# Ape$^+$: A Faster Ape with Static Model Guided Exploration

by

Yaxin Cheng

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Math
in
Computer Science

Waterloo, Ontario, Canada, 2022

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

**Abstract**

Ape is the state-of-the-art Android GUI testing tool, which implements a dynamic model internally to guide the testing process. At the time of writing, Ape was one of the most effective Android testing tools. Ape's interactions with Android devices partially rely on private APIs, which made it difficult to support newer Android versions. Aiming to solve this problem, we adopted Appium as the interaction layer. However, the introduction of Appium distorted Ape into a server-client structure, which brought a huge overhead and severely affected the efficiency. Besides, Ape naturally tries to test all widgets. Nevertheless, in scenarios where an application only needs to be partially tested, such strategy limits the effectiveness due to the inability to prioritize activities of interest.

In this study, we introduce Ape$^+$, which boosts the efficiency of Ape but avoids using private Android APIs. We reconstruct Ape as a monolithic on-device testing tool by replacing Appium, the communication layer between Ape and the device, with UiAutomator. We solved technical challenges, such as supporting drag function and fetching current activity names, and experiments show that efficiency improvements among the applications are between 10% to 40% compare to Ape with Appium.

We also analyze different static analyses tools to find the one whose static model is informative enough to bring extra knowledge to Ape for activity prioritization. Using instrumentation, we improve the accuracy of widget matching, which is an essential step to bridge the gap between the dynamic model and the static one and combine both synergistically. We also introduce a priority decay strategy to mitigate false information produced by static analysis, and a path finding algorithm to help Ape$^+$ navigate between activities using both models. Our experiments show, for two selected applications with informative models, Ape$^+$ is able to cover every activity 37% and 57% faster.

We believe that Ape$^+$ is a decent testbed with maintainability and extensibility for conducting research on automated Android GUI testing in the future.

**Acknowledgements**

I would like to thank my supervisor Chengnian Sun for the support and guidance on my study and life. I would also like to thank Yongqian Tian for his help on properly phrasing this academic thesis.

I would like to thank Professor Aafer and Professor Nagappan for reviewing my thesis and providing feedback.

I thank my grandparents and other family members for their love and support. My sincere thanks also go to Hinna Zhu and Mingjie Zhu for their company and love.

# Table of Contents

# List of Figures

# List of Listings

# List of Tables

# Chapter 1

# Introduction

Android applications have been increasingly popular over the decade. As the demands of developing Android applications soar, the technology to automatically test these applications also evolve rapidly [12][28][26][9][27][23][24][30]. One of the main aspects of Android testing focuses on testing the Graphic User Interface (GUI), because Android applications are essentially a collection of interfaces which users interact with directly. Any bug in the GUI would be directly exposed to users and exert negative influence upon the applications. Before automatic testing tools, testers used to either manually interact with the GUI or developing scripts to automate the process. This is extremely inefficient, due to the drastically varying application features and structures. Testers often have to upgrade the scripts after each version release. Outdated scripts cannot be trusted toward providing strong guarantees on the applications, as they may not be able to interact with the target applications correctly after updates. Requiring manual testing on applications is even worse. Due to the repetitiveness and strenuousness of this work, humans are prone to mistakes. It is exhausting and nearly-impossible work to develop scripts that resist updates and also provide strong bug-free guarantees. Eventually, this problem was eased with the birth of automatic testing tools as a complement to human interactions and testing scripts. The automatic testing tools analyze the GUI on display and automatically generate actions to interact with it. These tools lift the heavy burden from testers, as no predefined action scripts are required Also, they can potentially cover every aspect of an application given enough time and resources.

Ape [12] is a state-of-art automated Android testing tool focusing on GUI testing. Similar to other Android GUI testing tools [26][28][19][7][22], Ape is model-based. It implements a dynamic model internally, which supervises its interactions with the application in testing. The model is a collection of algorithms that accept and analyze UI hierarchy on

screen, and then output a sequence of actions. According to the paper, Ape was thoroughly tested with 1,316 applications from Google Play Store, and ended up finding 537 unique crashes in 42 error types from 281 of them [12]. In spite of the strong performance on bug-finding, Ape has a limited support for newer Android versions, because of its reliance on private APIs to interact with Android devices. Internally, these private APIs are called using reflections. However, Android is disabling the use of reflection since Android 11 [8]. This disabled Ape from supporting any Android OS above version 11. In order to solve this issue, we introduced Appium [5] as the communication layer, which does not use any private APIs. Appium is an open sourced mobile testing framework, and equips a series of functions to interact with an Android device. It provides functions, such as click, to deliver user interaction events to the device. Using Appium as a bridge, Ape and the device form a server-client structure, where Ape is the server that handles the logic computation and the Android device is the client who loyally acts with the server's instruction. However, this change brought in efficiency and maintainability problems to Ape. The sever-client structure is unnecessary, with heavy overhead due to the communication over the network. Given testing is almost always conducted with a limited time and resources, it is necessary to reduce the time wasted on communication. Also, it is more difficult to maintain and debug potential bugs in Ape, because of the server-client structure. The current strategy completely relies on logs, which is clearly ineffective.

Besides the drawbacks above, Ape can also suffer from effectiveness problems in scenarios, where an application only needs to be partially tested. Ape was initially designed to interact with every possible activities and widgets, and it does not have the ability to prioritize testing on specific activities. Consequentially, such design becomes less effective when a subset of all widgets is interesting. For example, when a new activity is introduced to some well-tested application, the preferred testing strategy would be focusing on the newly added activity, instead of testing every activity all over again. Especially under the limited time and resources, it would be a huge advantage if Ape could interact with the devices faster and be able to focus on a subset of activities of interest.

In this study, we aim to address the aforementioned drawbacks to upgrade Ape in terms of maintainability, efficiency, and effectiveness. More specifically, we replace Appium with UiAutomator [32], which runs completely on the device and provides APIs to interact with the device with faster speed. In addition, UiAutomator, as a first-party library, enables debuggers to be executed directly from the IDEs. We attempt to introduce a static model to Ape as prior knowledge about the target application, so that Ape can prioritize its testing based on the information from the static model. We chose Frontmatter, after examining multiple static analyses tools, including Frontmatter [18], Gator [29], ICCBot [37], and GoalExplorer [19], because it is the most stable and includes all the necessary features.

With the static model introduced, we implement a path finding algorithm that enables Ape to quickly navigate towards a specified activity. This algorithm gains knowledge about widgets from both dynamic model and static model, which assists Ape to rearrange the testing resources to a subset of activities of interest, and eventually enhance the testing effectiveness. We name the upgraded version Ape$^+$, as it shares the benefits of Ape, but also addresses its drawbacks with faster speed and static-model-guided exploration strategy.

There were a few challenges in making the enhancements. First and foremost, moving away from Appium means losing all the convenient tool functions provided by Appium. Interacting with an Android device through UiAutomator requires more cautions, as its APIs are more general and lower-level in comparison to Appium's. For example, `scroll` action is not provided by UiAutomator, and `grantRuntimePermission` function would fail silently if OS version is under 28. Moreover, introducing a static model as a guidance sounds simple, but it actually requires some efforts. Widget matching between the static model and the dynamic model is indispensable and can be problematic due to lack of shared information. There can also be information omission and false information spread across components in static models, which need to be properly handled. We will share the detailed information about these challenges, and how we crafted our solution to either solve or alleviate them.

Overall, this thesis answers the following research questions:

- **RQ1**: How much efficiency improvement UiAutomator has brought to Appium-based Ape?
- **RQ2**: Does informative static models enhance the efficiency of Ape$^+$?
- **RQ3**: Does false information in static models diminish the efficiency of Ape$^+$?
- **RQ4**: How static models changed the behaviours of Ape$^+$?

Our experiment results show promising outcome for the enhancements applied. The UiAutomator version of Ape has brought a roughly 10% to 30% speed bump compare to the Appium version. By conducting several experiments, we have also proven that an informative static model is a good aid to guide the exploration, and the false information contained in the static model is harmful to the testing.

We make the following major contributions:

1. We improved the efficiency of Ape by migrating from Appium to UiAutomator.

2. We analyzed multiple static analyses tools and discussed their advantages and disadvantages.

3. We introduced a novel strategy to enhance widget-matching between dynamic models and static models.

4. We designed a path finding algorithm using both dynamic model and static model to reach a specified destination.

In the following chapters, this thesis will start from introducing some background knowledge in §2. After that, we describe our approaches to address Ape's existing problems in §3. Specifically designed experiments were conducted to evaluate Ape$^{+}$ and answer the research questions, which will be detailed in §4. Following the evaluation, we introduce some related work in §5, and discuss the threats to validity and potential future work in §6. Finally, we have a conclusion for this thesis in §7.

# Chapter 2

# Background

## 2.1 Android Testing

Android applications are GUI-based applications. Each application consists of one or multiple activities [17] that serve different functionalities. One activity approximately equals to a single page. For example, a general weather application probably has a "city weather" activity to display weather information in a selected city, and a "forecast" activity that presents weather forecasts in the coming days. Each activity contains a number of interactive UI elements, called widgets [4], such as buttons, text fields, and switches. As the popularity of Android apps rises, it makes testing these GUI interfaces indispensable. However, due to the complex nature of GUIs, it is not an easy task to develop tests that cover every essential aspect. Testers often manually write testing scripts to generate interactions to hopefully capture bugs. Nevertheless, this is strenuous and ineffective in covering multiple different scenarios. To lift the burden, researchers came up with the idea of automatic testing tools, which explores and tests the GUI hierarchy automatically. The goal for an automatic testing tool is inherently the same, which is interacting with as many widgets as possible, hoping to trigger bugs. Without the predefined interaction sequences, automatic testing tools generate actions based on the current GUI hierarchy on display. Consequentially, even the application changes significantly in the future, it would not affect the functionality of such tools.

The state-of-the-art automatic testing tools are generally model-based [12][19][26][28][36]. A model is an encapsulation of the exploration algorithms. It digests the information about the current GUI hierarchy on screen, and outputs actions to interact with the widgets. The output actions are often sorted in a manner that maximizes the possibility of triggering

bugs. A model can be generated either statically or dynamically. As generated from two different paths, they thereby offer unique advantages and also carry different drawbacks.

## 2.2 Event Listeners

Event listeners [15] are callback methods associated to widgets. When interacting with a widget, *e.g.* touch or drag, it will trigger the related listener method, such as `onClick` or `onTouch`. The widget invoking a listener method would be passed in to the listener method as a parameter. Inside the listener method, it knows who the invoker is. Developers often program the logic of activity transitions in these listener methods, so that the transitions can be triggered by interacting with the widgets. For each type of interaction, a widget can have at most one listener method. However, each event listener can be potentially associated with multiple widgets. In that scenario, the listener method implements conditional statements, *e.g.* switch-statements, to distinguish each widget, and performs different functions when invoked by different widgets.

