Automated Cross-Platform Inconsistency Detection for Mobile Apps

Mattia Fazzini
Georgia Institute of Technology, USA
mfazzini@cc.gatech.edu

Alessandro Orso
Georgia Institute of Technology, USA
orso@cc.gatech.edu

***Abstract*—Testing of Android apps is particularly challenging due to the fragmentation of the Android ecosystem in terms of both devices and operating system versions. Developers must in fact ensure not only that their apps behave as expected, but also that the apps' behavior is consistent across platforms. To support this task, we propose DIFFDROID, a new technique that helps developers automatically find cross-platform inconsistencies (CPIs) in mobile apps. DIFFDROID combines input generation and differential testing to compare the behavior of an app on different platforms and identify possible inconsistencies. Given an app, DIFFDROID (1) generates test inputs for the app, (2) runs the app with these inputs on a reference device and builds a model of the app behavior, (3) runs the app with the same inputs on a set of other devices, and (4) compares the behavior of the app on these different devices with the model of its behavior on the reference device. We implemented DIFFDROID and performed an evaluation of our approach on 5 benchmarks and over 130 platforms. Our results show that DIFFDROID can identify CPIs on real apps efficiently and with a limited number of false positives. DIFFDROID and our experimental infrastructure are publicly available.**

I. INTRODUCTION

Testing is a difficult and costly activity in general. When testing Android apps, the task is further complicated by the extensive fragmentation of the Android ecosystem. Android apps must be able to run on a myriad of devices and operating systems; developers are thus faced with the problem of ensuring not only that their apps behave as expected, but also that the behavior of the apps is consistent across platforms. Given the large number of possible hardware and software configurations in Android, this makes it extremely difficult and expensive to perform adequate testing of an app [1].

The problem of *cross-platform inconsistencies* (*CPIs*) is therefore prevalent in the Android environment, where users can observe failures and unexpected behaviors that are caused by differences between their platform and those on which the app they are using was tested [2].

To mitigate this problem, and help developers identify inconsistencies in behavior before an app is released, in this paper we propose DIFFDROID, an automated technique whose goal is to identify CPIs by combining input generation, user interface (UI) modeling, and differential testing. More precisely, DIFFDROID takes as input an app and performs four main steps. First, it automatically generates a large set of test inputs for the app. Second, it runs the app with these inputs on a reference device and builds a UI model of the app. Third, it runs the app against the same inputs on a large set of different platforms. Finally, DIFFDROID compares the UI models of the app on these different platforms with the UI model of the reference device and reports the differences identified, suitably ranked and visualized, to the app developer.

In order to assess the effectiveness of our approach, we implemented DIFFDROID in a tool and performed an empirical evaluation on 5 real-world apps and over 130 different platforms. In the evaluation, DIFFDROID was able to find 96 inconsistencies due to differences in the version of the Android system used or in the screen configuration. Overall, our results show that DIFFDROID can identify CPIs on real apps efficiently, while generating only a limited number of false positives. Our implementation of DIFFDROID and our experimental infrastructure are publicly available at http://www.cc.gatech.edu/~orso/software/diffdroid.

The main contributions of this paper are:

- A new technique that combines input generation, UI modeling, and differential testing to automatically identify cross-platform inconsistencies in the UI of Android apps.
- A publicly available implementation of our technique that can be used to replicate our experiments or build on and extend our approach.
- An empirical evaluation, performed on a large number of platforms, that provides initial evidence of the effectiveness of our approach.

II. MOTIVATING EXAMPLE

To motivate our work we provide an example from a real-world app called DAILY DOZEN [3]—a diet tracking app that has been downloaded more than $50,000$ times and reviewed by more than $1,000$ users. Figure 1 shows the `MainAcitivity` of the app running on a LG G3 device, while Figure 2 shows the same activity running on a LG Optimus L70 device. Users can use this activity to track their daily food intake by clicking on the displayed checkbox elements.

Figures 1 and 2 show a CPI for the app. Users can tick the checkbox element associated with the "Cruciferous Vegetables" label on a LG G3 device, but they cannot do the same on an LG Optimus L70, as that checkbox element is not visible when the app runs on such device. This inconsistency is caused by a bug in the layout file associated with the `MainAcitivity` and is revealed because of the different

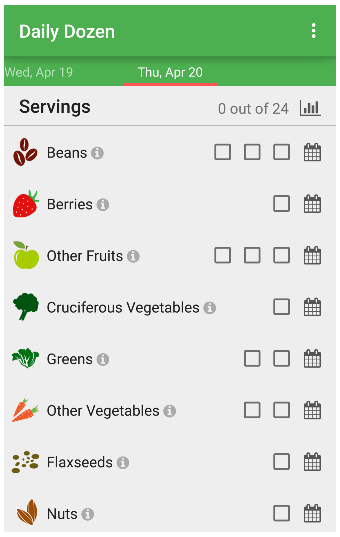

Fig. 1: DAILY DOZEN running on LG G3.

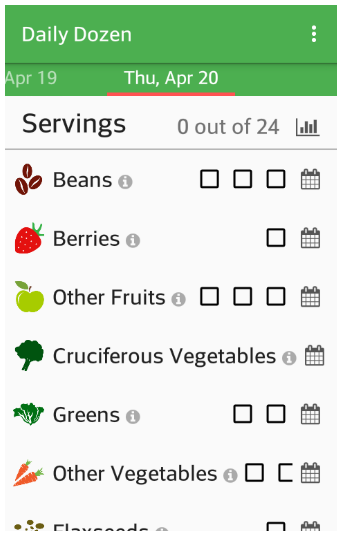

Fig. 2: DAILY DOZEN running on LG Optimus L70.

screen configurations (screen resolution and pixel density) of the two devices.

Bugs of this type can manifests in one of two ways: (1) the checkbox element is present on one device, but not on the other, or (2) the checkbox element is present on both devices, but its visual appearance is different. The checkbox element associated with the "Cruciferous Vegetables" label is an example of the former case. In this case, the difference can be visually perceived, but it can also be identified by comparing the UI hierarchies of the two devices. In fact, the UI hierarchy of the app running on the LG Optimus L70 device does not have a node representing the checkbox element, while such a node is present in the UI hierarchy of the app running on the LG G3 device. The rightmost checkbox element associated with the "Other Vegetables" label is an example of the latter case. In this case, the difference between the two devices can only be perceived visually, as both nodes are present the UI hierarchies of the two devices. These types of issues are far from rare because developers tend to use a limited set of devices (when not only one) during development and testing. In addition, these inconsistencies are hard to detect because this testing process tends to be mostly manual.

Figures 1 and 2 also highlight the challenges in finding CPIs on mobile devices. Because mobile devices can have different screen configurations, certain differences should not be classified as CPIs. For example, the "Nuts" label is displayed on the LG G3 device, while it is not displayed on the LG Optimus L70 device. This difference should not be considered a CPI: the label is part of a scrollable list, and the former device simply accommodates more list items due to its larger screen.

