Improving Automated GUI Exploration of Android Apps via Static Dependency Analysis

Wunan Guo, Liwei Shen,
School of Computer Science
Shanghai Key Laboratory of Data Science
Fudan University
Shanghai, China
wnguo17/shenliwei@fudan.edu.cn

Ting Su,
School of Software Engineering
Shanghai Key Laboratory of Trustworthy Computing
East China Normal University
Shanghai, China
tsuletgo@gmail.com

Xin Peng, Weiyang Xie
School of Computer Science
Shanghai Key Laboratory of Data Science
Fudan University
Shanghai, China
pengxin/18212010074@fudan.edu.cn

***Abstract*—Exploring GUIs of Android apps plays a key role in many important scenarios such as functional testing (e.g., finding crash errors), security analysis (e.g., identifying malicious behaviors) and competitive analysis (e.g., storyboarding app features). To automate GUI exploration, existing techniques often try to visit as many GUI pages as possible via specific strategies, e.g., random (like Monkey) or heuristic (like Stoat, A^{3}E). However, their effectiveness is still unclear and much under-explored. To this end, we conducted the *first* study in this paper to understand and characterize their limitations by carefully analyzing the coverage reports from a set of real-world, open-source apps. Through this study, we identified three key limitations due to the lack of dependency knowledge during exploration, i.e., *widget-page dependency*, *widget-widget dependency* and *system-event dependency*. To overcome them, we introduce *dependency-informed exploration*, an automated approach that leverages static dependency analysis to effectively improve GUI exploration performance. Given an app, our approach first constructs a GUI page transition model that captures the dependencies between GUI widgets, and then guides GUI exploration during a depth-first traversal. We realized our approach as a tool named GESDA, and evaluated it on 70 open-source Android apps. The results show GESDA outperforms existing state-of-the-art GUI exploration techniques, i.e., Monkey and Stoat. Additionally, GESDA uncovers 4 previously unknown crashes in 4 apps as a by-product of GUI exploration due to the benefit of dependency knowledge, while Monkey and Stoat have not discovered them.**

***Index Terms*—Android, Test, Exploration**

I. INTRODUCTION

Android applications (apps) are UI-centric whose pages are transited to each other by operating the GUI widgets. Automated GUI exploration of Android apps exercises the behavior of an app by generating relevant inputs such as clicks and scrolls [1]. Based on the mechanism, it can play a key role in many important scenarios. For example, GUI exploration has been widely leveraged for functional testing to find runtime errors such as crashes [2][1][3]. It is also used for security analysis to identify malicious behaviors of an app [4][5], and for competitive analysis to storyboard the app features [6][7].

To automate GUI exploration, existing techniques try to visit as many GUI pages as possible via specific strategies. These strategies guide the choice of the correct interactions for a given UI to improve the exploration effectiveness [8], reflected as the coverage on page transitions, on widget associated events and callbacks, and on the underlying program code and control logic. Among them, the random strategy, as the name suggests, freely chooses a widget to interact with. Monkey [9] is one of the current state-of-the-practice tools following the random strategy. It sends pseudo-random sequences of events to random locations on the screen. Besides, the heuristic strategy enhances the selection decision based on heuristics. In this category, Stoat performs the dynamic exploration supported by the weighted UI heuristics [10]. A^{3}E explores an app by means of a systematic depth-first traversal [11].

Existing work reports the achieved activity, method or line coverage applying Monkey and Stoat towards open-source and industrial apps [12][13][1][14]. These coverages are basically low, so that their effectiveness is still unclear and much under-explored. To this end, we conducted the *first* study using Monkey and Stoat on 70 real-world open-source Android apps to understand and characterize their limitations. Through the study, we found that the unexplored code is mainly due to the lack of dependency knowledge on the required events and the widgets state of an app. Three key dependencies are identified.

- The reach of a new page depends on triggering the event of the correct widget on the current page. There are quite a few cases where an event driving the transition to a new page is not triggered, therefore the callback method as well as the class of the target activity is not covered. We name the dependency as *widget-page dependency*.
- The execution of the code branches in a widget callback depends on the triggering of the specific widget as well as the states of associated widgets. For example, the callback execution flow of a button is affected by the state of a checkbox on the same page. The applied tools did not click the button again, or they did not modify the state of the checkbox when clicking the button repeatedly. Therefore, one of the code branches associated with the state of the checkbox is not covered. We name the dependency as *widget-widget dependency*.
- The execution of specific lifecycle callbacks depends on the triggering of specific system-level events. These tools did not simulate system events to activate the lifecycle callbacks such as `onSaveInstanceState`

and `onRestoreInstanceState` as needed. We name the dependency as *system-event dependency*.

These types of dependencies are hardly identified during the runtime exploration. For an app which has not been instrumented, it is difficult in most cases to determine whether a widget in the current page has an event handler, whether its event callback depends on the state of other widgets, and whether the current page has special lifecycle callbacks that need to be triggered by system events. However, we believe static analysis of the app can identify some of the dependencies in advance, thereby improving the efficiency and the coverage of dynamic exploration.

In this paper, we introduce *dependency-informed exploration*, an automated approach that leverages static dependency analysis to effectively improve GUI exploration performance. The process of the approach is two-stage. First, a dependency-integrated page transition model (dPTM) is statically constructed from an Android apk. The model uses page as the basic unit to describe the transitions between them. It captures the dependencies as the elements denoting the widgets with callbacks in a page (*widget-page dependency*), the widgets whose states influence a callback (*widget-widget dependency*), and the lifecycle callbacks of the page (*system-event dependency*). Second, a depth-first traversal based dynamic exploration is guided utilizing the static model. During a step of page exploration, the widgets with callbacks and the lifecycle callbacks are exercised preferentially, and the widgets whose callback are affected by the states of other widgets are fully exercised based on the combination of the states.

We realized our approach as a tool named GESDA (GUI exploration improved via static dependency analysis). We then evaluate the tool on the 70 Android apps to compare the coverage performance with the state-of-the-art GUI exploration techniques, i.e., Monkey and Stoat. The results show GESDA outperforms existing tools and the dependencies play a key role for coverage increment. Additionally, GESDA uncovers 4 previously unknown crashes in 4 apps as a by-product of GUI exploration due to the benefit of dependency knowledge, while Monkey and Stoat have not discovered them. We make GESDA open source. The tool and the data of the evaluation can be found in our replication package [15].

The rest of paper is organized as follows. Section II presents the empirical study. Section III overviews our approach, and the two stages are presented in Section IV and Section V respectively. Section VI briefs our tool and presents the comparative experiments. Section VII is the related work and Section VIII is the conclusion of the paper.

II. Coverage Study Based on Existing Tools

We introduce the empirical study in this section. Because the line coverage is to be measured, we pre-defined a rule for selecting the exploration objects that they can be instrumented by Jacoco [1]. Therefore, we selected 18 apps from Stoat benchmark[10] and 52 apps from F-Droid[16], totally 70 open-source apps which have been verified by instrumenting Jacoco successfully. Furthermore, we prepared an Android emulator configured with 2GB RAM, and Android Version 5.1.1 (API level 22). Then we kept one hour exercising time limit for each app using Monkey and Stoat separately. After the testing, the two co-authors of us manually read the reports generated by Jacoco, identifying the unexplored statements, the unexplored code branches, the unexplored methods, and even the unexplored classes. Finally we analyzed the limitations of the tools and characterized the reasons in terms of dependencies.

