Reverse Engineering iOS Mobile Applications

Mona Erfani Joorabchi
University of British Columbia
Canada
merfani@ece.ubc.ca

Ali Mesbah
University of British Columbia
Canada
amesbah@ece.ubc.ca

I. Abstract

As a result of the ubiquity and popularity of smartphones, the number of third party mobile applications is explosively growing. With the increasing demands of users for new dependable applications, novel software engineering techniques and tools geared towards the mobile platform are required to support developers in their program comprehension and analysis tasks. In this paper, we propose a reverse engineering technique that automatically (1) hooks into, dynamically runs, and analyzes a given iOS mobile application, (2) exercises its user interface to cover the interaction state space and extracts information about the runtime behaviour, and (3) generates a state model of the given application, capturing the user interface states and transitions between them. Our technique is implemented in a tool called iCrawler. To evaluate our technique, we have conducted a case study using six open-source iPhone applications. The results indicate that iCrawler is capable of automatically detecting the unique states and generating a correct model of a given mobile application.

***Keywords*-reverse engineering; mobile applications; iOS; model generation**

II. Introduction

According to recent estimations [1], by 2015 over 70 percent of all handset shipments will be smartphones, capable of running mobile applications.[1] Currently, there are over 600,000 mobile applications on Apple's AppStore [2] and more than 400,000 on Android Market [3].

Some of the challenges involved in mobile application development include handling different devices, multiple operating systems (Android, Apple iOS, Windows Mobile), and different programming languages (Java, Objective-C, Visual C++). Moreover, mobile applications are developed mostly in small-scale, fast-paced projects to meet the competitive market's demand [4]. Given the plethora of different mobile applications to choose from, users show low tolerance for buggy unstable applications, which puts an indirect pressure on developers to comprehend and analyze the quality of their applications before deployment.

[1]There are two kinds of mobile applications: Native applications and Web-based applications. Throughout this paper, 'mobile application' refers to native mobile applications.

With the ever increasing demands of smartphone users for new applications, novel software engineering techniques and tools geared towards the mobile platform are required [5], [6], [7] to support mobile developers in their program comprehension, analysis and testing tasks [8], [9].

According to a recent study [10], many developers interact with the graphical user interface (GUI) to comprehend the software by creating a mental model of the application. For traditional desktop applications, an average of 48% of the application's code is devoted to GUI [11]. Because of their highly interactive nature, we believe the amount of GUI-related code is typically higher in mobile applications.

To support mobile developers in their program comprehension and analysis tasks, we propose a technique to automatically reverse engineer a given mobile application and generate a comprehensible model of the user interface states and transitions between them. In this paper, we focus on native mobile applications for the iOS platform. To the best of our knowledge, reverse engineering of iOS mobile applications has not been addressed in the literature yet.

Our paper makes the following contributions:

- A technique that automatically performs dynamic analysis of a given iPhone application by executing the program and extracting information about the runtime behaviour. Our approach exercises the application's user interface to cover the interaction state space;
- A heuristic-based algorithm for recognizing a new user interface state, composed of different UI elements and properties.
- A tool implementing our technique, called iCrawler (iPhone Crawler), capable of automatically navigating and generating a state model of a given iPhone application. This generated model can assist mobile developers to better comprehend and visualize their mobile application. It can also be used for analysis and testing purposes (i.e., smoke testing, test case generation).
- An evaluation of the technique through a case study conducted on six different open-source iPhone applications. The results of our empirical evaluation show that iCrawler is able to identify the unique states of a given iPhone application and generate its state model accurately, within the supported transitional UI elements.

III. Related Work

We divide the related work in three categories: mobile application security testing, industrial testing tools currently available to mobile developers, and GUI reverse engineering and testing.

Mobile Application Security Testing. Security testing of mobile applications has gained most of the attention from the research community when compared to other areas of research such as functional testing, maintenance, or program comprehension. Most security testing approaches are based on static analysis of mobile applications [12] to detect mobile malware. Egele et al. [13] propose PiOS to perform static taint analysis on iOS application binaries. To automatically identify possible privacy gaps, the mobile application under test is disassembled and a control flow graph is reconstructed from Objective-C binaries to find code paths from sensitive sources to sinks. Extending on PiOS, the same authors discuss the challenges involved in dynamic analysis of iOS applications and propose a prototype implementation of an Objective-C binary analyzer [14]. Interestingly, to exercise the GUIs, they use image processing techniques. This work is closest to ours. However, their approach randomly clicks on an screen area and reads the contents from the device's frame buffer and applies image processing techniques to compare screenshots and identify interactive elements. Since image comparison techniques are known to have a high rate of false positives, in our approach we "programatically" detect state changes by using a heuristic-based approach.

Industrial Testing Tools. Most industrial tools and techniques currently available for analyzing mobile applications are manual or specific to the application in a way that they require knowledge of the source code and structure of the application. For instance, Kif (Keep It Functional) [15] is an open source iOS integration test framework, which uses the assigned accessibility labels of objects to interact with the UI elements. The test runner is composed of a list of scenarios and each scenario is composed of a list of steps. Other similar frameworks are Frank [16] and Instruments [17]. A visual technology, called Sikuli [18], uses fuzzy image matching algorithms on the screenshots to determine the positions of GUI elements, such as buttons, in order to find the best matching occurrence of an image of the GUI element in the screen image. Sikuli creates keyboard and mouse click events at that position to interact with the element. There are also record and playback tools for mobile applications such as MonkeyTalk [19]. However, using such tools requires application-specific knowledge and much manual effort.