## 2.3 Static Models

Static models are constructed through applying static analyses on layout resource files [20] and source code. Because widgets and GUI hierarchy are either programmed into the source code, or defined in the XML layout files. Through the analyses on them, the static analyses tool can obtain a general knowledge about the GUI of the application, including activities, connectivity of activities, and triggers that cause transitions between activities.

If we pack up such knowledge as a model, it provides testing tools a brief overview of how to test the target application. The most obvious benefit of this is the ability to predict possible transitions triggered by interactions. Because the static model includes the information about the trigger widgets, the testing tool can purposefully interact with them or avoid them. As a direct result, the testing tools would be able to stay at an activity longer if they want, or leave an activity more quickly. Since the static model also introduces such information about beyond the current activity, the testing tools' testing strategy can be adapted to target for gains in the long run. Based on the feature, the transition-aware functionality can be extended as a navigation for the testing tools to travel between activities. Eventually, it should enable testing tools to quickly reach any activity.

However, there are disadvantages lie in the nature of static models. One of the most significant problems is the omissions of information. As static models are generated based on the resource layout files and source code, they are essentially static representations of the applications. They cannot normally represent the complex behaviours of applications at runtime, therefore they are naturally over-abstracted. The direct reflection of the over-abstraction is missing of information, because during the static analyses, some runtime information cannot be discovered. For instance, the coordinate location for a widget is assigned at runtime, which cannot be detected by static analyses. Moreover, static analyses can introduce plenteous false information. Intrinsically, static analyses are based on certain assumptions to generate a result that closely represents an application. These assumptions can be biased under certain scenarios that lead to false information. For example, Gator [29] detects listeners based on certain keywords in the method names. Thus, it cannot detect some listeners using newer keywords introduced in newer Android versions. Overall, if some information is false or missing in the static model, it may misguide the testing tools into wrong paths.

## 2.4 Dynamic Models

Dynamic models are generated at runtime through constantly refining after collecting GUI information on the fly [12][7][26][36]. It starts with an empty model, and after each interaction, it refines the model with the newest GUI hierarchy information, along with a record of the previously triggered action and its outcome. As the testing process moves on, the dynamic models cover new widgets and transitions, and eventually become more robust and accurate. Intrinsically, a dynamic model is an accurate documentation of the GUI hierarchy, widgets, and transitions at runtime.

As a loyal recorder of application information, a dynamic model can give testing tools the most trustworthy information about actions and their expected outcomes. Using a dynamic model, a testing tool can have a clear vision over the history. For any visited activity, it can find the shortest path to it with ease. Moreover, building a dynamic model is much simpler and time-saving than building static models. For every received GUI information, dynamic models can be refined by capturing the key information and update related components. As the models are built gradually, it is much faster in comparison to building static models, where numerous of the layout resource files and source code is analyzed.

Nevertheless, dynamic models are not impeccable. One inevitable drawback is the tremendous memory consumption. As a type of fine-grained model, dynamic models ab-

stract each activity into multiple states to provide more accurate representations of the applications. For example, one activity with a hidden menu drawer may have a "menu opened" state and a "menu closed" state. As long as the available widgets differ, the same activity can be essentially stored as multiple different states. Therefore, saving all the different GUI and widget information for each different state inevitably drives up the memory consumption. In addition, a dynamic model does not provide information about unvisited activities. The direct outcome of such limitation is testing tools using dynamic models is unable to dive into the unknown and jump between activities easily. Due to the lack of overview, the testing tools can only make short-sighted strategies, while lack the ability to plan for long term gains. Hence, despite the precise information on the visited activities, dynamic models are not adept at guiding the test process to unvisited destinations.

## 2.5 Ape

Ape [12] is a state-of-art dynamic mode based Android testing tool. Inside of Ape, it implements a testing cycle to test applications. Such cycle is repeated multiple times to cover as many behaviors of the applications under test as possible.

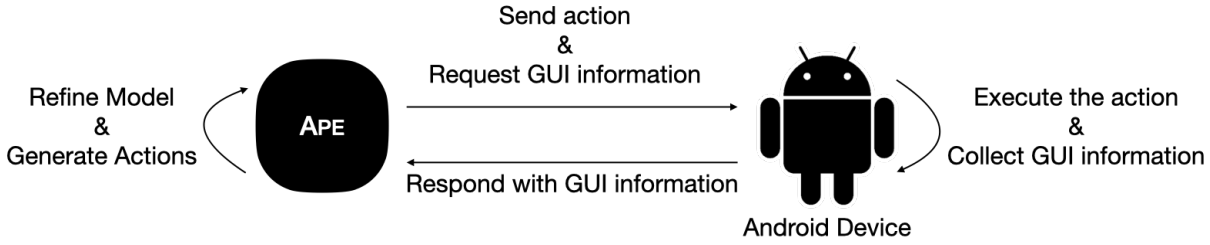
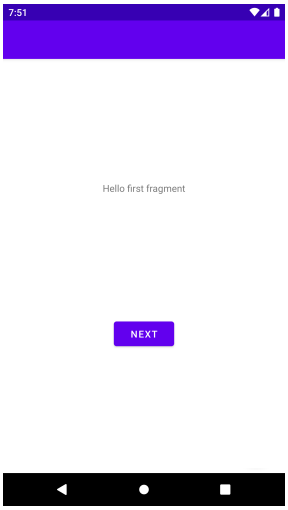



Figure 2.1: Ape Testing Cycle

As Figure 2.1 shows, at the beginning of every cycle, Ape retrieves the GUI information on display from the Android device. Upon receiving such information, Ape digests it and refines the dynamic model. After the refinement, the dynamic model generates a list of actions targeting on widgets that are showing on screen. To maximize the possible outcomes, these actions will be sorted based on multiple properties, such as the action type and the number of executions of such action. Ape prefers types like CLICK and SCROLL, and the less executed actions are also preferred. Among the most preferred actions, Ape randomly chose one and send it to the Android device. After the action being executed by the device, Ape would again request GUI hierarchy information from




```
<hierarchy rotation="0">
    <node index="0" text="" resource-id="" class="android.widget.FrameLayout"
     package="com.example.example" content-desc="" checkable="false" checked="false"
     clickable="false" enabled="true" focusable="false" focused="false" scrollable="false"
     long-clickable="false" password="false" selected="false" bounds="[0,0][1440,2392]">
        <node index="0" text="" resource-id="" class="android.view.ViewGroup"
         package="com.example.example" content-desc="" checkable="false" checked="false"
         clickable="false" enabled="true" focusable="false" focused="false" scrollable="false"
         long-clickable="false" password="false" selected="false" bounds="[0,84][1440,2392]">
            <node index="0" text="Hello first fragment" resource-id=""
             class="android.widget.TextView" package="com.example.example" content-desc=""
             checkable="false" checked="false" clickable="false" enabled="true" focusable="false"
             focused="false" scrollable="false" long-clickable="false" password="false"
             selected="false" bounds="[56,135][526,228]" />
            <node index="1" text="NEXT" resource-id="com.example.example:id/next_btn"
             class="android.widget.Button" package="com.example.example" content-desc=""
             checkable="false" checked="false" clickable="true" enabled="true" focusable="true"
             focused="false" scrollable="false" long-clickable="false" password="false"
             selected="false" bounds="[566,1598][874,1766]" />
        </node>
    </node>
</hierarchy>
```

Figure 2.2: Example GUI and its XML

the device to prepare for the next cycle. This process keeps repeating until a pre-specified goal, such as timeout, is reached.

Ape implements a dynamic model internally to instruct the exploration process. Initially, Ape starts with an empty model. After every testing cycle, Ape refines the model with the updated GUI hierarchy information. GUI hierarchy information is collected through UiAutomator [32]. UiAutomator is an official tool to write GUI test cases for Android application. It provides the functionality to interact with visible widgets on the device, and also the ability to search a certain widget. Ape is using the `dumpWindowHierarchy` function [33] provided by UiAutomator to dump the GUI hierarchy on screen into text in XML formatted text. Then, Ape analyzes the XML to obtain the GUI information.

There is an example GUI along with its XML in Figure 2.2. From the sample XML, it is easy to identify all the widgets: a `FrameLayout`, a `ViewGroup`, a `TextView`, and a `Button`. Each widget has some properties associated, such as resource ids, class names, text content, the bounds, their coordinates on screen, and interactions they each support. Moreover, the XML file also provides hierarchical information. For instance, the `Button`, with the resource id `next_btn`, is a child widget of the `ViewGroup`, and the `ViewGroup` is a child of the `FrameLayout`. Based on the above-mentioned information, Ape is able to create class instances for all widgets along with their properties. Each instance is linked with its parent instance and children instances. Therefore, all the widgets in an activity are effectively connected as a tree, which is later referred to as a GUI tree.

Each GUI tree is strictly associated with one activity, and one activity can potentially

have more than one GUI trees. Different GUI trees belonging to the same activity represent different states of that activity. Between these GUI trees, there are also direct or indirect connections by actions, transitions from one state to another are triggered by actions. These transitions can either be inter-activity or interstates within the same activity. Thus, the connected states and the transitions form a graph, which enables the exploration from one state to any other state.

During testing, Ape generates and arranges actions to maximize the opportunity of discovering bugs (*e.g.*, crashes and not-respondings). Based on the property information in GUI trees, Ape is able to identify all the supported actions, and thereby assembles a series of actions targeting these widgets. Randomly executing these widgets is not effective enough, so Ape also ranks the actions with reference to the action types and execution counts. Finally, Ape picks the most preferred action to execute, and puts the rest of them into a queue for the next round.

# Chapter 3

# Approach

Our approach improves Ape in two different aspects. After the initial attempt to introduce Appium as the communication layer to eliminate Ape's dependency on private APIs, we detected a noticeable performance decrease. Aiming to solve the performance issue, we first introduce UiAutomator [32] as the new communication layer and transform Ape back into the monolithic structure. With UiAutomator, Ape runs completely on the device, which removes the overhead of messaging over the network. Also, because of UiAutomator, Ape does not rely on any private APIs, so that it provides better supports to newer Android versions. Most importantly, the monolithic structure enhances Ape's maintainability by supporting debuggers. Secondly, we introduce a static model obtained from running a static analysis over on an Android app to aid the exploration of the activities inside the Android app. This static model should be informative enough to provide activity transitional information with their triggers. Based on the information, Ape should be able to find a path to quickly reach a specified destination activity and conduct testing there. We name our solution Ape$^{+}$. In this chapter, we are going to present the design and implementation details about the enhancements.