On average, Monkey achieved 56.7%, 45%, and 41.1% in terms of class, method and line coverage, while Stoat achieved 55.54%, 44.5%, and 40.6% correspondingly. The coverage by Stoat is slightly lower than Monkey which is inline with prior findings [12][17]. Another reason is that we only compared with Stoat's *weighted exploration strategy* used in the model construction phase and did not include the second MCMC fuzzing phase. The detailed coverage of both tools on the apps is given in Table I. In this study, we focused on the dependencies missed by both tools and named them separately.

1) Widget-page dependency: There are both about 40 apps (41 from Monkey and 38 from Stoat) whose partial code was not covered due to the lack of this dependency knowledge. During exploration, Monkey randomly selected the widgets on a page, while Stoat decided the widgets by their weights computed by heuristic rules. Their strategies do not take the dependency into account, therefore they were not able to execute all the widgets on a page sometimes. Figure 1 depicts an example of exercising *arXiv mobile*. There is a search icon (a magnifying glass surrounded by a yellow dotted frame) located at the top right of the page. Its event is defined in the layout file (`android:onClick`) and the callback (`searchPressed`) is defined in the class `arXiv`. The callback `searchPressed` can open a new page of class `SearchWindow`. Neither Monkey nor Stoat clicked the search icon during the exploration, thus the code surrounded by red dotted framework in `searchPressed` and even in the `SearchWindow` class was not covered.

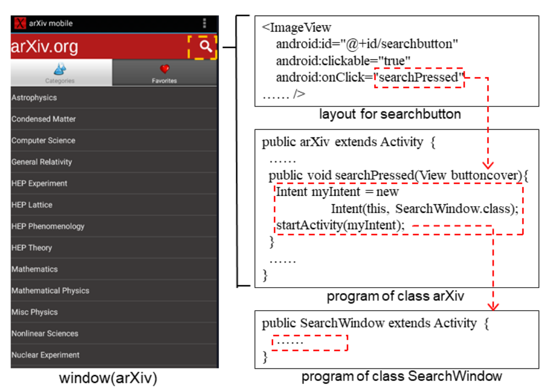

Fig. 1. Example of widget-page dependency

We also noticed a set of other similar scenarios where the

[1] Jacoco. https://www.jacoco.org/jacoco/

widgets in a dialog or a menu were not completely executed. This is also a *widget-page dependency* where the execution of the widgets in a dialog/menu depends on the opening of the dialog/menu, and further depends on the repeated triggering of the events responsible for opening the dialog/menu.

2) Widget-widget dependency: There are about 15 apps (15 from Monkey and 16 from Stoat) whose code involved in a branch was not covered due to the lack of this dependency knowledge. Monkey and Stoat focus on the selection of widgets but they do not care about the control logic and dependencies in the callbacks. Therefore they may not be able to cover all the branches in the callbacks by setting the states of other widgets. Figure 2 depicts an example of exercising *OI Notepad*. The left is a page of an opened dialog (`ThemeDialog`) with a OK button in the bottom-left and a checkbox (surrounded by the red solid framework) above this button. In the callback of the OK button, there exists a branch associated with the check state of the checkbox. Neither Monkey nor Stoat checked the checkbox before clicking the OK button during the exploration. Therefore the code surrounded by red dotted framework in `pressOk` was not covered.

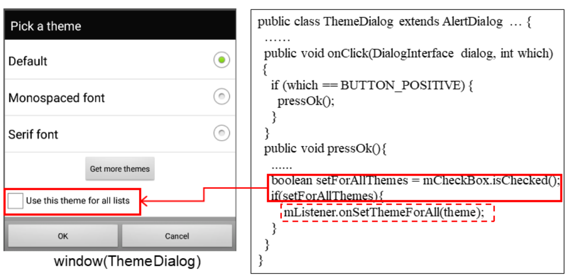

Fig. 2. Example of widget-widget dependency

3) System-event dependency: Lifecycle callbacks for Android activities such as `onCreate`, `onDestroy` are executed in a routine exploration process [18]. However, there are other kinds of lifecycle callbacks which are triggered by specific system events. Both Monkey and Stoat can simulate system events such as rotation, but in a random manner. Therefore, we found there are around 13 apps (12 from Monkey and 14 from Stoat) in which the callbacks including `onSaveInstanceState` and `onRestoreInstanceState` were not covered.

4) User-data related issue: Some pages depend on user input data to continue their business logic, for example a login page. If the data input randomly cannot be accepted, the app generally cannot transit to the other pages whose code is not covered. We found 6 apps which were explored by Monkey and Stoat each but the exploration is affected by the user-data related issue. However, we do not define it as a static dependency because the requisite for user data is hardly recognized during static analysis.

III. Approach Overview

The high-level overview of our approach is depicted in Figure 3. The approach is composed of two stages which are the model construction and the dynamic exploration.

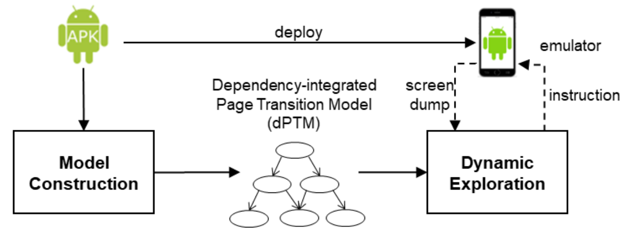

Fig. 3. Approach overview

Model construction is responsible for extracting a structural model of an Android app through static analysis. Taking an Android app as input, the stage generates a graph-style model named *Dependency-integrated Page Transition Model* (dPTM), with nodes representing app pages and edges representing transitions between the pages. The dependencies are integrated in the model according to the following principles. (1) The widgets with one or more event handles are the dependee UI elements of the *widget-page dependency* and should be completely recognized. (2) The widgets whose states affecting the control logic of a callback are the dependee elements of the *widget-widget dependency* and should be recognized and associated with the widget containing the callback. (3) The specific lifecyle callbacks, as the triggering object of the *system-event dependency*, should be recognized and pre-extracted from a page.

The dynamic exploration performs actions on a running Android app deployed in an emulator according to the depth-first traversal strategy assisted by dPTM. We leverage the depth-first strategy in order to systematically explore the app states by triggering the events of different widgets [11]. According to a typical exploration process, an exploration manager applying the strategy keeps interaction with the emulator running the app. The manager first sends the instruction of starting the app to start the exploration process. In each subsequent interaction, the manager generates the execution decision based on the screen dump returned by the emulator and the dPTM. The decision can be an instruction of executing a widget on the page (e.g., clicking a button, clicking a checkbox), or simulating a system event (e.g., rotating the screen). Once the emulator received the instructions, it performs the corresponding actions and returns back the screen dump to the manager after the action finishes.

The dependencies statically identified improve the dynamic exploration in making the execution decision. (1) The widgets existing in both the screen dump and dPTM are executed preferentially. This enables the code associated with the *widget-page dependency* explored as fully as possible. (2) A widget whose callback involves dependee elements of *widget-widget dependency* is executed completely. This enables to cover the branches determined by the combination of the states of the dependees. (3) The lifecycle callbacks of a page identified in dPTM are triggered by simulated events when entering this page for the first time. This enables the callbacks covered as needed.

IV. Model Construction

In this section, we first define the model, and then present the model construction process.

A. Model Definition

A Dependency-integrated Page Transition Model (dPTM) is a tuple $< P, p_0, T >$ where

- P is a non-empty set of pages of the app. In our approach, a page is an $Activity$, a $Menu$ or a $Dialog$. We take menu and dialog as the same level elements of the activity because they are also user interfaces that appear by executing widgets on the host activity.
- $p_0 \in P$ is the starting activity of the app. It is specified in the manifest file of the app.
- T is the set of transitions between the app pages. A transition exists when there is an invocation of starting another page (e.g., $startActivity$) from the current page.