GUI Reverse Engineering and Testing. Reverse engineering of desktop user interfaces was first proposed by Memon et al. in a technique called GUI Ripping [20]. Their technique starts at the main window of a given desktop application, automatically detects all GUI widgets and analyzes the application by executing those elements. Their tool, called Guitar, generates an event-flow graph to capture a model of the application's behaviour and generate test-cases.

For web applications, Mesbah et al. [21] propose a crawling-based technique to reverse engineer the navigational structure and paths of a web application under test. The approach, called Crawljax, automatically builds a model of the application's GUI by detecting the clickable elements, exercising them, and comparing the DOM states before and after the event executions. The technique is used for automated test case generation [22] and maintenance analysis [23] in web applications.

Amalfitano et al. [24] extend on this approach and propose a GUI crawling technique for Android applications. Their prototype tool, called A2T2, manages to extract models of a small subset of widgets of an Android application.

Gimblett et al. [25] present a generic description of UI model discovery, in which a model of an interactive software is automatically discovered through simulating its user actions. Specifically they describe a reusable and abstract API for user interface discovery.

Further, Chang et al. [26] build on Sikuli, the aforementioned tool, to automate GUI testing. They help GUI testers automate regression testing by programming test cases once and repeatedly applying those test cases to check the integrity of the GUI.

Hu et al. [27] propose a technique for detecting GUI bugs for Android applications using Monkey [28], an automatic event generation tool. Their technique automatically generates test cases, feeds the application with random events, instruments the VM, and produces log/trace files to detect errors by analyzing them post-run.

To the best of our knowledge, no work has been done so far to reverse engineer Objective-C iPhone applications automatically. Our approach and algorithms are different from the aforementioned related work in the way we track the navigation within the application, retrieve the UI views and elements, and recognize a new state, which are geared towards native iPhone user interfaces.

IV. Background and Challenges

Here, we briefly describe the relevant iPhone programming concepts [17] required for understanding our approach in Section V.

Objective-C is the primary programming language used to write native iOS applications. The language adds a thin layer of object-oriented and Smalltalk-style messaging to the C programming language. Apple provides a set of Objective-C APIs collectively called *Cocoa*. Cocoa Touch is a UI framework on top of Cocoa. One of the main frameworks of Cocoa Touch is `UIKit`, which provides APIs to develope iOS user interfaces.

Figure 1: The Olympics2012 iPhone application going through a UI state transition, after a generated event.

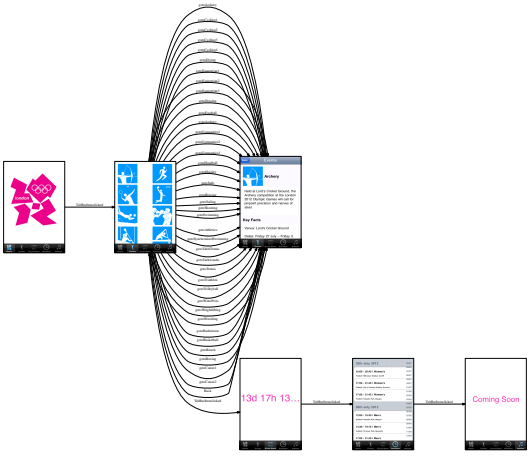

Figure 2: The generated state graph of the Olympics2012 iPhone application.

The Model-View-Controller design pattern is used for building iOS applications. In this model, the controller is a set of *view controllers* as well as the `UIApplication` object, which receives events from the system and dispatches them to other parts of the system for handling. As soon as an application is launched, the `UIApplication` main function creates a singleton *application delegate* object that takes control. The application delegate object can be accessed by invoking the shared application class method from anywhere in code.

At a minimum, a *window* object and a *view* object are required for presenting the application's content. The window provides the area for displaying the content and is loaded from the main *nib file*.[2] Standard UI elements, which are provided by the `UIKit` framework for presenting different types of content, such as labels, buttons, tables, and text fields are inherited from the `UIView` class. Views draw content in a designated rectangular area and handle *events*.

Events are objects sent to an application to inform it of user actions. Many classes in `UIKit` handle touch events in ways that are distinctive to objects of the class. The application sends these events to the view on which the touch occurred. That view analyzes the events and responds in an appropriate manner. For example, buttons and sliders are responsive to gestures such as a tap or a drag while scroll views provide scrolling behaviour for tables or text views. When the system delivers a touch event, it sends an action message to a target object when that gesture occurs.

View controllers are used to change the UI state of an application. A view controller is responsible for handling the creation and destruction of its views, and the interactions between the views and other objects in the application. The `UIKit` framework includes classes for view controllers such as `UITabBarController`, `UITableViewController` and `UINavigationController`. Because iOS applications have a limited amount of space in which to display content, view controllers also provide the infrastructure needed to swap out the views from one view controller and replace them with the views of another view controller. The most common relationships between source and destination view controllers in an iPhone application are either by using a *navigation controller*, in which a child of a navigation controller pushes another child onto the *navigation stack*, or by presenting a view controller *modally*. The navigation controller is an instance of the `UINavigationController` class and used for structured content applications to navigate between different levels of content in order to show a screen flow, whereas the modal view controllers represent an interruption to the current workflow.

