SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Applications

Cong Zheng[12], Shixiong Zhu[12], Shuaifu Dai[12], Guofei Gu[3], Xiaorui Gong[12], Xinhui Han[12], Wei Zou[12*]

[1]Beijing Key Laboratory of Internet Security Technology, Peking University
[2]Institute of Computer Science and Technology, Peking University
{zhengcong,zhushixiong,daishuaifu,gongxiaorui,hanxinhui,zou_wei}@pku.edu.cn

[3]SUCCESS Lab, Texas A&M University
guofei@cse.tamu.edu

ABSTRACT

User interface (UI) interactions are essential to Android applications, as many Activities require UI interactions to be triggered. This kind of UI interactions could also help malicious apps to hide their sensitive behaviors (e.g., sending SMS or getting the user's device ID) from being detected by dynamic analysis tools such as TaintDroid, because simply running the app, but without proper UI interactions, will not lead to the exposure of sensitive behaviors. In this paper we focus on the challenging task of triggering a certain behavior through automated UI interactions. In particular, we propose a hybrid static and dynamic analysis method to reveal UI-based trigger conditions in Android applications. Our method first uses static analysis to extract expected activity switch paths by analyzing both Activity and Function Call Graphs, and then uses dynamic analysis to traverse each UI elements and explore the UI interaction paths towards the sensitive APIs. We implement a prototype system *SmartDroid* and show that it can automatically and efficiently detect the UI-based trigger conditions required to expose the sensitive behavior of several Android malwares, which otherwise cannot be detected with existing techniques such as TaintDroid.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software (e.g., viruses, worms, Trojan horses); D.2.5 [**Software Engineering**]: Testing and Debugging —*Testing tools, Tracing*

General Terms

Security, Verification

*Corresponding author.

Keywords

Smartphone security, UI-based trigger condition, sensitive behavior, Android

1. INTRODUCTION

With the rapid development of smartphones, mobile Internet has changed the way we entertain, socialize and work. A recent news posted by Google shows that the number of Android applications reached more than 450,000 in its Google Play at the beginning of March 2012 [8]. The growth of the Android Market (the former name of Google Play) is amazing; the download number of applications passed 10 billion in December 2011 up from 3 billion in March 2011 [1]. Moreover, except for the official Android Market, some third-party vendors provide their own channels (e.g. Amazon Appstore, Samsung Apps) for users to download applications.

Unfortunately, Android developers can upload their applications to the Android Market or other third-party markets with little, if any, security vetting processes. Market operators should control the quality of applications and block malicious applications to their markets. Currently, however, except for Google's Bouncer [5], which is an automated scanning tool of the Android Market for detecting potentially malicious applications, other third-party markets have not taken effective actions to protect their users. Even though Google's Bouncer can detect some of malicious applications, these malicious applications may have been already downloaded by a huge number of users and stolen user's privacy or other valued information before Bouncer finds them. Therefore, it is very possible for users to download a malicious application from the Android Market or other third-party markets.

Note that behaviors of some Android applications could be difficult to be determined whether they are malicious or legitimate [20]. For the purpose of this paper, we do not determine them, but check all suspicious behaviors. In this paper, these behaviors are called "sensitive behaviors" [18]. In the Android system, behaviors should be executed by invoking APIs, which are referred to as "sensitive APIs" [11] related to sensitive behaviors.

A number of solutions including static analysis [15, 14, 19, 31] and dynamic analysis technologies [29, 17, 28] have been proposed to detect Android applications' sensitive behaviors. For example, TaintDroid presented by Enck et al. [17] can detect privacy leaks by using a dynamic taint-tracing method. However, it uses a passive detection method, which needs to know the conditions of triggering privacy leaking behaviors in advance. With the trigger conditions, the analyzer can then interact with applications manually to trigger and confirm its privacy leaking behaviors. Consider a sample that only leaks the IMEI after the user clicks a particular button on the screen, TaintDroid cannot detect this behavior unless the analyzer clicks the right UI button manually. Thus, the detection ability of TaintDroid is highly limited when facing malware samples that need UI-based trigger conditions. Hu et al. [23] proposed a method that feeds random events to the application. However, given the huge randomization space, it is very difficult to detect the sensitive behaviors efficiently. Gilbert et al. [22] tested a variety of categories of applications by generating random user events for 30 minutes. However, this can only achieve 40% or less code coverage in all cases.

In this paper, we are motivated to develop an automatic method to reveal UI-based trigger conditions of sensitive behaviors in Android applications. With the trigger conditions provided by our method, dynamic analysis tools, such as TaintDroid, will be able to automatically detect these sensitive behaviors. To reveal UI-based trigger conditions, we present a system called SmartDroid, which combines static analysis and dynamic analysis techniques. In the Android system, an Activity is the whole screen including buttons, text boxes, and other UI elements, with which the user can interact. Therefore, the main idea is that we use static analysis to discover the expected Activity switch paths that can lead to sensitive behaviors; then for each path, we apply dynamic analysis to enforce the application run along the path until sensitive behaviors are triggered. In the dynamic analysis, SmartDroid will try to interact with every UI element automatically in each Activity by traversing the view tree of the current Activity in the modified Android system. If the current Activity can jump to the next Activity in the Activity switch path, the current UI element is our expected element and will be saved. When we traverse UI elements in the last Activity, the sensitive behaviors will be triggered finally. At last, the sequence of all saved UI elements is the trigger condition, including the coordinate and UI event type.

We have implemented the SmartDroid system and evaluated it using several existing Android malwares with sensitive behaviors. The result shows that SmartDroid is very effective in revealing UI-based trigger conditions automatically.

Contributions. Our solution makes the following contributions:

- For the first time, we propose a novel method that combines the static analysis and the dynamic analysis to reveal UI-based trigger conditions. The key idea is to use the dynamic analysis technology to enforce the execution along the suspicious path obtained from static analysis.

- Our method can augment existing dynamic analysis tools with automatic UI interaction analysis capabilities. This is a great complement to current techniques and tools.

- We implement the SmartDroid system and have detected several real-world, complicated, Android malwares in the wild, which otherwise cannot be detected by existing tools such as TaintDroid.

Organization. Section 2 gives the intuition and overview of our work. Section 3 describes our system design. Section 4 presents the implementation of our system. Section 5 shows the evaluation of our system and some case studies. Section 6 discusses the limitations of our solution and suggests possible improvements. Finally, we describe related work in Section 7 and summarize our conclusions in Section 8.

2. INTUITION AND OVERVIEW

In this section, we present an example (Section 2.1) to better demonstrate the complex UI-based trigger mechanism of sensitive behaviors. After that, we briefly introduce our solution (Section 2.2) to reveal the UI-based trigger conditions.

