies of the obfuscated app are different from those of the original app. To fulfill the invisibility requirement (i.e., **C1**), we adjust properties (e.g., size and color) of injected view components to make them transparent to app users. Note that, since MLF modifies original layout files of the app, which makes the obfuscated app's runtime view hierarchies different from those of the original one. Thus, MLF can also exploit **W3**.

Implementation: We implement this UI obfuscation approach through 2 ways. One is to inject redundant view containers. An example is shown in Figure 1b. We regard the injected `FrameLayout` and `LinearLayout` as redundant because the view container has only one child node and the unique child node is also a view container. That is, if we remove the view container and place its child node to its position on the tree structure, no visual difference will be caused. By comparing the screenshot of the original layout (i.e., Figure 1a) with that of the modified layout (i.e., Figure 1b), we can see that it is difficult to differentiate between them.

The other way is to insert additional view widgets. Figure 1c gives an example of adding a `TextView` widget into the original layout file. To make the injected `TextView` widget transparent, we leave the `android:text` property unspecified to prevent the widget from being noticed through its textual contents. Note that the `TextView` is still not fully invisible because there will be a visible placeholder presented in the captured screenshot. To tackle this issue, in Figure 1d, we adjust the font size of the widget to a rather small value (e.g., *5dp* in this example), and thus it is difficult to find the difference between Figure 1a and Figure 1d.

4.2 Substituting Layout File (SLF)

Design: We replace the app's original layout files with fake ones so that static view hierarchies extracted by static layout based UI analysis methods will be different from the original ones. To fulfill the non-intrusiveness requirement (i.e., **C2**), we load the original assets at runtime so that the loaded layout files will substitute the fake ones and recover the original view hierarchies. Note that SLF is different from MLF because it does not modify the view hierarchies defined in the original layout files.

```
   input: p, the path of the asset file, which is going to be loaded.
1  Function substitute_layout_file():
2      mAssetManager = initialize_AssetManager()
3      mAssetManager.addAssetPath(p)
4      Collection<Resources> mResources = null;
5      if Build.VERSION.SDK_INT >= KITKAT then
6          mResourcesManager = obtain_ResourcesManager()
7          mResources = get_Resources(mResourcesManager)
8      else
9          mActivityThread = obtain_ActivityThread()
10         mResources = get_Resources(mActivityThread)
11     end
12     foreach resource in mResources do
13         resource.mAssets = mAssetManager
14     end
15 return
```

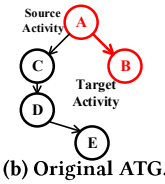
Figure 2: Algorithm for substituting layout file.

Implementation: Since `AssetManager` handles queries about layout contents, to implement the layout file substitution, we create a new `AssetManager` instance to load the asset file containing original layout definitions and use it to replace the original one created by Android framework so that the new instance will answer the queries from the obfuscated app (e.g., findViewById()). Figure 2 shows the algorithm for conducting this process. In line 2-3, we initialize a new instance of `AssetManager`, and call the `addAssetPath` method to let the created `AssetManager` instance load the original assets. Note that, since `addAssetPath` is a hidden method, we exploit the Java reflection to access it.

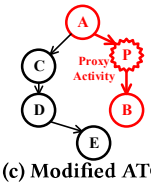
Then, we use the created `AssetManager` instance to replace the ones held by app activities or referenced by existing `Resources` objects. In practice, we only update the `AssetManager` instances referenced by `Resources` objects because we implement the layout file substitution in the `onCreate` method of the inherited `Application` class of the app. In this case, since the `onCreate` method will be executed before the creations of app activities and Android framework will make each app activity hold the `AssetManager` instance referenced by the associated `Resources` object (i.e., the one created by us), there is no need to substitute the `AssetManager` objects held by app activities. In line 5-11, we first collect `Resources` objects stored in the `mResourceReferences` field of the `ResourcesManager` instance, as well as the objects stored in the `mActiveResources` field of the `ActivityThread` instance. Then, in line 12-14, we use the newly created `AssetManager` instance to update the instance stored in the `mAssets` field of each collected `Resources` object.

```
1  // Code presented in the source Activity A.
2  public void onClick(*) {
3    Intent target = new Intent(*, B.class);
4    P.originIntent = target;
5    Intent proxy = new Intent(*, P.class);
6    startActivity(proxy); // A->P
7  }
8  // Code for the proxy Activity P.
9  public staic Intent originIntent;
10 protected void onCreate(*) {
11   startActivity(P.originIntent); // P->B
12   finish(); // necessary
13 }
14 // Code presented in the target Activity B.
15 public class B extends Activity
16 { /* nothing to be changed */ }
```

(a) Code snippet.



(b) Original ATG.



(c) Modified ATG.

Figure 3: Injecting proxy activity.

5 DISTORTING CONSTRUCTED ATG

To exploit **W2**, we design 2 basic UI obfuscation approaches to make the constructed ATG of the obfuscated app differ from that of the original app. First, we inject proxy activities to modify the ATG (§5.1). Second, we compromise the process of constructing the ATG to make the built ATG incomplete by using code obfuscation techniques: encoding string constants (in §5.2), rewriting function calls via the Java reflection (in §5.3), and patching app methods through the dynamic code loading (in §5.4).

5.1 Injecting Proxy Activity (IPA)

Design: We inject additional activity components into the app to modify its original ATG by strategically introducing more nodes and edges to the ATG. More precisely, the injected activities will serve as the proxy that intercepts direct transitions among original app activities. Note that app users may notice the injected proxy activities because they will be passively pushed into the app's back stack [45, 46]. In this case, if app users continuously click the BACK button of the device, proxy activities will be popped from the stack, re-rendered on the device screen, and will be seen by the app user. To tackle this issue for meeting the invisibility requirement (i.e., **C1**), we need to maintain the back stack of the app unchanged.

Implementation: Figure 3a shows the code snippet for dynamically injecting a proxy activity. The original and the modified ATG of the target app are shown in Figure 3b and Figure 3c, respectively. In line 5-6, we break the original activity transition $A \rightarrow B$, and set up a new transition from the source activity A to the proxy activity P (i.e., $A \rightarrow P$). Then, in line 11, we create a transition from the proxy activity P to the target activity B (i.e., $P \rightarrow B$). Hence, the original activity transition $A \rightarrow B$ is replaced by the new one $A \rightarrow P \rightarrow B$. Note that, currently, we just handle the explicit intent.