Challenges. Dynamic analysis of iOS applications has a number of challenges. Most iOS applications are heavily based on event-driven graphical user interfaces. Simply launching an application will not be sufficient to infer a proper understating of the application's runtime behaviour [14]. Unfortunately, most iOS applications currently do not come with high coverage test suites. Therefore, to execute a wide range of paths and reverse engineer a representative model, an approach targeting iOS applications needs to be able to automatically change the application's state and analyze state changes.

One challenge that follows is defining and detecting a new state of an application while executing and changing its UI. In other words, automatically determining whether a state change has occurred is not that straight forward.

Another challenge, associated with tracking view controllers, revolves around the fact that firing an event on the UI could result in several different scenarios as far as the UI

[2] A nib file is a special type of resource file to store the UI elements in.

is concerned, namely, (1) the current view controller could go to the next view controller (modally, by being pushed to the navigation stack, or changing to the next tab in a tab bar view controller) or (2) UI element(s) in the current interface could be dynamically added/removed/changed, or (3) the current view controller goes back to the previous view controller (dismissed modally or popped from the navigation stack), or (4) nothing happens. Analyzing each of these scenarios requires a different way of monitoring the UI changes and the navigation stack.

V. Our Approach

Our approach revolves around dynamically running a given iOS mobile application, navigating its user interface automatically, and reverse engineering a model of the application's user interface states and transitions between them. Figure 1 shows snapshots of an iPhone application (called Olympics2012, used in our case study in Section VII) UI state transition after an event. Figure 2 shows the automatically generated state graph of the same application. The figure is minimized because of space restrictions, and it is depicted to give an impression of the graph inferred by our approach.

Figure 3 depicts the relation between our technique and a given mobile application. The following seven steps outline our technique's operation.

Step 1 - Hooking into the Application: As soon as the application is started, our technique kicks in by setting up a shared instance object. As shown in Figure 3, we immediately hook into and monitor the application delegate object to identify the initial view controller and infer information about its UI components.

Step 2 - Analyzing UI Elements: After obtaining the initial view controller, we have access to all its UI elements. We keep this information in an array associated to the view controller. Meanwhile, our technique recognizes the different types of UI elements, such as labels, buttons, table cells, and tabs, and identifies which UI elements have an event listener assigned to them.

Step 3 - Exercising UI Elements: To exercise a UI element, we look for an unvisited UI element that has an event listener. As depicted in Figure 3, after gathering all the information about the event listeners, the UI object, and its action and target, we generate an event on that element and pass it to `UIApplication` object, which is responsible for receiving the events and dispatching them to the code for further handling.

Step 4 - Accessing Next View Controller: By observing the changes on the current view controller, we obtain the next view controller and analyze the behaviour. The event could lead to four scenarios: no user interface change, the changes are within the current view controller, going to a new view controller, or going to the previous view controller.

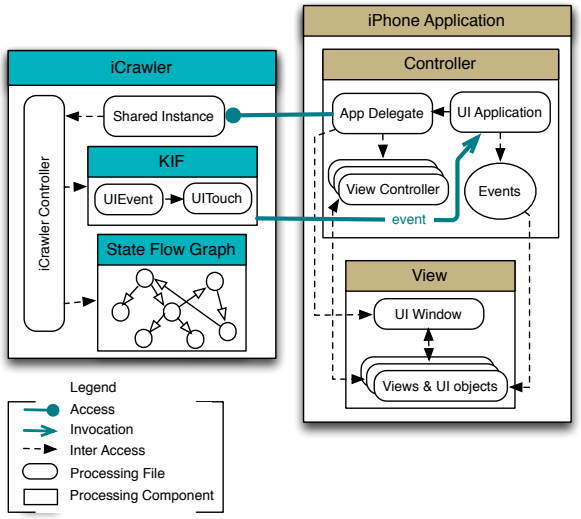

Figure 3: Relation between iCrawler and a given iPhone application. The right side of the graph shows key components of an iPhone application taken from [17].

Step 5 - Analyzing New UI Elements: After getting a new view controller, we collect all its UI elements. If the action has resulted in staying in the current view controller, we record the changes on the UI elements.

Step 6 - Comparing UI States: Once we get the new view controller and its UI elements, we need to compare the new state with all the perviously visited unique states. This way, we can determine if the action changes the current state or ends up on a state that has already been analyzed. If the state is not visited before, it is added to the set of unique visited states.

Step 7 - Recursive Call: We recursively repeat from step 3 until no other executable UI elements are left within the view controller and we have traversed all the view controllers.

We further describe our approach in the following subsections.