2.1 Example: The Horoscope App

The Horoscope App [7] in this example is intended to show your daily and monthly horoscopes. It connects to the Internet and sends the IMEI out of your device after you click certain buttons. We consider both behaviors of reading the IMEI and accessing the Internet as sensitive behaviors. Because, the IMEI is the only ID of smartphone devices and accessing the Internet may leak private information.

When this Horoscope App is started, the Android system creates an instance of the app's main Activity (an "Activity" provides user interfaces) depicted in Figure 1(a), which will pause 3 seconds and then start another Activity in Figure 1(b) using an Intent (an Activity is started with an Intent in Android system). There are two buttons for logging into Facebook and Twitter respectively and another button for setting your date of birth. Twelve icons represent twelve constellations on the screen. After you click one of twelve icons, it will display the Activity shown in Figure 1(c), which has two buttons for getting daily and monthly horoscope respectively. Once you click either of them, it switches to the Activity shown in Figure 1(d). In the last Activity, it reads and sends out the device's IMEI by using the sensitive APIs "android.telephony.TelephonyManager.getDeviceId()" and "org.apache.http.client.HttpClient.execute()" respectively.

If we were to use a dynamic analysis tool, such as TaintDroid, to test this sample, nothing would be detected, unless the analyzer understands how to click correct buttons manually. The method of feeding random events to the tool is also very weak on this sample. Especially when one random UI event clicks the wrong area, it will no longer reach the target Activity unless the sample is restarted to be analyzed again. For instance, when using MonkeyRunner[9] to generate a click event which clicks the advertisement bar shown in Figure 1(c), the browser will pop up so that subsequent UI events of MonkeyRunner will be out of context and therefore ineffective.

2.2 Overview of our approach

Since the UI-based trigger conditions can be quite complex, as described above, we seek to reveal them automatically and precisely. Figure 2 shows a schematic diagram, which includes the FCG (Function Call Graph) and the ACG

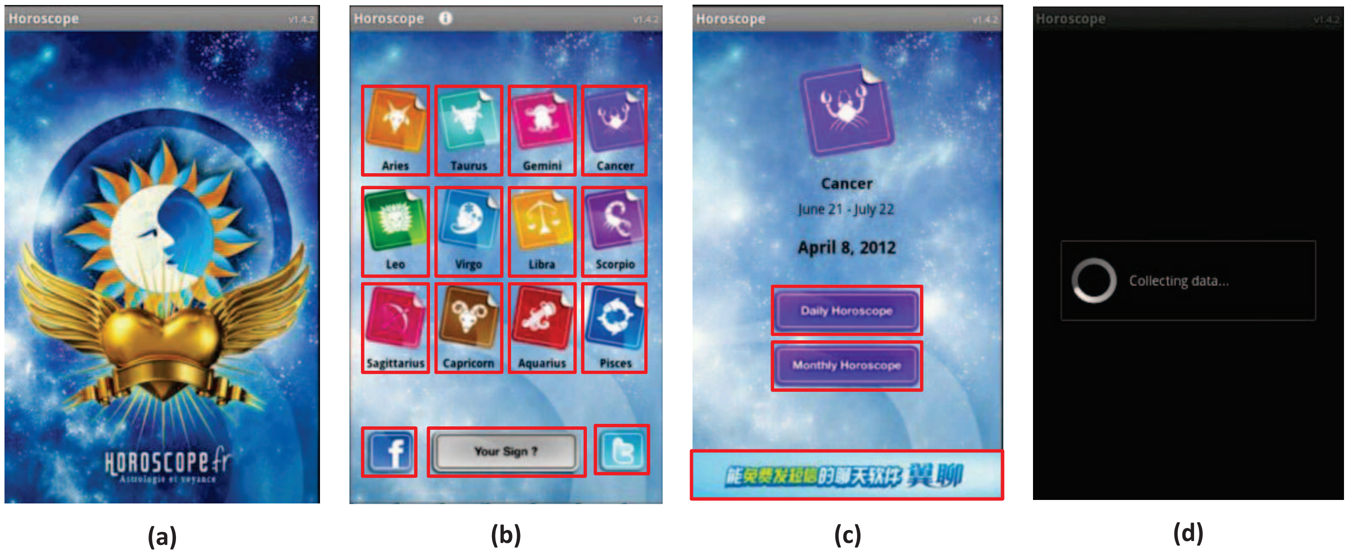


Figure 1: Screen shots of Horoscope App.

(Activity Call Graph). In the FCG, we can determine all function call paths to sensitive APIs. The start of the path is defined as the sensitive source function. In the ACG, we define the Activity, which is related to sensitive source functions, as a sensitive sink Activity. For example, in Figure 2, the Activity F is a sensitive Activity. It is linked with two types of sensitive source functions. When the current Activity switches to Activity F, the system will invoke the Activity F's "onCreate()" function automatically to render Activity F. Then the control flow of program will arrive at the "getDeviceId()" API along the function call path. In addition, there is a button in Activity F. When the user clicks this button, the system will invoke the corresponding "onClick()" function. The control flow will then execute the "sendTextMessage()" API to send an SMS message.

Our approach includes two stages: a static analysis stage and a dynamic analysis stage. The static analysis can produce the expected Activity switch path which will guide the dynamic analysis to determine how to interact with each Activity. In the dynamic analysis stage, we can find a sequence of UI elements which can make the sensitive APIs execute. The coordinates and UI event types of UI elements are our target UI-based trigger conditions.

In the static analysis stage, we first create the FCG and the ACG. It is easy to get the FCG by analyzing function call relationships using traditional control flow analysis [26]. But for the ACG, we must analyze all Intents in the invocations of the "startActivity" and "startActivityForResult" functions to obtain the source Activity and the target Activity, which are linked by an Intent. Finally, we extract the expected Activity switch paths which are from the main Activity to each sensitive sink Activity. The procedure is: (1) Get all sensitive source functions. We extract each function call path to sensitive APIs in the FCG. The first function in each path is the sensitive source function. (2) Get each sensitive sink Activity. We analyze which Activity the sensitive source function belongs to. (3) Get the expected Activity switch path. We use a depth-first searching algorithm in the ACG to get paths.

In the dynamic analysis stage, our goal is to determine which UI elements can make the application trigger the sensitive behaviors. But with the direction of expected Activity switch paths, the target transforms to know which UI elements can make the application run along the expected Activity switch paths. In order to know this, we first adopt an automatic method to interact with each UI element of Activity. We modify the Android system so that we can traverse and interact automatically with each UI element of an Activity after the creation of this Activity. In addition, to force the application to run along the expected Activity switch paths, we restrict the creation of the Activity by modifying the Android system. If our system interacts with wrong UI elements which make the application switch into a wrong Activity, we forbid the creation of this incorrect Activity. Finally, after the sensitive behavior is triggered, we can obtain a sequence of UI elements. While interacting with UI elements, we also record their coordinates and UI event types as the UI-based trigger conditions.