III. THE DIFFDROID TECHNIQUE

In this section, we present DIFFDROID, our technique for detecting CPIs on mobile devices. The basic idea behind DIFFDROID is to use differential testing to identify such inconsistencies. Figure 3 provides a high-level overview of our technique and shows its main phases. Given an app under test (*AUT*) and a reference device, in its *input generation phase*, DIFFDROID dynamically generates inputs with the goal of testing the app's functionality. Before providing the generated inputs to the app, the technique captures the UI state of the app by storing the tree of its UI hierarchy [4] and taking a screenshot of its appearance on the device. The technique logs UI hierarchy trees, screenshots, and generated inputs into the *trace*, which is the input to the following phase: the *test case encoding phase*. In this phase, our technique suitably analyzes the inputs together with UI hierarchy trees to generate a platform-independent test case. While doing so, the technique also creates a *UI model* of the app. The UI model is composed of a list of *window models*; and each window model contains a UI hierarchy tree and corresponding screenshot. We call this UI model the *reference UI model*, as it was generated using the reference device. The *test case execution phase* takes as input a set of test devices, executes the test case generated by the previous phase on the devices, and produces as output a *UI model* for each device (*test UI models*). Finally, in the *CPI analysis phase*, DIFFDROID performs a differential analysis to compare the reference UI model with the test UI models and generates a report that contains the detected CPIs. The *CPI report* is the output of our technique.

A. Input Generation

The input generation phase aims to test the functionality of the AUT on a reference device by dynamically generating inputs and providing them to the app. We describe this phase in Algorithm 1. The algorithm takes as inputs the reference device (rd), the AUT (AUT), and a timeout (T), and produces as output a trace ($trace$) that contains window models and generated inputs. We present the abstrax syntax of the trace produced by the algorithm in Figure 4.

The algorithm begins with an empty trace (line 2). It then starts the AUT (START) on the reference device (line 4) and subsequently enters its main loop, where it iterates until a timeout is reached.

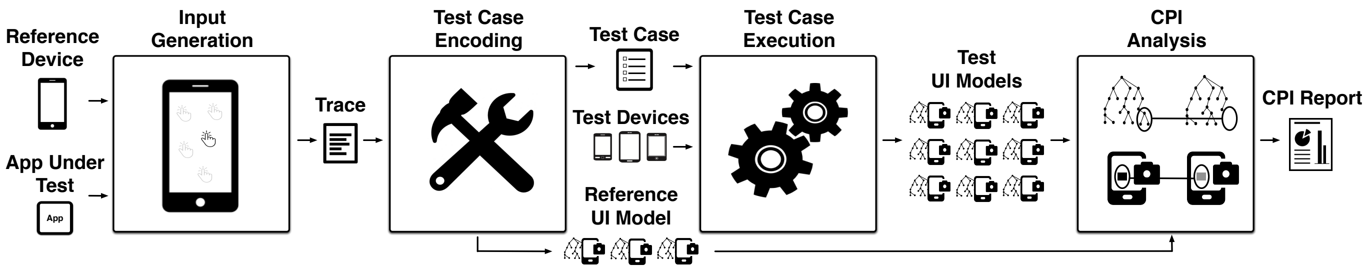

Fig. 3: High-level overview of the technique.

Algorithm 1: Input generation.

```
Input  : $rd$: reference device
         $AUT$: application under test
         $T$: input generation timeout
Output: $trace$: window models and generated inputs on reference device
1  begin
2      $trace = \emptyset$
3      $t$ = GET-CURRENT-TIME()
4      START($rd$, $AUT$)
5      while $T <$ GET-ELAPSED-TIME($t$) do
6          $root$ = GET-ROOT($rd$)
7          $tree$ = TRAVERSE($root$)
8          $screenshot$ = GET-SCREENSHOT($rd$)
9          $trace$.ADD(WINDOW-MODEL($tree$, $screenshot$))
10         $input$ = GENERATE-INPUT($w_1$, KEY, $w_2$, SYSTEM, $w_3$, TOUCH)
11         INJECT($rd$, $input$)
12         $trace$.ADD($input$)
13     return $trace$
```

In the first part of the loop iteration (lines 6-9), the algorithm retrieves the root node of the UI hierarchy (GET-ROOT) and traverses the UI hierarchy (TRAVERSE) to build a tree representation of such hierarchy. Each node in the tree is characterized by the following set of properties: $node\text{-}id$, $node\text{-}type$, $left$, $right$, top, $bottom$, $text$, $checked$, $enabled$, $focused$, $selected$, $clickable$, $checkable$, $focusable$, $scrollable$, and $long\text{-}clickable$. We selected this set of properties because it is the minimal set of properties that allows the technique to best differentiate nodes in the UI hierarchy (see Section III-D). After building a tree representation of the UI hierarchy, the algorithm captures a screenshot of the AUT (GET-SCREENSHOT). The algorithm then pairs the tree representation of the UI hierarchy with the screenshot of the AUT to define the current window model and adds the model to the trace.

In the second part of the loop iteration (lines 10-12), the algorithm generates an input (GENERATE-INPUT), provides the input to the AUT (INJECT), and adds the input to the trace. DIFFDROID generates three types of inputs (KEY, SYSTEM, and TOUCH) using a weighted random distribution, as done in related work [5]. (It is worth noting that the technique would also work with a different dynamic input generation approach, such as [6], [7], [8], [9], [10], [11], [12], [13].) Key inputs are characterized by the value of the key they are representing; system inputs express a change in the orientation of the device or data used to transfer control between components of the AUT; and touch inputs represent clicks or gestures on the device. Note that our technique does not currently remove inputs that do not affect the state of the AUT, but they could be discarded using an approach based on delta debugging [14].

$trace\text{-}def$	$::=$ ***trace*** $items$
$items$	$::=$ $window\text{-}model\text{-}def$ $input\text{-}def$
	$::=$ $\mid$ $window\text{-}model\text{-}def$ $input\text{-}def$ $items$
$window\text{-}model\text{-}def$	$::=$ ***window-model*** $tree\text{-}def$ $screenshot\text{-}def$
$tree\text{-}def$	$::=$ ***tree*** $root\text{-}reference\text{-}id^{*}$ $nodes$
$nodes$	$::=$ $node\text{-}def \mid node\text{-}def$ $nodes$
$node\text{-}def$	$::=$ ***node*** $reference\text{-}id^{*}$ $node\text{-}props$ $children\text{-}ids$
$node\text{-}props$	$::=$ $node\text{-}id^{\dagger}$ $node\text{-}type^{\dagger}$ $text^{\dagger}$ $checkable^{\ddagger}$
	$::=$ $clickable^{\ddagger}$ $focusable^{\ddagger}$ $scrollable^{\ddagger}$
	$::=$ $long\text{-}clickable^{\ddagger}$ $checked^{\ddagger}$ $enabled^{\ddagger}$ $focused^{\ddagger}$
	$::=$ $selected^{\ddagger}$ $left^{*}$ $right^{*}$ top^{*} $bottom^{*}$
$children\text{-}ids$	$::=$ $\mid$ $reference\text{-}id^{*}$ $children\text{-}ids$
$screenshot\text{-}def$	$::=$ ***screenshot*** $image$
$input\text{-}def$	$::=$ ***input*** $input\text{-}type$
$input\text{-}type$	$::=$ $key\text{-}input\text{-}def \mid system\text{-}input\text{-}def$
	$::=$ $\mid$ $touch\text{-}input\text{-}def$
$key\text{-}input\text{-}def$	$::=$ ***key*** $key\text{-}value^{\dagger}$
$system\text{-}input\text{-}def$	$::=$ ***system*** $system\text{-}input\text{-}type$ $system\text{-}input\text{-}props$
$system\text{-}input\text{-}type$	$::=$ ***rotate*** $\mid$ ***data***
$system\text{-}input\text{-}props$	$::=$ $exprs$
$touch\text{-}input\text{-}def$	$::=$ ***touch*** $\mid$ $coords$
$coords$	$::=$ $x\text{-}coord^{*}$ $y\text{-}coord^{*}$ $pointer\text{-}id^{*}$
	$::=$ $\mid$ $x\text{-}coord^{*}$ $y\text{-}coord^{*}$ $pointer\text{-}id^{*} coords$
$exprs$	$::=$ $expr \mid expr$ $exprs$
$expr$	$::=$ $boolean \mid number \mid string$

Fig. 4: Abstract syntax of the generated trace. "*" indicates that the value is a number, "†" indicates that the value is a string, and "‡" indicates that the value is a boolean.