A. Hooking into the Application

The process of accessing the initial view controller is different from the rest of the view controllers. Since our goal is to be as nonintrusive and orthogonal to the application's source code as possible, we determine the initial view controller by performing a low-level Objective-C program analysis on the application delegate object. To that end, we employ a number of runtime functions to deduce the initial view controller. We use the Objective-C runtime reference library [29], which provides support for the dynamic properties of the Objective-C language and works with classes, objects, and properties directly. It is effective primarily for low-level debugging and meta-programming. In addition, the Key-Value Coding (KVC) protocol [30] is used to access UI

```
- (void) icDismissModalVC:(BOOL) animated {
    [[NSUserDefaults standardUserDefaults] setBool:YES ↩
        forKey:@"IC_isDismissed"];
    // Call the original (now renamed) method
    [self icDismissModalVC:animated];
}
```

Figure 4: The new method in which we inject code to set the dismissed boolean and then call the original method.

objects at runtime. The KVC protocol assists in accessing the properties of an object indirectly by key/value, rather than through invocation of an accessor method or as instance variables [30].

Once the application delegate is accessed, we retrieve all the properties of this class and their names. After getting the property names in the application delegate, we call a KVC method to access an instance variable of the initial view controller using the property name string. This way, we are able to identify the type of initial view controller (e.g., `UITabBarController`, `UINavigationController`, just a custom `UIViewController`). This knowledge is required for setting up the initial state.

B. Analyzing UI Elements

In our approach, a UI *state* includes the current view controller, its properties, accompanied by its set of UI elements. Once we get the view controller, we read all the subviews, navigation items, as well as tool bar items of the view controller in order to record the corresponding UI elements in an array associated to the view controller. Having the required information for a state, we set a global variable to point to the current state throughout the program.

C. Exercising UI Elements

We fire an event (e.g., a tap) on each unvisited UI element that has an event-listener assigned to it. Since events are handled in different ways for different UI classes, for each UI element type, such as tables, tabs, text views, and navigation bar items, we recognize its type and access the appropriate view. As shown in Figure 3, we use KIF's methods to handle the event.

After an element is exercised, we use a delay to wait for the UI to update, before calling the main function recursively. Based on our experience, a 1 second waiting time is enough after firing different event types such as tapping on a table cell or a button, scrolling a table up and down, and closing a view.

D. Accessing the Next View Controller

After exercising a UI element, we need to analyze the resulting UI state. An event could potentially move the UI forward, backward, or have no effect at all.

At a low level, going back to a previous view controller in iPhone applications happens either by popping the view

```
+ (void)load {
  if (self == [UIViewController class]) {
    Method originalMethod = class_getInstanceMethod(↩
        self, @selector(↩
        dismissModalViewControllerAnimated:));

    Method replacedMethod = class_getInstanceMethod(↩
        self, @selector(icDismissModalVC:));

    swap(originalMethod, replacedMethod);
  }
}
```

Figure 5: Swapping the original built-in method with our new method in the `+load` function.

controller from the navigation stack or by dismissing a modal view controller. We monitor the navigation stack after executing each event to track possible changes on the stack and thus, become aware of the pop calls. However, being aware of dismissing a modal view needs to be addressed differently. Our approach combines reflection with code injection to track if a dismiss method is called. To that end, we employ the *Category and Extension* [31] feature of Objective-C, which allows adding methods to an existing class without subclassing it or knowing the original classes. We also use a technique called *Method Swizzling* [32], which allows the method implementation of a class to be swapped with another method.

We define a category extension to the `UIViewController` class and add a new method in this category (See Figure 4). We then swap a built-in method of the view controller, responsible for dismissing a view controller class, with the new method (See Figure 5). The static `+load` method is also added to the category and called when the class is first loaded. We use the `+load` method to swap the implementation of the original method with our replaced method. The `swap` method call swaps the method implementations such that calls to the original method at run-time result in calls to our method defined in the category. As show in Figure 4, we also call the original method, which is now renamed. Our method stores a boolean data in the defaults system. The iOS defaults system is available throughout the application, and any data saved in the defaults system will persist through application sessions. Therefore, after a dismiss call occurs, we set the dismissed boolean to `true`. At runtime, each time an action is executed, we check the dismissed boolean in the `NSUserDefaults` object to see if dismiss has occurred. We set this back to `false` if that is the case. This way we are able to track if the event results in going back to a previous view controller to take the proper corresponding action.

A new view controller could be pushed to the navigation stack, presented modally, or be a new tab of a tab bar controller. If the action results in staying in the current view controller, different state changes could still occur such as UI

Algorithm 1: State Change Recognition

```
   input : Set of weights for view controller properties (Wvc)
   input : Set of weights for UI element properties (We)
   input : Similarity threshold (τ)
   input : Set of the unique states visited (VS)
   input : Current state (cs)
   output: Similar state (s ∈ VS, otherwise nil)
1  begin
2      σ ← 0
3      foreach s ∈ VS do
4          σ ←
5          (Wvc(class) × (s.viewController.class ≡
           cs.viewController.class) +
6          Wvc(title) × (s.viewController.title ≡
           cs.viewController.title) +
7          Wvc(elements) × (s.uiElementsCount ≡
           cs.uiElementsCount))
8          foreach e1 ∈ s.uiElementsArray do
9              e2 ← GETELEMENTATINDEX(cs, e1)
10             σ ← σ + (We(class) × (e1.class ≡ e2.class) +
               We(hidden) × (e1.hidden ≡ e2.hidden) +
               We(enable) × (e1.enable ≡ e2.enable) +
               We(target) × (e1.target ≡ e2.target) + We(action)
               × (e1.action ≡ e2.action))
11         attributes ← Size(Wvc) + (s.uiElementsCount ×
           Size(We))
12         if ((σ/attributes) × 100) >= τ then
13             return s
14     return nil
```

element(s) dynamically being changed/added/removed, or a pop-up message or an action sheet appearing. If we do not notice any changes within the current state, we move further with finding the next clickable UI element. Otherwise, we need to conduct a state comparison to distinguish new states from already visited states. If the state is recognized as a new state, a screen shot of the interface is also recoded.