## 3.1 Enhancing Testing Efficiency

To understand the reasoning behind replacing Appium, we need to first understand how Appium works as part of Ape. As mentioned before, Ape runs testing cycles which interactively exchange information with the device. The exchange happens through Appium, which serves as a bridge connecting the Android device and Ape. As the Figure 3.1 shows, Appium is installed on the Android device. All the actions generated by Ape are sent to

Appium first. Appium then interprets the actions and executes related functions to trigger UI interactions on the Android device. When Ape tries to fetch the UI information, it sends a request to Appium, and Appium activates related functions on the Android device and returns the result back.

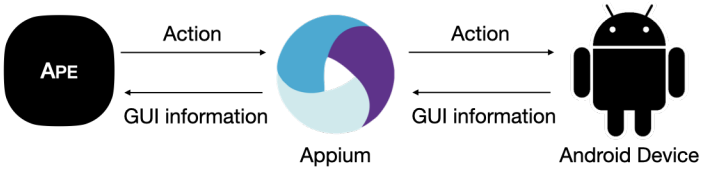



Figure 3.1: Ape Workflow with Appium

This essentially forms a server-client structure, where Ape, with the model and logic, is on the server side processing the GUI information and generating next actions; the Android device is the client that accepts and executes instructions from the server side. The communication between server and client is made possible through Appium, which uses Json-RPC calls under the hood. Each interaction happening between the server and client needs to go through Appium, which adds up a huge overhead. Appium was designed in a way to maximize the compatibility, as it supports both Android devices and iOS devices. However, the performance sacrifice for such compatibility is a waste for Ape, which only supports testing Android applications. In addition, because of the server-client structure, debugging Ape has been difficult, as the only viable way to debug Ape is through collecting and analyzing logs.

Consequentially, one straightforward answer to address these issues is to eliminate Appium and the server-client structure. Ape should obtain the ability to run completely on the device and to interact with the device directly without Appium in the middle. Since the elimination of the communication-over-network, the overhead is essentially minimized and the efficiency should be enhanced. Moreover, Ape should also support more efficient debugging tools, such as debuggers, which enables developers to easily pause the execution and inspect the ongoing process. Aiming to satisfy the requirements listed, we adopted UiAutomator.

UiAutomator [32] is an Android UI testing framework that supports interactions between widgets. It is a first-party library from Google, and it runs completely on the device. Ape using UiAutomator is a monolithic application runs on the device, as a result debuggers from Android Studio [3] or IntelliJ [16] can be used to inspect it. UiAutomator also provides APIs to trigger interactive actions on the device, including but not limited to

click, pinch, and swipe. There are also APIs that captures the current UI hierarchy on screen and dumps the information into XML files. We replaced Appium with UiAutomator to allow Ape to be debugged with a debugger and be executed on-device after rebuilding Ape as a monolithic application. In the following subsections, we discuss a few challenges we have faced in the process of replacing Appium with UiAutomator, and also provide our solutions to each of them.

### 3.1.1 Supporting Scrolling

To replace Appium, UiAutomator should take care of all the user interactions with the Android device that were previously handled by Appium. UiAutomator provides a class named `UiDevice`, which wraps multiple functions to generate user interactive events. Using a singleton instance of `UiDevice`, we could achieve click, swipe, rotate, and more. However, scroll is not provided by default in `UiDevice`. A scroll function should accept a list of coordinate points, then simulates the behaviour where the user puts their finger at the first point, then moves it from one point to the next, and eventually lifts that finger at the last point. It is an indispensable interaction users often do with any scroll-able views. For example, there are table views displaying a list of items, and users have to scroll to see all the items.

This important function is not provided, but we must have it. We first thought about finding a similar function, then make modifications based on it. There is a `swipe` function, which sounds similar, as scroll can be considered a variation of long swipe. However, this function is significantly different, as it simulates the event where the user touches a point with their finger, then move to a direction with their finger quickly lifting from the screen. Using `swipe` to replace scroll makes Ape unable to pause the scroll in the moving process or control the scrolling speed. As this approach does not solve the problem, we decided to build our own scroll function. We discovered a class called `MotionEvent` [25]. It can simulate user events such as touch down, move, and touch up. Hence, the scroll action can be broken down into multiple steps, and simulate each respective step using a `MotionEvent`. The first step would be touching down `MotionEvent` at the start point. Then we use multiple moving `MotionEvent` to move from one point to the next point. Finally, after reaching the last point, we send a touching up `MotionEvent` to lift the finger. Using the combination of `MotionEvent`s, we successfully simulate the scrolling behaviour.

### 3.1.2 Detecting Current Activity

GUI trees form the foundation for Ape's dynamic model. For each GUI tree, it is required to be associated with an activity. Therefore, it is important for Ape to correctly identify the current activity name. However, it is not an easy task for the current UiAutomator to detect the active activity on screen, since the `getCurrentActivityName` in `UiDevice` has been deprecated. Any function calls to this function will simply end up with an incorrect value. Besides this one function, we have also examined other proposed solutions found on a stack overflow post,[1] such as using `ActivityManager` and `ActivityThread`. But unfortunately, none of them worked.

Ultimately, we solved this challenge by dumping the system information through ADB [2], from which we capture the focused activity name using a regex. However, this brought in another problem. If Ape sends a request to fetch the activity name right before an activity transition, the activity name fetched could be the name of the source activity. Then Ape would associate the name of the source activity with the GUI tree from the target activity, which leads to inaccuracy. This is because ADB is a separate process, which does not pause the transition when fetching the activity name. It is not aware that there is an activity transition going on either. When ADB receives the request before the transition finishes, it would include the activity name of the source activity in the response. But when Ape receives the response, the transition is likely finished, which causes a mismatch of the activity names. We contemplated the problem and proposed two potential solutions. We firstly tried to subscribe to a callback service from `AccessibilityService` [1]. As the `AccessibilityService` is able to notify Ape about a window change, so that we can conservatively use the same activity name until a window is changed. We expect this call back to be the correct timing to refresh the activity name. Whenever a call back arrives about window changing, Ape would try to update the current activity name. However, this service was proven to be not precise enough. It does not always trigger the call back in time, and sometimes even not trigger the call back at all. This made the problem worse as the call back are unpredictable. As the first solution providing no sign of easing the problem, we decided to send a fetch request with a delay of 25 milliseconds after executing each action. The choice of 25 milliseconds was based on an experiment. We used a binary search strategy to narrow down the possible durations to complete an activity transition after executing an action. Eventually, we found that 25 milliseconds is the shortest time that is long enough to guarantee the end of a transition. Adding the delay ensures all the requests are sent after a transition, thus Ape is able to obtain the correct activity name on screen.

[1] Stack Overflow:https://stackoverflow.com/questions/11411395

## 3.2 Enhancing Testing Effectiveness

We have extended the Ape with UiAutomator to support static-model-guided exploration, so that it can rearrange the time and resources to test the activities of interest. We provide Ape$^+$ an informative static model, which is generated using a static analyses tool called Frontmatter [18]. Using this static model, Ape$^+$ prioritizes actions that lead the exploration closer to a specified target activity.

### 3.2.1 Overall Workflow

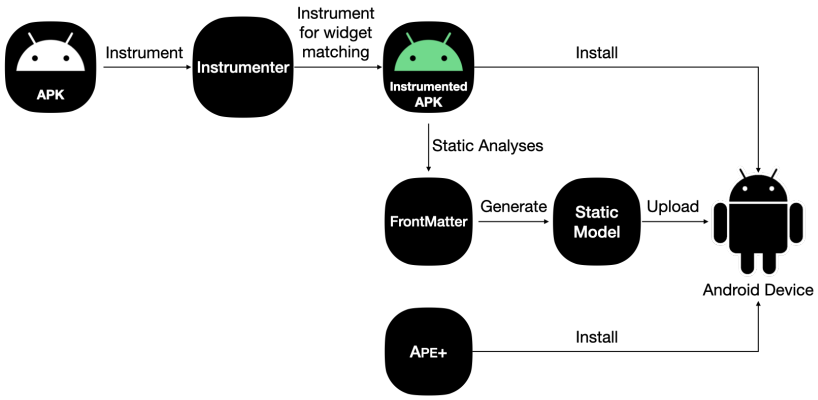



Figure 3.2: Ape$^+$ Workflow

This subsection talks about the general workflow of Ape$^+$. From Figure 3.2, the first step is to instrument the APK file for widget matching. This is our proposed optimization to the collaborations between static models and dynamic models. The instrumentation generates an instrumented APK, which will then be analyzed by Frontmatter [18] and also be installed on the Android device. Frontmatter generates a static model after performing static analyses on the instrumented APK. This static model will be uploaded to the device. Lastly, Ape$^+$ will be installed on that device, and the testing process starts.

### 3.2.2 Why Frontmatter

This subsection explains why we chose Frontmatter over other options, before diving into the implementation details. Frontmatter by far generates the model that best suits our

needs. It satisfied all the requirements we had: GUI hierarchy analysis, transition detection, and precise trigger identification. We have also experimented other static analyses tools, such as Gator [29], ICC-Bot [37], and GoalExplorer [19]. After experimenting, we have found that each candidate has their unique features and shortcomings.

Table 3.1: Feature Comparisons between Static Analyses Tools

| | Frontmatter | Gator | GoalExplorer | ICC-Bot |
|---|---|---|---|---|
| GUI Hierarchy Analysis | ✓ | ✓ | ✓ | × |
| Transition Identification | ✓ | ✓ | ✓ | ✓ |
| Trigger Listener Detection | ✓ | ✓ | ✓ | ✓ |
| Trigger Widget Detection | ✓ | × | ✓ | × |

Table 3.1 shows all the static analyses tools and their related features. First, Gator produces Window Transition Graphs, which contains GUI hierarchy information and transition information. However, the transitions in Window Transition Graphs only have listeners as the triggers, which is coarse-grained. Because a listener method can be potentially associated with multiple widgets, and implement different conditional branches to handle each widget differently. So that we cannot naively assume any widget associate with this listener would be able to trigger the transition. Besides, for some complex applications, Gator would crash during the analysis process and output no static model. ICC-Bot was not our first choice, because it does not contain the necessary GUI hierarchy information. In spite of its enhacement on detecting transitions between components in a fine-grained level, the lack of GUI hierarchy information makes it difficult to fit into Ape$^+$'s use case. Lastly, Screen Transition Graph from GoalExplorer could have been the most promising candidate. Nevertheless, a bug in their tool prevented us from generating any meaningful static model out of it. After excluding the three static analyses tools, Frontmatter remains the only candidate with all the required features. Frontmatter, besides satisfying the feature requirements, also shines with its trigger detection algorithm. It finds not only the trigger listeners but also the specific transition triggering branches. Frontmatter then associates each such branch with the widgets appear in the conditional statement. In this way, it narrows down the potential trigger widgets to the one that is related to the triggering branch. It does not solve the false positive completely, but it largely eases the problem. Finally, Frontmatter is the most stable among all the tools. For over one-hundred applications we tested, Frontmatter faithfully produces models for all the applications without crashes. In conclusion, Frontmatter was chosen because it is the most stable and more

fine-grained option over all other options, which provides GUI hierarchy analysis, trigger detection, and transition identification.