B. Test Case Encoding

The test case encoding phase aims to generate a platform-independent test case based on the content of the trace created by the input generation phase. We present this phase in Algorithm 2.

The algorithm takes as input the trace ($trace$) generated by the previous phase of the technique, and it produces two outputs: a platform independent test case (tc) and a UI model of the reference decvice ($RUIM$). The algorithm begins with an empty test case (line 2) and an empty UI model (line 3). It then processes the content of the trace in its main loop (lines 4-17).

In the first part of the loop iteration (lines 5-14), the algorithm processes window models. If the currently processed window model has the same tree representation (SAME-TREE) and the same screenshot (SAME-IMAGE) of a window model already added into the reference UI model (lines 7- 10), the model is discarded as superfluous. Function SAME-TREE performs a breadth-first traversal of two trees and compares the value of the properties of the traversed nodes. If the two trees have different structure, or if their nodes have different properties, the algorithm considers the two trees and corresponding window models to be different. Function

Algorithm 2: Test case encoding.

```
Input  : $trace$: window models and generated inputs on reference device
Output: $tc$: test case
        $RUIM$: UI model of the reference device
1  begin
2      $tc$ = $\emptyset$
3      $RUIM$ = $\emptyset$
4      foreach $item \in trace$ do
5          if $item \equiv$ WINDOW-MODEL then
6              $newModel$ = TRUE
7              foreach $wModel \in RUIM$ do
8                  if SAME-TREE($wModel$.GET-TREE(),
                     $item$.GET-TREE()) and
9                  SAME-IMAGE($wModel$.GET-SCREENSHOT(),
                     $item$.GET-SCREENSHOT()) then
10                     $newModel$= FALSE
11             if $newModel$ == TRUE then
12                 $RUIM$.ADD($item$)
13                 $stmt$ =
                   GENERATE-WINDOW-MODEL-STATEMENT($item$)
14                 $tc$.ADD($stmt$)
15         else
16             $stmt$ = GENERATE-INPUT-STATEMENT($item$)
17             $tc$.ADD($stmt$)
18     return $tc$, $RUIM$
```

SAME-IMAGE compares two screenshots using the complex wavelet structural similarity (CW-SSIM) index [15], which can range between zero (different images) and one (similar images). We motivate the use of this index to compute image similarity in Section III-D. If two screenshots do not have CW-SSIM index equal to one, we consider the corresponding window models to be different. If a window model is not redundant (lines 11-14), the algorithm adds the window model to the reference UI model. It also generates a test case statement (GENERATE-WINDOW-MODEL-STATEMENT) that builds a tree representation of the UI hierarchy and takes a screenshot of the device.

In the second part of the loop iteration (lines 16-17), the algorithm generates a platform-independent statement (GENERATE-INPUT-STATEMENT) that replicates the action of the input in the trace; it then adds the statement to the test case. We define generated statements as being platform-independent because they can run on any device independently from the operating system version and the device configuration (*e.g.,* screen size). The algorithm creates platform-independent statements following the principles presented in our previous work [16]. Platform-independent test cases allow our technique to collect UI models on many different devices, thus increasing the likelihood of identifying CPIs.

C. Test Case Execution

The test case execution phase aims to collect UI models from a set of test devices. This phase takes as inputs the test case generated by the previous phase of the technique and a set of test devices, and it executes the test case on the set of test devices. The execution is driven by two types of statements: WINDOW-MODEL-STATEMENT and INPUT-STATEMENT. Statements of the former type traverse the UI hierarchy of the AUT to build a tree representation of such hierarchy and capture a screenshot of the AUT. The tree and the screenshot are paired together to form a window model of the test device; this window model is then added to the UI model of the test device. Statements of the latter type provide an input to the AUT. These inputs are meant to exercise the AUT on the test device in the same way it was exercised in the generation phase.

The output of this phase is a mapping between test devices and corresponding UI models. Generation of UI models for test devices is amenable to parallelization, as the computation of a UI model for one device is completely independent from the computation of the UI model for a different device.

D. CPI Analysis

The CPI analysis phase aims to identify CPIs in the AUT and is the core of our technique. We present this phase in Algorithm 3. Algorithm 3 takes as inputs the reference UI model ($RUIM$) and the map of UI models of test devices ($TUIMMap$) generated by the previous phase. The algorithm produces as output a report that lists the identified CPIs ($CPIReport$); this report is also the overall output of DIFFDROID.

The algorithm begins with an empty CPI report (line 2), iterates over each window model ($rdModel$) in the reference UI model (lines 3-44), and compares the window model at hand to the corresponding window model ($tdModel$) in all test UI models (lines 5-44). The comparison between a reference window model and a test window model is divided into two steps. The first step (lines 6-33) matches nodes from the tree representing the UI hierarchy of the reference device ($rdTree$) to nodes from the tree representing the UI hierarchy of the test device ($tdTree$). The second step (lines 34-44) compares the visual representation (image) of matched nodes. The first step can detect *structural CPIs*, which consist of missing or additional nodes. The second step can detect *visual CPIs*, which consist of nodes with different visual representations. Considering the motivating example of Section II, the checkbox element associated with the "Cruciferous Vegetables" label is an example of a structural CPI, while the rightmost checkbox element associated with the "Other Vegetables" label is an example of a visual CPI.

The node mapping process begins by initializing the mapping ($nodeMappingMap$) between nodes in the reference tree and nodes in test tree to the empty value (line 10). For each node ($rdNode$) in the reference tree, the algorithm then computes a node similarity value ($nodeSim$) between the reference node and each node ($tdNode$) in the test tree using function COMPUTE-STRUCTURAL-SIMILARITY. This function computes a value between zero and one that represents the structural similarity of two nodes (see Section III-D1) If the similarity value is greater or equal than a threshold α, the algorithm stores the similarity value, together with the test node, in a list ($mappedNodeList$). Threshold α is used to avoid matching nodes that are too dissimilar. The choice of the value of α is related to function COMPUTE-STRUCTURAL-SIMILARITY and we describe it in Section III-D1. When the algorithm has processed all nodes in the test tree, it stores the mapping between the reference node and the computed list into $nodeMappingMap$ (line 19).

