CydiOS: A Model-based Testing Framework for iOS Apps

Anonymous Author(s)

ABSTRACT

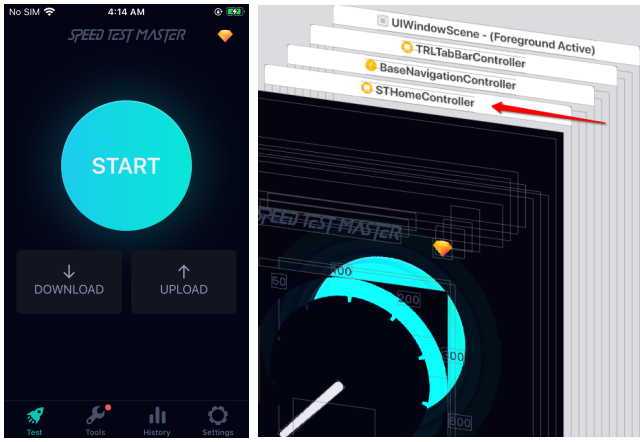
To make an app stand out in an increasingly competitive market, developers must ensure its quality to deliver a better user experience. UI testing is a popular technique for quality assurance, which can thoroughly test the app from the users' perspective. However, while considerable research has already studied UI testing on the Android platform, there is no research on iOS. This paper introduces CydiOS, a novel approach to performing model-based testing for iOS apps. CydiOS enhances the existing static analysis to build a more complete static model for the app under test. We propose an approach to retrieve runtime information to obtain real-time app context that can be mapped in the model. To improve the effectiveness of UI testing, we also introduce a potential-aware search algorithm to guide testing execution. We compare CydiOS with four representative algorithms(i.e., random, depth-first, stoat, and ape). We have evaluated CydiOS on 50 popular apps from App Store, and the results show that CydiOS outperforms other tools, achieving both higher code coverage and screen coverage. We will open source CydiOS at https://github.com/SoftWare2022Testing/CydiOS, and a demo video can be found at https://www.youtube.com/shorts/VeZpY92Fno4.

1 INTRODUCTION

Recent years have witnessed significant growth in the iOS market. According to recent estimations [15], iPhone sales reached a new record high in 2021, accounting for over 60% of the premium market. With the number of iPhone users continually increasing [21], more developers distribute their apps in App Store to get a market share. To promote user trust, App Store has a strict review process concerning the quality of each submitted third-party app [4]. Therefore, it is a significant concern for both app developers and vetters to ensure the app meets the quality expectation.

To improve apps' reliability, researchers have proposed many approaches to generate GUI tests to explore their functionalities. Random-based approaches [39, 57, 63, 76] inject arbitrary UI events into apps. However, without guidance, these approaches are time-consuming, and their test coverage cannot be guaranteed. To uncover the app behavior that is hard to be reached by random-based approaches, many systems [44, 53, 74] use symbolic execution to find all feasible program paths for test generation. However, it remains challenging to scale them in practice due to the high computational cost [54].

Model-based testing (MBT) is another popular solution, which generates test inputs based on the model to systematically explore the app. The model is a Finite State Machine (FSM) that reflects the behavior space of the app under test. Each state usually represents an app screen (i.e., an Activity in Android apps or a View Controller in iOS apps), and each transition between states is labeled with an input event. With guidance, MBT can get high coverage, thus outperforming other testing techniques [45, 65]. However, despite considerable research efforts [43, 46, 49, 54–56, 68, 70] focusing on performing MBT on Android apps, to the best of our knowledge, MBT on iOS apps remains unstudied. An effective MBT approach for iOS apps is highly demanded by both developers and researchers for reducing manual efforts to scale up the testing and evaluating iOS apps' performance and security.



(a) Home screen. (b) The hierarchy of home screen.

Figure 1: An example of Speed-Test-Master app.

To fill in the gap, in this paper, we design an effective MBT approach for iOS apps and implement a new tool named CydiOS by tackling the following technical challenges.

C1: Building a complete static model. To guide the testing, we need a complete model. A few previous work targeting Android apps builds the model by dynamically exploring apps. Although dynamic exploration can provide more trustworthy information, it is difficult to completely explore the behavior space [47], making the generated model only cover a small range of the behavior space. Therefore, researchers resort to statically analyzing apps' binary to generate the static model. However, applying static analysis to iOS apps' binary is non-trivial. State-of-the-art approaches [51, 59] perform data flow analysis directly on the disassembled code. Considering there are some indirect jumps in app binary, where the target address is stored in the memory and remains unknown until execution [22], these approaches cannot resolve some data flow paths correctly. For this reason, the accuracy of static analysis is hampered, which makes the built model incomplete and inaccurate.

C2: Obtaining app context. During UI testing, real-time app context (i.e., name of app screens) is retrieved to guide test input generation [45]. For example, we need app context to determine if the last event causes a screen transition or enters a new state in the model. For Android apps, adb [3], a tool provided by Google, can be employed to retrieve the name of the current `Activity`. However, there is an absence of such a tool for iOS apps.

In an iOS app, a view controller usually manages a screen of content, which is analogous to the `Activity` in Android. Therefore, we retrieve the name of view controller currently displayed on the device screen as the app context. However, unlike Android where an `Activity` represents a single screen, there can be multiple view controllers comprising a screen of an iOS app. For example, Figure

1a shows the home screen of the Speed-Test-Master app, and its hierarchy is displayed in Figure 1b. The home screen consists of 3 view controllers, and the GUI managed by `STHomeController` is displayed on the device. Considering that iOS developers can use multiple view controllers to comprise a single screen in many ways, it is challenging to determine the view controller which is responsible for the GUI displayed on the device screen.

To address **C1**, we use simulated execution [66], which simulates the code path execution on the emulator, to resolve the target of each indirect jump. Then, we enhance static analysis to build a complete model from apps' binary. To address **C2**, we gain comprehensive knowledge about common patterns of organizing view controllers by investigating Apple's developer documentation and forums [5, 18]. Based on it, we propose an approach to determine the view controller which is responsible for the GUI displayed on the device, so that we can obtain the real-time app context and map it to the model. To improve test effectiveness, we design a potential-aware search algorithm, which guides testing towards less traveled paths according to the model and real-time app context.

We compare CydiOS with four state-of-the-art UI testing algorithms (i.e., random, depth-first search, stoat [70], and ape [54]) on 50 popular apps downloaded from App Store. The result shows CydiOS outperforms other algorithms. Specifically, it consistently resulted in 24.2–154%, and 6.0–63.2% relative improvements over the above four algorithms in terms of average screen coverage and code coverage, respectively. In addition, we also evaluate the enhanced static analysis in CydiOS on 20 open source iOS apps, the result indicates it achieves 16% more precision and 35.6% more recall than existing approaches. To measure the performance of CydiOS driving the UI testing, we compare it with Apple's native UI testing driver XCUITest [17] and Appium [31] which is a commercial test automation framework. The result shows that CydiOS has the lowest cost in dumping GUI and injecting test actions. To further demonstrate the practical usefulness of CydiOS, we leverage it to perform UI fuzzing and detect privacy leakage in iOS apps.

Our major contributions are summarized as follows:

- To the best of our knowledge, we take the first step to investigate the UI testing on iOS platform.
- We design a novel model-based UI testing tool named CydiOS, which automates UI testing for iOS apps. We implement CydiOS by carefully tackling several technical challenges.
- After extensive evaluation, the results show that CydiOS can achieve both good screen coverage and code coverage. We also demonstrate two use cases to show the applicability of our tool in app fuzzing and privacy leakage detection.

2 BACKGROUND

In this section, we provide the necessary background about iOS UI in §2.1, and introduce the language feature of Objective-C in §2.2, which is the primary programming language for iOS app [24].

2.1 iOS UI

Window. Window is the top container for app user interface. Everything we can see on the device is in a Window [20]. Normally, an iOS app has only one Window [37] at a time.

View Controller. View controller (VC) represents a screen in the iOS app, which is similar to `Activity` in Android. Screen transition is the switch between two VCs. Every Window has a single VC served as its root VC, which is initially displayed on it [30].

