Contents

Chapter 1
Automatic Security Analysis of Android Applications with AppsPlayground

Vaibhav Rastogi, Yan Chen and William Enck

1.1 Introduction

Mobile devices such as smartphones have gained great popularity in response to vast repositories of applications. Most of these applications are created by unknown developers who may not operate in the users' best interests, leading to malware [17, 21] and frequent exposure of privacy sensitive information such as phone identifiers and location [10, 9, 11].

Researchers have proposed both static and dynamic security analysis techniques for smartphone applications. While static analysis approaches [9, 11, 12] scale to large numbers of applications, they do not capture runtime environment context such as configuration variables and user input. More importantly, code may be obfuscated to thwart static analysis, either intentionally or unintentionally (such as stripping symbol information of native binaries to reduce size).

On the other hand, TaintDroid [10] uses dynamic analysis to capture runtime environment context. However, the researchers had to manually navigate the user interfaces of each analyzed application to sufficiently exercise dangerous functionality. More recently, DroidScope [39] used dynamic analysis for malware forensics. Large-scale dynamic analysis however requires more than what has been proposed earlier – a fast analysis system and strategies to provide automatic code coverage.

In this chapter, we describe *AppsPlayground*, referred to as simply *Playground* for brevity, a framework for automated dynamic security analysis of Android applications. Playground is meant to analyze applications for both malware, i.e., apps that have a malicous intent, and grayware, i.e., apps that are not malicious but may still

Vaibhav Rastogi
Northwestern University, e-mail: vrastogi@u.northwestern.edu

Yan Chen
Northwestern University, e-mail: ychen@northwestern.edu

William Enck
North Carolina State University, e-mail: enck@cs.ncsu.edu

be annoying, for example, by leaking private information for a legitimate purpose but without user's awareness. From this point on, for the sake of conciseness, we will not particularly distinguish between malware and grayware and refer to them both as malware. An automatic dynamic analysis framework needs to possess not only detection techniques for identifying malicious or annoying functionality but also automatic exploration techniques to explore the application code as much as possible. Furthermore, the dynamic analysis environment needs to appear as real (in this case, a real smartphone) to the app as possible, lest a malicious app can easily detect the special environment and not show any malicious behavior. Finally any dynamic anlaysis system can also benefit from a preliminary static analysis, which can direct the analysis techniques to be applied at runtime.

In Playground, solutions to all the above requirements are integrated together in a modular manner. We use multiple detection techniques, ranging from taint tracing to kernel-level system call monitoring. For taint-tracing, we are able to seamlessly integrate and reuse TaintDroid [10], an already available taint-tracing engine with very good performance for Android into the rest of our system. In order to deal with root attacks in Android, we describe vulnerability conditions in Android as succint signatures in terms of system calls and kernel-level data structures. These signatures may easily be incorporated into a dynamic analysis.

For automatic exploration, we find that the nature of Android imposes non-conventional requirements on the exploration techniques that need to be used. Application code can be triggered by several kinds of system events and so such events need to be simulated. Moreover, most of the apps in Android provide GUI, which requires sophisticated GUI exploration schemes. Trivial approaches for GUI exploration such as fuzz testing have their advantages in their simplicity and, if designed properly, have the ability to eventually exhaustively explore a finite state space. They however take more time and are sometimes insufficient because application user interfaces have complex requirements such as login credentials for Internet services. Therefore, we also need to intelligently drive the user interface to exercise code implementing interesting and dangerous functionality. Our heuristic-based intelligent execution technique is able to avoid redundant exploration and is able to use contextual information to fill editable text boxes meaningfully.

Playground also enables a light-weight static analysis component, that can recognize use of specific APIs withing apps; such input is valuable for automatic event triggering and other kinds of guided UI exploration.

To demonstrate the practical advantage of Playground, we evaluated 3,968 from the official Android Market (now Google Play). We identified exposures of privacy sensitive information in 946 applications, flagged by the taint-tracing engine. Of these, 844 applications leaked phone identifiers (such as phone number and IMEI), and 212 applications leaked geographic location. We note that detecting privacy violations still requires manual confirmation, as TaintDroid only identifies that information has left the device over the network interface, and not privacy violations. For further validation, we also tested the applications used in the TaintDroid study. Playground's findings almost completely coincided with the findings manually made by the TaintDroid authors on the much smaller set of thirty applications they evaluated.

Furthermore, we also evaluated Playground on known malware samples, falling under diverse categories of root attacks and SMS trojans, and were able to detect the malicious nature of all of them.

Finally, to evaluate the performance of automatic GUI exploration, we compare our system with GUIRipper [23], a system that automatically generates test cases based on windowing elements in traditional desktop GUIs. It lacks advanced techniques such as filling in contextual data in text boxes and repeatedly exercising GUI widgets to achieve better code coverage, both of which we have found are often critical requirements when testing Android applications. Our comparison with an Android port of this system shows our technique to achieve a mean 30% improvement in terms of code coverage. Furthermore, we also compared intelligent exploration with fuzz testing. Both the techniques are included in our system and work in a complementary manner. We found that both the techniques achieve nearly identical code coverage but can be combined together to offer increased code coverage.

While many of the techniques used in Playground are well-known or not very challenging, they are combined together to form a very versatile analysis system. We describe the following key contributions in this chapter.