Algorithm 3: CPI detection analysis.

```
Input : $RUIM$: UI model of reference device
        $TUIMMap$: map of UI models of test devices
Output: $CPIReport$: set of cross device inconsistencies
1  begin
2      $CPIReport$ = $\emptyset$
3      for $i = 0;\ i <\ RUIM.length;\ \ ++i$ do
4          $rdModel$ = $RUIM$.GET($i$)
5          foreach $td \in TUIMMap$.KEY-SET() do
6              $//node\ mapping$
7              $tdModel$ = $TUIMMap[td]$.GET($i$)
8              $rdTree$ = $rdModel$.GET-TREE()
9              $tdTree$ = $tdModel$.GET-TREE()
10             $nodeMappingMap$ = $\emptyset$
11             foreach $rdNode \in rdTree$ do
12                 $mappedNodeList$ = $\emptyset$
13                 foreach $tdNode \in tdTree$ do
14                     $nodeSim$ =
15                     COMPUTE-STRUCTURAL-SIMILARITY($rdTree$,
                         $rdNode$, $tdTree$, $tdNode$)
16                     if $nodeSim \geq \alpha$ then
17                         $mappedNodeList$.ADD(MAPPING(
18                         $nodeSim$, $tdNode$))
19                 $nodeMappingMap[rdNode]$ = $mappedNodeList$

20             FIND-BEST-MAPPING($nodeMappingMap$)
21             $//structural\ comparison$
22             foreach $rdNode \in rdTree$ do
23                 if $nodeMappingMap[rdNode]$ == $\emptyset$ and
                     ¬WITHIN-DYNAMICALLY-SIZED-
                     ELEMENT($rdNode$, $nodeMappingMap$)
                     then
24                     $CPIReport$.ADD(STRUCTURAL-CPI($td$,
                         $rdNode$))
25                     $nodeMappingMap$.REMOVE($rdNode$)
26             foreach $tdNode \in tdTree$ do
27                 $mapped$ = FALSE
28                 foreach $rdNode \in rdTree$ do
29                     if $nodeMappingMap[rdNode]$.
30                     CONTAINS($tdNode$) then
31                         $mapped$ = TRUE
32                 if $mapped$ ==FALSE and ¬WITHIN-DYNAMICALLY-
                     SIZED-ELEMENT($tdNode$, $nodeMappingMap$)
                     then
33                     $CPIReport$.ADD(STRUCTURAL-CPI($td$,
                         $tdNode$))
34             $//visual\ comparison$
35             $rdScreenshot$ = $rdModel$.GET-SCREENSHOT()
36             $tdScreenshot$ = $tdModel$.GET-SCREENSHOT()
37             foreach $rdNode \in nodeMappingMap$.KEY-SET() do
38                 $tdNode$ = $nodeMappingMap[rdNode]$.REMOVE(0)

39                 $rdNodeImage$ = $rdScreenshot$.CROP($rdNode$)
40                 $tdNodeImage$ = $tdScreenshot$.CROP($tdNode$)
41                 $isInconsistency$ =
42                 COMPUTE-IMAGE-SIMILARITY($rdNodeImage$,
                     $tdNodeImage$)
43                 if $isInconsistency$ then
44                     $CPIReport$.ADD(VISUAL-CPI($td$,
                         $rdNodeImage$, $tdNodeImage$))
45     RANK($CPIReport$)
46     return $CPIReport$
```

Once $mappedNodeList$ is computed for all reference nodes, the algorithm computes the optimal mapping between reference nodes and test nodes using function FIND-BEST-MAPPING (line 20). For every reference node, this function sorts the elements in the $mappedNodeList$ in descending order based on their node similarity value. The function then finds the $mappedNodeList$ containing the element with the highest node similarity value, maps the reference node associated with the $mappedNodeList$ to the test node associated with the element of the list, and marks the reference node as processed. Finally, the function removes all occurrences of the test node from the $mappedNodeList$ associated with other reference nodes. This process continues until all reference nodes are processed.

After finding the best mapping between nodes in the reference tree and nodes in the test tree (line 20), the algorithm analyzes the mapping to find structural CPIs. The algorithm iterates over each node in the reference tree (lines 21-25) to find reference nodes that do not have a mapping to a node in the test tree. If such a node is found, it means that the node is present in the AUT while running on the reference device, but it is not present in the AUT while running on the test device. A challenging aspect for the classification of the missing nodes is given by the fragmentation of the Android ecosystem. Devices come with different screen configurations, and it could be normal that two devices have a different number of nodes in their UI hierarchies.

Consider again the motivating example of Section II, in which the MAINACITVITY displays a list of servings. When the app is running on the LG G3, the device displays eight elements. When the app is running on the LG Optimus L70, conversely, the device displays only seven elements. In this case, the eighth element of the list should not be classified as an inconsistency because the Android system does not represent a node in the UI hierarchy if the node is not visible. For this reason, the algorithm further analyzes the node using function WITHIN-DYNAMICALLY-SIZED-ELEMENT to determine whether or not the node should be reported as inconsistency. If the node does not have an ancestor that is scrollable, the node is reported as an inconsistency. Otherwise, if the node (1) has a scrollable ancestor and (2) has its preceding or subsequent sibling (depending on the position of the node in the tree) that is matched to a node that is visibile and it is not at the end of the dynamically sized element, then the node is also reported as an inconsistency. In all other cases, the node is not reported as an inconsistency. In case function WITHIN-DYNAMICALLY-SIZED-ELEMENT confirms that the node is an inconsistency, the node is added to the CPI report as a structural CPI. Similarly, the algorithm iterates over nodes in the test tree (lines 26-33) to find test nodes that do not have a mapping to a node in the reference tree. If such a node is found, and the node is not part of a dynamically sized element, the algorithm reports it as a structural CPI.

After these steps, the algorithm visually compares mapped nodes (lines 34-44). This part of the algorithm starts by retrieving (GET-SCREENSHOT) the screenshots of the reference and test devices from their window models. Then, for each node in the reference tree, the algorithm retrieves the test node mapped to it and creates two images ($rdNodeImage$ and $tdNodeImage$) from the two screenshots ($rdScreenshot$ and $tdScreenshot$) using function CROP. At this point, the algorithm compares the two images using function COMPUTE-IMAGE-SIMILARITY, which uses a decision tree classifier to recognize inconsistencies. We describe the decision tree classifier we use in Section III-D2. The function uses similar principles as the ones we discussed in the context of function WITHIN-DYNAMICALLY-SIZED-ELEMENT: it does not report as inconsistencies nodes that are partially visible because they are part of a dynamically sized element. If the classifier identifies an inconsistency, the algorithm reports that the reference node and the test node have a visual inconsistency (line 44).

The algorithm then ranks (RANK, in line 45) inconsistencies according to the following principles: (1) structural inconsistencies are ranked at the top, (2) visual inconsistencies that affect all devices with the same characteristics (e.g., version of the Android operating system) are ranked next, and (3) the remaining inconsistencies are listed last. Finally the algorithm returns the CPI report (line 46), which is the output of the technique.

1) Node Structural Similarity: Function COMPUTE-STRUCTURAL-SIMILARITY in Algorithm 3 computes the structural similarity between two nodes. The inputs to the function are a reference tree ($rdTree$), a reference node ($rdNode$), a test tree ($tdTree$), and a test node ($tdNode$). The output of the function is a value between zero and one ($nodeSim$) that indicates the similarity between the reference node and the test node. This function is necessary, as nodes in the tree are not required to have identifiers, and different versions of the operating system can use different node types to represent the same node.