Container View Controller. To facilitate the transition between VCs, iOS UI framework (i.e., UIKit [2]) provides container view controllers, including navigation controller and tab bar controller. Navigation controller internally maintains a stack to manage the VCs to be displayed [33]. Only the VC at the top of the stack will be displayed (i.e., when the Third View Controller in Figure 2a is displayed, it is on the top of the stack). App can invoke `pushViewController` function to push a new VC onto the top of the stack to navigate to it. On the newly presented VC, we can find a back button that is automatically created by the navigation controller for return back (see Figure 2a). Tab bar controller is another commonly used container VC in iOS. It keeps its child VCs in an array [34] and allows users to transit between different child VCs by selecting the array index. For example, in Figure 2b, the index for the currently displayed Item1Controller is 0.

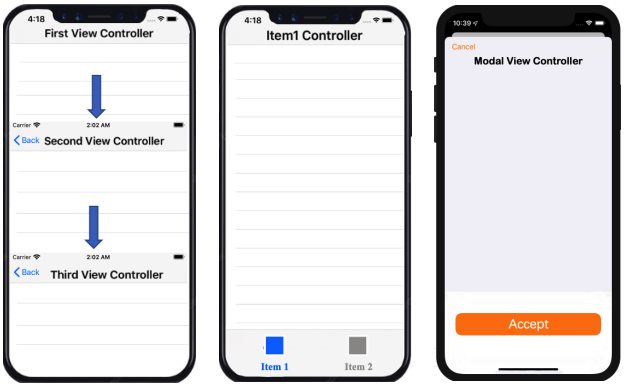
Screen Transition Basically, there are three styles of transition in iOS, namely Push, Tab, and Modal [36]. The Push and Tab transition can be achieved by utilizing container VCs (i.e., navigation controller for Push style, tab bar controller for Tab style). Modal is a bottom-up transition style that displays a separate screen to hide the current app screen. For example, the ModalViewController in Figure 2c is presented modally. Developers can invoke transition APIs provided by Apple framework to trigger the screen transition. We list the transition APIs in iOS in Table 1. Besides implementing VC transitions in the code (by setting transition source and destination in transition API), developers can create a segue object by connecting two VCs in the layout files to add a transition [40]. The segue object defined in the layout can be programmatically triggered by invoking `performSegueWithIdentifier` (Segue style).

Table 1: View Controller Transition APIs

Style	APIs
Push	pushViewController, popViewController, popToRootViewController, popToViewController
Tab	addChildViewController
Modal	presentViewController, dismissViewController
Segue	performSegueWithIdentifier

2.2 Message Dispatch Mechanism

Objective-C is the primary programming language for iOS [24], and it is a dynamic language because all method calls are resolved dynamically at runtime through the messaging dispatch function `objc_msgSend` [23]. For example, in Figure 4, the code in Line 4 will be translated into the code in gray color by the compiler. Every time the code in Line 4 is executed, the `objc_msgSend` function is invoked under the hood to jump to the method implementation (i.e., the implementation of `doJump`). Since all method invocations are delegated by `objc_msgSend`, it is difficult to extract a complete method call graph from the binary, which negatively affects further static analysis.



(a) Navi VC. (b) Tab Bar VC. (c) Modal VC.

Figure 2: View Controller Hierarchy.

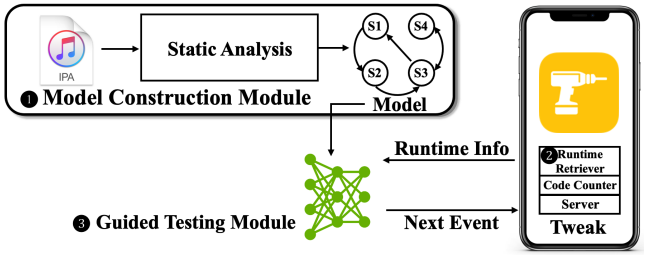


Figure 3: Architecture of CydiOS.

3 CYDIOS OVERVIEW

As shown in Figure 3, CydiOS consists of three modules:

❶ Static Model Construction Module : Given an iOS app, it statically analyzes the app's binary and layout files to build the model (i.e., a screen transition graph), so that Guided Testing Module can generate test inputs to trigger possible app behavior based on it. When statically analyzing app's binary, we perform data flow analysis to identify transition target. Due to the indirect jump, data flow analysis in existing practice is hampered because some data flow paths cannot be resolved. To build a more complete and accurate model, we employ a new technique named simulated execution to the enhance existing static analysis for iOS app (detailed in §4).

❷ Run-time Retriever Module: During testing, it retrieves run-time information including UI hierarchy and app context. UI hierarchy shows all the widgets displayed on the screen, which reflects all feasible UI events that we can execute. App context is used to map to a state in the model. The retrieved information will be fed to Guided Testing Module. Based on it, we can guide the test inputs generation to effectively explore the app. Run-time Retriever Module retrieves the name of view controller (VC) displayed on the screen as app context [45]. Since developers can organize multiple VCs to compose a screen in different ways, we propose an approach to identify the correct VC that is responsible for the GUI on the device (detailed in §5).

❸ Guided Testing Module: We design a potential-aware search algorithm to guide the testing based on the static model and run-time information. To explore the app more effectively, we estimate the potential of the widget to reach an unseen screen. Unlike existing testing algorithms [45, 54, 70] that only measure the potential of widget locally (i.e., by looking for the unvisited widgets in its next screen), we estimate its potential globally in the whole model, so that we can make a more optimal decision to guide the testing towards untraveled paths (detailed in §6).

4 STATIC MODEL CONSTRUCTION MODULE

This module builds a static model to guide test generation based on the app's expected behaviors. The model is a directed graph $G = (V, E)$, where node set V represents the view controllers (i.e, app screen), and edge set E represents transitions between view controllers (i.e., screen transitions). We take two steps to generate the model. First, we identify the source and destination view controllers (VC) in each screen transition (§4.1). Second, we find the widgets that trigger transitions so that we can label the edges in the model (§4.2). When building the static model, we need to identify the data flow between two VCs to detect a transition. However, due to the indirect jump, existing work [51, 52, 54] is limited in data flow analysis since they cannot identify some data flow paths. To build a complete and accurate model, we leverage simulated execution to tackle the problem.

4.1 Identifying Screen Transitions

Screen transitions can be implemented both in app's code and in layout files, we perform static analysis on them separately.

▹ **Analyzing App Binary**. Developers can programmably invoke a VC transition API (listed in Table 1) to trigger a screen transition. Therefore, we analyze transition APIs in app binary to find the potential transition between two VCs. Specifically, we build the app's method call graph (MCG) and traverse MCG to identify VC transition APIs. Due to the language feature of objective-C, all method invocations are transformed into the message by compiler (§2.2), we need to resolve the actual target of function calls to make MCG complete. In detail, we follow the existing work [51] to resolve the call target and then add the corresponding edges to MCG. After identifying transition APIs, we further determine their transition source and destination screen so that we can add an edge to the model. For each VC transition API, the source screen is the view controller that invokes it, so we look for its caller in MCG. For example, the source screen for the API in Line 6 is `AController`. Since the parameter of API (i.e., `target`) stores the object representing the destination screen, we perform data flow analysis on it to determine its value, so that the destination screen can be identified.