E. Comparing States

Another crucial step in our analysis is determining whether the state we encounter after an event is a new UI state. As opposed to other techniques that are based on image-based comparisons [26], [14], in order to distinguish a new state from the previously detected states, we take a programmatic, heuristic-based approach in which we compare the view controllers and all their UI elements of the application before and after the event is executed.

Deciding what constitutes a UI state change is not always that straight forward. For instance, consider when a user starts typing a string to a text field and that action changes the value of the text field's property, or when the sent button of an email application is enabled as soon as the user starts typing a body of the email. We need a way to figure out if these changes (changing text of a text field/label or enabling a button) should be seen as a new state. To that end, we propose a similarity-based heuristic to emphasize or ignore changes on specific properties of view controllers, their accompanying UI elements, and the elements' properties.

Our state recognition heuristic considers the following properties of view controllers: `class`, `title`, and the number of UI elements. In addition, for each UI element, it considers `class`, `hidden`, `enable`, `target`, and `action`. Although our algorithm can handle as many properties as required, we are interested in these attributes because we believe they are most likely to cause a visible UI state change. We consider a set of distinct weights for each of the aforementioned attributes of a view controller, denoted as $WVC = \{wvc1, wvc2, ..\}$ as well as another set of distinct weights for each of the aforementioned attributes of a UI element as $WE = \{we1, we2, ..\}$. The value of each weight is a number between 0 and 1. All weights have default values that can be overridden by the user's input if required. These default values are obtained for each weight through an experimental trial and error method (discussed in Section VII). The similarity, σ, between two UI states is a percentage calculated as follows:

$$\sigma = \left(\frac{\sum_{i=1}^{Size(WVC)} |WVC_i| VC_i + \sum_{j=1}^{Ne} \sum_{k=1}^{Size(WE)} |WE_k| El_j}{Size(WVC) + N_e \times Size(WE)} \right) \times 100$$

where, `VC` returns 1 if the property of the two view controllers are equal and 0 otherwise, `Size(WVC)` and `Size(WE)` return the total number of properties considered for a view controller and a UI element respectively. The second part of the summation calculates similarity of each of the elements' properties. `El` returns 1 if the property of the two UI elements is equal and `Ne` is the total number of UI elements. The total summation of the view controllers and elements is divided by the total number of properties.

Algorithm 1 shows our algorithm for checking the similarity of the current state (after an event) with all the visited states. It returns a similar state if one is found among the visited states. As input the algorithm gets two sets of distinct weights for view controller and UI element, a similarity threshold (τ), the set of unique states visited so far, and the current state.

For each visited state (line 3), we calculate the similarity of the two states by adding the similarity of the two view controllers' classes, titles and the number of UI elements (line 7). Then for each UI element in a visited state (line 8), the corresponding UI element in the current state (line 9) is retrieved and their similarity is calculated. Finally, we divide the similarity by the total number of attributes, which are considered so far, calculate the percentage (line 12) and compare it to the threshold. The algorithm assumes the two interfaces to be equivalent if the calculation of the aforementioned weight-based attributes are more than or equal to τ. In other words, we consider two UI states equal, if they have the same view controller, title, set of UI elements, with the same set of selected properties, and the same event listeners.

Table I: Experimental objects.

ID	Exp. Object	Resource
1	Olympics2012	https://github.com/Frahaan/2012-Olympics-iOS--iPad-and-iPhone--source-code
2	Tabster	http://developer.apple.com/library/ios/#samplecode/Tabster/Introduction/Intro.html#
3	TheElements	http://developer.apple.com/library/ios/#samplecode/TheElements/Introduction/Intro.html#
4	Recipes & Printing	http://developer.apple.com/library/ios/#samplecode/Recipes_+_Printing/Introduction/Intro.html#
5	NavBar	http://developer.apple.com/library/ios/#samplecode/NavBar/Introduction/Intro.html#
6	U Decide	http://appsamuck.com/day12.html

F. State Graph Generation

To explore the state space, we use a depth-first search algorithm and incrementally create a multi-edge directed graph, called a state-flow graph [21], with the nodes representing UI states and edges representing user actions causing a state transition.

VI. Tool Implementation: iCrawler

We have implemented our approach in a tool called iCrawler. iCrawler is implemented in Objective-C using Xcode 3. We use a number of libraries as follows.

DCIntrospect [33] is a library for debugging iOS user interfaces. It listens for shortcut keys to toggle view outlines and print view properties as well as the action messages and target objects, to the console. We have extended DCIntrospect in a way to extract a UI element's action message, target object, it's properties and values. We further use our extension to this library to output all the reverse engineered UI elements' properties within one of our output files.

To generate an event or insert textual input, we use and extend the Kif framework [15]. At runtime, iCrawler extracts UI elements with event-listeners assigned to them and collects information about the action message and target object of each UI elements. By recognizing the type of a UI element, iCrawler gains access to its appropriate view. Then it uses Kif's internal methods to generate an event on the view.