The function starts by comparing the identifiers of the reference and test nodes. If the identifiers are the same, and they are unique in both reference and test trees, the function sets the node similarity value to one and returns it. In this case, the function sets the similarity value to its highest value because the identifier is a property manually defined by the developer that is meant to uniquely identify nodes in the tree.

If identifiers are not unique, or they are different, the function checks the position of the two nodes in the tree by comparing their XPaths (using the path expression from the root of the tree). If the nodes have the same XPath, the function returns one as their similarity. If the path expressions of the two nodes differ in more than one path component, the function returns zero as similarity value (to avoid matches between nodes that are too distant in the tree). If only one path component is different, which may be due to small differences in the tree representation of the test devices, the function computes the similarity value based on the following properties of the nodes: $checkable$, $clickable$, $focusable$, $scrollable$, $text$, $checked$, $selected$, $long\text{-}clickable$, $enabled$, and $focused$. The similarity value, in this case, is given by the number of matching properties divided by the number of properties. For the evaluation of DIFFDROID we chose 0.9 as the value of α in Algorithm 3 to indicate that we do not want to match nodes having more than one property value that differs.

2) Decision Tree Classifier: Function COMPUTE-IMAGE-SIMILARITY in Algorithm 3 uses a decision tree classifier [17] to compute whether the visual representation of two nodes should be reported as a CPI. The decision tree classifier algorithm creates a model that predicts the value of a target variable based on a set of input variables. The algorithm learns the model using a set of training data. In our context, the training data corresponds to images of nodes from the UI hierarchies of apps running on different devices. The training set must also include a set of images exhibiting CPIs. After building the model, function COMPUTE-IMAGE-SIMILARITY follows the set of decisions in the model to predict the target variable. DIFFDROID uses the following variables as inputs to the the classifier:

Complex-Wavelet Structural Similarity Index. Our technique uses the Complex-Wavelet Structural Similarity (CW-SSIM) index [15] to compare the structural similarity of the content of two images. CW-SSIM is an image similarity metric robust to small rotations and translations in the images being compared. This characteristic makes the metric especially suitable in our context because different devices have different screen configurations; therefore, the visual representation of two nodes may present minor differences that should not be reported as CPIs.

Earth's Mover Distance of Color Histograms. DIFFDROID uses the Earth's Mover Distance [18] (EMD) of the color histograms of two images to compare the color composition of the images. EMD is a measure of the distance between two distributions and, intuitively, consists of the minimal cost that must be paid to transform one color distribution into the other. We decided to use this metric to take into account the fact that two images may have similar structure but display different colors.

Relative Ratio Change. Our technique uses the relative ratio change to assess whether two images differ significantly in their proportions. The relative ratio change is defined as $RRC = ((w_t/h_t) - (w_r/h_r))/(w_r/h_r)$, where w_t and h_t are the width and height of the test node, while w_r and h_r are the width and height of the reference node. This value allows DIFFDROID to identify nodes whose ratio is altered as a consequence of the placement of other nodes.

Optical Character Recognition Output. DIFFDROID uses the output of optical character recognition [19] (OCR) to assess whether two images display the same text. The classifier takes as input the value of the comparison (equal/not equal). We decided to use the output of OCR because nodes might have the same text in their tree representation but might display the text differently.

The target variable predicted by the classifier indicates whether the visual representation of two nodes should be reported as an inconsistency.

IV. IMPLEMENTATION

DIFFDROID's input generation module is built on top of Monkey [5], an input generator for Android apps that generates pseudo-random sequences of user and system events. We extended Monkey to encode generated inputs into the trace and to save UI hierarchies together with visual representations (screenshots) of the app being tested. The tool is able to inspect the screen content of the app using UiAutomation [20], which is a special accessibility service of the Android platform. The input generation module can run on any device without modifying the Android system.

TABLE I: Benchmarks used in the empirical evaluation.

ID	*Name*	*Category*	*Version*	*LOC (#K)*
A1	BuildmLearn	Education	2.5.0	23.6
A2	Daily Dozen	Health	10.3	6.3
A3	Kitchen Timer	Tools	1.1.6	4.3
A4	Outlay	Finance	1.1.3	8
A5	Translation Studio	Books	9.0	51.2

The test case encoding module uses the JavaPoet 1.8 library [21] to generate the source code of test cases. The implementation of DiffDroid extends the Espresso [22] framework to generate and execute test cases. The Espresso framework synchronizes test operations with the app being tested by waiting for UI events to be handled and for default instances of `AsyncTask` (computation that runs on a background thread) to complete. This capability allows DiffDroid to execute the same test case on different devices even when devices have different hardware configurations, which could lead to different timings in the execution of an app. However, there are cases in which apps perform background operations using non-standard means (e.g., direct creation and management of threads). The developer needs to manually handle such cases. For the benchmarks of Section V, this additional task was not necessary. The test case encoding module uses the implementation of CW-SSIM offered by pyssim [23] to compare screenshots taken by the input generation module and minimize the number of UI hierarchies and screenshots generated during test execution.

The test case execution module leverages the AWS Device Farm [24] to execute test cases on real devices. The module uses the AWS Command Line Interface (CLI) to automate the process of running test cases and retrieving execution artifacts (UI hierarchies and screenshots).

Finally, the CPI analysis module generates the decision tree classifier used to detect CPIs by leveraging the Weka data mining framework [25]. The CW-SSIM index used in the classifier is computed using pyssim, the EMD value is computed using the OpenCV [26] library, and the character recognition task is performed used Tesseract OCR [27] engine.

V. Empirical Evaluation

To determine the practicality and effectiveness of our technique, we performed an empirical evaluation of DiffDroid on a set of real-world apps and targeted the following research questions:

- **RQ1:** Can DiffDroid detect cross-platform inconsistencies in mobile applications while reporting a limited number of false positives?
- **RQ2:** What is the cost of running DiffDroid?
- **RQ3:** Are there similarities among devices exhibiting CPIs?

A. Experimental Benchmarks and Setup

For the empirical evaluation, we used a set of real-world Android apps. More specifically, we selected five open-source apps from GitHub [28]. We used open-source apps because the testing environment (Espresso) used in the implementation of our technique requires the source code of an app to build and run test cases for it. Our technique could be directly applied to app executables by changing testing framework.

TABLE II: Number of test devices divided by resolution and version of the operating system.

	Android Version					
Resolution	*19*	*21*	*22*	*23*	*24*	*25*
720 x 1280	18	2	6	2	0	0
768 x 1280	1	0	0	0	0	0
1080 x 1920	33	12	5	6	0	1
1440 x 2560	8	7	8	13	5	1
480 x 800	8	0	0	0	0	0
540 x 960	5	1	3	0	0	0
480 x 854	1	0	1	0	0	0

We selected apps based on three parameters: (1) presence of at least one known UI-based CPI in the app, (2) self-containment, and (3) diversity. In order to find apps containing at least one known CPI, we searched GitHub's tracker system for the following keywords: "android not clickable", "android cut off", and "android missing button". We used these keywords instead of more generic keywords, such as "android compatibility issue", to eliminate results that were not UI issues, which are out of scope for our technique. From the search results, we removed issues that did not correspond to Android apps and issues that we could not reproduce. Finally, we selected apps from different categories to have a diverse corpus of benchmarks, while prioritizing apps for which we did not have to build extensive stubs. Table I provides a summary description of the apps considered. Columns *Name*, *Category*, *Version*, and *LOC* report the name, category, version, and number of lines of code for an app.