P and T are compound elements which are defined as follows.

For each $p \in P$, p is a tuple $< pType, W, lcCallbacks >$ where

- $pType$ is the type identifier of the page, $pType \in \{Activity, Menu, Dialog, PseudoPage\}$. $PseudoPage$ represents an undetermined page object, used in a transition whose target page cannot be determined. It is also used to represent a page of an other app in an inter-app communication scenario.
- W is the set of widgets included in the page and each has at least one event handler. A widget $w \in W$ is further defined as a tuple $< wType, wId, wText, wEventHandler >$ in which $wType$ is the type of the widget such as $Button$ and $CheckBox$, wId and $wText$ are the resource id and the displayed textual characters of the widget. $wEventHandler$ is the set of event handlers associated with the widget.

  An event handler $wEh \in wEventHandler$ is defined as a tuple $< wEventType, wCallback >$. The former is the type of the event, for example *click*, *longclick*. The latter is a sub-tuple $< cbMethod, \{t_{cb}\}, wDepends >$. In it, $cbMethod$ is the callback method of the event. $\{t_{cb}\}$ is the set of transitions which can be triggered in the logic of the callback. $wDepends$ is the set of widget methods which are used to obtain the states of the widgets. These methods affect the control flow of the callback ($wCallback$), e.g. $checkbox1.isChecked$.
- $lcCallbacks$ is the set of specific lifecycle callbacks having been implemented in the code of the page.

A transition $t \in T$ is defined as a tuple $< p_s, p_d, wEh_t >$ where

- $p_s \in P$ and $p_t \in P$ are the source page and the target page of the transition respectively.
- $wEh_t \in w.wEventHandler$ is an event handler belonging to a $w \in p_s.W$. The transition is triggered by executing the callback of wEh_t.

B. Construction Process

The process of dPTM construction adopts a workflow integrated by the analysis on pages, transitions and dependencies. The process can be sketched by Algorithm 1, which takes an Android Apk file as input and finally generates the corresponding model. The algorithm first extracts the pages from the XML-based manifest file as well as the intermediate code representation (i.e., jimple code converted by Soot[19]) of the apk (line 2). For each page, it identifies the pre-specified lifecycle callbacks implemented in the page (line 4). The contained widgets are subsequently extracted based on the layout configuration and the intermediate code representation (line 5). For the widgets with event handles, transitions and possibly *widget-widget* dependencies are recognized through applying control-flow and data-flow analysis on the code of each callback (line 6-15). After traversing all the pages and their widgets, dPTM is synthesized (line 17).

Algorithm 1: dPTM-Construction

```
Input: apk: an Android Apk file
Output: dPTM
1  pages ← ∅, trans ← ∅
2  pages ← extractPages(apk)
3  foreach p ∈ pages do
4      p.lcCallbacks ← identifyLcCallbacks(p)
5      widgets ← extractWidgets(p)
6      foreach w ∈ widgets do
7          eventhandlers ← extractEventHandlers(w)
8          if |eventhandlers| > 0 then
9              foreach eh ∈ eventhandlers do
10                 t ← analyzeTransition(eh.wCallback)
11                 trans.append(t)
12                 eh.wCallback.{t_cb}.append(t)
13                 deps ←
                     analyzeDepends(eh.wCallback)
14                 eh.wCallback.append(deps)
15             end
16             p.W.append(w)
17         end
18     end
19 end
20 dPTM ← (pages, pages.getStartPage, trans)
```

The details of the construction steps are explained as follows.

1) Page extraction: Pages in the app are classified into Activity, Menu and Dialog. According to Android development specifications, activities are specified in the manifest file and are thus retrieved directly. Menus are hosted in an activity and are initialized by callback methods such as `onCreateOptionsMenu` and `onCreateContextMenu`. Therefore, we check whether there is such method in the code of the activity in order to extract the menu-type page. Dialogs, on the other hand, are identified by filtering out the app classes which directly or indirectly extend $android.app.Dialog$.

For each page, the pre-specified lifecycle callbacks are first identified. According to the coverage study, the most ignored callbacks are `onSaveInstanceState` and `onRestoreInstanceState`. Therefore, we currently recognize only these two callbacks by traversing and matching the names of callbacks.

2) Widget extraction: Widgets on a page are extracted subsequently. Activities and dialogs are pages containing various widgets with different capabilities such as $Button$ and $CheckBox$. They can be defined in the corresponding XML-based layout file, and can also be dynamically appended into the page through customized statements. In our approach, we reserve the widgets with at least one event handler since their response logic is the main target of dynamic exploration. We then leverage an extraction method starting from the events. On one hand, events of widgets registered in the layout file are identified for instance retrieving the configuration like `android:onClick="onClick"`. The resource id, type, text of the widget can be obtained and the callback method of the event is located. On the other hand, events which are specified using listener registration methods (e.g., `setOnClickListener`) are identified by scanning the code. We further adopt data-flow analysis on the caller of the method to locate its declaration statement which is a `findViewById` invocation when the widget is specified in the layout, or a `new` instantiation method when the widget is dynamically created. In this case, the resource id of the widget is obtained (if it is specified) and its type and text are retrieved from the layout file or from the arguments of the method invocations.

Menus are usually composed of hierarchical menu items each is regarded as a widget with corresponding event handler. There exist static and dynamic ways to define the interface of a menu. The former is to specify the constituents of a menu in the resource file and then to load the resource in the program. The latter uses menu-related methods (e.g., `addSubMenu`, `add`) to construct the menu structure. Unlike the other kinds of widgets, deep-level menu items are displayed through selecting their ancestor menus. Therefore, in the process of identifying the hierarchical menu items, we additionally need to record the display path of each menu item, which contains the sequence of the ancestor menus to be selected.

3) Transition analysis: The transitions between different pages are generally identified by the methods invoked in the callback of a widget. It means to locate the pre-specified methods in the callback and analyze the parameters of the method invocation to identify the target page. When the method is identified, a transition is created between the different pages. Otherwise, if there is no such method identified, we create a transition pointing to itself from the source page. We then explain the rules based on the methods specified.

- Methods prefixed with $startActivity$ such as `startActivity` and `startActivityForResult` indicate a transition to a new page which is determined by the $intent$ object. Since the intent can be explicit or implicit, we analyze the target of the transition as an activity belonging the app, or as a pseudo page ($PseudoPage$) belonging to a third-party app.
- `Dialog.show` indicates a transition to an instantiated dialog. The declaration class of the dialog can be retrieved by the caller of the method.
- `Dialog.dismiss` indicates a transition from a dialog to its host page. Since a dialog can be instantiated from arbitrary pages, the target page of the transition is set to a pseudo page ($PseudoPage$).
- The transition to a menu (including $optionMenu$ and $contextMenu$) is a special case in transition analysis. Because an $optionMenu$ is hosted in an activity, the transition from the activity to the menu is determined when the menu is extracted. On the other hand, a $contextMenu$ is registered on widgets using `registerForContextMenu` in lifecycle method such as `onCreate`. In this case, the transition can be determined from the host page to the menu with a $longclick$ event handler of the widget which registers the event.

4) Widget-widget dependency analysis: Based on the study from Section II, we conclude the most common widgets leading to *widget-widget dependency* and their state query methods, i.e, `CheckBox.isChecked`, `RadioButton.isChecked`, `Switch.isChecked` and `ToggleButton.isChecked`. These methods return boolean value thus can be involved as a condition.