### 3.2.3 Frontmatter's Static Model

A static model Frontmatter generates is hierarchical JSON file, with three parts at the top level: `packageName`, `activities`, and `transitions`. The `packageName` section indicates the package name belonging to the target application. The `activities` section is a big map containing all the activities and their associated widgets. All the widgets, along with their properties, are listed in a tree structure under the related activities. The `transitions` section contains a list of triples, where each triple is a source activity name, target activity name, and a list of ids of trigger widgets.

```
{
  "packageName": "com.example.Example",
  "activities": [ {
      "name": "com.example.Example.IntroActivity",
      "layouts": [ {
          "viewClass": "android.widget.ListView",
          "resourceId": "menu_drawer",
          "guid": "m7QKijqLpW",
          "children": [ {
              "viewClass": "android.widget.Button",
              "resourceId": "next_btn",
              "contentDescription": "",
              "text": "Next",
              "guid": "PH7OIVAKBf",
          } ]
      } ]
    },
    {
      "name": "com.example.Example.MainActivity",
      "layout": [ ]
    }
  ],
  "transitions": [ {
      "scr": "com.example.Example.IntroActivity",
      "tar": "com.example.Example.MainActivity",
      "trigger": [ "PH7OIVAKBf" ]
  } ]
}
```

Listing 3.1: Example of Static Model

A simplified static model looks like Listing 3.1. This application has a package name "com.example.Example". There are two activities in this sample application. The first activity, named `com.example.Example.IntroActivity`, has a button with resource id `next_btn` under a drawer widget with resource id `menu_drawer`. The second activity,

named `com.example.Example.MainActivity`, has no widget. Both the drawer and the button have the field `guid`, that does not appear in the dynamic model. It is a globally unique id generated in the static analyses process, but apparently it does not exist in the actual widget, so it cannot be used to identify widgets at runtime. it also lists a possible transition in the `transition` section, from `com.example.Example.IntroActivity` to `com.example.Example.MainActivity`. The `trigger` part includes one available trigger, whose guid is "PH70IVAKBf". If we look up this guid in the `activities` section, we would find the button, with the resource id `next_btn`. It means that this button is the trigger for the transition from `com.example.Example.IntroActivity` to `com.example.Example.MainActivity`.

### 3.2.4 Widget Matching

Widget matching is an essential but challenging step to achieve the goal of making two models collaborate. To interact with a specific widget described in the static model, Ape$^+$ needs to first find such widget in the dynamic model. The process of finding the widget in the dynamic model using the information provided in the static model is called "widget matching". Practically, there exists no way to match two widgets with absolute confidence, mainly because of the lack of shared information between dynamic models and static models. Especially, there does not exist a universally unique identifier that can be used to distinguish widgets from each other.

The current strategy for matching widgets is to calculate a confidence score based on the similarities of multiple properties, such as "resource id" [14], "text content", and "class name". A higher confidence score means a higher possiblity that the two widgets match. For any two giving widgets, one from the static model and one from the dynamic model, each pair of matched property values contributes certain points to the overall confidence score. Because the likelihoods of matching properties differ, the weights assigned to the matched properties are also different. Among all the properties shared between static widgets and dynamic widgets, the "resource id" has the highest weight. The reason is that "resource id" is normally unique within a certain scope [13]. Thus, matched resource ids are often a strong indicator for matched widgets. Nonetheless, resource id is often missing in widgets, as it is not a mandatory property. As a result, when multiple widgets exist without resource ids, the matching process can be less precise, due to the miss of the most important indicator.

However, the miss of resource ids can also be viewed as an opportunity to increase the match-ability. We propose a strategy is to create and supply globally unique ids to the

widgets that without resource ids. Through matching on the globally unique ids, widgets can be matched with even more confidence. Before talking about details of supplying resource ids to the widgets, it is necessary to understand where resource ids are associated with widgets. The first place is in XML layout resource files, where widgets are defined with their properties, including resource ids. The second place is in source code files where widget instances can be instantiated, then resource ids can be assigned to the newly created instances. Our strategy handles the second scenario only based on the assumption that most resource-id-less widgets defined in layout files are static and non-interactive, such as FrameLayouts or ViewGroups. For non-interactive widgets, there is not much value in matching or assigning resource ids to them.

Our solution supplies unique resource ids only to the widgets that are created through source code. There are three possible methods to instantiate a widget instance in source code:

1. `ViewBinding.inflate` [35]
2. `View.findViewById`
3. `new WidgetClass`

Method 1 inflates a widget defined in an XML layout file. It is the same as creating widgets through layout resource files, so we decided not to handle this case. Method 2 fetches and constructs a single widget instance based on a provided resource id. Since this widget already has a resource id, there is no need to supply another id to it. Finally, Method 3 creates a new widget instance using a constructor of the widget class. This is the case where our solution handles.

The process of supplying unique ids targeting method 3 is straightforward. The first step is extracting and parsing the resource file contained in the target APK file and collect a set of resource ids. The resource file, normally named as `resources.arsc`, is at the top level of the APK file, and stores all the resource related data, including resource ids. For a newly created resource id, it is essential to guarantee its uniqueness by checking in the set. Then it needs to be inserted into the resource file, so that the id can be correctly read and used in the static analyses and dynamic analyses. Next, we use FlowDroid [6] to analyze the APK file and identify all the "new" statements that are instantiating widgets. FlowDroid is a static analysis tool based on Soot [11] that is able to decompile the APK file into source code in Jimple format [34]. Using the source code, it is possible to identify, find, and modify statements and expressions. After each "new" statement, we insert a `setId(int)` statement with the newly created unique resource id as the argument. As some widgets are

created then assigned resource ids, it is important to insert our `setId(id)` statement right after new statement to avoid overriding the original ids. This is necessary to avoid potential bugs, as the original ids may be referred in other parts of the source code. Ultimately, we update the generated resource ids to resource file, and repackage the modified content into a new APK file.

We have also adopted a strategy used in ATUA [26], where they include the hierarchical information between static widgets and dynamic widgets to enhance widget matching. For example, when the two widgets match, ATUA would further recursively compare their parent nodes to gain more confidence score. However, one problem exists as Ape's GUI tree is based on compressed GUI information. In compressed GUI information, nested layouts or nested groups may be collapsed into a single layout or group. Thus, even two widgets match, their parents do not necessarily match. With knowing the existence of this difference, we implement the hierarchical match as a subsequence match. We add confidence scores based on the maximal number of matched ancestor nodes in the GUI hierarchy. Specifically, we compare the parent of a static widget to the parent of a dynamic widget. If the two nodes match, we recursively match the grandparents, otherwise, we compare the grandparent of the static widget to the parent of the dynamic widget. Eventually, when either of them reach the root, the recursion ends.

### 3.2.5 Path Finding

Once the instrumented APK file is generated, it is fed to Frontmatter to generate a static model and installed on a device for testing. Ape has the functionality to prioritize actions based on different factors to maximize the opportunity of triggering bugs. Using the similar strategy, Ape$^+$ also prioritizes the actions whose target widgets are likely leading the exploration process closer to the designated target.

In order to arrive at the activity of interest, Ape$^+$ needs to find a path from the current activity. A path is essentially a sequence of actions or static widgets that can trigger transitions. These actions and static widgets combined should trigger transitions eventually leading Ape$^+$ to the destination. Our approach consists of three steps to find the path. The first step starts with a breadth first search from the current activity using only the dynamic model. For every activity found in the process, Ape$^+$ saves its name and the actions needed to reach this activity from the current activity. This step finishes when no more activities can be found. If the target activity appears in that set, the related sequence of actions would be returned as the path. Otherwise, the second step is kicked off. Ape$^+$ extends the existing paths to all the activities found in step one using

the static model. At this phase, Ape$^+$ initiates another breadth first search starting from all the dynamically reachable activities. Using the static model, it attempts to find new and unreached activities. For each such activity, the static widgets leading to it will be appended after the sequence of actions. This phase terminates after a specified timeout or when all paths leading to target activity are found. For every path reaching the destination, it is a sequence of actions from the dynamic model plus a series of widgets from the static model. If the destination activity is accessible, step three is to find the shortest and the most reliable path among all the options. Ape$^+$ ranks each path based on the number of static transitions and the total number of transitions. The overall score is deducted for each static transition, mainly because static transitions can potentially be false. Reducing the number of static transitions would enhance the overall stability. Ape$^+$ always chooses the path with the highest score. However, if there are more than one path, then Ape$^+$ prefers the shortest one which has fewer overall transitions. If the destination remains missing after above phases, or any action is unavailable in the process, Ape$^+$ switches to the random exploration mode trying to either reach the next activity or enable the unavailable action.

```
Input  : Target, CurrentActivity, DynamicModel, StaticModel
Output: Path
DynActivities, Actions ← BFS(DynamicModel, CurrentActivity);
if Target ∈ DynActivities then
    return GetActions(Actions, Target);
else
    StaticActivities, Widgets ← BFS(StaticModel, DynActivities);
    if Target ∈ StaticActivities then
        return GetActions(Actions, Target) + GetTriggerWidgets(Widgets, Target);
    else
        RandomlyExplore(Target);
    end
end
```

**Algorithm 1:** Path Finding

When a path is determined, Ape$^+$ would explore with following its guidance. It follows the path as described in Algorithm 2 to choose the most optimal actions to perform. At each testing cycle, Ape$^+$ obtains a list of possible actions from the dynamic model after it analyzing the current GUI hierarchy. For each action, it increases the priority of each action if it matches the transition action or its target widget matches with the transition widget described in the path.

```
Input: Path, DyanmicModel
Actions ← ActionsFromModel(DynamicModel);
Transition ← GetTransition(Path);
foreach Action ← Actions do
    if Transition is Action ∧ Transition = Action then
        IncreasePriority(Action);
    else
        if Transition is Widget ∧ Match(Transition, GetTargetWidget(Action)) then
            IncreasePriority(Action);
        end
    end
end
```