For example, in Figure 2, we want to find the conditions of triggering the behavior of sending SMS. In static analysis, we get two expected Activity switch paths ("$Main \rightarrow B \rightarrow F$" and "$Main \rightarrow C \rightarrow F$"). Suppose there are three buttons in Main Activity. They are A,B,C buttons which can switch into Activity A, Activity B and Activity C respectively. For path "$Main \rightarrow B \rightarrow F$", the Main Activity will not switch if our system clicks the Button A or Button C. But it will switch into Activity B if our system clicks the Button B. In the meantime, we record the coordinates of Button B and the "onClick" event type. In Activity B, there is another key button which can switch into Activity F. After our system clicks one button in Activity F, this application sends a message. We also add this button into the sequence of UI conditions. So, there are three buttons as the UI-based trigger condition for this sensitive behavior.

According to our approach, we can reveal UI-based trigger conditions for sensitive behaviors automatically and precisely.

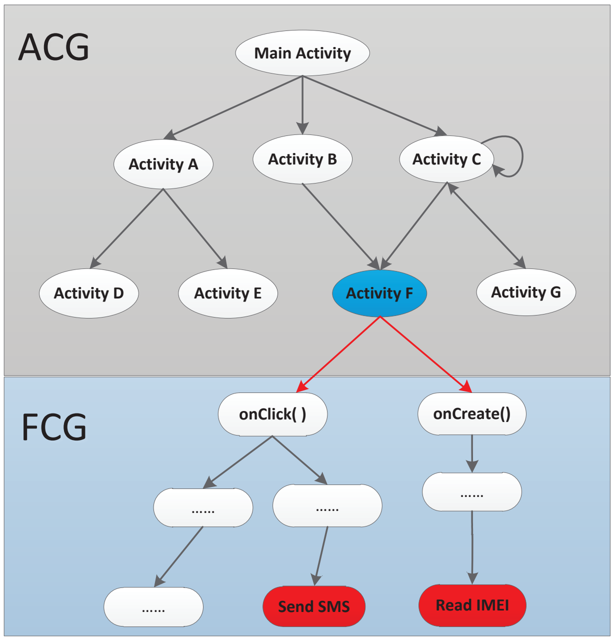


Figure 2: The Schematic Diagram for Trigger Mechanisms of Sensitive Behaviors.

3. SYSTEM DESIGN

Figure 3 provides a high-level overview of our SmartDroid system. It consists of two key components: the Static Path Selector and the Dynamic UI Trigger. The Static Path Selector is used to select the expected Activity switch paths which can lead to sensitive behaviors. The Dynamic UI Trigger can reveal the trigger conditions by interacting with UI elements according to these expected Activity switch paths.

3.1 Static Path Selector

The Static Path Selector is used to find the right Activity switch paths. If an application runs along the expected Activity call path, it is possible to trigger the sensitive behavior, otherwise there is no possibility to trigger the sensitive behavior. For example, in Figure 2, if we click the button A, the Main Activity will switch into Activity A. The first step is wrong in this case, so it is not possible to trigger the behavior of sending SMS. There are three steps in the Static Path Selector: disassemble, construct the FCG and construct the ACG. We introduce these three steps in the following sections.

3.1.1 Disassemble

With an application, we want to know which possible sensitive behaviors it has, so we must inspect the APIs used in this application. In order to do this, we disassemble this application. Considering the accuracy of existing "Dalvik bytecode-to-Java bytecode" translators, we prefer to operate and analyze directly on the Dalvik bytecode. The smali code is an intermediate representation of the Dalvik bytecode. The format of smali code is very convenient for the static analysis than the original format of the Dalvik bytecode. So we choose to do static analysis on the smali code level. Meanwhile, it is easy to get the smali code of an APK with existing tools.

3.1.2 Construct FCG

In this step, we want to find the FCG, in which all the child nodes are sensitive APIs. So, we first find all sensitive APIs in the smali code. In the smali code, it is very easy to know the function calling relationships according to the "invoke" instructions. We then use an interactive algorithm to search function call paths of these sensitive APIs. However, many indirect call instructions and event-driven calls in Android applications should not be ignored. In fact, some of the indirect call instructions come from the Java polymorphism. We use the methods proposed by Woodpecker [25] to solve this, which uses a conservative method for indirect function calls. It adds links by semantics for event-driven calls. Finally, we can obtain an entire FCG.

3.1.3 Construct ACG

Based on our approach, we should know the entire ACG of Android application. In the Android system, Activity switching is done through the Intent, which is a message that declares a recipient and optionally includes data, and it is used to start a new component. Usually, when an application starts a new Activity, it creates an Intent and invokes the "startActivity" or "startActivityForResult" functions. Generally, developers create one Intent corresponding to one "startActivity" or "startActivityForResult" invoking instruction. Applications use Intents to communicate with components in an application or with other applications. Additionally, the system sends Intents to applications as event notifications. Our goal is to find all Intents, and analyze the source Activity and target Activity linked by these Intents. We first introduce the Intent in the Android system and then introduce the method we used to analyze the Intent.

There are two kinds of Intents: explicit Intent and implicit Intent. An explicit Intent can specify its particular recipient by name, whereas an implicit Intent just specifies an action to the system. The actual recipient is determined by system according to the AndroidManifest.xml. In the AndroidManifest.xml, all components (i.e., Activity, Receiver) in this application define which Intents they can receive and the specified actions.

For an explicit Intent, it is easy to know its target Activity according to its definition. But for an implicit Intent, we need to match the target Activity according to actions in AndroidManifest.xml. If there is no matched Intent, we think this Intent can only be received by system components (i.e., sms box, email) or other applications. There are six kinds of constructors to construct Intent objects:

a) Intent()

b) Intent(Intent o)

c) Intent(String action)

d) Intent(String action, Uri uri)

e) Intent(Context packageContext, Class<?> cls)

f) Intent(String action, Uri uri, Context packageContext, Class<?> cls)

In constructor a), it just initializes a null Intent. After that, the "setClass()", "setComponent()" method or "setAction()" method are used to define the target Activity or bind