During the analysis process, we sequentially scan each statement in the callback. When it comes to a decision statement, each condition belonging to the decision is analyzed whether it relates to a widget state. There are different situations for the identification.

- If the condition is a method invocation of `isChecked`, the caller is retrieved and matched with an extracted widget based on the data flow.
- If the condition is a boolean variant, the variant is traced backwards through the data flow to locate its assignment statement. If the statement is a method invocation of `isChecked`, we apply the previous rule to get the dependee widget.

Once a widget is determined, the expression combined with the widget object and its state query method (e.g., $checkbox1.isCheck$) is added into the set $wCallback.wDepends$.

V. Dynamic Exploration

Dynamic exploration exercises an app and tracks the state of the app. We define the *exploration state* as the state of a page that is currently visible during the exploration process. An exploration state, denoted by S_e is synthesized with the screen dump of the page obtained from an emulator and the knowledge from the dPTM. It is recomputed after each transition by a widget execution. S_e is defined as a tuple $< pId, pType, W_s, W_d, lcCbks >$ where

- pId and $pType$ are the same as those in dPTM when matching from the screen dump to the model. We assume p as the object denoting the current page.

- $w_s \in W_s =< w \in p.W, wEh \in w.wEventHandler, exectuable >$. w_s is an actual widget in the screen dump that has its counterpart in $p.W$. Meanwhile, $executable$ is the flag indicating whether an event of the widget needs to be triggered. In particular, if a widget has more than one event handler, there should be the same number of tuple instances in order to facilitate the exploration.
- $w_d \in W_d =< w \notin p.W, executable >$. w_d is an operable widget dynamically extracted from the current page but does not has the counterpart in $p.W$. $executable$ means whether w_d can be executed.
- $lcCbk \in lcCbks =< Cbk \in p.lcCallbacks, triggered\}$. $triggered$ denotes whether the lifecycle callback has been triggered or not.

We sketch the process by Algorithm 2. The algorithm is performed by an explorer manager. The manager interacts with an emulator by continuously receiving the screen dump and sending the execution instructions according to the algorithm. The algorithm takes the dPTM of the app as input and a $timeout$ indicating the time the exploration can spend.

The algorithm first launches the app (line 1), then computes the exploration state based on the currently visible page after each transition by $computeState$ (line 3). If the page is visited for the first time, the exploration state is synthesized based on the page object from dPTM and the screen dump.

- One task in this step is to identify the static extracted widgets (a $w \in$ dPTM.$P.W$) in the current page. We adopt the following rules to achieve matching. If $w.wId$ is not empty, it retrieves the correspondent widget on the current page by the id. Otherwise, it finds the widget by matching the properties including $wType$, $wText$ and $wEventType$. When there are multiple widgets found in an activity, it randomly selects one of them. If no actual widget found, w is not counted in $E_s.W_s$ and a warning is reported. In addition, for an identified widget with more than one event handler, we duplicate the corresponding number of w_s with the different event handlers.
- $E_s.W_d$ is collected by retrieving the remaining operable widgets on the page. We recognize the operable widgets by checking whether the attributes such as $clickable$, $longClickable$ and $scrollable$ are true in the screen dump.
- $E_s.lcCbks$ is consistently ported from $w.lcCallbacks$ and each $lcCkb.triggered$ is set as false initially.

On the other hand, if the page is not visited the first time, the exploration state is recovered from the saved page states. In this case, $E_s.W_s$ and $E_s.W_d$ are re-identified but the $executable$ value remains. It should be noted that the exploration states related to a same page may be different reflected in the number of w_d, thus requiring the recomputation of the state. For example in the case of self-transition from a page to itself, a widget execution may load a fragment whose widgets are counted in the host activity but we cannot find their counterparts in the static model.

Algorithm 2: DynamicExploration

Input: dPTM, $timeout$

1 launchApp
2 **repeat**
3     $E_s \leftarrow$ computeState($curPage$)
4   **foreach** $lcCbk \in E_s.lcCbks, lcCbk.triggered = false$ **do**
5   triggerLcCallback($lcCbk$)
6   **end**
7   **if** $|\{E_s.W_s.w_s | E_s.W_s.w_s.exectuable = true\}| > 0$, **then**
8   select any w_s from the above set
9   **if** $|w_s.wEh.wCallback.wDepends| > 0$ **then**
10   setUncoveredDependsState ($w_s.wEh.wCallback.wDepends$)
11   **end**
12   execute w_s with $w_s.wEh$
13   **if** *checkExecutable*(w_s) = *fase* **then**
14             $w_s.exectuable \leftarrow false$
15   **end**
16   save E_s as page state
17   **end**
18   **else**
19   **if** $|\{E_s.w_d | E_s.W_d.w_d.exectuable = true\}| > 0$ **then**
20   select a w_d from the above set by top-down order and execute w_d
21   set $w_d.exectuable \leftarrow false$
22   save E_s as page state
23   **end**
24   **else**
25   go back to the previous page
26   **end**
27   **end**
28 **until** *timeout*;

After the state computation, the algorithm fist triggers the lifecycle callbacks as needed (line 4-6). Our approach specify the two callbacks, i.e., `onSaveInstanceState` and `onRestoreInstanceState`. Therefore, we apply a strategy of rotating the screen in the emulator to simulate the system events that are consumed by the callbacks.

After that an actual widget is to be executed by the following rules.

- Choose a w_s whose $exectuable$ is true and execute the widget (line 7-12). One major concern here is to handle with the *widget-widget* dependencies included. The algorithm guides to perform operations on the widgets to set their states (e.g., check a checkbox, check a radiobutton) which has not been covered beforehand (line 9-11). For example, assume $w_s.wEh.wCallback.wDepends =$

$\{chk1.isChecked, chk2.isChecked\}$ and $(chk1.isChecked = true, chk2.isChecked = true)$ has been covered, the operations leading to $(chk1.isChecked = true, chk2.isChecked = false)$ are performed this time.

Another major concern is to determine whether the widget can be executed again (line 13-15). For a widget whose callback involves *widget-widget* dependencies, $w_s.executable$ is set to false when all the combinations of the sates have been covered. For a widget whose execution transits to a menu or dialog, $w_s.executable$ is set to false when all the $executable$ of the widgets on the menu or the dialog is false. A widget with the both characteristics needs to consider these two conditions together. Finally, the exploration state also indicating the latest state of the current page is saved and it may be recovered in the further iterations.

- When there is no executable w_s on the page, the algorithm executes a selected w_d from $E_s.W_d$ in order to cover the possibly existing code that is not recognized in static analysis (line 20). Once the widget is executed, $w_d.executable$ is set to false (line 21) and the page state is saved (line 22).
- When there is no executable widget on the page, the algorithm guides to go back to the previous page by means of simulating the $back$ button of the emulator (line 25).

There is an exception handling mechanism designed for the exploration process. On one hand, when a runtime exception is raised, the exception is reported and the exploration is performed again from the start page. On the other hand, transitions to third-party apps by implicit intents are also regarded as an exception. In this case, the exploration manager tries to return back to the original app by simulating the back button clicking. If it is unsuccessful, it performs the exploration from the start page.

VI. Tool and Evaluation

Currently, GESDA is implemented by two major modules responsible for the static model construction and the dynamic exploration. We apply a set of off-the-shelf tools for analyzing an app, i.e., Apktool [20] is used to decompile an apk and obtain the resources, Soot [19] is used to analyze the intermediate code representation, and FlowDroid [21] is used to obtain the call graph of methods. Additionally, we develop the exploration manager as an Android app which contains UI Automator[22] and communicates with UI Automator by sending instructions based on the exploration algorithm.