The analysis performed by DiffDroid relies on the use of a reference device. We selected an LG G3 running Android *22* as reference device for the empirical evaluation because we had the device, and it did not exhibit any of the CPIs already known in the benchmarks. To compute the results of Section V-B, we executed the input generation phase of DiffDroid on the reference device with a timeout of 10 minutes. We chose this value because in previous work [29] we found that a set of dynamic input generation tools for Android apps hit their maximum coverage within 10 minutes of execution.

DiffDroid's analysis also requires a set of test devices. We used the AWS Device Farm [24] for this purpose. The AWS Device Farm is an app testing service provided by Amazon that allows to run tests on real mobile devices. Table II reports the number of devices used in the empirical evaluation grouped by resolution (*Resolution*) and version of the operating system (*Android Version*). The versions of the operating system were the ones available to us and supported by the technologies used for the implementation of DiffDroid. The total number of devices used was 147.

Finally, DiffDroid uses a decision tree classifier to recognize CPIs. We trained the classifier using the following

TABLE III: Results of running DiffDroid. For each benchmark considered: *D(#)* = number of test devices; *WM(#)* = number of window models; *N_R(#)* = number of nodes in UI hierarchy trees of the reference device; *N_T(#)* = average number of nodes in UI hierarchy trees per test devices; *CPI_S(#)* = number of structural CPIs; *CPI_F(#)* = number of functional CPIs; *CPI_V(#)* = number of CPIs related to changes in the version of the Android system; *CPI_C(#)* = number of cosmetic CPIs; *FP(#)* = false positives reported by the technique.

ID	*D(#)*	*WM(#)*	*N_R(#)*	*N_T(#)*	*CPI_S(#)*	*CPI_F(#)*	*CPI_V(#)*	*CPI_C(#)*	*FP(#)*
A1	135	19	491	465.2	0	2	7	14	1
A2	138	22	1199	1174.8	2	0	0	22	4
A3	129	13	286	276.6	2	3	0	2	1
A4	125	14	505	481.8	0	1	0	17	2
A5	136	17	486	466	2	3	0	19	8

TABLE IV: Cost of running DiffDroid. *T_G(s)* = time to encode inputs and compute the reference UI model; *T_E(s)* = average test execution time per device; *T_S(ms)* = average time to compare UI hierarchies per device; *T_{CW}(s)* = average CW-SSIM computation time per device; *T_{EMD}(s)* = average EMD computation time per device; and *T_{OCR}(s)* = average OCR computation time per device.

ID	*T_G(s)*	*T_E(s)*	*T_S(ms)*	*T_{CW}(s)*	*T_{EMD}(s)*	*T_{OCR}(s)*
A1	2080	474.9	139	443.8	14.8	172
A2	1102	512.2	581	575.7	50.8	586.5
A3	2772	329.9	115	246.4	11.9	134.5
A4	851	651.9	210	275	15.1	103.7
A5	1803	376	166	217.5	14.1	153.5

procedure. First, we selected one device from each category (combination of resolution and Android version) in Table II and used this set of devices to compute the training set. We then collected CW-SSIM index, EMD value, OCR output, and relative ratio change (inputs to the classifier) for all the nodes in the view hierarchies showing the known UI-based CPIs. (These nodes are not included in the results of the evaluation.) This procedure produced $5,558$ entries on which to train the classifier. We labeled the entries either `true` or `false` based on whether they represented CPIs or not, respectively. We labeled entries by looking at their visual representation and labeled as `true` entries such that, compared to the reference entry, (1) differed in their content structure, (2) differed in terms of color, (3) differed in terms of visibility, (4) visualized a different text, and (5) had a different aspect ratio. Following these guidelines, we labeled 282 entries. We used the Weka data mining framework [25] to generate a C4.5 decision tree classifier. The framework created a classifier of size 33 with 17 leaves in 0.09 seconds. We evaluated the classifier using 10-fold cross validation, resulting in a precision of 0.978 and recall of 0.957. The CPI analysis phase was performed on a workstation with 64GB of memory, one Intel Xeon i7-6700K Skylake 4.0GHz processor, running Ubuntu 14.04.

B. Results

*1) **RQ1**:* To answer RQ1, we applied our technique to the experimental benchmarks. Table III reports the results of the evaluation.

The first part of Table III (columns *D(#)*, *WM(#)*, *N_R(#)*, and *N_T(#)*) provides a picture of the scale of the analysis. For each benchmark: column *D(#)* reports the number of test devices used in the test execution phase; column *WM(#)* provides the number of window models generated by the test case encoding phase; *N_R(#)* is the number of nodes in the UI hierarchy trees for the reference device; and *N_T(#)* is the average number of nodes in the UI hierarchies for the test devices. The number of devices used for each benchmark differs because, when running the evaluation, certain devices were not available in the AWS Device Farm. For unavailable devices, we attempted to run test cases three times before moving forward. Columns *N_R(#)* and *N_T(#)* differ for two reasons: the app might contain a structural CPI, and different devices display a different number of nodes for dynamically sized elements (e.g., list containers). The total number of nodes analyzed across all benchmarks and devices is $387,174$.

The second part of Table III (columns *CPI_S(#)*, *CPI_F(#)*, *CPI_V(#)*, and *CPI_C(#)*) presents the CPIs reported by DiffDroid. We analyzed CPIs reported by our technique and classified them in four categories: inconsistencies in the UI hierarchy tree that affect the functionality of the app (*structural CPIs*, *CPI_S(#)*); inconsistencies in the visual representation of a node that affect the functionality of the app (*functional CPIs*, *CPI_F(#)*); inconsistencies generated by the version of the Android system used to run the benchmark (*version CPIs*, *CPI_V(#)*);and inconsistencies in the visual representation of a node that do not affect the functionality of the app because the user can infer their meaning given the context in which they are visualized (*cosmetic CPIs*, *CPI_C(#)*). Structural CPIs correspond to the inconsistencies with the same name we discussed in Section III-D, while functional CPIs, version CPIs, and cosmetic CPIs correspond to the visual CPIs that we also discussed in Section III-D. The results presented in this section are deterministic, as the classification part of the technique is itself deterministic. In addition, we also classified CPIs reported by DiffDroid that did not correspond to an inconsistency as false positives (*FP(#)*). Finally, we randomly selected 5 nodes in each benchmark on all test devices ($3,315$ total), checked for possible false negatives, and did not find any.

Our technique found CPIs in all the benchmarks analyzed: 6 structural CPIs, 9 functional CPIs, 7 version CPIs, and 74 cosmetic CPIs. We now provide an example from each category to better illustrate the identified CPIs and how we classified them.