To accurately instruct the proxy activity to launch the target activity, we use the variable *originIntent* in line 9 to store the intent object used to launch the target activity by the original app. Since this variable contains the information (e.g., class name) about the target activity, the proxy activity can pass it to the `startActivity` API in line 11 to launch the target activity. To make the proxy activity invisible to app users, after the target activity has been launched (e.g., B in Figure 3c), we actively remove the proxy activity (e.g., P in Figure 3c) from the app's back stack by calling the `finish` function of the `Activity` class (i.e., line 12). Consequently, even if app users navigate back from the target activity to the source activity, the transition flow will not be blocked by the proxy activity, and thus app users will not notice the presence of the proxy activity.

```
1  // Code presented in the source Activity, S.
2  public void onClick(View view) {
3    Intent intent = new Intent();
4    String encode = "eman_ytivitca"; // the encoded string
5    String decode = new StringBuilder(encode).reverse().toString();
6    if (condition_1) // intent.setClassName(S.getPackageName(),"activity_name")
7      intent.setClassName(S.getPackageName(), decode);
8    if (condition_2) { // intent = new Intent(S.this, activity_name.class)
9      Class<?> targetActivityClass = Class.forName(decode);
10     intent = new Intent(S.this, targetActivityClass);
11   }
12   startActivity(intent); // original code
13 }
```

Figure 4: Encode string constant.

```
1  // Code presented in the source Activity, S.
2  public void onClick(View view) {
3    Intent i = new Intent(S.this, TargetActivity.class);
4    // startActivity(i) // original invocation
5    Class<?> clazz = Class.forName("Activity"); // assume S extends Activity
6    Method method = clazz.getMethod("startActivity", new Class[] {*});
7    method.invoke(S.this, i); // call the framework API, Activity.startActivity
8  }
```

Figure 5: Rewrite function call.

5.2 Encoding String Constant (ESC)

Design: ATG builders resolve the intent objects passed to activity transition related APIs to find the target activity (i.e., the receiver of each intent object). Developers usually explicitly specify the intent receiver through its class name (i.e., *condition_1* in Figure 4) or its corresponding `java.lang.Class` object (i.e., *condition_2* in Figure 4). Exploiting these observations, this UI obfuscation approach encodes string constants, especially the class names of target activities, to make it difficult for ATG builders to correctly find the target of each activity transition. Consequently, the constructed ATG of the obfuscated app will be incomplete.

Implementation: Figure 4 shows a simple example of encoding the class name of the activity. In line 4, the string is encoded by reversing the order of characters, "`activity_name`", which is the class name of the target activity. It is worth noting that other sophisticated string encoding algorithms can also be employed to prevent ATG builders from getting the original string. In line 5, the decoding method is invoked to recover the original string. Furthermore, if the app specifies the intent receiver using its class name, the *decode* variable, a substitution of the original string, will be sent to the `setClassName` method (i.e., line 7). Otherwise, the decoded string will be passed to the `Class.forName` API (i.e., line 9) to retrieve the `java.lang.Class` object of the target activity, which will be used to specify the receiver (i.e., the target activity) of the intent object.

5.3 Rewriting Function Call (RFC)

Design: Locating the invocations of activity transition related APIs is another critical step in building the app's ATG. Based on this observation, this UI obfuscation approach hides such invocations to ATG builders by rewriting such function calls through Java reflection. Consequently, the constructed ATG of the obfuscated app will have less edges (or even less nodes) than the correct one.

Implementation: Figure 5 shows an example of rewriting the function call. In line 5-7, the original call to the `startActivity` method is rewritten using the Java reflection. We can also encode the names

```
input: p, the path of the dex file, which is going to be loaded.
       c, the context of the app's Application object.
       m, the signature of the method going to be patched.
1  // recover the modified app method
2  Function dynamic_code_loading(c, m):
3      dex = load_patch_file(c, p)
4      parent_classloader = c.getClassLoader()
5      patch_classloader = new ClassLoader(parent_classloader)
6      source_method = load_method(parent_classloader, m)
7      target_method = load_method(patch_classloader, m)
8      if the modified app method is a callback function then
9          replace_method(source_method, target_method)
10     end
11 return
```

Figure 6: Algorithm for patching app method.

of activity transition related APIs to make it harder for ATG builders to locate and analyze such essential function calls.

5.4 Patching App Method (PAM)

Design: We can also prevent ATG builders from identifying activity transition related APIs by first removing the bytecode of methods, which contain invocations of these APIs, and then dynamically loading them at runtime. Since ATG builders cannot find activity transition related APIs in the bytecode of the obfuscated app, the constructed ATG will have less edges than that of the original app. We implement this approach using dynamic app patching tools (e.g., `tinker` [17], `andfix` [4], `nuwa` [12], and `amigo` [3]).

Implementation: Figure 6 shows the algorithm for dynamically patching the app methods that call activity transition related APIs. We first extract the bytecode of such methods from the original app and store it to a dex file. In line 3, we call the framework API, `DexFile.loadDex`, to load this dex file. In line 4-5, we create a `ClassLoader` object and take the existing `PathClassLoader` instance of the obfuscated app as its parent class loader, which ensures the patched method can be executed correctly because the `ClassLoader` object can use its parent (i.e., the `PathClassLoader` instance) to find the related classes for executing the patched method. In line 6, we retrieve the app method to be patched from the `PathClassLoader` instance. Then, in line 7, we get the app method that contains the original bytecode of the target method from the created `ClassLoader` object. Subsequently, in line 8-10, we transform the obtained `java.lang.Method` objects to the corresponding `ArtMethod` objects by calling the `FromReflectedMethod` method declared in the `JNIEnv` class so that we can replace the app method loaded by the `PathClassLoader` instance with the one loaded by the created `ClassLoader` object to accomplish the patching.

However, it is worth to mention that we cannot replace the `ArtMethod` object of the instance method that are called by the app through the Java reflection at runtime. More specifically, since the modified method in the obfuscated app and the corresponding method in the patch file are loaded by different class loaders (i.e., the existing `PathClassLoader` instance and the created `ClassLoader` instance), if the method to be patched is invoked through the Java reflection, such a function call cannot pass the verification conducted by the Android runtime. To mitigate this problem, we just apply the `ArtMethod` replacement to the callback methods in the original APK (i.e., line 9). Additionally, we will use `DroidRA` [36] to check whether or not the modified callback methods will be invoked by other app methods via the Java reflection.

```
1  // Code presented in an Activity.
2  protected void onCreate(*) {
3    // original code is omitted
4    LinearLayout container =
5      findViewById(R.id.container);
6    TextView tv = new TextView(*);
7    tv.setText("inject");
8    tv.setTextSize(1); // tiny
9    tv.setTextColor(*); // transparent
10   container.addView(tv);
11 }
```

(a) Relevant code snippet.

```
<LinearLayout
  android:id="@+id/container" >
  // Original view widgets are omitted
  // Following is the injected TextView
  <TextView
    android:layout_height="*"
    android:layout_width="*"
    android:text="inject"
    android:textSize="1sp"
    android:textColor="#00000000" />
</LinearLayout>
```

(b) Modified view hierarchy.