To evaluate the effectiveness of GESDA, we aim to answer the following research questions.

RQ1: Compared with existing GUI exploration techniques, can GESDA improve the exploration effectiveness?

RQ2: How does the identified dependencies benefit the coverage?

RQ3: Can GESDA find crashes with the help of the identified dependencies?

We have explored 70 open-source Android apps by Monkey and Stoat. Therefore, we design further experiments utilizing GESDA on these apps based on the same environment setting, i.e., the same emulator and the same time limit (one hour) for exercising. Also, we only compared with Stoat's weighted exploration strategy in our experiment. Besides, in order to answer RQ2, we customized a tool named GESDA_DF which modifies Algorithm 2 by removing the parts taking advantage of the dependencies. GESDA_DF thus follows a primitive depth-first traversal strategy where the revised algorithm only reserves W_d during exploration.

A. RQ1: Exploration Effectiveness

The coverage statistics using Monkey, Stoat, GESDA_DF and GESDA is shown in Table I. For each tool, we list its coverage value (cv.) and coverage ratio (cr.) on the class level, method level, and code line level. The tools with the best coverage for each app is marked in the cells with gray background. In particular, two apps, i.e., $Addi$ and $InternetRadio$, cannot be explored by Stoat due to some compatibility issues, which were denoted by "-" in the corresponding cells.

On average, GESDA outperforms the other tools on all levels. The coverage on class level achieved on average 86 classes and 62.7%, at least 7 more explored classes and 6% higher than the others (Monkey has the highest 79 classes and 56.7% among the others). The coverage on method level achieved on average 366 methods and 50.6%, at least 34 more explored methods and 5.6% higher than the others (Monkey has the highest 332 methods and 45% among the others). The coverage on line level achieved on average 1776 lines and 46.5%, at least 205 more explored lines and 5.4% higher than the others (Monkey has the highest 1571 lines and 41.1%).

We choose two examples from the apps to describe the exploration effectiveness brought by GESDA.

Case 1 (*chanu*). *chanu* is a comic app composed of totally 29 menus distributed in 7 activities. Stoat activated 9 of them, while Monkey activated 13 of them. Based on the knowledge from *widget-page dependency* related to the menus, GESDA fully visited all the items on the menus, thus achieving a higher coverage. On the other hand, there are 6 activities containing the callbacks of `onSaveInstanceState` and `onRestoreInstanceState`. GESDA sent simulated screen rotation events as needed when being informed with their existence in advance. On the contrary, Stoat did not cover any of the callbacks while Monkey triggered one of them.

Case 2 ($MultiSms$)**.** $MultiSms$ is a chat app. There are three buttons (i.e., $mAddButton$, $mAddGroupButton$, $mSend$) with event handlers in the chat interface named $MultiSmsSender$. When clicking the $mSend$ button, the app transits to the $ListEntryActivity$ page, however it has no widget responsible for returning back to the previous page. In this case, Stoat and Monkey were stuck in $ListEntryActivity$ after clicking $mSend$. It is reported that Stoat triggered two of the buttons, while Monkey only triggered one. On the contrary, GESDA has the ability to return back to the previous page, helping to cover all the three buttons recognized in advance.