Translation Studio is a translation app. In the registration form of the app, there is an icon that, when clicked, presents a privacy note to the user. However, on certain devices, the icon is not present, and the user will miss the opportunity to read the privacy note. On these devices, the node of the icon is not present in the UI hierarchy tree of the app, and DiffDroid reports this difference as an inconsistency (structural CPI).

Kitchen Timer offers a timer functionality. The app can be used to start and stop three timers. If one of the timers is started, the label of the timer increases in size, moving the button to stop the timer at the bottom of the screen. On certain devices, the size of the button becomes small enough to prevent users from stopping the timer. In these devices, the node representing the button is present in the UI hiearchy tree, but its visual appearance differs from that of the corresponding node on the reference device. DiffDroid reports this difference as an inconsistency (functional CPI).

BuildmLearn is an app that assists users in developing Android apps. The app has a menu that can be used to navigate the app. The color of the background of the items in the menu is different when the app is running on devices using Android version 19. In these devices, the color of the background is similar to the color of the text of the menu items, making difficult to read the entries in the menu. DiffDroid reports this difference as an inconsistency (version CPI).

Outlay helps users track their expenses. Users can enter their expenses using a numpad. On certain devices, only roughly one fourth of the numbers is visible. Also in this case, DiffDroid reports this difference as an inconsistency (cosmetic CPI).

We looked at the nature of the structural, functional, and cosmetic CPIs mentioned above, and discovered tha they can be fixed by changing properties of corresponding elements in the layout files for the apps. We reported the issues found, and their possible solutions, to the developers of the apps involved.

DiffDroid also reported 16 false positives for the five benchmarks we considered. The false positives reported can be grouped into two categories. The first category (14 false positives) includes nodes that display text with additional spacing at the end. This behavior causes test nodes to have a big relative ratio change, leading the classifier to report them as inconsistencies. To address this issue, we plan to leverage OCR to recognize text boundaries and compute relative ratio changes based on such boundaries. The second category (2 false positives) includes test nodes whose image differed from the reference one in the color distribution, but the difference is such that it cannot be perceived by the human eye. This characteristic resulted in a significantly high EMD value, leading the classifier to report these nodes as inconsistencies. To reduce the number of this kind of false positives, we plan to investigate how the number of bins in the computation of the EMD value affects false positives and performance.

Overall, we feel that the current number of false positives generated by DiffDroid is acceptable. (Moreover, they can be further reduced through improvements of the technique.) We therefore believe that the results presented in this section provide initial evidence that DiffDroid can detect CPIs in mobile applications while reporting a limited number of false positives.

*2) **RQ2**:* To answer RQ2, we measured the time taken by each phase of the technique to process the experimental benchmarks. Table IV summarizes the results and reports: the time required to encode dynamically generated inputs as a test case while computing the UI model of the reference device ($T_G(s)$); the average test case execution time per device ($T_E(s)$); the average time required to compare reference UI hierarchies with test UI hierarchies per device ($T_S(ms)$); the average time required to compute CW-SSIM indexes per device ($T_{CW}(s)$); the average time required to compute EMD values per device ($T_{EMD}(s)$); and the average time required to extract text with OCR per device ($T_{OCR}(s)$).

The values in column $T_G(s)$ show that the cost to compute the UI model based on the dynamically generated inputs is not low (but still acceptable), which validates our choice of not performing this task during the input generation phase. The average time to execute test cases is less than the time taken to generate inputs. This happens mainly because test cases are saving significantly less UI hierarchies and screenshots. (The only exception is A4, for which we had to add a 60sec sleep time to make sure the test would go past the login screen on the test devices.) In the worst case (A4), test cases took a total of $1,358$ minutes to execute (D from Table III times T_E from Table IV). During the evaluation we took advantage of the fact that this task is highly parallelizable and executed test cases on 10 devices at the time, thus reducing the cost roughly by an order of magnitude.

Finally, the last part of Table IV shows that the time required to compare reference UI hierarchies to test UI hierarchies is negligible compared to the time to compute values for the features of the classifier. In the worst case (A2), the CPI analysis phase took $2,791$ minutes to complete (D from Table III times the sum of T_S, T_{CW}, T_{EMD}, and T_{OCR} from Table IV). This task is also highly parallelizable. and when running the evaluation we analyzed eight devices at a time. Finally, the most expensive part of the CPI analysis phase consists of the computation of CW-SSIM indexes, which across all apps and all devices took 351.7 seconds on average.

Based on these results, we can conclude that the analysis performed by DiffDroid can run overnight, at least for the cases considered.

*3) **RQ3**:* To answer RQ3, in Table V we ranked devices based on the number of CPIs they exhibited, with the device

TABLE V: Devices with the highest number of CPIs in our evaluation. Column *AV* (Android Version) reports the version of the Android system running on the device.

Device	*Resolution*	*Density*	*AV*
LG Optimus L70	480 x 800	207	19
Samsung Galaxy S3 Mini	480 x 800	233	19
Samsung Galaxy J1 Ace	480 x 800	217	19
Samsung Galaxy J1 Duos	480 x 800	217	19
Samsung Galaxy S Duos	480 x 800	233	19
Samsung Galaxy Grand Neo Plus	480 x 800	187	19
Intex Aqua Y2 Pro	480 x 854	218	19
Samsung Galaxy Light	480 x 800	233	19
Samsung Galaxy Star Advance	480 x 800	217	19
Samsung Galaxy Note 2	720 x 1280	267	19

exhibiting the highest number of CPIs at the top. For each device: column *Device* shows the name of the device; columns *Resolution* and *Density* shows the pixel resolution and density, respectively, of the device; and column *AV* reports the version of the Android system running on the device. The top nine devices all have low values for resolution and density, and no other test device has these characteristics. This result suggests that developers should consider to include a device with these characteristics when testing their apps. While looking at the relation between inconsistencies and device characteristics, however, we also observed that considering testing devices solely based on resolution and density would have not allowed us to identify all the inconsistencies reported in Table III. In fact, there are inconsistencies that derive from different hardware configurations of the devices (e.g., the presence of a physical menu button). It is also worth noting that, even if all devices in Table V happen to run Android version 19, we could not find any reason why that this version should be particularly problematic.

VI. Threats To Validity

As it is the case for most empirical evaluation, there are both construct and external threats to validity associated with our results. In terms of construct validity, there might be errors in the implementation of our technique. To mitigate this threat, we extensively inspected the results of the evaluation manually. In terms of external validity, our results might not generalize to other apps or CPIs. In particular, we only considered a limited number of apps. This limitation is an artifact of the complexity involved in manually inspecting results deriving from executions on a large set of devices (over 130). To mitigate this threat, we used randomly selected real-world apps from different domains.

VII. Related Work

The fragmentation of the Android ecosystem has been studied in the literature [30], [31], [2], [32], [33], [34], [35], [36]. Han and colleagues [30] are among the first to study the issues generated by such fragmentation. Their work systematically analyzes bug reports from two popular mobile device vendors and proposes a method for tracking fragmentation. In this line of research, Holzinger and colleagues [31] discuss the challenges involved in developing apps due to the differences in size and display resolution of different devices. Our work helps developers in this challenging task by automatically identifying inconsistencies across devices.

The work on Android fragmentation led to studies on device prioritization for app testing [37], [38], [39]. The recent work from Lu and colleagues [37], in particular, proposes a technique to prioritize Android device models for individual apps, based on mining large-scale usage data. We believe this line of research to be complementary to ours, as it could be integrated within DiffDroid in case resources are constrained in terms of number of test devices considered.