Previous approaches [51, 52, 59] directly perform data flow analysis on assembly/IR code, thus they cannot resolve the target of indirect jumps. As a result, some data flow paths cannot be correctly resolved, leading to missing some VC transitions. To tackle this problem, we employ simulated execution to execute the corresponding code snippet on an emulator (i.e., Unicorn [38]) to resolve the target of indirect jumps. Specifically, for the VC transition API whose destination can not be determined through data flow analysis, we first perform use-def analysis on its parameter to find the instruction that defines it. Then, we conduct simulated execution from the definition instruction to this API invocation. After that, we read the value of API's parameter to identify the destination of this transition. For example, to determine the destination screen

```
01 @implementation AController {
02 -(void)btn1Click{
03     BController *vcB = [[BController alloc] init];
04     [self doJump:vcB];}
Code in 0x04 will be compiled into
    objc_msgSend(self= objc_getClass(@"AController"),
                 op= NSSelectorFromString(@"doJump"),...=vcB)
05 -(void)doJump: (BController) target{
06     [self presentViewController:target];}
```

Figure 4: Example VC Transition Code.

```
//  CYButton.h
@interface CYButton : UIButton
01 @property (nonatomic,copy) void (^tapBlock)();//declare a block
//  CYButton.m   attach block to CYButton action
02 [self addTarget:self action:@selector((^tapBlock))];
-------------------------------------
@implementation CYController
03 - (void)viewDidLoad {
04   CYButton *btn = [[CYButton alloc] init];
05   [btn addTarget:self action:@selector(onClick)]; // 1)
06   UITapGestureRecognizer *tap = [[UITapGestureRecognizer alloc]
                  initWithTarget:self action:@selector(onClick)];
07   [btn addGestureRecognizer:tap]; // 2)
08   btn.tapBlock = ^{
09      // screen transition};}
-------------------------
10 - (void)onClick {// screen transition}
```

Figure 5: Add an action to button programmatically.

```
UIStoryboardPushSegueTemplate
   UIDestinationViewControllerIdentifier = AViewController
   ...
UIRuntimeEventConnection
  UILabel =  buttonAction:    //action method
  UISource = UIButton
```

Figure 6: Reverse-engineered layout files.

of API in Line 6 of Figure 4, we first find the definition instruction for `target` in Line 3. Then, we perform simulated execution from Line 3 to Line 6 and read the register that saves `target` when the execution reaches Line 6. Since the register usually holds a pointer, by analyzing the memory data it points to, we can find that `target` is an instance of `BController`.

▹ **Analyzing Layout Files**. Besides programmatically defining the VC transitions in the code, developers can create a segue in the layout files to connect two VCs. We use ibtool [19] to decompile the layout files and then extract the VC transitions from them (i.e., from the `SegueTemplate` items, as shown in Figure 6).

4.2 Identifying Widget

We label each screen transition in the model with the widget that triggers it, which guides the testing algorithm (§6) to perform targeted testing by selecting the widget that navigates to the target screen. In iOS app, each responsive widget (e.g., Button) is associated with an action method, which is a callback to handle user event. For example, the `onClick` in Line 10 of Figure 5 is an action method, which will be executed when the user taps on button `btn`. Therefore, we first find the widget that invokes transition API in its action method, and then label the corresponding edge (i.e., screen transition) in the model with it.

Since the action method can be set either statically in the layout files or programmatically in code, we analyze them separately.

▹ **Analyzing Layout Files**. We retrieve the name of action method on widget from layout files (e.g., from `EventConnection` items shown in Figure 6) and then inspect the method implementation in binary. By checking the control flow of action method, we determine if it invokes VC transition API. If so, we label the previously identified screen transition in the model (§4.1) with that widget.

▹ **Analyzing App Binary**. Besides setting action in layout files, developers can programmatically add an action to a widget in two ways. First, as shown in Line 5 of Figure 5, developers can use `addTarget` method to set the action on a widget. Accordingly, we resolve the value of parameters `target` and `action` to connect the widget to its action method leveraging techniques in §4.1. Second, developers can attach the action to a gesture detector (Line 6) and then add this gesture detector to a widget (Line 7) as the event handler. Accordingly, we track the data flow from the gesture detector (i.e., `UITapGestureRecognizer` in Line 6), to find the widget that holds it. By doing so, we can connect the gesture action to widget and label the corresponding screen transition with this widget.

In practice, instead of passing a defined method (Line 5 of Figure 5), developers can pass arbitrary action to `addTarget` with block, which is an anonymous function in Objective-C (e.g., the right-hand side of Line 8 is a block). To do so, developers first create a block pointer `tapBlock` (Line 1) in `CYButton`, and set it as `CYButton`'s action method (Line 2). After that, they can encapsulate the code into a block and assign it to `tapBlock` (Line 8), so that `CYButton`'s action method will point to this block of code. To handle it, based on the observation that the block assignment (in Line 8) is done by a method that is automatically generated by Apple's framework (i.e., code in Line 8 will be compiled into `[btn set_tapBlock_Block]`), we use a regular expression to find all block assignment methods. Then, we perform data flow analysis on each block assignment method to find the block implementation, which is further inspected to determine whether it contains a transition API invocation.

5 RUN-TIME INFORMATION RETRIEVER

This module retrieves the necessary run-time information (i.e., UI hierarchy and app context), which is further used to generate test inputs. In particular, we leverage UI hierarchy to find all feasible UI events (i.e., widget) on the screen, and app context to map the run-time state to the corresponding state in the model. Then, our exploration strategy (§6) can use the information to decide the proper UI events to be executed.

To obtain the app context, we capture the name of view controller (VC) displayed on the device. Since developers can organize multiple view controllers in different patterns to compose a single screen, we propose an algorithm to identify the correct VC that is responsible for the GUI on the device in §5.1. In model-based testing, for the purpose of avoiding redundant tests, visited widgets need to be recorded to keep the testing history [70]. Since the widget in iOS doesn't have attributes for uniquely identifying it, we propose a new approach to uniquely label each widget (§5.2). Moreover, to drive UI testing, we introduce how CydiOS injects UI events in §5.3.

This module is implemented as an iPhone extension. We build it using theos [32], a development tool integrated with Cydia Substrate framework [10] which allows analysts to hook the code of apps. Apart from capturing run-time UI information, to drive UI

testing, the module uses a built-in server to receive test commands from host computer and injects corresponding UI actions to apps.

5.1 Retrieving App Context

Since each state in the model denotes a separate view controller, to map each run-time state to its corresponding state in the model, we obtain the view controller displayed on the device, i.e., app context.

However, as the example in Figure 1, there can be multiple view controllers organized in different patterns to compose a single screen, therefore it is challenging to determine the correct VC that is responsible for the GUI on the device, which complicates the task of obtaining app context. For example, if the model built in §4 contains a transition from `STHomeController` (in Figure 1) to another VC. When the testing reaches the screen shown in Figure 1, if we wrongly treat the `BaseNavigationController` as app context, we cannot detect this transition according to the model. To mitigate this problem, we identify the view controller that is responsible for the GUI on the current screen and name it the topmost VC.

Specifically, we investigate Apple's developer documentation and forums [5, 18] to gain comprehensive knowledge about common patterns of organizing view controllers (see §2.1) and design Algorithm 1 to identify topmost VC. Specifically, we first obtain the current displayed Window (Line 2), which is the top container for the app UI (§2.1). Then, we traverse the view controller hierarchy on it from its root view controller (Line 3). As we find there are 3 common patterns to present a VC on the device, during the traverse, ① if the current VC is a container VC (e.g., Figure 2a, 2b), since it contains multiple child VCs, we invoke the related API to obtain the VC currently displayed by it (Line 7 and Line 9). For example, if the current VC is a Navigation controller, we obtained the VC which is currently at the top of the navigation stack by invoking `topViewController`. ② If the current VC separately displays a modal VC on its top (e.g., Figure 2c), we obtain the pop-up modal VC by invoking `presentedViewController` [35] (Line 11). ③ In practice, we find developers can also implement custom container by adding child VCs to a normal VC (e.g., `UIViewController`). Therefore, in Line 13, we will retrieve the current displayed child VC in the custom container [9]. By iteratively repeating this process, we can identify the topmost VC, which is responsible for the GUI on the current screen.

After determining the topmost VC of the current screen, we recursively traverse its subviews to dump the UI hierarchy on the screen, which reflects all feasible UI events. During traversing, we filter out the invisible widgets (i.e., whose x coordinate of its top left corner exceeds the screen width, the `isHidden` attribute is true, or the height is zero).

Since the `makeKeyAndVisible` method defined in `UIWindow` gets fired when app is launched and is executed in the main thread (i.e., UI thread), we hook it to retrieve the app run-time information. Specifically, we create a timer task to periodically retrieve the run-time information and synchronize it with host computer. The time interval can be adjusted by users. Currently, our iPhone extension captures app context and UI hierarchy including widget's class name, type, size, location, text, and screenshot. To offload work from UI thread (i.e., main thread), our iPhone extension does IO (file and network IO) operations in low priority threads.