TABLE I
COVERAGE STATISTICS OF MONKEY, STOAT, GESDA_DF AND GESDA

App		Class Level								Method Level								Line Level								T(s)
Name	KLOC	Monkey		Stoat		GESDA_DF		GESDA		Monkey		Stoat		GESDA_DF		GESDA		Monkey		Stoat		GESDA_DF		GESDA		
		cv.	cr.(%)	cv.	cr.(%)	cv.	cr.(%)	cv.	cr.(%)	cv.	cr.(%)	cv.	cr.(%)	cv.	cr.(%)	cv.	cr.(%)	cv.	cr.(%)	cv.	cr.(%)	cv.	cr.(%)	cv.	cr.(%)	
36C3 Schedule	3.3	47	72	44	67	45	69	51	78	238	53	204	45	216	48	260	58	1546	47	1126	34	1223	37	1782	54	40
A Photo Manager	11.3	120	39	201	65	148	48	192	62	573	28	1024	50	674	33	934	46	2832	25	5051	44	3387	30	4545	40	47
Addi	19.6	54	14	-	-	48	12	48	12	235	11	-	-	193	9	193	9	3775	19	-	-	3477	17	3477	17	7
ADSdroid	0.2	9	90	9	90	9	90	9	90	31	86	29	80	31	86	31	86	187	86	168	77	197	91	197	91	7
Amaze Debug	17.8	158	40	162	41	155	39	170	43	719	27	808	30	739	28	821	31	3703	20	4512	25	3893	22	4833	27	6
AnyMemo	12.2	223	38	160	27	240	41	309	54	889	33	630	24	828	31	1211	46	3692	30	2539	20	2836	23	4905	40	8
Apod Classic	0.4	19	90	19	90	19	90	19	90	66	68	67	69	64	65	69	71	238	62	256	67	240	63	262	69	12
arXiv Explorer	1.6	43	86	35	70	32	64	44	88	312	76	284	69	243	59	316	77	1234	75	1158	71	1094	67	1240	76	6
arXiv mobile	2	31	55	30	53	22	39	36	64	95	48	83	47	78	40	114	58	810	41	825	42	727	37	982	50	7
Atomic	7.9	70	37	21	11	27	14	27	14	392	28	69	5	100	7	100	7	1941	24	342	4	476	6	476	6	10
AudioBook	1	41	66	28	45	30	48	56	90	127	47	74	27	89	33	195	72	455	45	263	26	304	30	769	76	8
AudioMeter	0.3	14	93	14	93	14	93	14	93	54	85	50	79	46	72	51	80	281	87	271	84	265	82	281	87	7
Binaural Beats	3	92	86	88	83	81	76	97	91	356	69	330	64	293	57	382	74	2152	71	2001	66	1796	59	2243	74	8
Bop-MusicPlayer	4.4	91	48	91	48	95	50	99	52	306	34	271	30	280	31	305	34	1405	32	1183	27	1227	28	1534	35	7
Budget	2.5	49	72	53	77	44	64	48	70	255	56	301	66	249	55	289	64	1241	49	1528	60	1183	46	1529	60	6
Car Report	7.3	182	75	172	71	97	40	97	40	718	50	733	51	316	22	316	22	3249	44	3342	46	1238	17	1238	17	4
CEtoolbox	1.7	15	69	16	69	16	69	16	69	110	70	114	72	88	56	88	56	1068	61	1133	65	1048	60	1048	60	5
chanu	36.8	397	34	329	28	306	26	486	41	1831	28	1374	21	1180	18	2439	37	9588	26	7751	21	6656	18	13505	36	29
Dalvik Explorer	1.4	38	80	40	85	40	85	43	91	149	62	187	77	165	68	179	74	739	52	969	69	854	61	953	68	5
DemocracyDroid	1.1	26	39	26	39	22	32	36	54	68	28	68	28	54	22	97	40	364	32	375	33	341	30	390	34	15
Dev GrowTracker	6.3	132	42	126	41	99	32	130	41	366	34	374	35	283	26	347	32	1988	31	1836	29	1403	22	1782	28	6
Diary	2	12	54	11	50	10	45	12	54	102	41	97	39	92	37	107	43	569	29	548	28	431	25	587	30	6
Diccionario castellano	0.2	15	88	13	76	14	82	15	88	49	70	48	68	44	62	50	71	170	69	129	52	157	64	174	71	8
Eternity Wall	4.7	63	33	98	51	55	28	57	30	203	24	331	39	180	21	189	22	859	18	1570	33	776	16	805	17	5
Finance Manager	3.4	131	59	130	58	124	56	140	63	396	50	405	51	373	47	420	53	1428	42	1451	42	1324	39	1491	44	10
Fissure	1.1	6	19	19	86	19	86	19	86	11	8	80	62	77	59	78	60	64	6	443	41	415	39	422	40	7
Free Mobile Netstat	3	45	51	46	52	42	47	45	51	174	42	184	45	165	40	177	43	1257	41	1342	44	1190	39	1238	41	8
FTP Server	3.3	26	31	26	31	26	31	29	35	128	31	120	29	109	26	142	34	709	21	606	18	616	18	780	23	13
G-droid	6	93	48	78	40	98	51	155	80	322	35	248	27	297	32	520	56	2332	38	1358	22	2351	39	3711	61	23
Goblim	1.4	32	42	26	34	27	35	33	44	87	36	77	31	79	32	92	37	451	31	426	29	439	30	484	33	8
Hendroid-beta	11.2	114	36	130	41	109	39	131	42	489	21	526	22	507	21	528	22	1936	17	1965	17	1940	17	1984	17	41
InternetRadio	0.6	29	82	-	-	24	68	30	85	94	73	-	-	70	54	97	75	467	79	-	-	331	56	484	82	9
inventum	2.4	53	49	56	51	53	49	57	52	191	29	205	31	178	27	204	31	648	27	674	28	602	25	675	28	11
Just Craigslist	2.2	18	48	28	75	28	75	36	97	47	46	77	76	75	74	97	96	386	17	1713	77	1338	60	1504	67	7
Just Notes	0.6	20	86	20	86	17	73	20	86	51	80	52	82	38	60	54	85	343	59	452	78	290	50	373	64	7
KeePassDroid	9	48	14	30	9	30	9	169	52	159	9	83	4	102	5	613	35	645	7	330	3	371	4	2751	30	17
KindMind	1.7	34	59	40	70	34	59	44	77	108	48	144	64	112	50	146	65	619	37	914	55	718	43	986	59	7
klaxon	0.7	16	55	17	58	17	58	18	62	44	34	50	39	49	38	53	41	205	28	227	31	214	29	235	32	4
KouChat	5.3	70	41	88	52	82	49	83	49	352	30	444	38	381	33	393	34	1373	25	1691	31	1501	28	1506	28	10
library	2.5	12	24	12	24	12	24	13	26	23	11	23	11	23	11	26	12	154	6	156	6	154	6	181	7	6
Lock Pattern Generator	0.7	22	75	24	82	21	72	22	75	76	58	93	71	71	54	81	62	446	68	506	77	403	61	442	67	4
LogicalDefence	0.2	12	92	11	84	12	92	12	92	47	81	46	79	47	81	47	81	204	87	210	90	203	87	203	87	6
man man	2.5	75	65	59	51	77	67	82	71	242	50	150	31	218	45	249	51	892	36	523	21	793	32	948	38	7
Markor	8.4	109	63	114	66	97	56	117	68	709	45	678	43	519	33	740	47	3344	39	3215	38	2462	29	3461	41	11
MatLong	3.1	49	66	40	54	40	54	43	58	285	49	208	35	203	35	209	36	1487	47	1053	33	952	30	1088	34	16
MoneyWallet	25.3	309	35	357	40	151	17	179	20	1441	31	1754	38	690	15	815	17	6056	23	7821	30	3021	11	3555	14	10
Multi Sms	0.8	11	33	17	51	16	46	20	60	25	19	59	48	39	30	74	58	151	18	308	37	257	31	359	43	4
nanoConverter	1.2	17	40	20	47	16	38	20	47	46	37	61	49	44	35	56	45	356	28	456	36	311	24	439	35	5
Net Monitor	3.4	38	45	33	39	50	60	61	73	158	39	137	33	194	48	257	63	1497	44	1382	40	1655	48	2031	59	9
News	7.8	135	43	138	44	126	40	138	44	425	26	477	29	408	25	490	30	2132	27	2212	28	1885	24	2196	28	37
Ninja	4	76	52	76	52	64	44	73	50	326	45	320	45	214	30	298	41	1684	42	1678	42	1323	33	1539	38	10
Oinotepad	2.9	34	51	49	74	45	68	51	77	151	40	231	62	203	54	248	67	991	34	1482	51	1223	42	1631	56	10
OpenManga	11.1	140	44	87	27	114	36	137	43	629	31	357	17	511	25	642	32	2822	25	1479	13	2231	20	3037	27	36
primitive ftpd	3.6	40	38	65	63	54	52	56	54	174	25	294	42	229	33	242	34	791	22	1368	38	1072	30	1124	31	12
PWMP	1.9	64	70	61	67	37	40	67	73	252	61	229	55	168	40	263	63	1045	54	900	46	694	36	1099	57	17
RedReader	12.5	131	24	165	31	142	27	142	27	444	19	555	24	484	21	484	21	1901	15	2443	19	2182	17	2190	17	15
ringdroid	1.4	20	76	15	57	14	53	16	61	116	67	75	43	91	53	111	64	1007	69	682	47	880	60	1015	70	6
Sanity	4.6	87	34	41	16	56	22	56	22	330	28	151	12	208	17	208	17	922	19	381	8	603	12	605	12	11
Shopping List	4.2	140	74	149	78	144	76	144	76	597	52	766	67	643	56	644	57	2046	48	2750	64	2290	53	2311	54	13
Silectric	1.3	52	98	52	98	32	60	52	98	214	81	217	82	122	46	219	83	1076	83	1062	82	616	47	1082	83	8
sNotePad	0.4	21	91	22	95	15	65	22	95	55	62	67	76	36	40	74	84	240	59	296	73	158	39	321	79	7
Starke Verben	0.7	23	88	25	95	23	88	23	88	104	81	102	80	97	76	101	79	613	85	607	84	583	81	594	82	13
SyncMyPix	5.2	56	41	74	54	50	37	74	54	137	19	187	26	126	18	187	26	624	12	826	15	467	9	827	16	8
Tinte Webcoms	3.2	100	59	43	25	43	25	112	66	381	57	107	16	107	16	407	61	1696	53	453	14	446	14	1821	57	14
Tshot	12.4	285	50	171	30	268	47	319	56	1096	41	668	25	904	34	1362	51	4245	34	2499	20	3750	30	5483	44	39
Vinyl DEBUG	12.6	229	58	247	63	209	53	232	59	1319	44	1437	48	1127	38	1260	42	4791	38	5316	42	4056	32	4564	36	48
vlilleChecker	2.1	55	55	54	54	52	52	55	55	271	44	264	43	257	42	271	44	851	41	841	40	794	38	870	42	18
Zapp	2.6	91	84	79	73	84	77	101	93	489	72	381	56	434	64	525	77	1792	70	1393	54	1572	61	1925	75	19
charmplayer	1	36	97	35	93	30	81	36	97	130	76	137	80	113	66	139	81	781	75	842	81	552	53	831	80	8
WikiPedia	22.2	380	43	432	48	415	46	438	49	1677	28	2079	35	1951	33	2209	37	6375	28	7994	35	7211	32	8420	37	50
Average	5.2	79	56.7	76	55.54	70	52.5	86	62.7	332	45.0	322	44.5	280	40.0	366	50.6	1571	41.1	1509	40.6	1338	37.0	1776	46.5	13

In general, the effectiveness is improved especially for the apps with menus or dialogs. GESDA can generally guarantee the full coverage of the widgets in these pages. On the other hand, the code of branches related to *widget-widget dependency* is also covered in the study but the size of the code is relatively small. However, we believe the exploration efficiency will be better improved if a branch involves complex logic, and even the transitions to other activities.