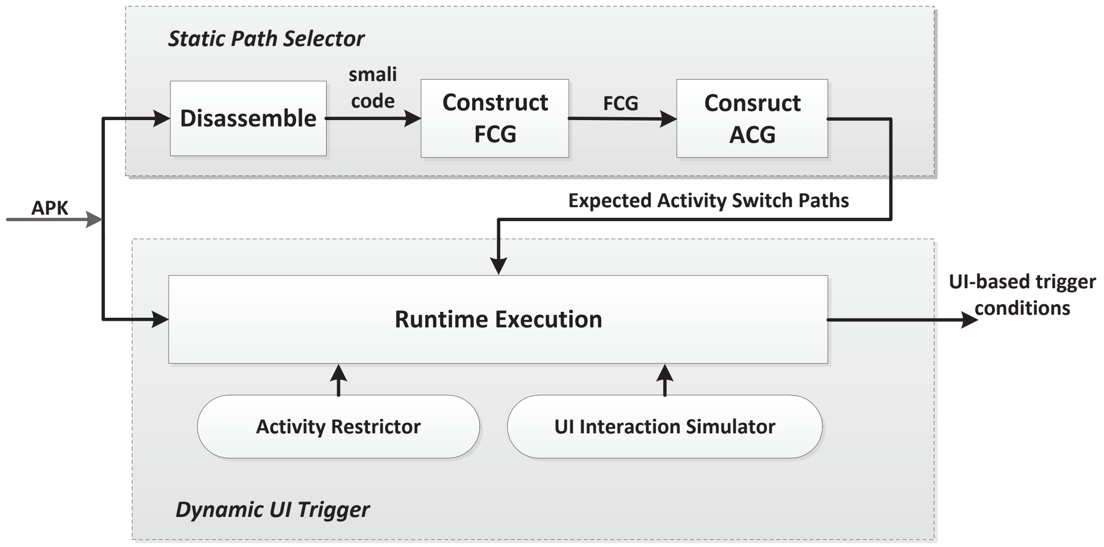


Figure 3: The Architecture and Workflow of SmartDroid.

an action to its target Activity respectively. Hence, it is equivalent to use Intent constructors in c), d), e) and f). In constructor b), an Intent is created by copying from another Intent. We must further analyze the Intent's parameters to know the source Activity and target Activity. In constructors c) and d), they define an implicit Intent by assigning the actions to the Intent. From the AndroidManifest.xml, we can determine the target Activity with actions. In constructor e) and f), they define an explicit Intent so that we can get the target class easily. Sometimes, the "packageContext" parameter in constructor e) is assigned by the current Activity's name. If it is, we can also get the source Activity.

Every target Activity can be obtained by analyzing the Intent constructors, but sometimes the source Activity is not assigned in constructors. We should continue to analyze the source Activity by a further step. We adopt a rudimental method to find the source Activity. If the function in which the Intent is defined is an UI event function, this function should be in a user event handler class. This handler class is always an inner class of an Activity class. In the smali code, the prefix name of an inner class is the same as the name of its Activity class. So, we can determine the source Activity's name. In addition, this UI event function is the edge between the source Activity and target Activity in the ACG. However, if the function in which the Intent is defined is not an UI event function, there are extra steps to do. We must extract the function call-in paths of this function from the FCG. For each path, we analyze every function backward. If a function, whose type is non-static and its class is Activity type, is found, its class is the source Activity. This function is the edge between the source Activity and target Activity in the ACG. At last, we can obtain a source Activity according to each path.

After we get all Activity switch relationships, we can build the ACG. With the ACG, we should continue to get expected Activity switch paths. Firstly, we can get the function call paths to sensitive API from the FCG. From the first function in the call path, we can know the sensitive sink Activity. There are just two types of the sensitive source function: UI event functions (e.g., onClick and onLongClick) and Activity related functions (e.g., onCreate and onStart). For the UI event function, it is registered in an Activity and always defined in an Activity class. For the Activity related function, it is a member function of an Activity class. So, it is easy to know the sensitive sink Activity related with the sensitive source function. Finally, we extract expected Activity switch paths which are from the node of the main Activity to nodes of sensitive sink Activity in the ACG.

3.2 Dynamic UI Trigger

If we only depend on static analysis, it is very difficult to match the UI element and its corresponding UI event functions. The reasons are: 1) Sometimes developers do not bind the UI element to its UI event functions in the "onCreate" function of Activity. It is difficult to find where they are bound. 2) Another problem is that many UI elements are bound to the same UI event function. Even though we know that an UI event function can lead to a sensitive behavior, we still do not know which UI element is related to this behavior exactly. So, we use dynamic analysis to solve this problem. In our Dynamic UI Trigger, there are three components: UI Interaction Simulator, Activity Restrictor and Runtime Execution Environment.

3.2.1 *Runtime Execution*

The Dynamic UI Trigger is based on a Runtime Execution environment. We build this Runtime Execution environment on a modified Android emulator. We have modified some codes of the Android framework and then compiled them to build a new emulator. In order to monitor the sensitive behaviors, we add some codes into the functions of the sensitive APIs and log the APIs' names and parameters when these APIs are called. The reason why we log the parameters is that some APIs (e.g., "ContentResolver.query()") can trigger different sensitive behaviors with different parameters. We run applications in this new emulator and collect the runtime log. Then we can monitor the log to confirm whether some sensitive behaviors happen. If we find a sensitive behavior, we can retrieve the related UI information in the log as the UI-based trigger condition of this sensitive behavior.

3.2.2 UI Interaction Simulator

In order to make our dynamic analysis automated, the first step we should do is to develop the UI Interaction Simulator to interact with UI elements in the Activity automatically. The idea of implementing the UI Interaction Simulator is: we modify the Activity codes of the Android framework to traverse all of UI elements in one Activity. Views in an Activity are organized into a hierarchy tree. The view object in the top-level window is the root of any other views in the Activity. We use a depth-first traversal algorithm in the tree and retrieve information of every UI element. When we traverse the UI element, we can trigger some of its event listeners according to edges of the expected Activity switch path, because the edge represents possible UI events so that we need not trigger all of event listeners. Meanwhile, we record the UI information (e.g., size and coordinates on the screen) when triggering its event listeners.

3.2.3 Activity Restrictor

We must overcome the challenge of generating expected user interactions automatically. The key point for this challenge is to know which UI elements are expected to be interacted with in the current Activity. Since the expected Activity switch path has been obtained by the Static Path Selector, we just need to determine which UI elements can lead the switch to the next Activity in that path. These UI elements belong to our result UI-based trigger conditions.

Our Activity Restrictor can determine which UI elements in the current Activity are the UI-based trigger conditions. If there is a new Activity after interaction with a UI element, but this new Activity is not the next expected Activity, the Activity Restrictor will prevent the creation of that new Activity. So, in this way, we can ensure that application runs along the expected Activity switch path. To develop the Activity Restrictor, we choose to modify some codes of "startActivityForResult" function in the Android framework. Because "startActivity" function is implemented by invoking the "startActivityForResult" function. In the "startActivityForResult(Intent, int)" function, we analyze its first parameter which represents an Intent. From this Intent object, we can easily know the new Activity by calling "intent.getComponent().getClassName()". Hence, the Activity Restrictor can determine whether it could prevent the creation of a new Activity according to the expected Activity switch path.

4. IMPLEMENTATION

We have implemented a SmartDroid prototype that consists of a mixture of an modified Android emulator, shell scripts and Python scripts. Specifically, the smali code in the static analysis is generated by the open-source apktool tool [4].

4.1 Dataflow Analysis

We analyze Intents in the smali code. Since now there is not a runtime and debug environment for the smali code, we can only implement a dataflow analysis technology on the smali code. In particular, our target is to know the two Activities linked by an Intent. For the constructor a), we use the dataflow analysis to search the parameters in "setClass()", "setComponent()" or "setAction()" methods to know the target Activity. For other constructors, we use the dataflow analysis to search parameters of constructors to know the source Activity and target Activity.