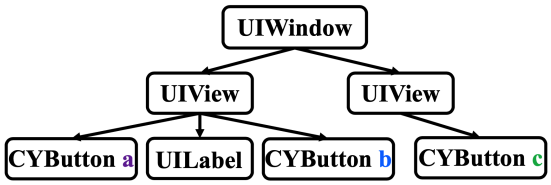


Figure 7: UI hierarchy of AController.

5.2 Labeling UI Element

After dumping the UI hierarchy, we need to uniquely label each UI widget within it for the purpose of recording the testing history by saving the unique ID of visited widgets. The recorded testing history can be used by the testing algorithm (§6) to guide testing. For example, we can determine if a UI widget has been explored before, or whether it will navigate to another screen.

Previous work [71] simply uses some attributes of widget as the identifier, such as `accessibility id` and `Label`, which cannot be used to uniquely identify all widgets in the app. Specifically, in practice, we find that most developers leave the `accessibility id` blank, while the `Label` attribute is not available in non-texture UI elements (e.g., `UIImageView`).

To tackle the problem, we design a new approach to generate a unique label for each widget. Specifically, we label each widget with the screen name and its path in the UI hierarchy of this screen. For example, Figure 7 displays the UI hierarchy of `AController`. To label `CYButton b`, we trace the path from the root view (i.e., UIWindow) to it, and then concatenate class names of views in the path. That is, the label for `CYButton b` is `<AController,UIImageView/CYButton>`.

However, in some cases, a parent view can have several child views of the same class, making it hard to distinguish between them. For example, the two children of `UIView` have the same type `CYButton`, we cannot differentiate them through class name. To address the issue, we first put child views into different lists according to their types, so that we can use the index in the list to uniquely label each child view. For example, the label for `CYButton b` would be `<AController,UIView[0]/CYButton[1]>`.

Since UITableview can reuse cells within it, we cannot use index to uniquely label each cell. For example, in Figure 8, each `UITableViewCell` which represents a single blue row in the table, is a child view of the UITableview. At first, the index for "Cell 0" is 0. If we vertically scroll the table and scroll out "Cell 0", at this time "Cell 1" is on the top and the index for it becomes 0. As a result, it causes "Cell 1" to be misidentified as "Cell 0", although these two cells can have different functionalities. To handle the case, we use the row in the table to differentiate them instead of using index.

5.3 Injecting UI Events

CydiOS injects UI events to drive UI tests. Specifically, we integrate PTFakeTouch [26], a user action simulation framework, into our iPhone extension. For the common widgets (e.g., `UIButton` and `UIImageView`), we inject click events on them. For `UIScrollView`, we simulate a scroll action. For widgets like `UISearchBar` that receive user textural input, we feed them a random string by directly modifying their text attribute.

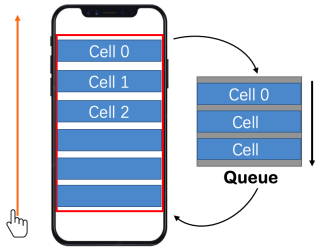


Figure 8: Reusing table cells.

Algorithm 1: Retrieve the top most view controller

```
function findTheTopMostVC()
    keyWindow = [UIApplication sharedApplication].keyWindow;
    rootController = keyWindow.rootViewController;
    topMostVC = rootController;
    while true do
        if topMostVC is NavigationController then
            topMostVC = [topMostVC topViewController];
        else if topMostVC is TabBarController then
            topMostVC = [topMostVC selectedViewController];
        else if topMostVC has presentedViewController then
            topMostVC = [topMostVC presentedViewController];
        else if [topMostVC.childViewControllers count] > 0 then
            topMostVC = [[topMostVC childViewControllers]
              lastObject];
        else
            break;
    return topMostVC
End Function
```

Apart from that, we also handle some unusual events since they can potentially interrupt UI testing. For example, when tapping on the search bar, the keyboard will pop up automatically, which actually creates a new window situated at the top of screen, preventing us from capturing the current UI hierarchy and app context. To handle it, our iPhone extension hooks `willMoveToWindow`, which gets fired when the keyboard is popped up, so that we can invoke `dismissKeyboard` API to dismiss keyboard from the screen.

6 POTENTIAL-AWARE UI TESTING

To facilitate UI testing, we propose a potential-aware exploration algorithm that guides UI exploration effectively. Specifically, we define the metric *exploration score* (ES) of a widget as its potential to discover an unseen screen. The potential for a widget to reach a new screen is two-fold. One is that it can directly navigate to a new screen (e.g., widget a1 in Figure 9 directly navigates to Screen B.), and the other is that it can indirectly reach the new screen (e.g., widget a1 is on a path toward screen E). Unlike previous work [54, 70] which only considers the potential of direct exploration, we also measure the potential of indirect exploration, thus we can perceive the widget's potential globally in the model and guide the testing more effectively. The exploration process is optimized by cherry-picking widgets for testing according to ES.

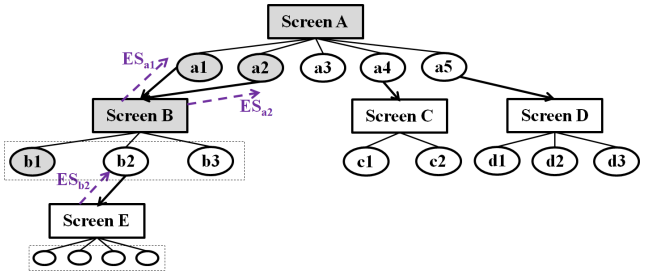


Figure 9: Dynamic transition graph. Rectangle node represents a screen, elliptical node represents a widget. solid line without arrow indicates a subordination between screen and widget, solid line arrow indicates the screen transition. Grey screens and widgets have been explored by testing.

In a nutshell, our algorithm is roughly comprised of three steps, including ❶ estimating the ES of widgets, ❷ selecting widgets for testing, and ❸ updating the ES of contextual widgets.

❶ Estimating ES of widget. When reaching a new screen $\mathcal{A}$, for each widget v_k on it, we estimate its ES as:

$$ES_k = \begin{cases} 1, & (\mathcal{A} \xrightarrow{v_k} \mathcal{B}) \in T_{\mathcal{A}}, \\ \mu, & \mathcal{A} \xrightarrow{v_k} \emptyset \wedge T_{\mathcal{A}} \neq \emptyset, \\ \delta, & T_{\mathcal{A}} = \emptyset, \end{cases} \tag{1}$$

where $T_{\mathcal{A}}$ represents a set containing all unexplored transitions from $\mathcal{A}$.

If v_k is identified to trigger a new transition from $\mathcal{A}$ to another screen, say $\mathcal{B}$, according to static analysis, we set its ES to be 1. If v_k is not identified to trigger a transition but there are still unexplored transitions from screen $\mathcal{A}$ (i.e., $T_{\mathcal{A}} \neq \emptyset$). we set its ES as an empirical parameter μ. The optimal empirical parameters may vary for different types of widgets, because some widgets such as `UIButton` are more interactive and tend to trigger a transition. We learn the empirical parameters for major widget types in iOS from an app set comprised of 50 representative iOS apps. When all statically detected transitions from screen $\mathcal{A}$ have been triggered (i.e., $T_{\mathcal{A}} = \emptyset$), we specify v_k to be a small value δ in case there may be a few false negatives in static analysis.

❷ Selecting widgets for testing. After estimating the ES of widgets on the screen, We select the widget for testing using the ϵ-greedy strategy, which is a common strategy to balance exploration and exploitation in reinforcement learning [67]. Specifically, the widget v_i will be selected with the probability

$$p_i = \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{|\mathcal{A}|}, & v_i = \arg\max\limits_{v_j \in \mathcal{A}} ES_j, \\ \dfrac{\epsilon}{|\mathcal{A}|}, & \text{otherwise.} \end{cases} \tag{2}$$

The value of ϵ is set as 0.2 by default. To test the selected widget, we employ the iPhone extension to inject an action (click or scroll) on it according to its type. If the selected widget fails to trigger a transition, we update its ES to 0.