Figure 7: Update view hierarchy.

6 ALTERING RUNTIME VIEW HIERARCHY

To exploit **W3**, we design 2 ways to make the retrieved runtime view hierarchies of the obfuscated app distinct from those of the original app. The first approach (in §6.1) dynamically creates invisible view components and adds them into original view hierarchies of the app so that runtime view hierarchies of the obfuscated app will have more widgets than those of the original app. The second approach (in §6.2) exploits the limitation of `UIAutomator`, which can only capture the view hierarchy of the topmost focused window, by crafting overlay windows to seize the focus from app windows. In this case, the retrieved runtime view hierarchies are the layouts of crafted overlay windows rather than app windows.

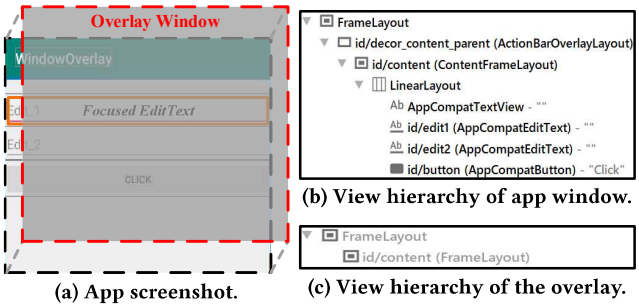
6.1 Updating View Hierarchy (UVH)

Design: We dynamically create invisible view components and add them into original view hierarchies of the app to arbitrarily change the runtime view hierarchies of the obfuscated app.

Implementation: Figure 7a shows the code snippet, which adds a newly created invisible `TextView` to the layout of an app window, and Figure 7b illustrates the modified runtime view hierarchy. More specifically, in line 4-5, we retrieve the view container, to which we inject an invisible `TextView` widget. In line 6-7, we initialize the `TextView` instance and specify its text content. To meet the invisibility requirement (i.e., **C1**), in line 8-9, we adjust the size and the color of the specified text content to make the `TextView` widget tiny and transparent. Finally, in line 10, we update the view hierarchy via adding the created `TextView` to the view container.

6.2 Misusing Overlay Window (MOW)

Design: We exploit the overlay to seize the window focus from the app component (e.g., activity) so that the runtime view hierarchies retrieved by `UIAutomator` refer to the layout of the overlay rather than the app window. To achieve this purpose, the overlay window should be focusable and drawn on top of the device screen. However, such focused overlay window may interfere with interactions between the app and the user. For example, key events (e.g., clicking the BACK button) and touch events (e.g., pressing the screen), which ought to be handled by concealed app components, are intercepted by the overlay. To address this issue, we forward intercepted user events to the proper app window rendered behind the overlay. Note that MOW is different from UVH because it exploits the vulnerability of `UIAutomator` to let it obtain the incorrect runtime view hierarchies rather than modifying the app's view hierarchies.



(a) App screenshot. (b) View hierarchy of app window. (c) View hierarchy of the overlay.

Figure 8: Misuse overlay window.

```
   input: views, an array includes all app windows' root views.
          params, an array contains all app windows' layout parameters.
1  Function onTouch(View view, MotionEvent event):
2  |  fRootView, fView, lRootView = get_focused_view(views, params)
3  |  imm = InputMethodManager.peekInstance()
4  |  imm.mCurRootView = fRootView
5  |  imm.focusIn(fView)
6  return
```

Figure 9: Algorithm for customizing touch event handler.

```
   input: views, an array includes all app windows' root views.
          params, an array contains all app windows' layout parameters.
1  Function onKey(*, KeyEvent event):
2  |  fRootView, fView, lRootView = get_focused_view(views, params)
3  |  imm = InputMethodManager.peekInstance()
4  |  if imm.isActive(fView) == true && fView.instanceof(EditText) then
5  |  |  if imm.hideSoftInputWindow(*) == true then
6  |  |  |  return
7  |  |  end
8  |  end
9  |  lRootView.dispatchKeyEvent(keyEvent)
10 return
```

Figure 10: Algorithm for customizing key event handler.

Implementation: To ensure that the overlay will be drawn on top of any other app windows, we adjust the window type of the overlay to `TYPE_PHONE` or `TYPE_APPLICATION_OVERLAY`. Figure 8a shows an example of such overlay window. The original view hierarchy of the app window and the one retrieved by `UIAutomator` when the target window is concealed by the overlay are shown in Figure 8b and 8c, respectively. Obviously, `UIAutomator` generates a different result due to our UI obfuscation approach.

Since placing an overlay on top of the app window may block common interactions between the covered window and the app user, we forward user events intercepted by the overlay to the proper concealed app window for achieving the invisibility and non-intrusiveness requirements (i.e., **C1** and **C2**). More precisely, touch events, as well as key (or button) events, need to be handled.

To dispatch intercepted touch events to the covered app window, we adjust the window size of the overlay to zero and enable the `FLAG_NOT_TOUCH_MODAL` property of the overlay window. Such configuration allows touch events to pass through the overlay, and in most of the cases, the touch events can be handled by the proper app window. However, if the editable view component (e.g., `EditText`) in the app window is going to consume the touch event, additional effort is required to reconnect the link between the editable widget and the soft input method. Specifically, we enable the `FLAG_WATCH_OUTSIDE_TOUCH` property of the overlay to monitor the touch event and customize the touch event handler, `OnTouchListener.onTouch`.

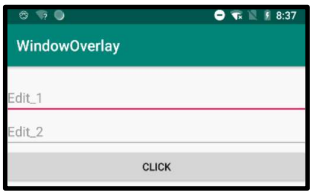
Figure 9 shows the algorithm for customizing the touch event handler registered in the overlay window. It takes in 2 inputs: *views* and *params*, which are retrieved from the `mViews` field and the `mParams` field of the `WindowManagerGlobal` class, respectively. Since the instance of `WindowManagerGlobal` is a singleton, the corresponding fields can be accessed through the Java reflection. The `mViews` field is an array, containing the root view of each app window's view hierarchy, e.g., the topmost `FrameLayout` in Figure 8b and 8c. The view containers (i.e., the root views) included in the `mViews` are organized according to the time when the windows are created. More precisely, the root view of the most recently created window is located at the tail of `mViews`. The `mParams` field stores the layout parameter of each app window, and accordingly, the recorded parameters are ordered depending on the window creation time as well. Hence, there is a one-on-one mapping relationship between each element in *mViews* and *mParams*.