In our dataflow analysis, we adopt the fixed-point algorithm [16], which is an iterative algorithm from the initial state to the fixed state. The fixed-point algorithm can statically determine which definitions may reach a given point in codes. After running the fixed-point algorithm, we traverse the CFG to know the values of parameters in Intent's constructions or "setClass()", "setComponent()" and "setAction()". To optimize the dataflow analysis between functions, we apply the method summaries [25].

4.2 Runtime Execution

The Runtime Execution environment is based on the Android emulator (Android 2.3.3 version). In order to accelerate the starting of the emulator, we save a snapshot when the emulator finishes starting, so the emulator can be started from its snapshot every time [2]. This method decreases the time of starting the emulator certainly.

After we install an Android application into the emulator, it should wait for an interval (10s in our implementation) before the interaction of the UI Interaction Simulator. In some applications, there are automatic sensitive behaviors after applications start. These automatic behaviors would affect the results of dynamic analysis if the UI Interaction Simulator interacts with the Activity immediately. For example, the advertisement in some applications shows by a thread which runs in asynchronous. Sometimes the advertisement component reads the IMEI and connects to the Internet in serval seconds. This interference can be avoided by waiting for an interval after applications start.

4.3 UI Interaction Simulator

When we traverse the UI tree, we will trigger the listeners of every UI element. For every UI element, we will check its type and check whether its corresponding listener exists. Then we invoke the corresponding trigger functions. Table 1 shows different view types, as well as their corresponding listeners and trigger functions. There is one exception: as the AdapterView has a list of views, we should invoke "performItemClick", "onItemLongClick" and "onItemSelected" functions by the position from 0 to the size of list. In this paper, we have not handled "View.OnTouchListener" and "View.OnKeyListener" event, since they have too many possible parameters to enumerate. This will be our future work.

Besides, we must consider what the best time is to traverse the UI tree in an Activity. If we traverse the UI tree early, the Activity may not be created fully and screen coordinates of the UI element may not be prepared. If we traverse the UI tree late, it could increase the time cost very much. Fortunately, we know that the screen coordinates will be prepared well once it is drawn. In the implementation of the Activity mechanism, there is an invocation of the "makeVisible" function which sends a message to the "ViewRoot" object to draw all views. So, we add some codes at the end of "makeVisible" function to send a message. Our message will be handled immediately after the prior message of drawing all views is handled. In the handler of our message, we traverse the UI tree of Activity and interact with UI elements.

TYPE	LISTENER	TRIGGER METHOD
View	OnClickListener	performClick()
View	OnLongClickListener	performLongClick()
View	OnFocusChangeListener	onFocusChange()
Adapter View	OnItemClickListener	performItemClick()
Adapter View	OnItemLongClickListener	onItemLongClick()
Adapter View	OnItemSelectedListener	onItemSelected() onNothingSelected()
CompoundButton	OnCheckedChangeListener	setChecked()

Table 1: Different view types and their corresponding listeners and trigger methods.

APP name	Package Name	Sensitive Behavior	a	b	c	d	e	f
Horoscope	fr.telemaque.horoscope	Read IMEI	64	10	54	12	16.34	44.32
a.payment.operaupdater	com.soft.android.appinstaller	Send SMS	6	5	2	1	8.32	28.21
Dalton.The.Awesome	com.depositmobi	Send SMS	3	3	1	1	8.93	27.43
htc.notes	com.depositmobi*	Send SMS	2	2	1	1	6.24	27.13
mobileagent	opera.updater	Send SMS	1	1	1	1	5.73	26.72
kate.v2.2	com.android.installer.full	Send SMS	1	1	0	0	6.09	27.26
android.icq	pushme.android	Send SMS	0	1	1	1	5.15	25.52

Table 2: Evaluation Results.
a) "startActivity/startActivityForResult" number, b) Activity number in the ACG, c) Expected Activity switch paths number, d) Confirmed activity switch paths number, e)Static analysis time (s), f) Dynamic analysis mean time per path (s)

5. EVALUATION

In this section, we present the evaluation results of testing several Android malwares, which are listed in Table 2. Except for the Horoscope sample which is introduced in Section 2, we find other six Android malware families among 19 applications. In these families, the "a.payment.operaupdater" is provided by the Antiy [3] and others are obtained from the Contagio [6], which is a public share platform of Android malwares. Here, we do not list names of all applications in Table 2, but just select one malware in each family. Even though these six families have the same behavior of sending SMS, their disassemble codes and user interfaces are different. For example, the samples with the "com.depositmobi" package name have two buttons in the first Activity, but there is only one button in the samples with the "com.depositmobi*" package name. In these six families, the trigger condition is very simple, since it just needs one-click on a button in the first Activity.

We have not tested our system by a large amount of Android applications. Because it is difficult to find Android applications in the wild, which need UI-based conditions to trigger sensitive behaviors. In the characterization of the majority of exiting Android malwares [30], most malicious behaviors of Android malwares are activated by the system-wide Android events, such as the "BOOT_COMPLETED" and the "SMS_RECEIVED". Nevertheless, from some samples we captured, there are some samples having UI-based trigger conditions to circumvent existing dynamic analysis tools effectively. We believe more vicious developers will apply this kind of UI-based activations into their sensitive applications in the future.

We describe our evaluation results in Section 5.1 and then present case studies of some samples in Section 5.2.

5.1 Results Overview

We evaluate the SmartDroid system by samples above and the results are listed in Table 2. Since the application's codes and user interfaces are same if they have the same package name, we just list the results for each package name.

The invocation number of the "startActivity" and "startActivityForResult" functions, the Activity number in the ACG and the number of expected Activity switch paths are analyzed by the Static Path Selector. For each expected Activity switch path, we manually verify the path by tracing the disassembled smali codes. Then we run the application in the Dynamic UI Trigger once for each expected Activity switch path. Finally, we can confirm each path whether it is feasible and also get the final UI-based trigger conditions. From the results, some paths are not feasible with some reasons, which will be elaborated in the following section 5.2.

For the processing time, we analyze these samples on the AMD Opteron 64 X4 2376 with 4GB of memory. For the static analysis time, it depends on the code size of samples and the invocation number of the "startActivity" and "startActivityForResult" functions with the reason that the Static Path Selector must scan every instruction and analyze the Intent in parameters of the "startActivity" and "startActivityForResult" functions. For the time of dynamic analysis, we analyze samples for each expected Activity switch path and then calculate the mean time per path. In the dynamic analysis stage, the processing time consists launching the emulator, installing the apk into the emulator, waiting for automatic behaviors and interacting with UI elements automatically. In our experiment, the interaction time mainly relies on how many Activities the expected Activity switch path have. The Horoscope app has more than three Activities in the expected Activity switch path and other samples just have only one Activity in the expected Activity switch path. Hence, the processing time of the Horoscope app is more than the time of other samples.