**Algorithm 2:** Priority Adjusting

### 3.2.6 Mitigating False Information

From the path, Ape$^+$ is able to find the preferred action and its expected activity after triggering. However, false information is inevitable in static models, thus Ape$^+$ also needs to handle situations like false triggers. Showing in Algorithm 3, Ape$^+$ keeps track of the expected activity before executing a selected action. Unless in the random exploration mode, whenever after an action is executed, Ape$^+$ checks the reached activity against the expected activity. If the two activities match, Ape moves on to work on to the next action until it reaches the destination. If not, Ape$^+$ considers it a failure and decays the priority of that action, potentially treating it as a false positive case. With the decayed priority, such action will less likely to be chosen again. The decay is based on a modified sigmoid function to reduce the priority gradually and smoothly. The first a few failures would only mildly reduce the priority, and the rates for punishment keep increasing with the failures accumulating. However, even after a tremendous amount of failures, an action would not be blocked completely. Because the reasons of failures can be complicated, Ape$^+$ prefers not to assertively block any failed actions as false positive cases. Instead, the gradually decayed priorities leave the door open for potential future retries.

```
Input: Path
Procedure DecreasePriority():
    Input: Action
    f ← CountFailedAttempts(Action);
    if f ≠ 0 then
        Priority ← GetPriority(Action);
        Tolerance ← GetFalsePositiveTolerance();
        DecayedPriority ← Priority/(exp(f − Tolerance) + 1);
        SetPriority(DecayedPriority, Action);
    end
end
Action ← SelectAction();
ExpectedActivity ← GetTarget(Path, Action);
ExecuteAction(Action);
Activity ← GetCurrentActivity();
if Activity = ExpectedActivity then
    MoveOn(Path);
else
    DecreasePriority(Action);
end
```

**Algorithm 3:** Path Following

# Chapter 4

# Evaluation

The proposed solution was evaluated on a Linux machine with CPU 2 x Intel(R) Xeon(R) Gold 5217 CPU @ 3.00GHz and 384 GB RAM. All the tests were conducted on an Android emulator with the default hardware configuration of Nexus 6 and SDK version 24. After each round of testing, the emulator was cleared and restarted. Each test was carried out sequentially to avoid potential affects between concurrent emulators. We picked our benchmark applications from the Themis paper [31], along with randomly chosen applications from F-Droid [10]. We filtered out applications whose model has fewer than three transitions. This left us thirty applications. Then we removed five applications that cannot be run in the emulator. Eventually, there are twenty-five benchmark applications for the experiments. We initially intended to evaluate Ape$^+$ against GoalExplorer [19]. However, due to a bug in their tool, we could not get it running. Therefore, we eventually only tested Ape$^+$ against UiAutomator version of Ape (referred to as "Ape" later). Since both Apes interact with devices through UiAutomator, any difference between them should be related to the static model.

Related experiments are designed to answer the following research questions:

- **RQ1**: How much efficiency improvement UiAutomator has brought to Appium-based Ape?
- **RQ2**: Does informative static models enhance the efficiency of Ape$^+$?
- **RQ3**: Does false information in static models diminish the efficiency of Ape$^+$?
- **RQ4**: How static models changed the behaviours of Ape$^+$?

## 4.1 RQ1: How Much Efficiency Improvement UiAutomator Has Brought to Appium-Based Ape?

We designed this experiment with all the target applications for measuring the efficiency enhancement. The twenty-five testing applications cover a wide range of widgets, including but not limited to buttons, scrollviews, texts, and a variety of layouts. We have configured both Ape$^+$ and Ape using Appium to test the applications. Since there is no specific goal set for Ape$^+$, it is essentially Ape with UiAutomator. In general, the path finding algorithm would not cause any measurable impact upon the performance of Ape$^+$. As mentioned before, a cycle is the base unit to count the number of iterations Ape interacts with the device. In each cycle, Ape requests GUI hierarchy information from the device, refines its model, then executes an action. We set both Apes to test for thirty testing cycles and eighty testing cycles, so that we can see the impacts of UiAutomator in shorter term and longer term. Due to the randomness in Ape, we conduct each testing process ten times, and measure the time consumption for both Apes. Between each iteration, we clean and restart the emulator to completely wipe out any remaining data that may potentially affect the outcome. Eventually, we calculate the average time consumption and a variance for each application.

Reading the Table 4.1, for all applications, the time consumption dropped since switching to UiAutomator. However, it is also obvious that the efficiency improvement is not consistent across all applications. Taking `AndBible-3.0.286` as an example, the 30 cycles efficiency has increased 44%, as time consumption was reduced from 70.05 seconds to 39.73 seconds; and 80 cycles efficiency has increased 36%, as time consumption dropped from 148.98 seconds to 95.13 seconds. However, the same cannot be said about `nextcloud-30100090`, whose improvement is quite limited. The time consumption for 30 cycles was reduced by barely 5 seconds, which is about 11% of efficiency increase; and for 80 cycles, the time consumption was reduced by 19.09 seconds, which is around 17%. Among all the applications, the scales of such efficiency improvement vary. It is likely caused by the different testing paths and different widget distributions.

We have also found two applications, `de.hskl.contacts_1` and `wpandroid-11.3`, causing a bug in Appium constantly after roughly 30 seconds of execution. This bug would crash the experiment, and that is why we do not have a precise time consumption collected from the two applications.

In conclusion, switching to UiAutomator has been proven to enhance the performance of Ape by simplifying the structure and reducing the overhead. Even though there are improvements for all applications, the scales of enhancements are not identical. For all the

Table 4.1: Average Time Comparison between Ape with UiAutomator and Ape with Appium. The Cells With N.A. Means a Bug in Appium Crashed the Experiment. The Numbers in Parenthesis Are Variances

| | 30 cycles | | 80 cycles | |
|---|---|---|---|---|
| | UiAutomator | Appium | UiAutomator | Appium |
| amaze-file-manager-3-2-1 | 37.98 (2.80) | 53.06 (2.30) | 96.91 (18.08) | 119.14 (9.71) |
| AndBible-3.0.286 | 39.73 (4.98) | 70.50 (217.68) | 95.13 (29.96) | 148.98 (533.55) |
| AndBible-3.1.309-beta | 40.07 (2.60) | 59.90 (25.12) | 96.91 (7.87) | 144.43 (470.51) |
| AnkiDroid-2.6 | 42.98 (1.36) | 62.86 (35.95) | 91.35 (305.99) | 140.78 (279.49) |
| AnkiDroid-2.7 | 42.79 (0.82) | 65.11 (27.38) | 94.49 (6.65) | 140.96 (368.95) |
| AnkiDroid-2.9.1 | 36.39 (3.89) | 51.09 (0.55) | 97.31 (7.09) | 107.23 (6.61) |
| AnkiDroid-2.9.4 | 36.59 (1.54) | 50.84 (0.24) | 95.58 (6.41) | 106.09 (2.78) |
| AnkiDroid-2.9 | 36.11 (4.16) | 52.69 (6.71) | 95.00 (6.09) | 105.76 (0.84) |
| APV_APP_v1.0 | 39.33 (2.53) | 45.86 (1.64) | 87.23 (32.33) | 98.44 (0.91) |
| Barcode_Scanner_v4.7.8 | 42.30 (2.54) | 56.56 (2.20) | 90.10 (4.50) | 125.10 (8.34) |
| de.hskl.contacts_1 | 35.84 (3.44) | N.A | 96.85 (17.00) | N.A |
| My_Tracks_v5.6.3 | 36.14 (1.68) | 54.46 (5.34) | 82.21 (1.86) | 123.16 (7.30) |
| nextcloud-30100090 | 44.08 (3.52) | 49.30 (3.86) | 96.40 (13.01) | 115.49 (29.06) |
| Omni_Notes_v6.1.0 | 39.74 (1.91) | 50.71 (2.40) | 94.24 (14.44) | 121.07 (24.94) |
| OpenLauncher_Alpha_v0.3.1 | 37.42 (4.05) | 59.52 (0.72) | 96.91 (9.71) | 128.28 (27.76) |
| ToDont | 32.54 (0.81) | 50.90 (3.81) | 93.52 (20.85) | 121.10 (140.30) |
| wpandroid-11.3 | 40.68 (1.17) | N.A | 97.66 (28.94) | N.A |
| wpandroid-12.9 | 38.31 (4.18) | 48.36 (1.02) | 94.75 (16.81) | 117.12 (14.17) |
| wpandroid-13.1 | 37.69 (6.07) | 49.11 (2.79) | 99.09 (24.00) | 116.82 (18.89) |
| wpandroid-13.3 | 36.86 (0.39) | 49.85 (1.22) | 97.90 (21.66) | 115.33 (13.98) |
| wpandroid-13.6 | 37.23 (0.41) | 49.31 (5.42) | 97.70 (5.48) | 115.86 (50.48) |
| wpandroid-13.7 | 37.38 (0.36) | 50.51 (3.25) | 96.52 (5.36) | 115.62 (21.83) |
| wpandroid-14.9 | 38.05 (0.73) | 50.73 (1.33) | 94.84 (12.02) | 118.59 (13.70) |
| wpandroid-8.1 | 39.65 (1.76) | 57.84 (6.52) | 96.18 (4.69) | 123.15 (13.85) |
| wpandroid-9.2 | 43.76 (22.63) | 57.05 (3.69) | 103.49 (9.97) | 122.89 (9.84) |

applications tested, the improvements of efficiency roughly range from 11% to 44% for 30 cycles. For 80 cycles, the enhancements range from 9% to 36%.

## 4.2 RQ2: Does Informative Static Models Enhance the Efficiency of Ape$^+$?

The introduction of static models to Ape is to bring in prior knowledge about the target applications. In an ideal situation, the static models should contain all possible transitions between activities, and it should be able to guide Ape to quickly explore every activity. Therefore, we designed this experiment verifying wether an informative static model as prior knowledge is actually helpful for the exploration. We define an informative static model as one that contains mostly accurate information about the GUI hierarchy, activities, and transitions, with mimimal percent of false information. We manually inspected all the applications along with their static models, and picked two whose static models are informative. These two applications are `rocks.poopjournal.todont` (referred to as `ToDont`) and `de.hskl.contacts` (referred to as `ContactBook`). `ToDont` has six activities, and `ContactBook` has ten. We configured Ape$^+$ to cover as many activities as possible within a thirty-minute time range. For comparison purposes, we also test Ape with UiAutomator, referred to as Ape, for thirty minutes. The time consumption to cover all activities will be collected. Due to the randomness, the experiment was repeated ten times, and between each iteration, the emulator was wiped out and restarted.

As the Table 4.2 presents, Ape$^+$ outperforms Ape on both applications. On `ToDont`, Ape$^+$ spent on average 39 seconds to discover all the activities, while Ape spent 90.9 seconds. The 90th percentile of time consumption reflects that in most cases Ape$^+$ spent fewer than 39.7 seconds, while Ape took roughly 131.1 seconds. It has also shown similar results with `ContactBook`, where Ape$^+$ spent 526 seconds on average to cover all activities, while Ape spent 833 seconds. The 90th percentile of time consumption also favours Ape$^+$, which took 1,000.3 seconds most of the time, while Ape took 1,601.5 seconds. The tests on both applications demonstrate that informative static models have positive impacts on the exploration process in terms of time consumption.