The `onTouch` method is the customized touch event handler registered in the overlay, and its main task is to adjust the improper binding between the focused view and the soft input method. By scrutinizing the binding process of the soft input method [9], we notice that, in normal cases, the soft input method will be linked with the focused view in the focused window. However, since the specially designed overlay is always the focused window, the focused view (e.g., the `EditText` widget in Figure 8a) cannot be connected with the soft input method because it is included in the app window, which is concealed by the overlay and does not have the window focus. To actively rebuild the binding, in line 2, we invoke the auxiliary method, `get_focused_view`, to obtain the view that gets focused, the root view that contains the focused view, and the last focusable root view in the `mViews` field (i.e., the last element of the *views* parameter). The corresponding results are stored in variables `fView`, `fRootView`, and `lRootView`, respectively. After obtaining such information, in line 4, we set the `mCurRootView` field of the `InputMethodManager` singleton to `fRootView`, the root view containing the focused view `fView`. After that, in line 5, we call the `focusIn` method to instruct the `InputMethodManger` instance to rebuild the binding. Consequently, the editable widget in the concealed app can accurately respond the dispatched touch event.

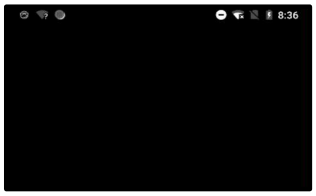
Since the key event will also be received by the focused window, we customize the key event handler of the overlay to forward blocked key events to the covered app window. The algorithm is shown in Figure 10. Note that the inputs and the auxiliary method (i.e., `get_focused_view`) of this algorithm are the same as those in Algorithm 9. The `onKey` method is the key event handler, and the event forwarding process consists of 2 steps. First, in line 3-8, the overlay consumes the BACK button event and hides the activated soft input method if the focused view component is editable. Second, in line 9, the `dispatchKeyEvent` method is invoked to forward the intercepted key events to the topmost app window, and the corresponding root view takes the charge of finding the proper view component to handle such key event.

```
1 // Code presented in an Activity.
2 public void onCreate(*) {
3   setContentView(R.layout.activity_layout); // original code
4   getWindow().addFlags(WindowManager.LayoutParams.FLAG_SECURE);
5   /* the remaining original code is omitted */
6 }
```

(a) Relevant code snippet.



(b) Original app screenshot.



(c) Guarded app screenshot.

Figure 11: Prevent app snapshot.

Table 2: Weaknesses of representative studies.

Idx	App Analysis	Repackaging Detection	Automatic Testing
W1	Gator[47]	DroidEagle[51],ResDroid[48]	N/A
W2	A3E[21],Gator[47]	ViewDroid[65],MassVet[27]	N/A
W3	N/A	RepDroid[64],Soh et al.[49]	GUIRipper[20],Stoat[50] DroidBot[37],Paladin[38]
W4	N/A	Malisa et al.[41]	N/A

7 GUARDING APP SCREENSHOT

To exploit **W4**, we design an approach (PAS) to prevent screenshots from being captured by the snapshot taker.

Design: We leverage the window property, `FLAG_SECURE`, to prevent the app's screenshots from being captured by common snapshot takers (e.g., tools [7, 15, 16] built upon the shell command, `screencap`, or the framework class, `VirtualDisplay`). Once the property is enabled, the visual content of the protected app window will not be included in the captured snapshot. Consequently, analysis results of runtime screenshot based methods become invalid.

Implementation: Figure 11a shows an example of using the window property, `FLAG_SECURE`, to prevent the visual content of an activity from being presented in the screenshot. In line 4, we call `getWindow` to get the corresponding `Window` instance, and then invoke `addFlags` to add the `FLAG_SECURE` property to the obtained `Window` instance. Figure 11b and 11c show the screenshots taken by `scrcpy` [15], a popular snapshot taker, before and after `FLAG_SECURE` has been enabled, respectively.

8 EVALUATION

We implement 9 basic UI obfuscation approaches in a new tool named UIObfuscator with 6,143 lines of Java code and 2,633 lines of Python code. We evaluate the effects of UI obfuscation on representative automated UI analysis methods for 3 kinds of applications, including UI centric app analysis, UI based repackaged app detection, and UI driven app testing, and answer 3 research questions (i.e., **RQ1/2/3**). Table 2 summarizes the studies under examination and their weaknesses. Moreover, we assess the extra overhead introduced by the UI obfuscation approaches and conduct a user survey to evaluate whether they fulfill the invisibility requirement and the non-intrusiveness requirement to answer **RQ4** and **RQ5**.

8.1 Data Set

Apps used for evaluation were downloaded from `F-Droid` [8]. We filter out some apps according to the following 3 requirements:

∗ **R1**: The app should be able to be processed by `Apktool` [6] and `Soot` [53] because UIObfuscator is built on top of them.

∗ **R2**: The app should meet the requirements of automated UI analysis tools that are used to evaluate UIObfuscator.

∗ **R3**: The app can run on Android 5.1.1, where we deploy the UI driven app testing tools. For the ease of exploring the app's UI states, its activity transition should not start from the login activity.

Based on these requirements, we randomly select 200 apps to form the origin APK set. Then, we apply each of UI obfuscation approaches to these apps and generate 1800 obfuscated apps.

Table 3: The effect on UI centric app analysis.

APK	A3E				Gator					
Set	$diff_n$	p_{wilcox}	$diff_e$	p_{wilcox}	$diff_n$	p_{wilcox}	$diff_e$	p_{wilcox}	$diff_v$	p_{wilcox}
origin	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$
MLF	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	+210.4%	$1.6e^{-18}$
UVH	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$
IPA	+52.5%	$2.7e^{-25}$	+43.6%	$2.7e^{-25}$	+31.6%	$5.9e^{-26}$	+82.2%	$1.4e^{-11}$	0.0%	$1.0e^{0}$
ESC	-92.8%	$7.9e^{-20}$	-92.8%	$7.1e^{-20}$	0.0%	$1.0e^{0}$	-47.7%	$1.5e^{-11}$	0.0%	$1.0e^{0}$
RFC	-84.7%	$1.4e^{-19}$	-85.1%	$1.5e^{-19}$	0.0%	$1.0e^{0}$	-48.6%	$8.4e^{-12}$	0.0%	$1.0e^{0}$
SLF	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	-52.5%	$6.8e^{-20}$
PAM	-69.1%	$9.3e^{-19}$	-70.8%	$9.0e^{-19}$	0.0%	$1.0e^{0}$	-45.9%	$2.6e^{-11}$	0.0%	$1.0e^{-5}$
MOW	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$
PAS	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$	0.0%	$1.0e^{0}$

8.2 RQ1: How does UI Obfuscation affect UI Centric App Analysis?

Tools: We select 2 representative open-source UI centric app analysis tools (**A3E** [21] and **Gator** [47]) to analyze the obfuscated apps. A3E performs static bytecode analysis to construct the ATG of the app. It first locates the APIs related to activity transition, and then performs data analysis on the `Intent` objects passed to such APIs to create ATG nodes and build ATG edges. By treating broadcast receivers and services as ATG nodes, A3E also considers the connections between these components as ATG edges. Gator not only constructs the ATG but also recovers the static view hierarchies of the app. To build the ATG, Gator treats all activities declared in `AndroidManifest.xml` as ATG nodes, and adopts an approach similar as A3E to build ATG edges. To recover the app's static view hierarchies, Gator first parses layout files to construct the basic view hierarchies, and then performs static reference analysis to find the dynamically generated view components and appends them into the basic view hierarchies.