We also tracked the progressive coverage of the tools over time. Figure 4 reports the coverage on line level over the time threshold we used, i.e., 60 minutes. The plot illustrates the average coverage achieved across all the 70 apps. It should be noted that we started timing for these tools after the app loads its main page. Therefore, the initial coverage is greater than zero. The plot shows that GESDA surpassed the others from the beginning, and achieved 90 percent of the final coverage in about 10 minutes. The progressive coverage on the method level and the class level is also consistent with the figure, thus is omitted here for paper limitation (all the data can be found in our replication package [15]). These time-related data also proves the exploration effectiveness improved by GESDA.

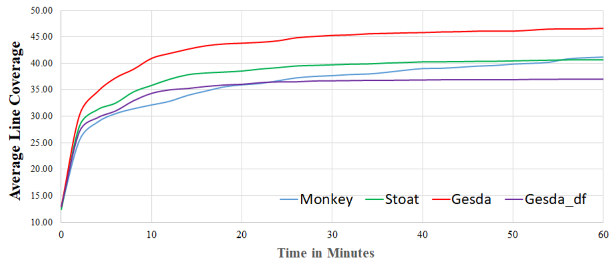

Fig. 4. Progressive coverage on line level

B. RQ2: Benefit of Static Dependencies

We compare the coverage between GESDA and GESDA_DF. The results show that GESDA performs better (at least not worse) than GESDA_DF for each app. It indicates the identified dependencies play an effective role in the exploration process, leading to explore more classes, methods and code lines.

We also use the data in RQ1 to explain the benefits. In these cases, GESDA_DF has a lower coverage than Monkey and Stoat. When taking advantage of the dependencies, the coverage exceeds those two tools because the exploration is guided to cover all the recognized widgets and the recognized code branches.

However, there are still situations where GESDA and GESDA_DF have the same coverage. For example, $ADSdroid$ is a simple app consisting only of activities, without menus and dialogs. The dependency only contains the widgets which can trigger a page transition. These widgets can also be covered by a conventional depth-first traversal. Therefore, the identified dependencies can benefit the coverage if they exist.

We also recorded the time spent on static analysis (i.e., model construction) for each app which is listed in the last column of Table I. It varies according to the number of the activities and callbacks in an app, however up to 50 seconds. It spent on average 13 seconds which is considered acceptable as an offline task.

C. RQ3: Crash Detection

Through the automated GUI exploration of the 70 open-source apps, we totally uncovered 7 crashes in 6 apps. In it, 4 crashes in 4 apps listed in Table II were previously unknown and were uncovered only by GESDA, while the remaining 3 crashes could be discovered by Monkey or Stoat (one of them by Monkey, the other two by Stoat). In particular, all the crashes were uncovered benefiting from the knowledge of *widget-page dependency*.

The first crash appears in a menu item (*open-M-file*) clicking scenario in $Addi$. The menu containing the item is opened by simulating the menu key on the home page. Actually the menu has two items causing different crashes. Besides the listed one, the other crash was uncovers by both GESDA and

TABLE II
CRASH UNCOVERED BY GESDA ONLY

#	app	operations causing crash
1	Addi	Addi $\stackrel{menukey}{\rightarrow}$ optionMenu $\stackrel{open-M-file}{\rightarrow}$ crash
2	ringdroid	RingdroidEditActivity $\stackrel{save}{\rightarrow}$ saveDialog $\stackrel{save}{\rightarrow}$ successDialog $\stackrel{close}{\rightarrow}$ crash
3	Silectric	UsageActivity $\stackrel{add}{\rightarrow}$ addDialog $\stackrel{add}{\rightarrow}$ crash
4	Bop-Music Player	MainScreen $\stackrel{menukey}{\rightarrow}$ optionMenu $\stackrel{quit}{\rightarrow}$ HomeScreen $\stackrel{restart}{\rightarrow}$ MainScreen $\stackrel{menukey}{\rightarrow}$ crash

Monkey. Monkey clicked the other item (*create-M-file*) but neglected the *open-M-file* item since it did not open the menu again. However, GESDA uncovered the both crashes through a complete traversal. It takes advantage of the dependency indicating that the menus should be opened repeatedly.

The second and third crashes appear in scenarios of dialog widget clicking in $ringdroid$ and $Silectric$ respectively. The former is caused by clicking the $close$ button on the $success$ dialog opened by another $save$ dialog. The $save$ dialog is opened by clicking the $save$ button on the $RingdroidEditActivity$ page. The latter is caused by clicking the add button of a dialog opened by the add button on the $UsageActivity$ page. They both benefited from the *widget-page dependency* so that all the widgets in the dialogs were explored by repeatedly opening the dialogs while Monkey and Stoat did not guide in this regard.

The forth crashes appears in $Bop-MusicPlayer$ where the simulating of the menu key causes the crash after the app is restarted by clicking the $quit$ button on the option menu. Among the tools, only GESDA guided the operations, i.e., clicking the $quit$ button and restarting the app due to the dependency knowledge.

D. Limitations and Threats to Validity

There are some limitations of GESDA. First, GESDA may fall into a "quagmire". It means to constantly explore the newly appeared widgets whose logics are the same, so that the coverage cannot be increased and GESDA has no chance to leave the page. For example, when exercising $MoneyWallet$, GESDA fell in a $Calendar$ page where there was new clickable date icons appeared when clicking an existing date icon. GESDA thus has a low coverage on this app. Future work may integrate Repetition-Avoidance technique [14] to tackle this issue. Second, GESDA does not have the ability to send broadcasts like Monkey, therefore it cannot cover the code related to broadcast receiver. For example, GESDA performs poor for $Atomic$ because it is a chat client consuming series of broadcasts. GESDA does not pay attention to it currently but can be improved by recognizing the callback of broadcast receiver and sending the broadcasts when necessary.

The main threat to external validity is the selection of the apps. In order to compare the exploration effectiveness with Monkey and Stoat, we carefully choose some apps from Stoat benchmark and supplement others from F-Droid. These apps with different design structures and different sizes are able to demonstrate the ability of GESDA. However, we acknowledge that these apps may not be broadly representative. The threat to internal validity lies in the coverage analysis. We currently analyze the reports manually, thus causing possible inaccuracies. To alleviate the threat, we arranged two persons to review the reports and check the results with each other.

VII. RELATED WORK

A. Static Analysis for Android Apps

Static analysis for Android apps has been used for clone detection [23][24], security assessing [25][26], automated test cases generation [27][28], and so on. The modeling of the GUI behavior of an app is a basis thus receiving a lot of attentions. Among them, A^3E is the first to build a static model of an Android app which constructs the Static Activity Transition Graph (SATG) by data-flow analysis [11]. Gator [29] constructs a Window Transition Graph (WTG) which adds components like menus and dialogs on the SATG. GoalExplorer [30] further extends the static model by adding other component such as fragments, drawers, and broadcast receivers. There are also some work aiming to construct a static model by inter-component communication analysis. EPICC [31] is the first to extract inter-component communication. Based on it, IC3 [32] improves the extraction ability of inter-component communication. StoryDroid [33] extends IC3 and adds the fragment information. Our approach relies on a static model to improve the exploration effectiveness. Compared with the above techniques, the model we customized captures the dependencies benefitting exploration.