However, we should also consider the time consumed in the static model generation phase. As shown in Table 4.3, the average time to generate a static model using Frontmatter varies across different applications. Taking `ToDont` as an example again, generating the static model took 27.32 seconds on average. The time to cover all activities using Ape$^+$ has saved 51.9 seconds on average. Effectively, the saved time is approximately to 24.58 seconds after subtracting the model generation time from the average time saved.

In conclusion, for both applications, Ape$^+$ achieved a visible drop in time consumption, as informative static models provide guidance in the process, and irrelevant interactions are potentially avoided. When Ape$^+$ is able to reach the activities of interest quicker with

Table 4.2: Time Consumption in Seconds for Testing `ToDont` and `Contact Book` with Informative Static Models

| | ToDont | | Contact Book | |
|---|---|---|---|---|
| | Ape$^+$ | Ape | Ape$^+$ | Ape |
| | 73 | 231 | 609 | 198 |
| | 35 | 52 | 319 | 1,202 |
| | 35 | 120 | 309 | 1,553 |
| | 35 | 36 | 267 | 207 |
| | 35 | 89 | 182 | 531 |
| | 35 | 52 | 130 | 678 |
| | 36 | 53 | 274 | 185 |
| | 35 | 110 | 911 | 227 |
| | 35 | 53 | 1,804 | 2,038 |
| | 36 | 113 | 455 | 1,511 |
| Average | 39 | 90.9 | 526 | 833 |
| Max | 73 | 231 | 1,804 | 2,038 |
| Min | 35 | 36 | 130 | 185 |
| 90 Percentile | 39.7 | 131.1 | 1,000.3 | 1,601.5 |
| 10 Percentile | 35 | 50.4 | 176.8 | 196.7 |
| Median | 35 | 71 | 314 | 604.5 |

less waste, it also means an increase in efficiency. On average, Ape$^+$ saved nearly 57% of time on exploration for `ToDont`, and the time consumption dropped 37% for `Contact Book`, if not considering the time spent on model generation. Even considering the time spent on model generation, the static model also provided a 27% enhancement for `ToDont` and 32% enhancement for `Contact Book`.

## 4.3 RQ3: Does False Information in Static Models Diminish the Efficiency of Ape$^+$?

In real world scenarios, static models inevitably contain false information or miss essential data. In order to discover the relations between the quality of static models and the outcomes of tests, we designed another experiment.

Table 4.3: Time Consumption for Frontmatter to Generate Static Models

| | Average | Variance |
|---|---|---|
| amaze-file-manager-3-2-1 | 640.91 | 253.01 |
| AndBible-3.0.286 | 499.83 | 369.02 |
| AndBible-3.1.309-beta | 411.26 | 1056.81 |
| AnkiDroid-2.6 | 686.67 | 428.69 |
| AnkiDroid-2.7 | 626.00 | 27345.50 |
| AnkiDroid-2.9.1 | 1146.92 | 13939.76 |
| AnkiDroid-2.9.4 | 1767.07 | 2211.75 |
| AnkiDroid-2.9 | 824.74 | 8485.28 |
| APV_APP_v1.0 | 10.83 | 0.74 |
| Barcode_Scanner_v4.7.8 | 58.23 | 5.26 |
| de.hskl.contacts_1 | 39.82 | 4.71 |
| My_Tracks_v5.6.3 | 636.20 | 214.94 |
| nextcloud-30100090 | 462.90 | 832.15 |
| Omni_Notes_v6.1.0 | 109.42 | 20.59 |
| OpenLauncher_Alpha_v0.3.1 | 23.53 | 8.69 |
| ToDont | 27.32 | 1.17 |
| wpandroid-11.3 | 741.14 | 37.32 |
| wpandroid-12.9 | 846.88 | 117.76 |
| wpandroid-13.1 | 792.34 | 120.32 |
| wpandroid-13.3 | 826.01 | 101.71 |
| wpandroid-13.6 | 586.27 | 185.78 |
| wpandroid-13.7 | 589.45 | 65.98 |
| wpandroid-14.9 | 731.08 | 650.12 |
| wpandroid-8.1 | 286.37 | 28.24 |
| wpandroid-9.2 | 533.30 | 87.67 |

Because there is no automatic way to prove whether a transition is false information or not, we again manually inspected one application, `AnkiDroid-2.7`, and its related static model. We picked three activities as the destination for Ape$^+$ to explore,

1. `com.ichi2.anki.MyAccount` (referred to as `MyAccount`)
2. `com.ichi2.anki.FilteredDeckOption` (referred to as `FilteredDeckOption`)
3. `com.ichi2.anki.NoteEditor` (referred to as `NoteEditor`)

We picked `MyAccount` because the path leading to it has some noise. Specifically, one transition in the path has listed five wrong trigger widgets along with a single correct trigger widget. We expect it to confuse Ape$^+$ when it tries to trigger a transition using the information provided by the static model, and leads Ape$^+$ to irrelevant activities. `FilteredDeckOption` was chosen because one static transition in the path is infeasible, as every trigger listed in that transition is wrong. Even though the correct trigger remains missing, the existence of false triggers would delay Ape$^+$ from switching to random exploration mode. We expect Ape$^+$ to reach this activity after many false attempts. Lastly, we chose `NoteEditor` as a control group, because the path to this activity does not contain false information. We expect Ape$^+$ to reach this activity quickly without many troubles. Again, we test Ape$^+$ against Ape, with a timeout limit set for thirty minutes. Because Ape does not support exploring with a specific destination, it would be exploring in the random exploration mode. To alleviate the potential threat of randomness, we repeated the experiment ten times for each destination and recorded the time consumption.

Table 4.4: Comparison between Activities

| | NoteEditor | | MyAccount | | FilteredDeckOption | |
|---|---|---|---|---|---|---|
| | Ape$^+$ | Ape | Ape$^+$ | Ape | Ape$^+$ | Ape |
| | 193.71 | 346.45 | 307.00 | 264.52 | 219.81 | 529.15 |
| | 42.84 | 497.90 | 609.40 | 40.69 | 1,013.97 | 150.98 |
| | 625.26 | 375.17 | 531.33 | 101.13 | 882.74 | 562.94 |
| | 102.50 | 496.54 | 228.17 | 462.85 | 1,542.30 | 370.54 |
| | 411.57 | 157.49 | 225.11 | 17.54 | 907.06 | 39.67 |
| | 249.93 | 360.49 | 233.71 | 366.56 | 301.07 | 291.97 |
| | 52.33 | 364.98 | 400.65 | 406.17 | 329.74 | 432.49 |
| | 291.41 | 423.29 | 260.47 | 400.46 | 1,172.33 | 438.44 |
| | 381.05 | 325.26 | 379.14 | 183.97 | 863.40 | 207.52 |
| | 460.13 | 422.65 | 567.01 | 86.82 | 742.39 | 160.14 |
| Mean | 281.07 | 377.02 | 374.20 | 233.07 | 797.48 | 318.39 |
| Median | 270.67 | 370.07 | 343.07 | 224.25 | 873.07 | 331.26 |
| Max | 625.26 | 497.90 | 609.40 | 462.85 | 1,542.30 | 562.94 |
| Min | 42.84 | 157.49 | 225.11 | 17.54 | 219.81 | 39.67 |
| 90th Percentile | 476.64 | 496.68 | 571.25 | 411.84 | 1,209.33 | 532.53 |
| 10th Percentile | 51.38 | 308.48 | 227.86 | 38.37 | 292.94 | 139.85 |

Table 4.4 demonstrates negative impacts that were caused by false information and

omission of information. Ape$^+$ took 281.07 seconds to reach `NoteEdtior`, as the path to it is clear and correct, while traditional Ape spent 377.02 seconds on the road. This again strengthens the claim that an informative static model is beneficial to the exploration. However, The situation got worse when Apes try to reach `MyAccount`. Because of the noise in the transition, Ape$^+$ spent extra 60% of time on average compare to the time Ape spent. The specific numbers are 374.20 seconds for Ape$^+$, and 233.07 seconds for Ape. As expected, Ape$^+$ triggered lots of transitions that led it to irrelevant activities, or activities far away from the destination before actually triggering the correct transition. The noise had successfully slowed down Ape$^+$ from reaching the destination. The last activity revealed that omission of information may cause a much worse slow-down compare to false information. During the exploration to `FilteredDeckOption`, due to the wrong transition information, the time consumption for Ape$^+$ nearly doubled that of Ape. On average, traditional Ape took 318.39 seconds to reach `FilteredDeckOption`, while Ape with static model wasted lots of time on irrelevant activities before finally crawled to the destination after 797.48 seconds. As expected, due to the miss of correct trigger, Ape$^+$ kept triggering wrong transitions to irrelevant activities. Because the correct trigger is missing, Ape$^+$ would keep trying the wrong triggers until they are considered not possible and Ape$^+$ switches to random exploration mode. In the random exploration mode, Ape$^+$ has no difference from Ape, and therefore it could reach the destination after wasting all the time on the wrong triggers.

This experiment reveals that false information and omission of information in static models could lead the exploration process to false and unrelated activities, which eventually causes a decline in efficiency. For `MyAccount`, where a correct trigger is buried by false triggers, the efficiency dropped roughly 38%; while for `FilteredDeckOption`, the efficiency dropped approximately 60%.

## 4.4 RQ4: Does the Static Models Change Ape's Behaviours

In this section, we discuss the impact brought by static models to Ape$^+$'s behaviours. The experiment was set up to test each target application ten times using both Ape$^+$ and Ape. Each testing iteration takes up to thirty minutes. In the process, we keep track of the activities reached, and the paths they took to reach the activities. Eventually, we analyze the total number of unique activities and total number of unique transitions covered in all testing iterations.