Result: Table 3 lists the results. Precisely, $diff_n$ shows the average ratio of the changed number of ATG nodes. It is calculated via $diff_n = avg(|N_i' - N_i| \div N_i)$, where N_i and N_i' separately denote the number of ATG nodes of the original app and the obfuscated app. $diff_e$ shows the average ratio of the changed number of ATG edges. It is calculated via $diff_e = avg(|E_i' - E_i| \div E_i)$, where E_i and E_i' separately denote the number of ATG edges of the original app and the obfuscated app. $diff_v$ shows the average ratio of the changed number of view components. It is calculated via $diff_v = avg(|V_i' - V_i| \div V_i)$, where V_i and V_i' separately denote the number of view components in recovered view hierarchies of the original app and the obfuscated app. To assess the statistical confidence

of the results, we conduct Wilcoxon signed-rank test [44, 56] on the number of ATG nodes, ATG edges, and view components of the obfuscated app and the original app, individually. p_{wilcox} in Table 3 denotes the p-value of the test. Specifically, if the p-value is smaller than 0.05, it implies that the distribution of the number of ATG nodes, ATG edges, or view components of the obfuscated app is different from that of the original app, which suggests the constructed ATG or the recovered view hierarchies of the obfuscated app are totally different from those of the original app.

The results show that ESC, RFC, and PAM, which compromise the process of constructing the ATG, obviously reduce the number of nodes and edges included in constructed ATGs of obfuscated apps. For A3E, more than 69% of ATG nodes and ATG edges of original apps are excluded in the ATGs of obfuscated apps. Meanwhile, for Gator, around 45% of ATG edges of original apps are not presented in the ATGs of obfuscated apps. Since Gator treats activities declared in the manifest file as ATG nodes, ESC, RFC, and PAM have no impact on the number of ATG nodes of obfuscated apps because they will not introduce additional activities.

IPA, which injects an activity to proxy each activity transition of the original app, introduces 82.2% more edges in the obfuscated app's ATG constructed by Gator. Since A3E fails to resolve the `Intent` objects indicating the transitions between the proxy activity and other app activities, IPA introduces 43.6% more edges in the obfuscated app's ATG constructed by A3E.

MLF and SLF, which manipulate the app's layout files, make the recovered static view hierarchies of obfuscated apps different from those of original apps. In particular, we instruct MLF to insert 20 invisible `TextView` to each app activity's layout file, which makes the number of view components in the recovered static view hierarchies of obfuscated apps 210.4% times greater than that of original apps. Meanwhile, SLF replaces each app activity's layout file with the one that contains only a `LinearLayout` widget, which makes the number of view components in the recovered view hierarchies of obfuscated apps just 47.5% of the original one. Although Gator can handle dynamically generated view components, it fails to find the view containers, to which we insert `TextView` widgets. Consequently, UVH could not change the recovered view hierarchies.

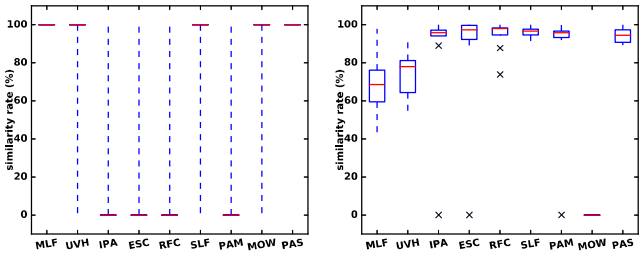
Answer to RQ1: UI obfuscation approaches, including IPA, ESC, RFC, and PAM, make the constructed ATGs of obfuscated apps dramatically different from those of original apps. Moreover, MLF and SLF make the recovered static view hierarchies of obfuscated apps significantly distinct from those of original apps.

Table 4: The effect on UI based repackaged app detection.

Tool	MLF	UVH	IPA	ESC	RFC	SLF	PAM	MOW	PAS
ViewDroid	100.0%	91.0%	1.8%	7.2%	1.8%	91.0%	1.8%	91.0%	100.0%
RepDroid	68.8%	74.9%	86.2%	86.9%	94.3%	96.3%	86.3%	0.0%	94.4%

8.3 RQ2: How does UI Obfuscation affect UI Based Repackaged Apps Detection?

Tools: Although several UI based repackaged app detection systems [27, 41, 48, 49, 51, 64, 65] have been proposed, only a few of them are open-source. Hence, we just use **ViewDroid** [65] and



(a) ViewDroid. (b) RepDroid.

Figure 12: The effect on UI based repackaged app detection.

RepDroid [64] for evaluation. ViewDroid performs static bytecode analysis to build apps' ATGs, and measures the similarity among ATGs to identify repackaged apps. RepDroid uses `UIAutomator` to get runtime view hierarchies of the app, and analyzes the obtained view components to explore app activities and build the layout-group graph (LGG), which represents transitions among different view hierarchies. Then, it calculates the similarity between the obfuscated app and the original app according to their LGGs.

Result: Table 4 lists the average similarity between obfuscated apps and original apps, and Figure 12 shows the boxplots of these results. We can see that IPA, ESC, RFC, and PAM dramatically decrease the similarity calculated by ViewDroid, because these UI obfuscation approaches make the constructed ATGs of obfuscated apps significantly different from those of original apps. Since other UI obfuscation approaches do not change ATGs of original apps, they will not affect the detection results of ViewDroid.

Moreover, MLF, UVH, MOW reduce the similarity measured by RepDroid, because these approaches make the runtime view hierarchies of obfuscated apps distinct from those of original apps. Since the retrieved runtime view hierarchies of the apps obfuscated by MOW are totally different from the ones of original apps, the similarity drops to 0%. As MLF and UVH just modify the runtime view hierarchies of original apps to a certain extent, the similarity percentage only decreases to 69% and 75%, respectively. Other UI obfuscation approaches will not change runtime view hierarchies of original apps, and thus they do not affect the results of RepDroid.