B. Automated GUI Exploration

Automated GUI exploration performs according to specific strategies. Monkey [9] is the state-of-the-practice tool applying the random strategy. Dynodroid [34] also follows the random strategy to generate UI events and system-level events, and it generates more system-level events than Monkey such as incoming phone calls and geolocation changes. PUMA [35] is a dynamic analysis framework that provides a random exploration implemented by Monkey. Sapienz [2] extends Monkey and leverages a genetic algorithm to maximize code coverage and fault detection while minimizing the length of the generated test sequences. Our approach also follows a random strategy when selecting the pre-recognized widget, and involves system events from the lifcycle perspective.

There is another strategy for exploration by constructing a finite state machine (FSM). GUIRipper [36] and MobiGUITAR [37] are the first two approaches aiming to construct a FSM by depth-first exploration. ORBIT [38] improves the efficiency of GUIRipper by analyzing the app's source code to get the relevant events of the current activity. Besides constructing SATG, A^3E implements a depth-first exploration strategy named Depth-First Exploration. Unlike GUIRipper, when there exists no event to enter a new activity, it will return back to the previous activity instead of restarting the exploration. Stoat [10] (upgraded to Stoat+ [39]) computes the priority of each widget in a state by the heuristics composed of the execution frequency, event type and the number of widgets after the execution. APE [40] uses a model-based approach, but its model is dynamically refined according to the attributes of UI elements. Its core idea is to balance the model precision and scalability. Different from the traditional FSM-based work, our approach makes use of the dependency knowledge and integrates into a basic depth-first traversal.

C. Combination of static and dynamic analysis

There are some studies leveraging the combination of static and dynamic analysis to improve the test efficiency or to find specific crashes. TrimDroid [27] employs static analysis to extract dependencies meanwhile combining Robotium to improve GUI-based testing. The dependencies of TrimDroid is used to reduce the combinatorics in testing. Fax [41] combines static activity launching context construction and dynamic exploration to achieve multiple-entry testing of android apps. APEChecker [42] combines static analysis and dynamic UI exploration to uncover asynchronous programming errors in android apps. However, our approach has a different purpose which is to enhance GUI exploration in order to increase the coverage and to uncover unknown crashes [43].

D. Coverage Study of Existing Tools

There have been some studies comparing the exploration coverage of exiting tools. Choudhary et al. [1] conducted a study to analyze 7 main testing tools at the time on 60 apps by the metrics including code coverage. Zeng et al. [44] analyzed the limitations of Monkey in exploring an industrial app WeChat and then compared the line coverage and activity coverage between their proposed approach and Monkey. They further analyzed the uncovered code of Monkey and summarize six main reasons [14]. Wang et al. [12] conducted a systematic study by applying 6 tools on 68 industrial apps, and compared their coverage on the method and activity level. Compared with their work, we also conducted an empirical study investigating the coverage on class, method and line using Monkey and Stoat, and we went further to understand the key limitations due to the lack of dependency knowledge.

VIII. CONCLUSION

In this paper, we propose an automated approach leveraging static dependency analysis to improve GUI exploration. The identified three types of dependencies are captured in a structural model and are used to guide the widgets selection during exploration. We developed a prototype and conducted experiments on 70 open-source apps. The results verify the effectiveness of GESDA in exploration and crash detection.

As a preliminary work, our approach and tool still have potentials for development such as constructing a more comprehensive model by involving components like drawers and fragments. In addition, the limitations discussed require resolved in the future.

References

[1] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (E)," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 429–440.

[2] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[3] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2018.

[4] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou, "Behavioral analysis of android applications using automated instrumentation," in *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*. IEEE, 2013, pp. 182–187.

[5] S. N. Dutia, T. H. Oh, and Y. H. Oh, "Developing automated input generator for android mobile device to evaluate malware behavior," in *Proceedings of the 4th Annual ACM Conference on Research in Information Technology*, 2015, pp. 43–43.

[6] A. Memon, I. Banerjee, B. N. Nguyen, and B. Robbins, "The first decade of gui ripping: Extensions, applications, and broader impacts," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 11–20.

[7] A. D. C. Guerreiro, "Gui ripping of ios mobile applications," *Hamburg University of*, 2016.

[8] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1070–1073.

[9] Monkey. (2020) Monkey. [Online]. Available: http://developer.android.com/tools/help/monkey.html

[10] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 245–256.

[11] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 641–660.

[12] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of android test generation tools in industrial cases," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 738–748.

[13] L. Sell, M. Auer, C. Frädrich, M. Gruber, P. Werli, and G. Fraser, "An empirical evaluation of search algorithms for app testing," in *IFIP International Conference on Testing Software and Systems*. Springer, 2019, pp. 123–139.

[14] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Towards getting there in an industrial case," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 253–262.

[15] Gesda and evaluation data. [Online]. Available: https://github.com/Agesda/gesda20

[16] F-Droid. [Online]. Available: https://f-droid.org/en/

[17] Y. Li, Z. Yang, Y. Guo, and X. Chen, "A deep learning based approach to automated android app testing," *arXiv preprint arXiv:1901.02633*, 2019.

[18] Android activity lifecycle. [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle.html

[19] Soot. [Online]. Available: https://sable.github.io/soot/

[20] Apktool. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[21] Flowdroid. [Online]. Available: https://blogs.uni-paderborn.de/sse/tools/flowdroid/

[22] Ui automator. [Online]. Available: https://developer.android.com/training/testing/ui-automator

[23] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 37–54.

[24] ——, "Andarwin: Scalable detection of semantically similar android applications," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 182–199.

[25] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 280–291.

[26] P. Gadient, M. Ghafari, P. Frischknecht, and O. Nierstrasz, "Security code smells in android icc," *Empirical software engineering*, vol. 24, no. 5, pp. 3046–3076, 2019.

[27] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android applications," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 559–570.

[28] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-droid: Automated system input generation for android applications," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 461–471.

[29] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for android (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 658–668.

[30] D. Lai and J. Rubin, "Goal-driven exploration for android applications," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, 2019, pp. 115–127.

[31] D. Octeau, P. D. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013, pp. 543–558.

[32] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. D. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 77–88.

[33] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: automated generation of storyboard for android apps," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 596–607.

[34] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for android apps," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013, pp. 224–234.

[35] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan, "PUMA: programmable ui-automation for large-scale dynamic analysis of mobile apps," in *The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'14, Bretton Woods, NH, USA, June 16-19, 2014*, 2014, pp. 204–217.

[36] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon, "Using GUI ripping for automated testing of android applications," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, 2012, pp. 258–261.

[37] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE Software*, pp. 53–59, 2015.

[38] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.

[39] T. Su, L. Fan, S. Chen, Y. Liu, L. Xu, G. Pu, and Z. Su, "Why my app crashes understanding and benchmarking framework-specific exceptions of android apps," *IEEE Transactions on Software Engineering*, p. to appear, 2020.

[40] T. Gu, C. Sun, X. Ma, C. Cao, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[41] J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, and B. Liang, "Multiple-entry testing of android applications by constructing activity launching contexts," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.

[42] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, "Efficiently manifesting asynchronous programming errors in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated*

Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, 2018, pp. 486–497.

[43] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 408–419.

[44] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Are we really there yet in an industrial case?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 987–992.