❸ Updating the ES of contextual widgets. After ESes of new widgets have been estimated (see ❶) or the ES of a tested widget has been updated (see ❷), we also need to update ESes of their

contextual widgets. Formally, we update ESes of contextual widgets in a recursive fashion. Without loss of generality, assume that v_j is a widget on the screen $\mathcal{S}_1$ and clicking on v_j triggers the transition to $\mathcal{S}_2$, i.e., $\mathcal{S}_1 \xrightarrow{v_j} \mathcal{S}_2$. We update the ES of v_j as

$$ES_j = \eta \sum_{v_i \in \mathcal{S}_2} p_i ES_i \quad \text{for } v_j \in \mathcal{S}_1 \wedge \mathcal{S}_1 \xrightarrow{v_j} \mathcal{S}_2, \tag{3}$$

where η is a discount factor and p_i is the probability that v_i on screen $\mathcal{S}_2$ will be selected using a ϵ-greedy strategy.

By iteratively repeating the process described in Eq. (3), we can propagate the potential change of a widget throughout the whole model. It enables us to reach deeper app space, because the potential of widgets in deep space will also be perceived. Taking an example from Figure 9, after exploring b2, the testing reaches screen E. Then we first update the ES of b2. Since a1 and a2 can navigate to Screen B where b2 is on, we also update their ES.

During the testing, if the app enters an unseen screen, we will add this transition to the model. Therefore we can dynamically refine the model.

7 EVALUATION

We evaluate CydiOS by answering 4 research questions (RQs).
RQ1: Can CydiOS accurately build the model for iOS app?
RQ2: Can CydiOS accurately identify the current app context?
RQ3: Can CydiOS achieve higher coverage than existing work?
RQ4: Can CydiOS drive testing more effectively than existing work?

7.1 Environment Setup

Four jailbroken devices are used in the experiments: an iPhone 6s (iOS 10.3.1), an iPhone 6s (11.2.6), an iPhone 7 (13.3.1), and an iPhone SE (13.4.1). The host computer is an iMac with 8 cores and 32 GB memory. Devices and host computers connect to the same local WiFi for communication.

7.2 RQ1: Model Construction

Methodology. To evaluate the model generated by CydiOS, we randomly download 20 open-source Objective-C iOS apps from the list [8]. When downloading apps, we skip those that we cannot build. For each downloaded app, we use Xcode to build it for arm64 architecture, and apply our static analysis to the app's binary to extract static model. By manually reading source code and layout files, we get ground truth and compare with the generated model.
Result. Table 2 lists the experiment results. Cruiser represents the model we generated by following the previous approaches (i.e, analyzing the assembly code) [51, 59]. "Transition" column represents the screen transitions discovered by tools. "Widget" column represents the widget that triggers screen transition. On average, the precision of CydiOS in identifying VC transitions reaches 100%, and the recall reaches 94.9%. The precision in identifying widgets is 95.5% and the recall is 76.3%. The reason for precision loss is: (1) Some apps have action methods (e.g., onClick in Figure 5) with the same name, we don't try to distinguish them. Consequently, the wrongly correlated action method leads to false positives. The reasons for recall loss are: (1) Some methods defined in UIKit are not implemented in app code so we cannot simulate their execution, resulting in the incomplete data flow. (2) When performing inter-procedure def-use analysis, some data flow cannot be identified due to indirect jump. As a result, we cannot find the target instructions to start simulation execution, causing false negatives.

Table 2: Compare the model built by CydiOS and Cruiser.

App Name	Cruiser				CydiOS			
	Transition		Widget		Transition		Widget	
	Precison	Recall	Precison	Recall	Precison	Recall	Precison	Recall
XCFApp	34/41	34/78	21/27	21/83	75/75	75/78	77/83	77/83
SXBaiduDoctor	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8
HPYZhiHuDaily	5/8	5/9	6/7	6/9	9/9	9/9	8/8	8/9
Pica	42/51	42/65	33/36	36/59	62/62	57/65	51/53	51/59
LovePlayNews	18/18	18/25	8/10	8/20	25/25	25/25	11/13	11/20
UCToutiaoClone	7/7	7/7	6/6	6/6	7/7	7/7	6/6	6/6
OpenEyesDemo	5/8	5/15	2/6	2/8	13/13	13/15	8/8	8/8
DpWeibo	13/18	13/33	6/10	6/27	31/31	31/33	20/21	20/27
whatsapp-ios	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
ToDoList	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
Timi	10/11	10/15	6/8	6/11	15/15	15/15	10/10	10/11
DDNews	6/6	6/7	6/6	6/7	7/7	7/7	7/7	7/7
WhatsUrName	2/3	2/5	1/2	1/4	5/5	5/5	4/4	4/4
loveFreshPeak	3/3	3/4	2/3	2/4	4/4	4/4	3/3	3/4
DAudiobook	13/15	13/15	8/9	8/14	15/15	15/15	13/13	13/14
Shop-for-iOS	11/11	11/25	9/10	9/25	18/18	18/25	18/18	18/25
Concentration	4/7	4/8	8/8	8/8	8/8	8/8	8/8	18/25
Nebula	22/31	22/43	15/21	15/39	41/41	41/46	27/30	27/39
ZFCityGuides	5/5	5/11	4/6	4/11	11/11	11/11	8/8	8/11
eyepetizer	14/19	14/25	8/10	8/21	24/24	24/25	18/18	18/21
	82.7%	56.2%	80.8%	43.1%	100%	94.9%	95.5%	76.3%

False negatives in Cruiser are mainly caused by the indirect jump, and false positives in Cruiser are mainly caused by the imprecise reference analysis. In the following example, the return type of the method `instantiateViewControllerWithIdentifier` is a `UIViewController` pointer. When resolving `vc` (which is a `AVC` pointer), Cruiser analyzes the allocation on the right-hand side to determine its type, and thus it mis-identifies `vc` as a `UIViewController` pointer. Similarly, false negatives are also generated when widgets are initialized with the method `loadNibNamed`. We use simulated execution to handle these special cases.

```
1 AVC *vc = [Storyboard
      instantiateViewControllerWithIdentifier:@"AVC"];
2 [self presentViewController: vc]
```

Compared with previous approaches, the model built by CydiOS is more complete and accurate, which proves that simulated execution significantly enhances static analysis.

Answer to RQ1: `CydiOS` achieves a high precision and recall when building the model.

7.3 RQ2: App Context

Methodology. We re-use the apps in §7.2 to evaluate whether `CydiOS` can identify the correct view controller which is responsible for the GUI on the device (i.e., the topmost VC). For each app, we randomly go through 3 screens, and then use `CydiOS` to identify the name of topmost VC. By manually reading apps' source code and using the "debug view" functionality of Xcode, which can inspect view controller hierarchy on the screen, we can determine the topmost VC on the screen and validate the result of `CydiOS`.
Result. CydiOS can identify the topmost VC for 20×3 screens. The precision is 100%, which proves that Algorithm 1 is effective.

Answer to RQ2: CydiOS can correctly identify the top most VC on the screen as app context.

7.4 RQ3: Testing Coverage

Methodology. To study the effectiveness of CydiOS, we compare it with the commonly used DFS testing strategy [45, 69] and Fastmonkey [14], an open source tool that injects random events into iOS apps, resembling Monkey in Android [39]. In addition, we adapt state-of-the-art techniques on Android, stoat [70] and ape [54], according to their source code. We compare CydiOS with these tools with regard to both app screen coverage and code coverage on 50 popular apps from App Store. Note that, higher screen coverage strongly indicates a thorough exploration of app space, while higher code coverage indicates a more complete testing of app functionalities. In detail, we run each tool 3 hours for each app. For stoat, following the original settings, we assign 1 hour for model construction and 2 hours for UI testing. To reduce the measurement bias, we run each app 3 times and calculate the average result.