Answer to RQ2: UI obfuscation approaches, including IPA, ESC, RFC, and PAM, will compromise the analysis results of activity transition for repackaged app detection. Meanwhile, MLF, UVH, and MOW, can make the obfuscated app evade the runtime view hierarchy based repackaged app detection.

8.4 RQ3: How does UI Obfuscation affect UI Driven App Testing?

Tools: Existing app testing tools use 2 methods to traverse an app's activities: (1) random event generation based methods; (2) runtime view hierarchy based methods. The former (e.g., Monkey [11] and Sapienz [42]) explores the app without considering view components in the current app window, and thus we exclude them in this study. Since some runtime view hierarchy based tools (e.g., GUIRipper [20] and Dynodroid [39]) are only available on quite early Android versions (e.g., systems before Android 4.0), we exclude them due to the compatibility issue [55]. Therefore, we use

Table 5: The effect on UI driven app testing.

APK Set	Stoat		DroidBot		Paladin	
	$diff_{act}$	p_{wilcox}	$diff_{act}$	p_{wilcox}	$diff_{act}$	p_{wilcox}
origin	0.0%	$1.0e^0$	0.0%	$1.0e^0$	0.0%	$1.0e^0$
MLF	-5.4%	$0.13e^0$	0.0%	$1.0e^0$	-10.8%	$0.07e^0$
UVH	-4.2%	$0.55e^0$	0.0%	$1.0e^0$	-7.9%	$0.57e^0$
IPA	0.0%	$1.0e^0$	0.0%	$1.0e^0$	-4.6%	$0.27e^0$
ESC	+1.8%	$0.72e^0$	-9.0%	$0.57e^0$	-10.8%	$0.16e^0$
RFC	+4.4%	$0.61e^0$	-6.3%	$0.55e^0$	-6.7%	$0.13e^0$
SLF	+4.2%	$0.55e^0$	0.0%	$1.0e^0$	-2.1%	$0.55e^0$
PAM	-2.4%	$0.94e^0$	-12.3%	$0.13e^0$	0.0%	$1.0e^0$
MOW	-71.2%	$0.01e^0$	-67.2%	$0.01e^0$	-71.5%	$0.01e^0$
PAS	0.0%	$1.0e^0$	0.0%	$1.0e^0$	-2.1%	$0.57e^0$

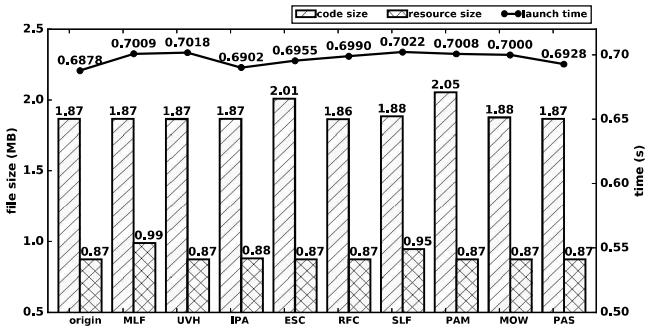


Figure 13: Overhead of UI obfuscation approaches.

Stoat [50], **DroidBot** [37], and **Paladin** [38] in this evaluation. They use the runtime view hierarchies retrieved by `UIAutomator` to find a particular view component, to which the next simulated user event is sent. Moreover, we set their timeout value to half an hour.

Result: Table 5 lists the results. Precisely, $diff_{act}$ shows the average ratio of explored activities. It is calculated via $diff_{act} = avg(|A'_i - A_i| \div A_i)$, where A_i and A'_i separately denote the explored activities of the original app and the obfuscated app.

We find that MOW can obstruct the activity exploration conducted by Stoat, DroidBot, and Paladin. More specifically, since the obtained view hierarchies refer to the layouts of the overlay windows, no valid view components can guide the generation of the next user event, and thus such tools stuck at the app's entry activity.

Answer to RQ3: MOW prevents the runtime view hierarchy based app testing tools from recognizing the view components in the app windows. Hence, no matter how long such tools run, only the entry activity of the app will be explored.

8.5 RQ4: How is the Overhead Induced by UI Obfuscation?

We evaluate the additional code size and delay introduced by UIObfuscator. Note that directly comparing the APK size of the obfuscated app with that of the original app may be inaccurate because in some cases the former may be even smaller than that of the latter. The reason may be that the APK packaging tool (`aapt` [1]) or the APK alignment utility (`zipalign` [19]) is more applicable for obfuscated apps. Thus, we unzip APK files and calculate the average size of bytecode (dex files) and resource files. Then, we compare the average code size and resource size of obfuscated apps with those of original apps to determine the size expansion caused by our UI obfuscation approaches. To measure the introduced delay, we use `ADB` [2] to launch each app in the origin and the obfuscated APK data sets for 10 times and calculate the average launch time. We notice that the launch time for an app in 10 tests will not quite vary from each other. Thus, launching each app 10 times is enough. Afterwards, we compare the average launch time of obfuscated apps with that of original apps to estimate the additional overhead.

Result: The results shown in Figure 13 illustrate that our UI obfuscation approaches only introduce small size expansions and little launch delays. Specifically, PAM introduces the biggest code size expansion, which is 0.17 megabytes on average (9.1% of original apps' average code size), and MLF causes the largest resource size expansion, which is 0.12 megabytes on average (13.8% of original apps' average resource size). Moreover, SLF incurs the longest launch delay, which is around 15 milliseconds on average.

Answer to RQ4: UIObfuscator only introduces small size expansions and little launch delays to the obfuscated app.

8.6 RQ5: Do Our UI Obfuscation Approaches Fulfill the Invisibility and the Non-intrusiveness Requirements?

To answer this research question, we conduct a user survey on 25 students, all of whom are familiar with Android system. In detail, we give each participant the same 5 randomly selected original apps and their corresponding obfuscated apps (50 apps in total). Then, we ask the participants to test the apps using 2 emulators, one for running each original app and another for running the obfuscated app. Specifically, once the participants performed an operation on the original app, we ask them to perform the same operation on the obfuscated app and observe whether the UIs and the functionality of the obfuscated app are the same as those of the original app. After the participants finish the test on a pair of apps, we ask them to answer the question: to what extent (100%/75%/50%/25%/0%) the UIs and the functionality of the pair of apps are the same.

Result: All participants reported that the UIs and the functionality of the apps obfuscated by our UI obfuscation approaches are totally (100%) the same as those of their corresponding original apps. It suggests that our UI obfuscation approaches are transparent to app users and will not obstruct user interactions with obfuscated apps.