Table 4.5: Coverage for All Applications. All the Values in Brackets Are the Numbers of Univocally Covered Activities or Transitions

| | Activities | | | Transitions | |
|---|---|---|---|---|---|
| | Total | Ape | Ape$^+$ | Ape | Ape$^+$ |
| amaze-file-manager-3-2-1 | 7 | 3 (1) | 2 (0) | 343 (132) | 364 (153) |
| AndBible-3.0.286 | 31 | 18 (3) | 19 (4) | 284 (121) | 290 (127) |
| AndBible-3.1.309-beta | 31 | 20 (2) | 21 (3) | 501 (201) | 527 (227) |
| AnkiDroid-2.6 | 21 | 16 (2) | 14 (0) | 465 (184) | 449 (168) |
| AnkiDroid-2.7 | 21 | 15 (0) | 15 (0) | 384 (145) | 391 (152) |
| AnkiDroid-2.9.1 | 22 | 11 (1) | 12 (2) | 296 (121) | 310 (135) |
| AnkiDroid-2.9.4 | 22 | 14 (1) | 15 (2) | 443 (172) | 440 (169) |
| AnkiDroid-2.9 | 22 | 14 (0) | 16 (2) | 57 (19) | 66 (28) |
| APV_APP_v1.0 | 11 | 7 (0) | 7 (0) | 427 (149) | 461 (183) |
| Barcode_Scanner_v4.7.8 | 9 | 7 (0) | 7 (0) | 727 (249) | 802 (324) |
| de.hskl.contacts_1 | 10 | 10 (0) | 10 (0) | 392 (148) | 408 (164) |
| My_Tracks_v5.6.3 | 35 | 16 (0) | 16 (0) | 508 (184) | 527 (203) |
| nextcloud-30100090 | 36 | 3 (0) | 3 (0) | 333 (129) | 351 (147) |
| Omni_Notes_v6.1.0 | 14 | 6 (1) | 7 (2) | 572 (230) | 632 (290) |
| OpenLauncher_Alpha_v0.3.1 | 8 | 7 (0) | 7 (0) | 319 (125) | 338 (144) |
| ToDont | 6 | 6 (0) | 6 (0) | 555 (219) | 582 (246) |
| wpandroid-11.3 | 76 | 7 (1) | 6 (0) | 117 (42) | 139 (64) |
| wpandroid-12.9 | 84 | 6 (0) | 6 (0) | 338 (130) | 358 (150) |
| wpandroid-13.1 | 80 | 6 (0) | 6 (0) | 325 (128) | 341 (144) |
| wpandroid-13.3 | 80 | 6 (0) | 6 (0) | 104 (33) | 131 (60) |
| wpandroid-13.6 | 78 | 6 (0) | 6 (0) | 94 (29) | 115 (50) |
| wpandroid-13.7 | 78 | 6 (0) | 6 (0) | 127 (41) | 147 (61) |
| wpandroid-14.9 | 81 | 6 (0) | 6 (0) | 456 (174) | 448 (166) |
| wpandroid-8.1 | 62 | 4 (0) | 4 (0) | 476 (192) | 453 (169) |
| wpandroid-9.2 | 67 | 5 (0) | 5 (0) | 430 (170) | 429 (169) |

Table 4.5 shows the activity coverage and the edge coverage for all the applications. The numbers in brackets are the numbers of univocally covered activities or transitions. For all the applications, most activities are commonly discovered by both Ape$^+$ and Ape, with merely few exceptions. There are nine cases where one Ape beats another by one or two more univocally covered activities, while most of the covered activities remain shared. For small and simple applications, both Apes are able to discover all activities within the limited time. For large and complicated applications, both Ape did equally poorly within the 30 minutes time range. For example, `wpandroid-13.3` has 80 activities, but both of Ape

only covered the same 6 activities. The average numbers of univocally covered activities show that Ape covers 0.55 unique activities and Ape$^+$ covers 0.68 unique activities. Using paired T test on the numbers of univocally covered activities, we can get a two-tailed P value equals 0.3269. It demonstrates that there is not much behaviour difference between Ape$^+$ and Ape in terms of activity coverages. However, the path coverage tells a slightly different story. For all the applications, both Apes have numerous transitions that are not covered by the other. Even for small and simple applications, like `ToDont`, the numbers of uniquely covered transitions are 216 (Ape) and 234 (Ape$^+$). On average, Ape has 130.05 univocally covered transitions and Ape$^+$ univocally covered 144.63 transitions. Again, we applied paired T test on the numbers of univocally covered transitions, and we got a two-tailed P value equals 0.0004. Statistically, this difference is extremely significant.

In conclusion, there is nearly no significant difference between the two Apes in terms of activity coverages. It provides a key insight that the introduced static model and upgraded path finding algorithm do not enahance Ape$^+$ to discover new undiscovered activities. However, these updates deviated Ape$^+$ from the original search path, which resulted in the large number of exclusively covered transitions.

# Chapter 5

# Related Work

***Static Model Guided Exploration.*** At the time of writing, we have noticed a concurrent work in guiding explorations with a predefined static model. Ngo *et al.* developed ATUA [26], an update-driven testing tool. It focuses on testing updated methods between two versions of the same application, then use a static model, Extended Window Transition Graph, to guide DroidMate2 [7] to test widgets associated with the updated methods. Their work is more fine-grained, as it focuses on the function level, while Ape$^+$ only tries to reach a specified activity at this moment. Ape$^+$ has also learned an important widget matching technique from ATUA, which is matching with the hierarchical information. Similarly, Peng *et al.* proposed CAT [28] which accepts user provided goals, then maps these goals to related widgets. CAT traces back on a call graph from the provided goals to determine the widgets of interest. It then utilizes a test generation tool, Droidbot [21], to generate tests to targeting these widgets. CAT does not control how Droidbot generates tests, until it reaches an activity where widgets of interest exist. It will then take over the testing process when such activities are reached. Ape$^+$ on the other hand, does not rely on third party test generators, and it controls the generation of every action. Each generated action is rearranged in the most optimal order to effectively guide the exploration process to the destination. Lai *et al.* developed GoalExplorer [19] which builds a highly-fine-grained static model as the sole information source to guide the exploration. GoalExplorer emphasizes the importance of a static model that closely represents an application. With its informative static model, GoalExplorer does not rely on a dynamic model to do the exploration. Unlike GoalExplorer, Ape$^+$ utilizes both models, the static and the dynamic, to explore an application. As discussed before, static models are prone to false positives and miss of information. Thus, we believe that exploring with the combination

of both models is more optimal.

***Static Analyses Tools.*** All the related static analyses tools are based on Soot[11] to collect information from the APK files. Rountev *et al.* [29] developed Gator, which produces the Window Transition Graph. Window Transition Graphs contain the information of widgets, activities, and transitions between activities. The trigger information between widgets are listener methods, which is coarse-grained. As mentioned beofre, a listener function can internally implements multiple conditional branches, and the transition may only be triggered by specific widgets. Thereby, listing listeners as triggers narrows down the scope of widgets, but it is not accurate enough to avoid false trigger widgets. Ngo *et al.* [26] introduced Extended Window Transition Graph. As the name suggests, it is an extended version of the Window Transition Graph from Gator. It tracks back from listeners to widgets, and includes the related widgets as part of the model. Based on ATUA's unique use case, information about updated methods between two versions of the same application is also included in the model. Similarly, these trigger widgets are not accurate enough due to the nature of listener methods. Yan *et al.* [37] presented ICCBot, whose model focuses solely on transitional information. ICCBot attempts to divides activities into components, and find relations between the components. Because it can subdivide an activity into multiple components, it will essentially find more transitions. The transitional information is therefore more fine-grained as compare to Window Transition Graph. However, this tool is unable to analyze UI hierarchy or widget information. Kuznetsov *et al.* [18] developed Frontmatter, which provides an optimized trigger detection strategy. Frontmatter does not naively assume any widget associated with a trigger listener would trigger the transition. Instead, for every listener function, Frontmatter breaks it down based on the conditional branches. Next, Frontmatter analyzes the conditions associated with each such branch. If there is any widget is referred in the condition, Frontmatter would link the conditional branch to that widget. Using this strategy, Frontmatter is able to narrow down the correct trigger widgets for each transition. Lai *et al.* [19] proposed Screen Transition Graph, as a fine-grained alternative to Extended Window Transition Graph. Its static model, Screen Transition Graph, contains various node representing different screens the application has. Each node has its unique widget arrangement. It records transitions between different screens in hope of closely represnting the runtime behaviours of the application.

***Android Testing Tools.*** There are also related automatic Android testing tools. Gu *et al.* developed Ape [12] that is able to abstract content on screen into a model and constantly refine it with a decision tree. This thesis is based on Ape, and tries to address the drawbacks it has. Borges *et al.* [7] constructed a highly efficient testing generator platform, DroidMate2, that provides out of box testing strategies and monitoring tools. DroidMate2 can also be used as a testing tool by itself to generate test inputs targeting given appli-

cations. ATUA is using DroidMate2 to generate test cases. Wang *et al.* [36] developed ComboDroid. The corner stone of this tool is the observation that a long test input can be decomposed into relatively independent pieces, and enumerating different small pieces can form long and better test inputs. ComboDroid has the ability to extract the small pieces in the testing process, and it can also accept user inputs about the small pieces. There is also Humanoid introduced by Li *et al.* [22]. Humanoid uses a deep learning approach to learn and generate actions that resemble human interactions. As a result, Humanoid is able to put first the test inputs that users think are more important. TimeMachine [9] developed by Dong *et al.* is another interesting Android testing tool. It is able to sense that the testing process is not making progress. When it happens, TimeMachine would return the testing process to a previous saved spot in time, and resume testing with a different path. Each time TimeMachine chooses a different path, it tries to avoid getting stuck and make progress in a different direction. Such time travelling functionality is achieved by conducting the test in a virtual machine, and saving snapshots in the testing process. Whenever it needs to go back in time, a saved snapshot is reloaded.

# Chapter 6

# Discussion

## 6.1 Threats to Validity

This study focuses on improving the maintainability and efficiency by switching to UiAutomator from Appium based Ape, and enhancing the testing effectiveness by introducing static-model-guided exploration. The validity of the results may be affected by a number of factors.

### 6.1.1 Internal Validity

***Randomness.*** Ape's exploration involves a lot of randomness: from randomly selecting actions to randomly generating throttles. Therefore, the random choices may increase or decrease the amount of time needed to complete certain tasks, *e.g.* discovering an activity. This is a potential threat to the validity of our results. To mitigate this threat, we repeat the test process multiple times. All the comparisons are conducted on the averages and percentiles. We believe this is beneficial to make the results more reliable.

***Static Models.*** Even though we have leveraged multiple static analyses tools, including Gator [29], IccBot [37], and Screen Transition Graph [19], we still cannot guarantee the models Frontmatter [18] generated describe the applications the most accurately. There are around twenty applications that Frontmatter failed to find more than three transitions between activities. Even for the applications where Frontmatter successfully analyzed, there are false information existing in the models. Since there is no ground truth about the correct number of transitions in an application, it is impossible to automatically evaluate

the qualities of static models. For the experiments related to model qualities, we relied on manual inspections to pick a small subset of models. Due to the lack of quality assurance and clear selecting criteria, the manual work may affect the final results.