To demonstrate the effectiveness of the SmartDroid system, we discuss, in more detail, a few cases which have an UI-based trigger condition to start sensitive behaviors. Our system is able to handle these cases and reveal their UI-based trigger conditions.

5.2 Case Studies

In this section, we evaluate our SmartDroid system in several case studies to demonstrate the effectiveness of the system.

5.2.1 *The Horoscope App*

In the Horoscope app, there are mainly three types of sensitive behaviors: reading the IMEI, getting the location information and connecting to the Internet. But according to sensitive function call paths, we know that this application gets the location information and connects to the Internet automatically for advertising after it starts. User interaction can only lead to connect to the Internet, as well as read the IMEI.

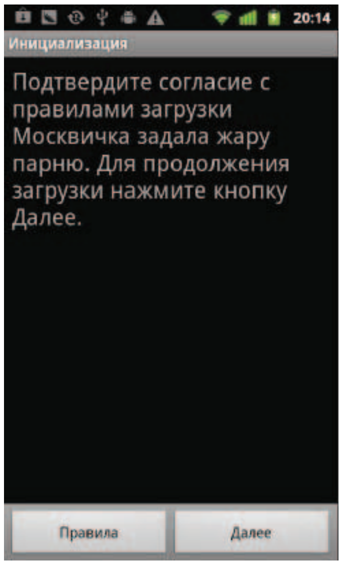
We have analyzed all 72 Intents in this application by static analysis, and get the target Activity and the source Activity linked by the Intent. Then we construct the ACG shown in Appendix A. There are five sensitive sink Activities labeled with red color, in which "CheckConnection" Activity is related to the sensitive behavior of reading the IMEI, and other Activities are only related to the sensitive behavior of connecting to the Internet.

There are 54 expected Activity switch paths from the "EntryPoint" Activity to sensitive sink Activities in the ACG. For each path, we run this application in the Dynamic UI Trigger to force this application to execute along the path. After dynamic analysis, we confirm that there are only 12 paths leading to sensitive behaviors. In order to find the reason why other paths cannot lead to sensitive behaviors, we analyze the disassembled smali codes manually. The reason is that actually there is not a button in the "MonthHoroscope", "DayHoroscope" and "Day2Horoscope" Activity for connecting to the Internet. We realize that the button for connecting to the Internet is generated at runtime according to the returned value from its server. So, it is incorrect that the "MonthHoroscope", "DayHoroscope" and "Day2Horoscope" Activity are sink sensitive Activities determined by static analysis.

The SmartDroid can output the detailed UI-based trigger conditions. For example, the expected Activity switch path *(EntryPoint → My_iHoroscope → My_Sign → ChooseSign → CheckConnection)* can lead to read the IMEI. The condition for this behavior is displayed in Table 3. From this table, we can know a detailed sequence of user interactions including the UI element's type, coordinates, height, width and its corresponding event type. Provided these user interaction information, it is easy to replay this sensitive behavior automatically by the MonkeyRunner.

5.2.2 *The a.payment.operaupdater*

The a.payment.operaupdater [10] is an Android malware which is a fake browser software to entice users to click one button for activating this application. But if users click the certain button, this malware will send SMS to deduct the phone fees. In this case, there are two buttons in the first Activity shown in Figure 4. If we click the right button, it will send SMS. So, the UI-based trigger condition of this sample is one-click event on the right button of the first Activity.



Подтвердите согласие с правилами загрузки Москвичка задала жару парню. Для продолжения загрузки нажмите кнопку Далее.

Figure 4: The a.payment.operaupdater.

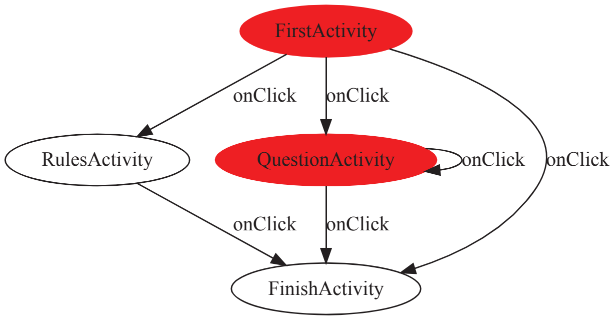


Figure 5: The ACG of a.payment.operaupdater.

The ACG of this sample is showed in Figure 5. There are two sink sensitive Activities and two expected Activity switch paths in the ACG. However, by dynamic analysis we confirm only one path ("FirstActivity") is feasible. The Dynamic UI Trigger cannot walk along the path ("FirstActivity" → "QuestionActivity"). After analyzing the disassembled smali codes, we find that when users click the right button in "FirstActivity" Activity, there will be a judgement on the size of the "this.loc" array. If the size of the "this.loc" is zero, this application can switch from "FirstActivity" Activity to "QuestionActivity" Activity. But by inspecting the config file of this application, the size of the "this.loc" must not be zero. So, "FirstActivity" Activity always switches into "FinishActivity" Activity when uses click on the right button.

From this case, we can clearly know that some logics in applications may affect the execution of expected Activity switch paths extracted by the Static Path Selector. But this would not affect the final trigger conditions, because all of final trigger conditions are confirmed by the Dynamic UI Trigger. So, it just would improve the processing time of dynamic analysis.

Activity from	Activity to	UI	Event	x	y	h	w
EntryPoint	My_iHoroscope	None	None	–	–	–	–
My_iHoroscope	My_Sign	android.widget.ImageView	onClick	83	405	46	154
My_Sign	ChooseSign	android.widget.ImageView	onClick	110	85	100	100
ChooseSign	CheckConnection	android.widget.ImageView	onClick	83	299	46	154

Table 3: One UI path of Horoscope App that will reach to Read IMEI behavior.

5.2.3 The Kate.v2.2

The samples with the "com.android.installer.full" package name are very special. The number of the expected Activity switch path is zero, because the sensitive source function in the FCG is the "onCreate" function of the main Activity. In other words, the behavior of sending SMS is automated after the application starts. However, the Dynamic UI Trigger can detect one trigger condition of this behavior. There is no sensitive behavior found before the interaction with UI elements in the main Activity. However, when the Dynamic UI Trigger clicks the only one button in the main Activity, this application sends three SMSs.

We analyze the disassembled smali code and conclude that the behavior of sending SMS indirectly relies on the click event. A thread is started when the main Activity starts. In this thread, there is an implementation that it sends SMS only if the "startsend" variable is equal to 1. But the default value of the "startsend" variable is 0. In the "onClick" function of that button, the "startsend" variable is assigned with 1.

In this case, the SmartDroid is able to reveal this simple indirect UI-based trigger condition. But now, it cannot reveal some complex indirect conditions. That is our future work which will be discussed in the following Section 6.