Answer to RQ5: All our UI obfuscation approaches meet the requirements of invisibility and non-intrusiveness.

9 DISCUSSION AND THREAT TO VALIDITY

UI obfuscation is different from code obfuscation, because it aims at making obfuscated apps circumvent UI centric analysis, while code obfuscation cannot achieve this purpose in most of cases. However, an app may employ UI obfuscation and code obfuscation together to evade both UI centric and code centric analysis. Moreover, we can adopt code obfuscation (e.g., packing [57, 58, 66] and encryption) to protect the implementations of our UI obfuscation approaches from being analyzed and evaded. Note that, although we use code obfuscation techniques to implement ESC, RFC, and PAM, such

work cannot be done by general code obfuscation tools, such as `ProGuard` [14], which just obfuscates names of app classes, methods, and fields.

Our work is valuable for both the research community and the industry community. Our observations (in §3) and insights obtained from the experiments (in §8) inform researchers the limitations of existing automated UI analysis methods. Our UI obfuscation approaches (in §4, §5, §6, §7) can be adopted by app developers to protect their apps (e.g., bank apps [29]) from being inspected by adversaries. For example, apps can employ MOW to prevent adversaries from using UI driven app testing tools to fuzz them.

The main threat to the external validity of our work is the representativeness of the APKs and the tools, which are used to evaluate the effectiveness of our UI obfuscation approaches. Specifically, we only assess the impacts of UI obfuscation on a limited number of representative UI centric app analysis frameworks, UI based repackaged app detection systems, and UI driven app testing tools. To reduce this threat, in future work, we will use UIObfuscator to obfuscate more apps and then use these obfuscated apps to evaluate more tools that conduct automated UI analysis for Android apps.

Threats to our work's internal validity come from 2 aspects. On one hand, the non-determinism of the evaluated runtime view hierarchy based tools affect the internal validity. To mitigate it, we may apply these tools to analyzing each app under study multiple times. On the other hand, the time threshold on executing UI driven app testing tools influences the internal validity. To reduce the threat, in future work, we will increase the default timeout for a large-scale study.

10 RELATED WORK

10.1 App Obfuscation

Although recent work studied app obfuscation techniques [33, 34, 40, 54], none of them examined UI obfuscation. Maiorca et al. [40] evaluated the performance of anti-malware solutions and found that most of them are resilient to trivial code obfuscation techniques but fail to handle advanced protection mechanisms. Faruki et al. [33] evaluated the effectiveness of existing app deobfuscation tools. They found that existing tools (e.g., `Androguard` [5]) fail to decode real-word apps. Hammad et al. [34] evaluated the effectiveness of anti-malware products against code obfuscation and found that obfuscation techniques can negatively affect the detection results of anti-malware products. Wang et al. [54] thoroughly characterized the obfuscated iOS apps by using statistical language models.

10.2 UI Centric App Analysis

According to the analysis target, we divide UI centric app analysis methods into 4 types. First, static layout based methods [47] analyze static view hierarchies of the app, which are constructed by parsing the app's layout files. Thus, these methods are vulnerable to MLF and SLF. Second, static activity transition based methods [21, 23, 28, 59, 60] analyze ATG of the app, which is built by performing static analysis on the app's bytecode to identify transition relationships among app activities. Accordingly, these methods are vulnerable to IPA, ESC, RFC, and PAM. Third, runtime view hierarchy based methods [22, 30, 31, 43, 63] analyzes the app's runtime view hierarchies, which are obtained by `UIAutomator`. Thus, such methods are vulnerable to UVH and MOW. Fourth, runtime screenshot based methods [25, 26, 30, 31, 67] analyze the app's runtime screenshots. Thus, these methods suffer from PAS. Meanwhile, since these methods always use UI testing tools (e.g., Stoat [50]) to drive the app for getting screenshots, they are vulnerable to MOW.

10.3 UI Based Repackaged App Detection

Existing UI based repackaged app detection systems can be divided into 4 categories. First, static layout based systems [27, 51] characterize an app using its static view hierarchies. More specifically, they first resolve the layout files to construct static view hierarchies of the app. Then, they calculate the similarity of different apps via comparing their static view hierarchies. Hence, they suffer from UI obfuscation approaches, MLF and SLF. Second, static activity transition based systems [48, 61, 65] quantify the similarity between a pair of apps via comparing their ATGs. Hence, they suffer from UI obfuscation approaches, IPA, ESC, RFC, and PAM. Third, runtime view hierarchy based systems [41, 49, 64] leverage `UIAutomator` [18] to retrieve the app's runtime view hierarchies, which are further used to measure the app similarity. However, such methods suffer from UI obfuscation approaches, MLF, UVH, and MOW. Fourth, runtime app screenshot based systems [41] capture the app's snapshots, and then calculate the similarity among screenshots to determine whether the visual experience of an app is similar with another one. However, such methods suffer from the UI obfuscation method, PAS.

10.4 UI Driven App Testing

UI driven app testing tools [20, 37–39, 50] usually analyze runtime view hierarchies retrieved by `UIAutomator` to decide the proper view widget, to which the next simulated user event is sent. Therefore, they will be affected by the UI obfuscation method, MOW.

11 CONCLUSION

We conduct the *first* systematic investigation on UI obfuscation for Android apps and its effect on automated UI analysis methods. After pointing out the weaknesses of existing automated UI analysis methods, we design 9 UI obfuscation approaches and develop UIObfuscator, a new tool for automatically obfuscating Android apps' UI-related elements. We apply UIObfuscator to public available apps, and feed obfuscated apps to 3 kinds of tools that rely on automated UI analysis. The experimental results show that UI obfuscation can severely impact the performance of such tools while introducing little additional overhead to obfuscated apps. Our work sheds light on the limitations of existing automated UI analysis methods and enlightens developers about UI obfuscation approaches.

12 ACKNOWLEDGEMENT

We thank the anonymous reviewers for their helpful comments. This research is partially supported by the Hong Kong RGC Projects (No. 152279/16E, 152223/17E, CityU C1008-16G) and the National Natural Science Foundation of China (No. 61872057, 61702045, 61672297) and National Key R&D Program of China (2018YFB0804100, 2019YFB2101704) and the National Science Foundation under Grant (No. 1953893, 1953813, and 1951729).