At the end of the reverse engineering process, the state graph is transformed into an XML file using Xswi [34], which is a standalone XML stream writer implemented in Objective-C.

The output of iCrawler consists of the following three items: (1) an XML file, representing a directed graph with actions as edges and states as nodes. (2) screenshots of the unique states, and (3) a log of all the reverse engineered UI elements (including their properties, values, actions and targets), generated events and states.

VII. Empirical Evaluation

To assess the effectiveness of our reverse engineering approach, we have conducted a case study using six open-source iPhone applications.

Table II: Characteristics of the experimental objects.

ID	.m/.h Files	LOC (Objective-C)	Statements (;)	Widgets
1	22	2,645	1,559	398
2	21	1,727	286	14
3	28	2,870	690	21
4	23	2,127	508	7
5	20	1,487	248	10
6	13	442	162	15

We address the following research questions in our evaluation:

RQ1 Is iCrawler capable of identifying unique states of a given iPhone application correctly?

RQ2 How complete is the generated state model in terms of the number of edges and nodes?

RQ3 How much manual effort is required to set up and use iCrawler? What is the performance of iCrawler?

A. Experimental Objects

We include six open-source experimental objects from the official Apple sample code, Guithub, and other online resources. Table I shows each objects's ID, name, and resource. Table II presents the characteristics of these applications in terms of their size and complexity. We use Xcode Statistician[3] for collecting metrics such as the number of header and main files, lines of code (LOC) and statements. The table also shows the number of UI widgets within each application. The UI widget is a UI element, such as a tab bar view with all of its tab icons, a table view with all of its cells, a label or a button. The number of UI widgets is collected through iCrawler's output file, which logs all the UI elements and their properties.

B. Experimental Design

In order to address RQ1, we need to compare unique states generated by iCrawler to the actual unique states for each application. As mentioned before, iCrawler identifies the unique states through Algorithm 1 and keeps the screenshots of the unique states in a local folder. To form a comparison baseline, we manually run and navigate each application and count the unique states and compare that with the output of iCrawler.

To assess the iCrawler's generated state model (RQ2), we also require to form a baseline of the actual number of edges (i.e. user's actions that change the states) and states (unique and repetitive) to compare with the iCrawler's state model. Therefore, we manually run and navigate each application and count the edges and the states. Note that there are currently no other similar tools available to compare iCrawler's results against.

[3]http://xcode-statistician.mac.informer.com/

In order to address RQ3, we measure the time required to set up ICRAWLER and employ it to each of the given iPhone applications. The following series of manual tasks are required before ICRAWLER can start the analysis:

- The ICRAWLER framework should be added to the application's project under analysis.
- In order to enable ICRAWLER to access the delegating application object, the ICRAWLER's initialization line of code should be added to the built-in method, `application:didFinishLaunchingWithOptions:`.
- Finally, a preprocessor flag (`RUN_ICRAWLER`) needs to be added to the created Xcode target.

Further to investigate the performance of ICRAWLER for each application under test, we measure the time between calling ICRAWLER and when ICRAWLER finishes its job.

As we mentioned earlier, we obtain default values for the threshold and similarity weights by an experimental trial and error method for each of the applications. The best values that we have observed are: threshold (%70); weights include: view controller's `class` (0.8), `title` (0.8), and number of UI elements (0.8); UI element's `class` (0.7), `hidden` (0.7), `enable` (0.7), `target` (0.7), and `action` (0.7). These are also the values used in our evaluation, for all the experimental objects.

C. Results

Setting up ICRAWLER and utilizing it for a given iPhone application takes 10 minutes on average. The results of our study are shown in Table III. The table shows the number of `Unique States`, `Total States`, and `Edges` counted manually and by ICRAWLER. Further, the total number of `Generated Events` and `Total Time` for ICRAWLER are presented. We should note that the total time depends on the application and includes the delay (1 sec) we use after each action. The number of generated events is different from the number of detected edges. The events include all the user actions, while the edges are only those actions that result in a state change (including back-ward edges). For instance, scrolling a table or a view up and down counts as an event while it is not an edge in the state model. Another example, related to our state comparison algorithm, is a label that changes after executing a button, which ICRAWLER does not consider as a new state.

Below, we describe some of the results in Table III.

The `Olympics2012` (#1) application provides information about 38 sports in the Olympics 2012 as well as a timetable and a count down (See Figure 1 and Figure 2). According to Table III, ICRAWLER is capable of identifying the correct number of uniques states, total states, and edges within this application. The events include tapping on a tab bar item, scrolling up/down a view, scrolling up/down a table, tapping on a backward/forward button and tapping on a button which flips the view. The number of user actions, i.e., generated events, is 85 while the number of edges is 43 (including a back-ward edge). This is because user actions such as scrolling are not changing states and as a result they are not counted as edges. The number of uniques states is 6 while the number of total states is 81. This is because there are 38 buttons in this application which lead to a same UI state while presenting different data for 38 types of sports.

Table III: Results.