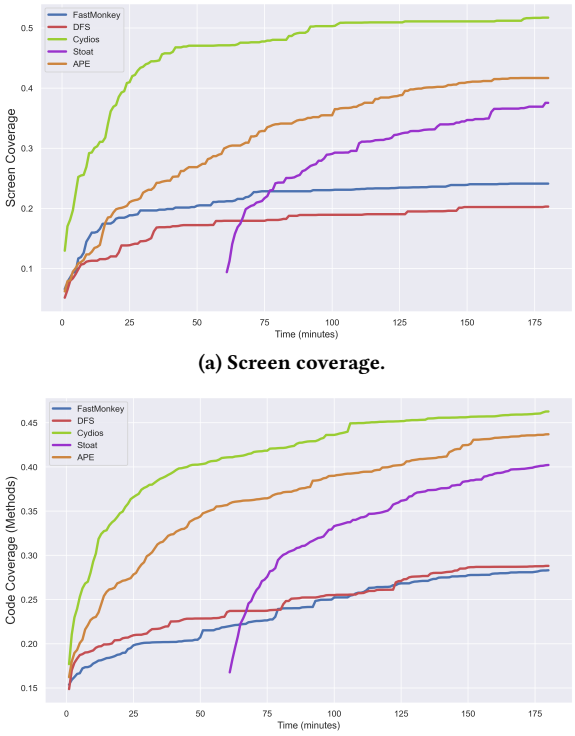
App Selection. We downloaded 50 apps in 23 categories from Apple Store to guarantee the comprehensiveness of testing. Note that, we only select the apps developed in Objective-C language. To this end, we search "swift" in app's symbol table [11] for filtering. Meanwhile, we skip the apps in the browser and game categories. To avoid potential legal risks, we also skip the apps which require a phone number or credit card to sign up with. The app list can be found in our github repository.

Code Coverage Measurement. Although Apple provides a built-in support in Xcode for code coverage to assist the unit tests of open-source app [27], there lacks a tool to measure the code coverage for iOS app binary. To address this issue, since all method invocations are executed by the messaging dispatch function `objc_msgsend` (introduced in §2.2), we leverage the method swizzling technique [6, 28] in Objective-C to dynamically instrument `objc_msgsend` to log all executed methods. Thus, we can measure how many app methods are covered by the testing.

Screen Coverage Measurement. To count the total number of screens in an app, we first use class-dump [7] to extract the header files from the app, and then look for the class that implements `UIViewController`, which is the base class for all custom VCs.

Result. Figure 10 illustrates the measurement results. "DFS" denotes the tool that adopts depth-first search strategy on our extension. In Figure 10a, we plot the mean of screen coverage minute-to-minute. We discover that CydiOS is very efficient in covering new screens. The screen coverage of CydiOS grows much faster than the other three tools. Figure 10b details code coverage trends of different tools. We count the number of executed methods in apps. We notice that the result is consistent with Figure 10a. CydiOS shows a significant advantage in testing effectiveness compared with other tools.

Due to lack of guidance, random strategy generates many repetitive and redundant test events, resulting in poor performance. Since DFS strategy always gets stuck in deep states, it cannot explore other paths, leading to low coverage. Specifically, "DFS" only restarts the app until it is crashed or in the background (e.g., navigate to the browser after clicking an ad), it rarely gets out of the current navigation path and moves to a new path. Even worse, within a single screen, many widgets can navigate to the same



(a) Screen coverage.

(b) Code coverage.

Figure 10: Testing result on 50 apps.

target. For example, on the home screen of a news app, clicking on different news goes to the same screen with different news content loaded. Therefore, for each news, "DFS" has to repeatedly go to the same "news detail" screen and explore all the widgets on it.

The reason why stoat did not has a good performance is threefold. First, stoat cannot build a dynamic model effectively for iOS apps and therefore fails to provide an effective guide to UI testing. When constructing the dynamic model, Stoat only considers executable widgets (e.g., Button). However, it cannot distinguish executable widgets from non-executable widgets (i.e., the widget that cannot be interacted with) in iOS apps, so it wastes much time exploring non-executable widgets. Meanwhile, since stoat gives "Back" button the lowest priority (the intent is to explore all executable widgets on a screen and then go back), it will get stuck in a path during the model construction. Second, when performing the UI testing, the event sequence length of test input generated by stoat is only 20, which is not enough to reach deep app states. Third, stoat wastes much time restarting apps and many test budget is spent on the initial screens. Since we learn the potential of each widget type empirically, CydiOS are more likely to click the executable widgets. Meanwhile, CydiOS only restart the app when the app crashes or in background, it can spend more test budget on exploring unvisited app space.

Although ape generates a longer event sequence (i.e., 300), it employs a greedy strategy rather than perceiving the widget potential globally. Also, ape is not able to identify the executable widgets. For these reasons, the performance of ape is worse than CydiOS.

Moreover, we study the reasons why CydiOS cannot reach some screens and execute some code. The main reason is that some

screens are inaccessible until the precondition is satisfied. For example, we are allowed to visit the screen that holds premium features only if we buy a VIP service. Meanwhile, some apps require a valid and meaningful textural input to go to the target screen and show the content correlation with user input. For example, on travel apps, we need to feed the name of a real city to the search bar, so that we can see the tourist information. Some apps have a weak network connection to the remote server, which causes the screen cannot be loaded, making CydiOS click on the "Back" button.

Answer to RQ3: `CydiOS` can achieve both higher screen coverage and code coverage compared with representative algorithms.

7.5 RQ4: Performance

Methodology. To validate if our iPhone extension can capture the UI hierarchy on the screen accurately and drive UI testing effectively, we compare it with Apple's native driver XCUITest, and a commercial UI testing framework Appium. We randomly choose 5 open-source apps. For each app, we randomly choose 3 screens and run three different UI drivers. We measure the time it takes these drivers to dump the UI hierarchy of the app screens, and to inject a click event on the screens. We also compare the UI hierarchies extracted by these three drivers. To get the ground truth, for each app, we use the "debug view" functionality provided by Xcode to capture the target UI Hierarchy.

Table 3: Compare the performance of CydiOS with other tools.

App Name	Screen	XCUITest			Appium			CydiOS		
		d_T	e_T	Ele	d_T	e_T	Ele	d_t	e_T	Ele
XCFApp	1	0.127	0.227	21	0.862	0.275	21	0.051	0.014	21
	2	0.146	0.236	30	1.198	0.283	30	0.045s	0.013	30
	3	0.115	0.190	12	0.604	0.290	12	0.043	0.017	12
DPWeibo	1	1.223	0.719	442	16.02	0.462	442	0.031	0.024	58
	2	0.137	0.193	22	0.566	0.270	22	0.015	0.008	22
	3	0.991	0.368	365	12.30	0.3984	365	0.033	0.012	47
DDNews	1	0.629	0.478	213	14.75	0.793	213	0.058	0.010	57
	2	0.974	0.741	420	20.66	0.893	420	0.063	0.007	64
	3	0.114	0.205	16	0.653	0.280	16	0.031	0.005	16
LoveNews	1	0.138	0.212	31	0.982	0.286	31	0.050	0.027	31
	2	0.368	0.239	51	1.937	0.291	51	0.058	0.027	51
	3	0.149	0.217	21	0.930	0.282	21	0.043	0.026	21
Diary	1	0.106	0.182	8	0.403	0.253	8	0.027	0.006	8
	2	0.122	0.191	20	0.518	0.293	20	0.040	0.010	20
	3	0.132	0.202	15	0.377	0.274	15	0.044	0.014	7

- d_T represents the time to dump UI hierarchy, e_T represents the time to execute a UI action.
- Ele represents the number of widgets in the UI hierarchy dumped by these tools.

Result. Table 3 reports the experimental results. The results show that `CydiOS` requires a minimum of time to take an action, which indicates it is able to run the test effectively. XCUITest uses a client-server architecture, it consumes extra time to deliver the command from the script running on our computer to the device. Considering the command from Appium script is first transferred from the Appium server to WDA, then from WDA to XCUITest, Appium takes more time to execute a command.