REFERENCES

[1] 2020. AAPT. https://developer.android.com/studio/command-line/aapt2.
[2] 2020. ADB. https://developer.android.com/studio/command-line/adb.
[3] 2020. Amigo. https://github.com/eleme/Amigo.
[4] 2020. AndFix. https://github.com/alibaba/AndFix.
[5] 2020. androguard. https://github.com/androguard/androguard.
[6] 2020. Apktool. https://ibotpeaches.github.io/Apktool/.
[7] 2020. CastScreen. https://github.com/JonesChi/CastScreen.
[8] 2020. F-Droid. https://f-droid.org.
[9] 2020. InputMethodManager. http://androidxref.com/8.0.0_r4/xref/frameworks/base/core/java/android/view/inputmethod/InputMethodManager.java.
[10] 2020. Instance Method. https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html.
[11] 2020. Monkey. https://developer.android.com/studio/test/monkey.
[12] 2020. Nuwa. https://github.com/jasonross/Nuwa.
[13] 2020. Overview - App resources. https://developer.android.com/guide/topics/resources/providing-resources.
[14] 2020. ProGuard. https://www.guardsquare.com/en/products/proguard.
[15] 2020. scrcpy. https://github.com/Genymobile/scrcpy.
[16] 2020. ScreenCapture. https://github.com/googlesamples/android-ScreenCapture.
[17] 2020. Tinker. https://github.com/Tencent/tinker.
[18] 2020. UIAutomator. https://developer.android.com/training/testing/ui-automator.html.
[19] 2020. zipalign. https://developer.android.com/studio/command-line/zipalign.
[20] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. In *Proc. ASE*.
[21] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proc. OOPSLA*.
[22] Carlos Bernal-Cárdenas, Kevin Moran, Michele Tufano, Zichang Liu, Linyong Nan, Zhehan Shi, and Denys Poshyvanyk. 2019. Guigle: A GUI Search Engine for Android Apps. In *Proc. ICSE*.
[23] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *Proc. USENIX Security*.
[24] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the app is that? deception and countermeasures in the android user interface. In *Proc. S&P*.
[25] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From UI design image to GUI skeleton: a neural machine translator to bootstrap mobile GUI implementation. In *Proc. ICSE*.
[26] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *Proc. ICSE*.
[27] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *Proc. USENIX Security*.
[28] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. StoryDroid: Automated Generation of Storyboard for Android Apps. In *Proc. ICSE*.
[29] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. 2020. An Empirical Assessment of Security Risks of Global Android Banking Apps. In *Proc. ICSE*.
[30] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proc. UIST*.
[31] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps. In *Proc. UIST*.
[32] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and X Wang. 2018. Things you may not know about android (un)packers: a systematic study based on whole-system emulation. In *Proc. NDSS*.
[33] Parvez Faruki, Hossein Fereidooni, Vijay Laxmi, Mauro Conti, and Manoj Gaur. 2016. Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions. *arXiv preprint arXiv:1611.10231* (2016).
[34] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A Large-scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-malware Products. In *Proc. ICSE*.
[35] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, KyuHyung Lee, and Taesoo Kim. 2019. Fuzzification: Anti-Fuzzing Techniques. In *Proc. USENIX Security*.
[36] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: Taming Reflection to Support Whole-program Analysis of Android Apps. In *Proc. ISSTA*.
[37] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: A Lightweight UI-guided Test Input Generator for Android. In *Proc. ICSE*.
[38] Yun Ma, Yangyang Huang, Ziniu Hu, Xusheng Xiao, and Xuanzhe Liu. 2019. Paladin: Automated Generation of Reproducible Test Cases for Android Apps. In *Proc. HotMobile*.
[39] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proc. FSE*.
[40] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. 2015. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security* 51 (2015), 16–31.
[41] Luka Malisa, Kari Kostiainen, Michael Och, and Srdjan Capkun. 2016. Mobile application impersonation detection using dynamic user interface extraction. In *Proc. ESORICS*.
[42] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proc. ISSTA*.
[43] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI. In *Proc. ASE*.
[44] John W. Pratt. 1959. Remarks on zeros and ties in the Wilcoxon signed rank procedures. *J. Amer. Statist. Assoc.* (1959).
[45] Chuangang Ren, Peng Liu, and Sencun Zhu. 2017. WindowGuard: Systematic Protection of GUI Security in Android. In *Proc. NDSS*.
[46] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards Discovering and Understanding Task Hijacking in Android. In *Proc. USENIX Sec*.
[47] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *Proc. CGO*.
[48] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. 2014. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proc. ACSAC*.
[49] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, and Lipo Wang. 2015. Detecting clones in android applications through analyzing user interfaces. In *Proc. ICPC*.
[50] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proc FSE*.
[51] Mingshen Sun, Mengmeng Li, and John Lui. 2015. DroidEagle: Seamless detection of visually similar Android apps. In *Proc. WiSec*.
[52] Y. Tang, Y. Sui, H. Wang, X. Luo, H. Zhou, and Z. Xu. 2020. All Your App Links are Belong to Us: Understanding the Threats of Instant Apps based Attacks. In *Proc. ESEC/FSE*.
[53] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java bytecode using the Soot framework: Is it feasible?. In *Proc. CC*.
[54] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. 2018. Software Protection on the Go: A Large-scale Empirical Study on Mobile App Obfuscation. In *Proc. ICSE*.
[55] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An Empirical Study of Android Test Generation Tools in Industrial Cases. In *Proc. ASE*.
[56] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Breakthroughs in statistics* (1945).
[57] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive unpacking of Android apps. In *Proc. ICSE*.
[58] L. Xue, H. Zhou, X. Luo, L. Yu, D. Wu, Y. Zhou, and X. Ma. 2020. PackerGrind: An Adaptive Unpacking System for Android Apps. *IEEE Transactions on Software Engineering* (2020).
[59] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static Control-flow Analysis of User-driven Callbacks in Android Applications. In *Proc. ICSE*.
[60] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android. In *Proc. ASE*.
[61] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static window transition graphs for android. In *Proc. ASE*.
[62] L. Yu, J. Chen, H. Zhou, X. Luo, and K. Liu. 2018. Localizing Function Errors in Mobile Apps with User Reviews. In *Proc. DSN*.
[63] Shengcheng Yu, Chunrong Fang, Yang Feng, Wenyuan Zhao, and Zhenyu Chen. 2019. LIRAT: Layout and Image Recognition Driving Automated Mobile Testing of Cross-Platform. In *Proc. ASE)*.
[64] Shengtao Yue, Weizan Feng, Jun Ma, Yanyan Jiang, Xianping Tao, Chang Xu, and Jian Lu. 2017. RepDroid: an automated tool for Android application repackaging detection. In *Proc. ICPC*.
[65] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proc. WiSec*.
[66] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. 2015. DexHunter: toward extracting hidden code from packed Android applications. In *Proc. ESORICS*.
[67] Dehai Zhao. Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: Vision-Based Linting of GUI Animation Effects Against Design-Dont Guidelines. In *Proc. ICSE*.