6. DISCUSSION AND FUTURE WORK

In this paper, we aim to automatically interact with UI elements in the Activity and enforce the execution of applications along the expected Activity switch path. Our system just focuses on the situation that the trigger conditions of sensitive behaviors are not data dependent on UI elements. In other words, we only consider the control dependency on UI elements. The indirect UI-based condition described in Section 5.2.3 is an example of the data dependency. In applications with indirect UI-based conditions, some comparison instructions determine whether the sensitive behavior can be started. However, the UI event function can assign the values to registers in the comparison instruction. So, there is a data dependency between the comparison instruction and the UI event function. In future work, we will analyze this kind of data dependency and add some edges of the data dependency into the FCG. With this process, the expected Activity switch paths will be more complete. Besides, there exist data dependencies when the Activity is created. The creation of some UI elements is related to the data from the servers or configuration files. If sometimes the data does not allow the creation of UI elements which can lead to sensitive behaviors, our system cannot reveal this hidden UI-based trigger conditions. This is our future work to conquer this challenge.

We have not considered the logic-based trigger conditions which can affect sensitive behaviors. For example, the zSone malware [30] invokes the SMS sending code when users click a button in the fifth time. Now our SmartDroid cannot detect this sample, but we are ready to adopt a rewriting apk method to solve this problem. In static analysis, we analyze each comparison instructions in disassembled smali codes. We want to determine whether each branch of the comparison instruction points to sensitive APIs. If one of branches points to a sensitive API, but others do not point to a sensitive API, we should continue to modify this key comparison instruction to enforce all branches to point to this sensitive API. Then we reassemble these modified smali codes to a new apk and sign it [12]. This technique can bypass the logic tricks and also get the logic-based trigger conditions in some cases. However, there may be an exception thrown at runtime if we modify every key comparison instruction. The judgement on which key comparison instruction is the real logic condition is difficult and we will continue to seek the solution for this problem.

Moreover, we just handle some of UI events without complex events, such as gestures. In order to simulate the correct gestures, we should identify each gesture in static analysis. That is a challenging work in the future. Our system has not covered some sensitive behaviors, such as rooting, which is existed in the wild so far. Given this, we should continue to examine possible root causes and explore future solutions. We do not consider the native codes, in which the developer can invoke the sensitive API in the C or C++ language by the JNI [9]. Disassembling native codes to analyze in the static analysis is our future work.

When coming to dynamic analysis, we traverse all UI elements in an Activity after the creation of the Activity. We choose to traverse it after the Activity finishes its "onResume" function to draw. But there is a possible situation that an application starts its sensitive behavior in an Activity's "onPause", "onStop", "onDestory" or "onRestart" functions, even though we have not found one sample like this in the wild so far. Our system could not detect the sensitive behaviors in this kind of samples. However, it is not very difficult to solve this problem. Our static analysis can get the sensitive source functions. If it finds that these Activity's functions belong to sensitive source functions, we will take the extra procedure in dynamic analysis phase. For the "onPause", "onStop" and "onDestory" functions, we kill this application's process according to its pid after the Activity has been created. For "onRestart" function, we can send a home key event to the emulator and then restart this application again. According to the Activity's life cycle, this solution can work out this problem.

7. RELATED WORK

Analyzing mobile applications is an emerging hot field in academic research. Our work covers both static analysis and dynamic analysis. On static techniques, PiOS [24] use data flow analysis and slicing techniques to check whether the iOS application leaks sensitive information. The AndroidLeaks [21] reverses the Dalvik bytecode to the java bytecode, then applies WALA to do static taint analysis. Enck et al. [18] developed a better decompiled tool ded to reverse Dalvik bytecode to Java source code with up to 98.04% recovery rate, and it applies the Fortify SCA static analysis suite to study 1,100 free Android applications. Grace et al. [25] developed Woodpecker which does control flow analysis based on the smali code. Chin et al. [15] proposed ComDroid, which can statically analyze the disassembled Dex bytecode from Dedexer and identify potential components' Intent vulnerabilities. However, those static analysis techniques are not sufficient in analyzing the GUI components such as button, and they do not have runtime information. Our SmartDroid makes use of the static analysis information to guide the dynamic analysis, and we can solve the UI paths.

On the side of dynamic techniques, TaintDroid [17] uses a system-wide dynamic tracking technology to identify each privacy leak. Our SmartDroid can solve the UI paths automatically, which is complementary to TaintDroid. Gilbert et al. [22] build the AppInspector which extends TaintDroid to track some types of implicit flows and confirms that the random input approach for automated analysis [23] is very poor and unaccepted. It also proposes to use the concolic execution approach to generate user inputs. Compared with AppInspector which covers all branches, our SmartDroid only executes suspicious paths provided by static analysis within a short execution time. Saswat et. al [27] build the CONTEST to generate input events to exercise smartphone apps including Android applications. But its intention is different from our SmartDroid. SmartDroid aims to know the effective user input events which make the sensitive behaviors happen, but CONTEST works on GUI testing and on alleviating path explosion in concolic testing. Besides, CONTEST just focuses on generating input events for a single Activity, but SmartDroid can generate input events intelligently for multiple Activities. Crowdroid [13] bypasses this automated generating inputs challenge by collecting traces from an unlimited number of real users based on crowdsourcing. It applies the method of monitoring system calls to detect Android malwares. But this approach needs many Android users to be convinced to install Crowdroid and it only protects users who install it. None of the above efforts support thoroughly generating UI-based trigger conditions for Android applications.

8. CONCLUSION

In this paper, we combine the static and dynamic analysis to automatically revealing UI-based trigger conditions in Android applications. Through static analysis, we first build the Activity Call Graph and Function Call Graph for an app, and extract expected Activity switch paths. Then, we use dynamic analysis to traverse each UI element and explore the UI interaction paths towards the sensitive APIs. We implement a prototype system SmartDroid, which augments existing dynamic analysis tools with automatic UI interaction analysis capabilities. The result shows that our SmartDroid system is very effective in revealing UI-based trigger conditions automatically.

Acknowledgements

This research is supported by the NDRC under Project "A Cloud-based service for monitoring security threats in mobile Internet" and the Beijing Technological Innovation Project (Grant No. Z111101055311052).

9. REFERENCES

[1] Android market growth. `http://android-developers.blogspot.com/2011/12/closer-look-at-10-billion-downloads.html`.

[2] Android snapshot. `http://pastebin.com/bCieGJVV`.

[3] Antiy corp. ltd. `http://www.antiy.com/cn/about/index.htm`.

[4] Apktool. `http://code.google.com/p/android-apktool/`.

[5] Bouncer. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`.

[6] Contagio. `http://contagiominidump.blogspot.co.il/search/label/Russian`.

[7] The horoscope app. `https://play.google.com/store/apps/details?id=fr.telemaque.horoscope`.