Other related work tries to find compatibility issues in Android apps [40]. Wei and colleagues propose a technique based on static analysis that identifies compatibility issues using an API-Context pair model. The issues identified by their technique are different from those identified by DiffDroid, as they are related to platform API evolution and drivers implementation. This technique could therefore also be combined with DiffDroid to identify a broader set of issues.

Finally, our work relates to the work on inconsistency identification for web apps [41], [42], [43], [44], [45]. Roy Choudhary and colleagues [41] propose a technique that crawls the web app under test in different browsers, collects DOM trees and screenshots for web pages in the app, and compares collected trees and images to identify inconsistencies. There are differences between mobile and web apps that prevent this and similar techniques to be straightforwardly applied in our context. This technique, for instance, runs browsers so that the size of their visible area is the same, which is an assumption that cannot be made for mobile apps.

VIII. Conclusion

Because of the fragmentation of the Android ecosystem, Android apps can exhibit inconsistencies in their behavior when they are run on different platforms. To help developers identify these behavioral inconsistencies before developers release their apps, we propose a technique called DiffDroid. DiffDroid aims to identify and suitably report behavioral inconsistencies in Android apps by combining input generation, behavior modeling, and differential testing.

We implemented and empirically evaluated DiffDroid by running it on 5 real-world benchmark apps and over 130 different platforms. Our results provide initial evidence that DiffDroid can identify CPIs on real apps efficiently and with a limited number of false positives.

In future work, we will first extend our evaluation by considering additional apps. Second, we will perform a user study with app developers to assess how useful is the information our technique provides and how effective is the format in which it is provided. Third, we will extend DiffDroid so that it handles inconsistencies other than visual ones (e.g., inconsistencies in the state of the app after an event is processed). Fourth, we will investigate whether precision and accuracy of CPIs identification can improve by using a multi-class classifier approach. Fifth, we will investigate techniques for suggesting repairs for the CPIs identified by DiffDroid. Finally, and more on the engineering side, we will investigate ways to automatically build stubs for the apps being tested, so as to speed up the execution on the different devices, enforce determinism, and in general allow for performing test executions in a sandbox.

Acknowledgments

This work was partially supported by the National Science Foundation under grants CCF-1320783 and CCF-1161821, and by funding from Amazon under the AWS Cloud Credits for Research program, Google, IBM Research, and Microsoft Research.

References

[1] Recode, "Salesforce will only support Nexus and Samsung Galaxy phones to avoid Android fragmentation," https://www.recode.net/2016/7/18/12217580/salesforce-support-nexus-samsung-galaxy-android.

[2] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real Challenges in Mobile app Development," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2013.

[3] Google, "Daily Dozen," https://play.google.com/store/apps/details?id=org.nutritionfacts.dailydozen.

[4] ——, "UI Overview," https://developer.android.com/guide/topics/ui/overview.html.

[5] ——, "Monkey," https://developer.android.com/studio/test/monkey.html.

[6] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An Input Generation System for Android Apps," in *Proceedings of the 2013 Joint Meeting on Foundations of Software Engineering*, 2013.

[7] 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 IEEE/ACM International Conference on Automated Software Engineering*, 2012.

[8] W. Yang, M. R. Prasad, and T. Xie, "A Grey-box Approach for Automated GUI-model Generation of Mobile Applications," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, 2013.

[9] 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, and Applications*, 2013.

[10] 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, and Applications*, 2013.

[11] 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*, 2014.

[12] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: Segmented Evolutionary Testing of Android Apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[13] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated Concolic Testing of Smartphone Apps," in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2012.

[14] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Transactions on Software Engineering*, 2002.

[15] M. P. Sampat, Z. Wang, S. Gupta, A. C. Bovik, and M. K. Markey, "Complex wavelet structural similarity: A new image similarity index," *IEEE transactions on image processing*, 2009.

[16] M. Fazzini, E. N. d. A. Freitas, S. Roy Choudhary, and A. Orso, "Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests," in *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation*, 2017.

[17] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[18] Y. Rubner, C. Tomasi, and L. J. Guibas, "The earth mover's distance as a metric for image retrieval," *International journal of computer vision*, 2000.

[19] C. Patel, A. Patel, and D. Patel, "Optical character recognition by open source OCR tool tesseract: A case study," *International Journal of Computer Applications*, 2012.

[20] Google, "UiAutomation," https://developer.android.com/reference/android/app/UiAutomation.html.

[21] Square, "JavaPoet," https://github.com/square/javapoet.

[22] Google, "Espresso," https://google.github.io/android-testing-support-library.

[23] A. Vacavant, C. Godfrey, and J. Terrace, "pyssim," https://github.com/jterrace/pyssim.

[24] Amazon, "Device Farm," https://aws.amazon.com/device-farm.

[25] University of Waikato, "Weka," http://www.cs.waikato.ac.nz/ml/weka.

[26] OpenCV, "OpenCV," http://opencv.org.

[27] Google, "Tesseract OCR," https://github.com/tesseract-ocr/tesseract.

[28] GitHub, "GitHub," https://github.com/.

[29] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet? (E)," in *Proceedings of the 2015 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[30] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs," in *Proceedings of the 2012 Working Conference on Reverse Engineering*, 2012.

[31] A. Holzinger, P. Treitler, and W. Slany, "Making apps useable on multiple different mobile platforms: On interoperability for business application development on smartphones," in *International Conference on Availability, Reliability, and Security*, 2012.

[32] H. Li, X. Lu, X. Liu, T. Xie, K. Bian, F. X. Lin, Q. Mei, and F. Feng, "Characterizing smartphone usage patterns from millions of Android users," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, 2015.

[33] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011.

[34] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[35] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[36] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *2014 IEEE Symposium on Security and Privacy (SP)*, 2014.

[37] X. Lu, X. Liu, H. Li, T. Xie, Q. Mei, D. Hao, G. Huang, and F. Feng, "PRADA: Prioritizing android devices for apps by mining large-scale usage data," in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[38] S. Vilkomir and B. Amstutz, "Using combinatorial approaches for testing mobile applications," in *2014 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2014.

[39] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan, "Prioritizing the devices to test your app on: A case study of android game apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[40] L. Wei, Y. Liu, and S.-C. Cheung, "Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps," in *2016 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[41] S. R. Choudhary, H. Versee, and A. Orso, "Webdiff: Automated Identification of Cross-Browser Issues in Web Applications," in *2010 IEEE International Conference on Software Maintenance (ICSM)*, 2010.

[42] A. Mesbah and M. R. Prasad, "Automated Cross-Browser Compatibility Testing," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[43] S. R. Choudhary, M. R. Prasad, and A. Orso, "Crosscheck: Combining Crawling and Differencing to Better Detect Cross-Browser Incompatibilities in Web Applications," in *2012 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[44] V. Dallmeier, M. Burger, T. Orth, and A. Zeller, "Webmate: a Tool for Testing Web 2.0 Applications," in *Proceedings of the Workshop on JavaScript Tools*, 2012.

[45] S. Roy Choudhary, M. R. Prasad, and A. Orso, "X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.