ID	Manual			ICRAWLER				
	Unique States	Tot. States	Edges	Unique States	Total States	Edges	Gen. Events	Total Time (Sec)
1	6	81	43	6	81	43	85	88
2	11	16	17	9	12	11	18	18
3	6	16	15	6	16	15	27	29
4	6	13	10	3	5	4	8	10
5	8	14	13	3	5	4	7	9
6	2	13	2	2	13	2	12	13

Events within `Tabster` (#2) include tapping on a tab bar item, scrolling up/down a table, tapping on a table cell, tapping on a backward/forward button, tapping on a dismiss/present button and writing a text. When exercising UI elements which require text input through keyboard, we used a dummy string based on the keyboard type e.g., numeric, alphanumeric, url or email address input. As it is shown in Table III, our approach is able to identify 11 edges and 9 uniques states. However `Tabster` has the tab bar view with a "more page" feature and ICRAWLER supports an ordinary tab bar view (without the "more" feature) at this time. As a result, there is a difference between the number of uniques states and edges in baseline and ICRAWLER.

Actions within `TheElements` (#3) application include tapping on a tab bar item, scrolling up/down a table, tapping on a table cell, tapping on a back-ward/forward button and tapping on a button which flips the view. ICRAWLER is successfully able to cover the states and edges of `TheElements`. Here, we disabled a button, which closes the application and forwards the user to the AppStore.

The `Recipes & Printing` application (#4) browses recipes and has the ability to print the browsed recipes. Here, the difference between manual and ICRAWLER results in Table III is due to ignoring the states and actions involved with printing.

For tables, one could think of different strategies to take: (1) generate an event on each and every single table cell, (2) randomly click on a number of table cells (3) generate an event on the first table cell. In our technique, once ICRAWLER encounters a table view, it scrolls down and up to ensure the scrolling action works properly and it does not cause to any unwanted crashes, e.g., by having a specific character in an image's url and trying to load the image on a table cell. ICRAWLER then generates an event on the first row and moves forward. This works well for table cells that result to the same next view. However,

there are cases in which table cells lead to a different view. `NavBar` (#5) is such a case. There are five different table cells within this application, which go to different UI states. Thus we witness a difference between the number of edges or states counted manually and by ICRAWLER. This is a clear empirical evidence suggesting that we need to improve our table cell analysis strategy.

D. Findings

The results of our case study show that ICRAWLER is able to identify the unique states of a given iPhone application and generate its state model correctly, within the supported UI elements and event types. Generally, it takes around 10 minutes to set up and use ICRAWLER. The performance of ICRAWLER is acceptable. For the set of experimental objects, the minimum analysis time was 9 seconds (5 states, 4 edges, 7 events) and the maximum was 88 seconds (81 states, 43 edges, 85 events).

VIII. Discussion

Limitations. There are some limitations within our current implementation of the approach. Although it is minimal, the users still need to complete a few tasks to set up ICRAWLER within their applications manually. There are also some UI elements such as the tool bar, slider, page control, and search bar, which are not supported currently. In addition, while ICRAWLER currently supports the most common gestures in iOS applications such as tapping on a UI element, inserting text, and scrolling views, there is no support yet for other advanced gestures such as swiping pages and pinching (e.g., zooming in and out images).

Threats to Validity. The fact that we form the comparison baselines manually could be a threat to internal validity. We did look for other tools to compare our results against, without success. Manually going through the different applications to create baselines is labour intensive and potentially subject to errors and author's bias. We tried to mitigate this threat by asking two other students to create the comparison baselines.

Additionally, the independent variables of weights and threshold within our state recognition algorithm have a direct effect on our dependent variables such as number of unique states and edges. As a result, choosing other values for these independent variables rather than our default values, could result in difference in the outcome. As mentioned in the evaluation section, we chose these optimal values through a series of trial and error experiments.

In our attempt to gather the experimental objects, we noticed that there is a small collection of open-source iPhone applications available online – note that we could not use applications available in AppStore for our experiment since we needed access to their source code. Even though, this made it difficult to select applications that reflect the whole spectrum of different UI elements in iPhone applications, we believe the selected objects are representative of the type of applications ICRAWLER can reverse engineer. However, we acknowledge the fact that, in order to draw more general conclusions, more mobile applications are required.

Applications. There are various applications for our technique. First of all, our technique enables automatic interaction with the mobile application. This alone can be seen as performing smoke testing (e.g., to detect crashes). In addition, the state model inferred can be used for automated test case generation. Further, using the model to provide a visualization of the state space supports developers to obtain a better understanding of their mobile applications. The approach can be extended to perform cross-platform testing [35], i.e., whether an application is working correctly on different platforms such as iOS and Android, by comparing the generated models. Finally, other application areas could be in performance and accessibility testing of iOS applications.

IX. Conclusions and Future Work

As smartphones become ubiquitous and the number of mobile applications increases, new software engineering techniques and tools geared towards the mobile platform are required to support developers in their program comprehension, analysis, and testing tasks.

In this paper, we presented our reverse engineering technique to automatically navigate a given iPhone application and infer a model of its user interface states. We implemented our approach in ICRAWLER, which is capable of exercising and analyzing UI changes and generate a state model of the application. The results of our evaluation, on six open source iPhone applications, point to the efficacy of the approach in automatically detecting unique UI states, with a minimum level of manual effort required from the user. We believe our approach and techniques have the potential to help mobile application developers increase the quality of iOS applications.