Meanwhile, we discover that `CydiOS` has higher efficiency in dumping UI hierarchy. It is surprising that `CydiOS` outperforms Apple's native driver XCUITest. The reason is that XCUITest in practice dumps many widgets that are invisible on current screen. For example, the DPWeibo(a Twitter-liked app) only displays 2 tweets on the device. However, the UI hierarchy dumped by XCUITest contains 137 tweets. From the text attributes of these twitters, we find that they are the tweets to be shown after a swipe down. Another example is Diary app, we find that the widgets on the covered screen

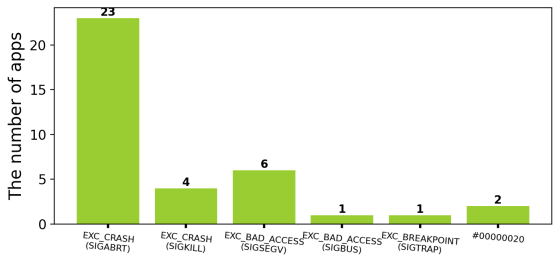


Figure 11: Fuzzing result.

(e.g., Figure 2c) will also be dumped by XCUITest. Therefore the XCUITest has to process more redundant widgets (e.g, invisible widgets) to build the UI hierarchy. For Appium, which runs XCUITest as the backend, it takes more time to dump UI hierarchy. Besides the cost on message transfer, the main reason is that Appium takes many efforts to parse the UI hierarchy obtained from XCUITest. It decodes the data and generates XPath for each widget, which is very time-consuming. As shown in Table 3, the time for Appium to dump a UI hierarchy increases corresponding to the number of widgets within it. Since the UI hierarchy from XCUITest contains many redundant widgets, it could explain why Appium performs worse than `CydiOS` while we also generate an XPath-liked identifier. This is also the reason why Appium on iOS runs tests slower than on Android platform, given that Appium on Android is based on UiAutomator [41], which will not dump the widgets that are not displayed on the current screen. This experiment also shows the benefits of Algorithm 1 (i.e., to identify top most VC), which not only identifies the correct app context, but also helps dump the correct UI hierarchy.

Answer to RQ4: `CydiOS` can effectively drive UI testing compared with other testing frameworks.

8 APPLICATION

We further present two use cases to demonstrate CydiOS's practicality in finding bugs(§8.1) and detecting users' personal information leakage in iOS apps(§8.2).

8.1 Fuzzing

We randomly download 200 apps developed in OC from Apple Store. For each app, we run CydiOS to test it for 1 hour. We detect the bugs by capturing app's crash log through Xcode [16] and manually analyzing the stack trace in the log to check if the crash is caused by app code. The result is shown in Figure 11. CydiOS detects 37 bugs crashes from 32 apps in 6 exception types. The most common exception type is EXC_CRASH (SIGABRT), which is most likely to be caused by some unhandled exception in the code. We have reported these crashes to developers. The testing result shows the promising ability of CydiOS in UI fuzzing.

8.2 Privacy Leakage Detection

Tracking libraries are widely used in iOS apps to collect user data for user profiling. Compared with Android [62, 72], little has been done to study user information leaked to tracking libraries in iOS. While

dynamic analysis is an effective way to learn data leakage, due to the absence of an automated UI testing tool for iOS, the recent work [58] only analyzes network traffics at application launch time. CydiOS can facilitate dynamic analysis to detect more privacy leakage.

We use HKET app as an example, which uses Google Analytics for user tracking. We first read the online documentation provided by Google Analytics to find all tracking APIs, through which developers can pass user data to it. After that, we use Frida [12] to hook these APIs and use CydiOS to run the app for 30 minutes. By analyzing the log, we find it leaks user behavior (e.g., visited screen, purchase information) and device ID to Google Analytics.

9 RELATED WORK

9.1 Automated UI Testing for Mobile Apps

Model-based Testing. Model-based testing drives the test generation according to the constructed model. Since the accuracy and completeness of model is important to the correctness of test generation, many efforts have been made to model construction. A3E [45], Gator [75] build a static window transition graph(WTG), which may contain infeasible transition due to the over-approximation of static analysis. Stoat [70], APE [54],MobiGUITAR [43],DroidBot [60] use the information monitored at runtime to build model. However, the built model is incomplete since dynamic execution cannot guarantee coverage. ProMal [61] is a hybrid approach, which applies dynamic analysis to verify the static WTG, and leverages machine learning to predict unverified transitions.

Testing strategy is another important component for MBT. MobiGUITAR [43],DroidBot [60],GUIRipper [42] simply use depth-first strategy to explore the model space, thus their performance is limited. Stoat [70] adopts a probabilistic fuzzing strategy. It distributes different probabilities of being selected for the widgets in the model, and generates optimized test inputs by iteratively evolving the model.

Search-based testing. Some researchers adopt search based testing to explore and optimize test inputs. Sapienz [65] employs a genetic algorithm that optimizes randomly generated tests to maximize code coverage and fault revealing. EvoDroid [64] uses evolutionary algorithms that generate more complex test sequences to reach the deep point in app. ACTEVE [44] applies concolic testing to identify valid UI events to explore more code branches.

9.2 Existing iOS UI Testing Frameworks

UIAutomation [29] is a native framework provided by Apple, which provides convenient APIs to help testers to write test cases. With UIAutomation deprecated from XCode8, Apple replaces it with XCUITest framework [17]. Based on it, Facebook proposes a WebDriver server named WebDriverAgent (WDA) [13], which invokes XCUITest framework to achieve remote control of iOS devices. While the master branch of WDA is archived by Facebook, Appium [31] creates a forked branch and uses WDA to proxy the test commands from Appium script to XCUITest framework. EarlGrey [73] is another UI testing framework based on the XCUITest, developed by Google. It provides strong UI synchronization to enhance the test stability. However, all these tools requires human-implemented scripts, which makes them inefficient. Because writing test scripts can be labor-intensive and the quality of scripts cannot be guaranteed. Meanwhile, all these frameworks cannot retrieve app context information to map the current app state in the model, thus cannot be applied to model-based testing.

9.3 iOS Analysis

PIOS [51] generates a control-flow graph for iOS app, and performs reachable analysis on it to detect privacy leaks. Based on PIOS, Cruiser [59] build a view controller graph to detect hidden screens in Crowdturfing apps. iRiS [50] combines static analysis with binary instrumentation to detect private APIs misuse. Chen et al. [48] identify potentially harmful iOS libraries by comparing their API sequences with Android counterparts. Feichtner et al. [52] lift iOS binary to LLVM IR, and apply static analysis to IR code to find improper usage of cryptographic APIs.

9.4 Cross-platform Test-cases Migration

Some works focus on cross platform test-cases mitigation. Qin et al. [68] propose a framework, namely TestMig to mitigate the iOS test cases to Android platform. In detail, to map the UI events, it calculate the similarity between iOS UI controls and Android UI controls based on the UI control attributes (e.g., ID, label). While TestMig is only allowed to generate test inputs for Android apps, Talebipour et al. [71] presented MAPIT for bi-directional test case transfer. MAPIT can automatically translate the source test script (Appium, Robot [25]) into the script for target platform. It uses Appium to extract the UI hierarchy and map the UI event by calculating the visual and textual similarity of the widget on the screen.

10 THREAT TO VALIDITY

In this work, we identified the following major threats to validity. (1) Intrusiveness: The iPhone extension in CydiOS works on jailbroken iPhones and hooks app to retrieve runtime information. However, in practice, we find some apps either implement jailbreak detection or enable anti-hook protection. To mitigate this threat, we can leverage the tools (e.g., shadow [1]) to bypass the detection. (2) Textual input: Some widgets like search bar, receive user textual input. In this paper, we simply feed them a meaningless string. However, in some cases, it cannot meet the condition (i.e., input should be a city name) and trigger app behaviors, thus reducing the testing effectiveness. In the future, we will follow previous work [65] to extract statically-defined strings in the app, which proves to improve the performance of search-based testing for web app,and use these strings as textual input.

11 CONCLUSION

Automated UI testing is a popular method to inspect app's functionalities and improve its reliability. In this paper, we take the first step to study the UI testing on iOS platform. Specifically, we propose CydiOS, a model-based testing tool for iOS app. CydiOS builds a static model for target app via an enhanced static analysis. During the testing, we use an iPhone extension to retrieve runtime information, which is further used by a potential-aware search algorithm to guide testing effectively. Our evaluation results on real world apps show that CydiOS can achieve both good screen coverage and code coverage.