### 6.1.2 External Validity

***The Benchmark Application Collection.*** Our benchmark applications are initially collected from the Themis [31] paper. Even though the application collection has over fifty applications, there were nearly half of them being filtered out. Among the remaining applications, a few of them are different versions of the same applications. We believe the lack of diversity would be a threat to our testing outcomes. To mitigate this threat, we randomly explored F-Droid [10] to collect more applications. Eventually, we have in total twenty-five benchmark applications, with eleven of them being unique applications.

***Server Scheduling.*** All our experiments were conducted on the same Linux server in a sequential order. These experiments were conducted over days and nights. However, the server executes scheduled daily jobs during late midnight, which may take up to several hours. Some of these jobs are known to be performance hungry, and it potentially affects the overall performance of the server. During executions of the daily jobs, our experiements may be slowed down. This is a threat to the validity of our result. If two tasks can be executed in different conditions, then their results can never be compared directly and fairly. To verify if the jobs actually affected our experiements, we conducted extra experiments. We purposefully selected some tasks that were executed over the night, then executed them again in the daytime. The results from the daytime executions were compared to the results from the night time executions. It showed no sign of significant difference between the two. In conclusion, the scheduled daily jobs do not visibly slow down the experiments, as the performance affect is negligible.

## 6.2 Future Work

The solution we proposed is not perfect, and some future work can be done to improve it for better testing outcomes.

### 6.2.1 Static-Model-Guided Exploration

The current static-model-guided exploration finds the most optimal path to one specified target activity. Once reaching the destination, Ape$^+$ would continue to conduct testing to thoroughly examine the target activity. For the future work, this can be further enhanced in two directions. First and foremost, Ape$^+$ should be able to support multiple destinations. For all the destinations, Ape$^+$ should sort them in a prioritized order and find the most optimal testing path to cover all of them. Secondly, the exploration can be enhanced in supporting more types of destinations. These destinations can be more fine-grained, such as widgets or specific actions. These fine-grained destinations can be directly related to the changed widgets or functions, which would further reduce the testing scope and increase the overall efficiency. In conclusion, static-model-guided exploration should support multiple destinations and each of them being more diverse and fine-grained.

### 6.2.2 Instrumentation

The current instrumentation for widget matching falls short for GUI widgets defined in layout files, as the current strategy focuses solely on the widgets created through source code. Even though most resource-id-less widgets defined in layout resource files are static, it is still possible to have a few such widgets being interactive. Therefore, one of the future works would be improving the instrumentation to support instrumenting widgets defined in layout resource files. It requires the instrumentation tool to understand AXML file format used in APKs to correctly parse and modify the resource layout files. Once every widget is instrumented with a resource id, the widget matching process would be easier with more confidence.

### 6.2.3 Static Analyses

The proposed solution adopts Frontmatter [18] as the static analyses tool. However, static models generated by Frontmatter are far from perfect. First of all, Frontmatter falls short on certain applications where transitions or triggers cannot be correctly detected. We have found approximately ten such cases during our evaluation, where Frontmatter produces a model with no transition or trigger at all. Besides, there are many false positive cases for some complicated applications after manual inspections. As shown in the evaluation chapter, false positives and miss of information would negatively and severely impact the exploration. Therefore, one future work would be optimizing Frontmatter to enhance the

transition and trigger detections. At the same time, Frontmatter should also minimize the number of false information in the models.

Besides that, Frontmatter requires a huge amount of memory and time. The recommended memory allocation for Frontmatter is 40 GB. To thoroughly analyze one application, the time consumption ranges from 10 seconds to 30 minutes, depending on the application complexities. This tremendous overhead shadows the efficiency benefits brought to Ape$^+$. Hence, it is necessary to ease the resource hunger of Frontmatter, and also accelerate it, so that it takes a reasonable amount of time.

Moreover, the transitional information of the current static model is activity-based. It essentially ignores any transition that happens within the same activity. Adding the missing transitions would help the navigation, as it provides more details inside each activity. Therefore, it would be helpful in the future to extend Frontmatter support component-level transitions, like in ICCBot, or screens, like in Screen Transition Graph.

### 6.2.4 Benchmark for Static Analyses Tools

We chose Frontmatter because it is the most stable tool that also provides the most accurate trigger detection algorithm. However, such decision was made without any scientific metrics, because there currently exists no systematic way to evaluate different static analyses tools. The reason behind is the lack of ground truth for applications, as the exact number of widgets, transitions, and their triggers cannot be easily obtained without human intervention. Consequentially, decisions on static models or static analyses tools are based on feelings instead of concrete metrics. Technically, selecting the correct static analyses tool would dramatically benefit the test, it would be necessary to have a benchmark to evaluate static models in the future. This benchmark should provide a group of selected applications with the a known number of widgets, transitions, and related triggers. Any selected static model should be evaluated against the ground truth. We can then obtain the number of false or missing transitions, triggers, or widgets to understand the quality of a static model.

# Chapter 7

# Conclusion

Ape is a state-of-the-art automatic testing tool focusing on testing Android GUI. In this study, we presented our solution, Ape$^+$, to address the two main problems Ape has. The first problem is the maintainability and efficiency problem. In order to solve Ape's reliance on private APIs, we introduced of Appium as the communication layer. Because of Appium, Ape was transformed into a server-client structure, which brought in a huge overhead and was hard to maintain. We solved it by replacing Appium with UiAutomator, and rebuilding Ape as a fully on-device application. This approach enabled debuggers to be used on Ape, eliminated the dependency on private APIs, and also improved the efficiency. During the transition, we realized UiAutomator does not support scroll action out of the box, so we simulated it with a combination of `MotionEvent`s. In addition, retrieving current activity name was not possible with functions provided by UiAutomator. So we adopted a solution to find the information from a system dump generated through ADB. The second problem is the inability to prioritize activities of interest, which made it less effective when testing a part of an application. We attempted to solve this problem by introducing a static model, generated by Frontmatter, as prior-knowledge and an assistant. During explorations, the static model provides extra information about trigger widgets and their destination activities, so that Ape$^+$ can quickly travel to a specified activity with ease. To make widget matching between static models and dynamic models more accurate, we have developed an instrumentation tool that is able to assign unique resource ids to the widgets that do not have one. For the false information potentially contained in every static model, we have a priority decay policy to minimize its influence. Lastly, we conducted a series of experiments to demonstrate the improvements. By comparing the time consumption of between Ape with UiAutomator and Ape with Appium, it shows that the performance gains are roughly between 10% to 40% among applications. In addition, using manually

selected applications and activities to perform static-guided-explorations, we have proven that informative static models helps applications explore faster, and false information, on the other hand, slows down the process. Finally, we contemplated the covered activities and transitions of Ape$^+$ and Ape among all the testing applications. It demonstrated that the introduction of static models does not necessarily help Ape$^+$ discover unvisited activities, but it changes the exploring paths. In the end, we discussed the threats to validity and potential future work. We believe this study will be helpful for other researchers who are trying to adopt UiAutomator or trying to guide explorations with static models.

# References


[1] Accessibility service. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService. Accessed: 2022-05-08.

[2] Android debug bridge. https://developer.android.com/studio/command-line/adb. Accessed: 2022-05-06.

[3] Android studio. https://developer.android.com/studio. Accessed: 2022-06-30.

[4] App widgets overview. https://developer.android.com/guide/topics/appwidgets/overview. Accessed: 2022-06-24.

[5] Appium. http://appium.io. Accessed: 2022-06-23.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[7] Nataniel P. Borges Jr., Jenny Hotzkow, and Andreas Zeller. *DroidMate-2: A Platform for Android Test Generation*, page 916–919. Association for Computing Machinery, New York, NY, USA, 2018.

[8] Compatibility framework tools. https://developer.android.com/guide/app-compatibility/test-debug. Accessed: 2022-07-27.

[9] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel testing of android apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 481–492. IEEE, 2020.

[10] F-droid. https://f-droid.org. Accessed: 2022-05-08.

[11] Sable Research Group et al. A framework for analyzing and transforming java and android applications, 2018.

[12] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical gui testing of android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 269–280. IEEE, 2019.

[13] View.ids. https://developer.android.com/reference/android/view/View#ids. Accessed: 2022-05-06.

[14] Id resource. https://developer.android.com/guide/topics/resources/more-resources#Id. Accessed: 2022-06-24.

[15] Input events overview. https://developer.android.com/guide/topics/ui/ui-events. Accessed: 2022-07-19.

[16] Intellij idea. https://www.jetbrains.com/idea/. Accessed: 2022-06-30.

[17] Introduction to activities. https://developer.android.com/guide/components/activities/intro-activities. Accessed: 2022-06-24.

[18] Konstantin Kuznetsov, Chen Fu, Song Gao, David N Jansen, Lijun Zhang, and Andreas Zeller. What do all these buttons do? statically mining android user interfaces at scale. *arXiv preprint arXiv:2105.03144*, 2021.

[19] Duling Lai and Julia Rubin. Goal-driven exploration for android applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 115–127. IEEE, 2019.

[20] Layout resource. https://developer.android.com/guide/topics/resources/layout-resource. Accessed: 2022-06-24.

[21] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE, 2017.

[22] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun 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)*, pages 1070–1073. IEEE, 2019.

[23] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 94–105, 2016.

[24] monkeyrunner. https://developer.android.com/studio/test/monkeyrunner. Accessed: 2022-06-27.

[25] Motionevent. https://developer.android.com/reference/android/view/MotionEvent. Accessed: 2022-07-07.

[26] Chanh Duc Ngo, Fabrizio Pastore, and Lionel Briand. Automated, cost-effective, and update-driven app testing. *ACM Transactions on Software Engineering and Methodology*, 2020.

[27] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. Reinforcement learning based curiosity-driven testing of android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–164, 2020.

[28] Chao Peng, Ajitha Rajan, and Tianqin Cai. Cat: Change-focused android gui testing. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 460–470. IEEE, 2021.

[29] Atanas Rountev and Dacong Yan. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 143–153, 2014.

[30] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256, 2017.

[31] Ting Su, Jue Wang, and Zhendong Su. Benchmarking automated gui testing for android against real-world bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 119–130, 2021.

[32] Write automated tests with ui automator. https://developer.android.com/training/testing/other-components/ui-automator. Accessed: 2022-05-04.

[33] Uiautomator.dumpwindowhierarchy. https://developer.android.com/reference/androidx/test/uiautomator/UiDevice#dumpwindowhierarchy. Accessed: 2022-06-25.

[34] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[35] View binding. https://developer.android.com/topic/libraries/view-binding. Accessed: 2022-06-24.

[36] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. Combodroid: Generating high-quality test inputs for android apps via use case combinations. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 469–480, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. Iccbot: Fragment-aware and context-sensitive icc resolution for android applications. 2022.