Future Work. There are several opportunities in which our approach can be enhanced and extended for future research. The immediate step would be to extend the current version of ICRAWLER to support the remaining set of UI elements within `UIKIT` such as the tool bar, slider, page control, and search bar. We also plan to evaluate the tool on more complex industrial iOS applications. Other directions we will pursue are using this technique for smoke testing of iPhone applications as well as generating test cases from the inferred state model. Furthermore, we intend to expand ICRAWLER to support iPad applications. We are currently extending ICRAWLER with reverser engineering analysis at the binary level.

References

[1] Berg Insight, "The mobile application market," http://www.berginsight.com/ReportPDF/ProductSheet/bi-app1-ps.pdf.

[2] "App Store Metrics," http://148apps.biz/app-store-metrics/.

[3] "Android market stats," http://www.appbrain.com/stats/.

[4] H. Kim, B. Choi, and W. Wong, "Performance testing of mobile applications at the unit test level," in *Proceedings of the 3rd International Conference on Secure Software Integration and Reliability Improvement*. IEEE Computer Society, 2009, pp. 171–180.

[5] H. Muccini, A. D. Francesco, and P. Esposito, "Software Testing of Mobile Applications: Challenges and Future Research Directions," in *Proceedings of the 7th International Workshop on Automation of Software Test (AST)*. IEEE Computer Society, 2012.

[6] J. Dehlinger and J. Dixon, "Mobile Application Software Engineering: Challenges and Research Directions," in *Proceedings of the Workshop on Mobile Software Engineering*. Springer, 2011, pp. 29–32.

[7] A. I. Wasserman, "Software engineering issues for mobile application development," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, ser. FoSER'10. ACM, 2010, pp. 397–400.

[8] M. Janicki, M. Katara, and T. Paakkonen, "Obstacles and opportunities in deploying model-based gui testing of mobile software: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 313–341, 2012.

[9] D. Franke and C. Weise, "Providing a Software Quality Framework for Testing of Mobile Applications," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2011, pp. 431–434.

[10] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How Do Professional Developers Comprehend Software?" in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2012, pp. 255–265.

[11] B. A. Myers and M. B. Rosson, "Survey On User Interface Programming," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, ser. CHI'92. ACM, 1992, pp. 195–202.

[12] M. Chandramohan and H. B. K. Tan, "Detection of mobile malware in the wild," *Computer*, vol. 99, 2012.

[13] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *18th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2011.

[14] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna, "Challenges for Dynamic Analysis of iOS Applications," in *Proceedings of the Workshop on Open Research Problems in Network Security (iNetSec)*, 2011, pp. 65–77.

[15] "KIF iOS Integration Testing Framework," https://github.com/square/KIF.

[16] "Frank: Automated Acceptance Tests for iPhone and iPad," http://www.testingwithfrank.com/.

[17] iOS Developer Library, "Apple's developer guides," https://developer.apple.com/library/ios/navigation/#section=ResourceTypes&topic=Guides.

[18] "Project SIKULI," http://sikuli.org.

[19] "MonkeyTalk for iOS & Android," http://www.gorillalogic.com/testing-tools/monkeytalk.

[20] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," in *Proceedings of The 10th Working Conference on Reverse Engineering*. IEEE, 2003, pp. 260–269.

[21] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes," in *ACM Transactions on the Web (TWEB)*, vol. 6, no. 1. ACM, 2012, pp. 3:1–3:30.

[22] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 35 –53, 2012.

[23] A. Mesbah and S. Mirshokraie, "Automated analysis of CSS rules to support style maintenance," in *Proceedings of the 34th ACM/IEEE International Conference on Software Engineering (ICSE'12)*. IEEE Computer Society, 2012, pp. 408–418.

[24] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *Proceedings of the Workshops at IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE Computer Society, 2011, pp. 252–261.

[25] A. Gimblett and H. Thimbleby, "User interface model discovery: towards a generic approach," in *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, ser. EICS '10. ACM, 2010, pp. 145–154.

[26] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," in *Proceedings of the 28th international conference on Human factors in computing systems*, ser. CHI '10. ACM, 2010, pp. 1535–1544.

[27] C. Hu and I. Neamtiu, "Automating GUI testing for Android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. ACM, 2011, pp. 77–83.

[28] "UI/Application Exerciser Monkey," 2010, http://developer.android.com/guide/developing/tools/monkey.html.

[29] "Objective-C Runtime Reference," https://developer.apple.com/library/ios/documentation/Cocoa/Reference/ObjCRuntimeRef/ObjCRuntimeRef.pdf.

[30] "NSKeyValueCoding Protocol Reference," https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Protocols/NSKeyValueCoding_Protocol/NSKeyValueCoding_Protocol.pdf.

[31] "Categories and Extensions," http://developer.apple.com/library/ios/#documentation/cocoa/conceptual/objectivec/chapters/occategories.html.

[32] Cocoa developer community, "Method Swizzling," http://cocoadev.com/wiki/MethodSwizzling.

[33] "DCIntrospect," https://github.com/domesticcatsoftware/DCIntrospect.

[34] "XSWI: XML stream writer for iOS," http://code.google.com/p/xswi/.

[35] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering (ICSE'11)*. ACM, 2011, pp. 561–570.