REFERENCES

[1] 2022. A jailbreak detection bypass for modern iOS jailbreaks. https:/https://github.com/jjolano/shadow/.
[2] 2022. A lightweight and modular front-end framework for developing fast and powerful web interfaces. https://getuikit.com/.
[3] 2022. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb.
[4] 2022. App Review. https://developer.apple.com/app-store/review/.
[5] 2022. Apple Developer Forums. https://developer.apple.com/forums/.
[6] 2022. Brief discussion about iOS swizzle . https://juejin.cn/post/6844903856497754126.
[7] 2022. class-dump: Generate Objective-C headers from Mach-O files. https://github.com/nygard/class-dump.
[8] 2022. Collaborative List of Open-Source iOS Apps . https://github.com/dkhamsing/open-source-ios-apps.
[9] 2022. Current ChildView Controller. https://stackoverflow.com/questions/14405490/current-childview-controller/.
[10] 2022. Cydia Substrate. http://www.cydiasubstrate.com/.
[11] 2022. Detect if iOS App is written in Swift. https://stackoverflow.com/questions/32882208/detect-if-ios-app-is-written-in-swift.
[12] 2022. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/.
[13] 2022. Facebook WebDriverAgent. https://github.com/facebookarchive/WebDriverAgent/.
[14] 2022. Fastmonkey. https://github.com/zhangzhao4444/Fastmonkey.git.
[15] 2022. Global Premium Smartphone Market Sales Reach Highest Ever in 2021. https://www.counterpointresearch.com/global-premium-smartphone-market-2021/.
[16] 2022. How to view crash reports in XCode. https://stackoverflow.com/questions/69123921/how-to-view-crash-reports-in-xcode.
[17] 2022. introduction to ios test automation with xcuitest. https://testautomationu.applitools.com/introduction-to-ios-test-automation-with-xcuitest/chapter1.html/.
[18] 2022. iOS App Dev Tutorials. https://developer.apple.com/tutorials/app-dev-training/.
[19] 2022. iOS Interface Builder utility, implemented in python. https://github.com/davidquesada/ibtool.
[20] 2022. iOS UIWindow. https://developer.apple.com/documentation/uikit/uiwindow.
[21] 2022. iPhone Users and Sales Stats for 2022. https://backlinko.com/iphone-users/.
[22] 2022. Jump with Indirect Operand. http://www.c-jump.com/CIS77/ASM/FlowControl/C77_0040_jump_indirect.htm.
[23] 2022. objc_msgsend. https://developer.apple.com/documentation/objectivec/1456712-objc_msgsend.
[24] 2022. Programming with Objective-C: About Objective-C. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html.
[25] 2022. robot framework. https://robotframework.org/.
[26] 2022. Simulate touch events for iOS User mode. https://github.com/Ret70/PTFakeTouch.
[27] 2022. Testing with Xcode Code Coverage. https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/07-code_coverage.html.
[28] 2022. The Right Way to Swizzle in Objective-C. https://newrelic.com/blog/best-practices/right-way-to-swizzle.
[29] 2022. The UIAutomation Driver for iOS. http://appium.io/docs/en/drivers/ios-uiautomation/.
[30] 2022. The View Controller Hierarchy. https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/TheViewControllerHierarchy.html.
[31] 2022. The XCUITest Driver for iOS. https://appium.io/docs/en/drivers/ios-xcuitest/.
[32] 2022. theos framework. https://github.com/theos/theos.
[33] 2022. UINavigationController. https://developer.apple.com/documentation/uikit/uinavigationcontroller.
[34] 2022. UITabBarController. https://developer.apple.com/documentation/uikit/uitabbarcontroller.
[35] 2022. UIViewController presentedviewcontroller. https://developer.apple.com/documentation/uikit/uiviewcontroller/1621407-presentedviewcontroller.
[36] 2022. Understanding Navigation in iOS. https://guides.codepath.com/ios/Understanding-Navigation-in-iOS.
[37] 2022. Understanding Windows and Screens . https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/WindowAndScreenGuide/WindowScreenRolesinApp/WindowScreenRolesinApp.html.
[38] 2022. Unicorn Engine Introduction. https://ctf-wiki.mahaloz.re/reverse/unicorn/introduction/.
[39] 2022. Using Segues. https://developer.android.com/studio/test/other-testing-tools/monkey.
[40] 2022. Using Segues. https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/UsingSegues.html.
[41] 2022. Write automated tests with UI Automator. https://developer.android.com/training/testing/other-components/ui-automator.
[42] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 258–261.
[43] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2014. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE software* 32, 5 (2014), 53–59.
[44] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.
[45] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 641–660.
[46] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 238–249.
[47] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. 2016. Curiousdroid: automated user interface interaction for android application analysis sandboxes. In *International Conference on Financial Cryptography and Data Security*. Springer, 231–249.
[48] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 357–376.
[49] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. *Acm Sigplan Notices* 48, 10 (2013), 623–640.
[50] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2015. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 44–56.
[51] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications.. In *NDSS*. 177–183.
[52] Johannes Feichtner, David Missmann, and Raphael Spreitzer. 2018. Automated Binary Analysis on iOS: A Case Study on Cryptographic Misuse in iOS Applications. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. 236–247.
[53] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 419–429.
[54] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.
[55] Vignir Gudmundsson, Mikael Lindvall, Luca Aceto, Johann Bergthorsson, and Dharmalingam Ganesan. 2016. Model-based Testing of Mobile Systems–An Empirical Study on QuizUp Android App. *arXiv preprint arXiv:1606.00503* (2016).
[56] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. 204–217.
[57] Taeyeon Ki, Alexander Simeonov, Chang Min Park, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. 2017. Fully automated ui testing system for large-scale android apps using multiple devices. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 185–185.
[58] Konrad Kollnig, Anastasia Shuba, Reuben Binns, Max Van Kleek, and Nigel Shadbolt. 2021. Are iPhones Really Better for Privacy? Comparative Study of iOS and Android Apps. *arXiv preprint arXiv:2109.13722* (2021).
[59] Yeonjoon Lee, Xueqiang Wang, Kwangwuk Lee, Xiaojing Liao, XiaoFeng Wang, Tongxin Li, and Xianghang Mi. 2019. Understanding {iOS-based} Crowdturfing Through Hidden {UI} Analysis. In *28th USENIX Security Symposium (USENIX Security 19)*. 765–781.
[60] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 23–26.
[61] Changlin Liu and Xusheng Xiao. 2021. ProMal: precise window transition graphs for Android via synergy of program analysis and machine learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 144–146.

[62] Xing Liu, Jiqiang Liu, Sencun Zhu, Wei Wang, and Xiangliang Zhang. 2019. Privacy risk analysis and mitigation of analytics libraries in the android ecosystem. *IEEE Transactions on Mobile Computing* 19, 5 (2019), 1184–1199.

[63] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 224–234.

[64] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 599–609.

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

[66] Leon J Osterweil and Lloyd D Fosdick. 1976. Program testing techniques using simulated execution. *ACM SIGSIM Simulation Digest* (1976).

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

[68] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. Testmig: Migrating gui test cases from ios to android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 284–295.

[69] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI testing for Android applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 27–37.

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

[71] Saghar Talebipour, Yixue Zhao, Luka Dojcilović, Chenggang Li, and Nenad Medvidović. 2021. UI Test Migration Across Mobile Platforms. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 756–767.

[72] Yutian Tang, Haoyu Wang, Xian Zhan, Xiapu Luo, Yajin Zhou, Hao Zhou, Qiben Yan, Yulei Sui, and Jacky Wai Keung. 2021. A systematical study on application performance management libraries for apps. *IEEE Transactions on Software Engineering* (2021).

[73] Aditya Atul Tirodkar and Sundeep Singh Khandpur. 2019. EarlGrey: iOS UI automation testing framework. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 12–15.

[74] Heila Van Der Merwe, Brink Van Der Merwe, and Willem Visser. 2012. Verifying android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[75] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25, 4 (2018), 833–873.

[76] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for android: Are we really there yet in an industrial case?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 987–992.