[8] Introducing google play. `http://googleblog.blogspot.com/2012/03/introducing-google-play-all-your.html`.

[9] Jni. `http://developer.android.com/guide/practices/jni.html`.

[10] Operaupdater. `http://www.18digi.com/news/7361/tencent-security-laboratory,-december-11-mobile-phone-viruses/`.

[11] Sensitive apis. `http://www.android-permissions.org/`.

[12] A. K. Benjamin Davis, Ben Sanders and H. Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. In *Proceedings of the Mobile Security Technologies 2012*, MOST '12. IEEE, 2012.

[13] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM.

[14] P. P. Chan, L. C. Hui, and S. M. Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 125–136, New York, NY, USA, 2012. ACM.

[15] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[16] S. Dienst and T. Berger. Mining interactions of android applications static analysis of dalvik bytecode. Technical report, Department of Computer Science, University of Leipzig, Germany, May 2011. Technical Note.

[17] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[18] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[19] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[20] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[21] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th International Conference on Trust&Trustworthy Computing*, TRUST '12, pages 291–307, Vienna, Austria, 2012.

[22] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, MCS '11, pages 21–26, New York, NY, USA, 2011. ACM.

[23] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, New York, NY, USA, 2011. ACM.

[24] E. Manuel, K. Christopher, K. Engin, and V. Giovanni. Pios: Detecting privacy leaks in ios applications. In *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS '11, 2011.

[25] G. Michael, Z. Yajin, W. Zhi, and J. Xuxian. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS '12, 2012.

[26] J. Midtgaard and T. P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 287–298, New York, NY, USA, 2009. ACM.

[27] A. Saswat, N. Mayur, Y. Hongseok, and J. H. Mary. Automated concolic testing of smartphone apps. In *Proceedings of the ACM Symposium on Foundations of Software Engineering*, FSE '12, March 2012.

[28] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. "andromaly": a behavioral malware detection framework for android devices. *J. Intell. Inf. Syst.*, 38(1):161–190, Feb. 2012.

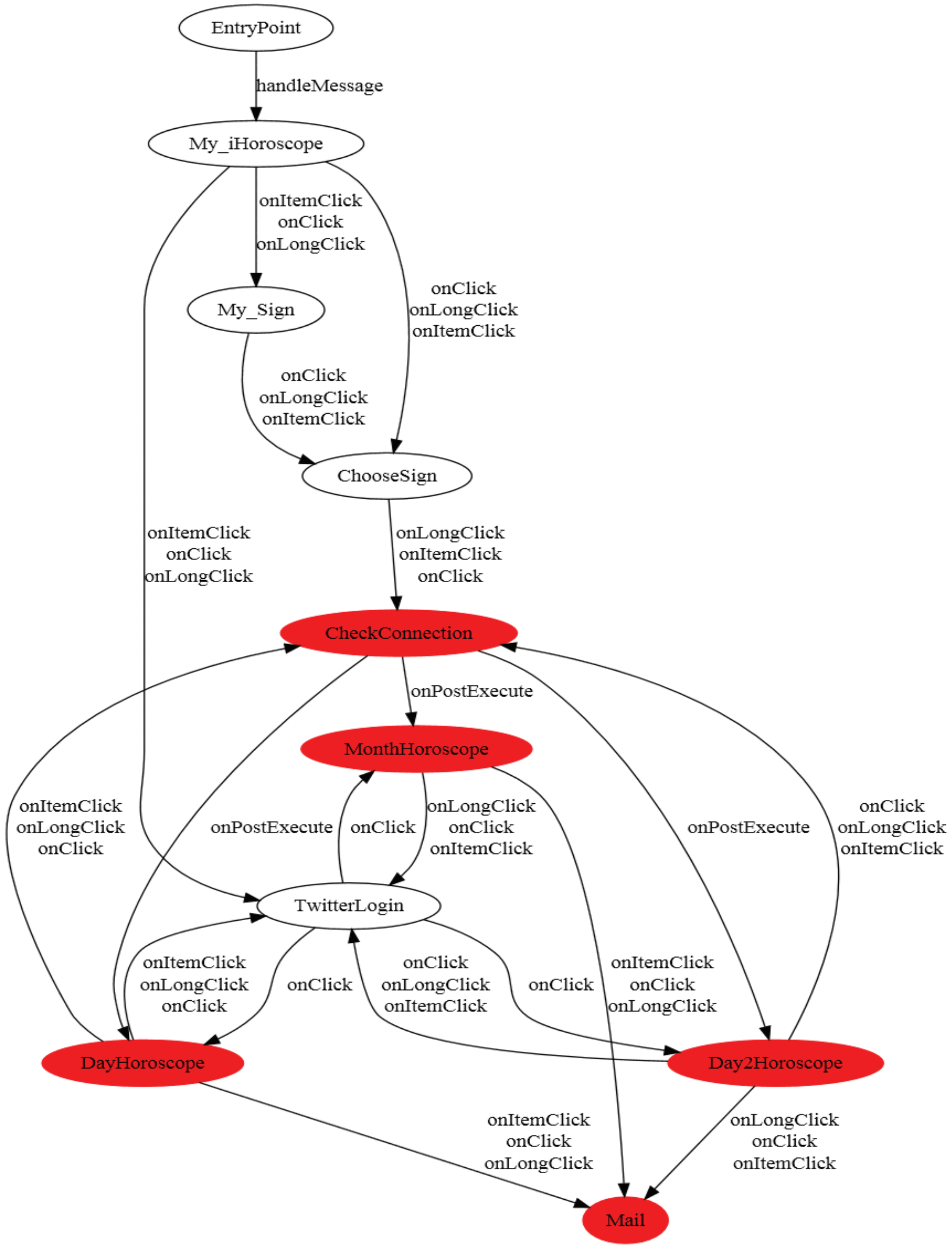
[29] B. Thomas, B. Leonid, S. Aubrey-Derrick, and A. C. Seyit. An android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, Malware '10, pages 55–62, 2012.

[30] X. J. Yajin Zhou. Dissecting android malware: Characterization and evolution. *Security and Privacy, IEEE Symposium on*, 0:95–109, 2012.

[31] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

APPENDIX

A. ACG OF HOROSCOPE


EntryPoint
handleMessage
My_iHoroscope
onItemClick onClick onLongClick
My_Sign
onClick onLongClick onItemClick
onClick onLongClick onItemClick
ChooseSign
onItemClick onClick onLongClick
onLongClick onItemClick onClick
CheckConnection
onPostExecute
MonthHoroscope
onItemClick onLongClick onClick
onPostExecute
onClick
onLongClick onClick onItemClick
onPostExecute
onClick onLongClick onItemClick
TwitterLogin
onItemClick onLongClick onClick
onClick
onClick onLongClick onItemClick
onClick
onItemClick onClick onLongClick
DayHoroscope
Day2Horoscope
onItemClick onClick onLongClick
onLongClick onClick onItemClick
Mail