- We propose AppsPlayground (or simply, Playground), a modular framework for scalable dynamic analysis of Android application.
- We identify the key requirements for automatically exploring Android applications. We use automatic system event triggering and propose and develop a new intelligent execution technique that can use contextual information to provide meaningful textual input.
- We describe vulnerability conditions for known vulnerabilities in Android as succint signatures that may be used in dynamic analysis. These vulnerability conditions are necessary for a system compromise.
- We implement the AppsPlayground framework for Android and evaluate 3,968 applications from the official Android app Market. Our analysis identified exposures of privacy sensitive information in 946 applications. Moreover, we were able to confirm the malicious nature of already known malware samples using this framework.
- we perform a thorough evaluation of the exploration techniques used in AppsPlayground based on instruction-level code coverage and compare our techniques to a previous technique of GUIRipper [23].

The remainder of this chapter proceeds as follows. Section 1.2 provides relevant background in Android and Section 1.3 gives an overview of Playground. Sections 1.4-1.7 provide detailed discussion of the techniques incorporated into Playground. Section 1.8 discusses the implementation of Playground. Section 1.9 describes our measurements with Playground. Section 1.10 discusses the effectiveness of the automatic exploration techniques employed. Section 1.11 presents related work and Section 1.12 concludes.

1.2 Android Background

Android is a widely popular and open source operating system designed for smartphones and other mobile devices. Applications are distributed primarily through app stores, or application repositories. Users are free to download applications from any app store, without much control of a central party. Likewise, developers have a number of choices to reach the users. Applications are often not reviewed for quality and security assurance before they reach the user. Given such openness of the current ecosystem around Android, it is unsurprising that Android malware is on the rise, and simultaneously, there are a number of grayware applications of which the users may need to be wary. The astronomical numbers of applications make the task of manually reviewing applications infeasible; hence automatic tools are needed. Playground is an effort in this direction.

While Android is based on Linux, it defines an entirely new middleware and GUI environment in which applications execute. Applications are mostly written in Java, which is compiled to Dalvik bytecode, which runs in a virtual machine similar to the Java virtual machine. Apart from Java, Android also allows parts of apps to be coded in native code.

Every Android application runs as an unprivileged user with Linux UIDs effectively being used to provide application sandboxes. Android applications are composed of *components*. There are four component types: *activity*, *service*, *broadcast receiver*, and *content provider*. The user interface is defined by one or more activity components. Services are meant to run in background while content providers manage access to data. Broadcast Receivers are registered with system services and can receive system events, such as reboot completed, or an SMS received, and so on. Once a broadcast receiver is registered to receive a system event, the code specified in the broadcast receiver is run whenever the system event is triggered. (This may sometimes not hold due to, for example, abort of a broadcast.) Most system events are guarded by permissions, which the app must declare and get approved for at installation time.

For automatic exploration, it is necessary to understand the GUI features in Android. Each activity corresponds to a screen displayed to the user. This screen is functionally equivalent to a traditional GUI window, the only difference being that only one screen is shown at a time (with minor exceptions), whereas traditional GUIs can typically display multiple windows.

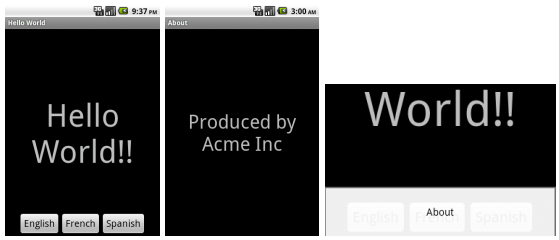
An application's GUI consists of several activities that invoke one another and possibly return results. At any point in time, only one activity has input focus and processing. This activity is referred to as the *active* activity. When one activity invokes another, the former is paused and the new activity is pushed to the top of the activity stack and made active. Once an activity has completed its work, it terminates, optionally returning a value, and the next activity on the stack is made active. Note that activities are not limited to invoking activities within the same application. A sequence of related activities on the stack is called a *task*.

There are several exceptions to this linear user interface workflow. First, at any point in time, the user can press the "home" button that returns the device display

to the launcher from which the user can start a new application, which begins a new task. Second, the user can return to a running application and its task by "long-pressing" the home button and selecting it from the displayed list of applications. Third, various flags can be used when invoking activities that reorder the activity stack (e.g., move an existing activity to the top of the stack).

The activity GUI layout is commonly defined in XML but may also be defined programmatically. As in traditional GUIs, an Android window consists of widgets, which are are referred to as *views* in Android terminology. The Android library supplies several useful views which may either be standalone (e.g., buttons) or act as containers for other views. In addition to the window layout, an activity can define a menu that appears when the user presses the physical "Menu" button on the phone.

Example. Figure 1.1 shows a simple example application. The application consists of two activities, "Hello World" and "About" (Figures 1(a) and 1(b), respectively). The "Hello World" activity has three buttons which bring up the "Hello World!!" message in three different languages. The "About" activity is non interactive. There is a menu attached to the "Hello World" activity, which we model as a separate window. After opening this menu, one may click on the only option (named "About") to go to the "About" activity. Figure 1.2 depicts the GUI hierarchy of the window in Figure 1(a).



(a) (b) (c)

Fig. 1.1 A simple application with three windows. Window (*a*) invokes window (*c*) which invokes window (*b*). (*c*) shows only the lower half of the screen emphasizing the menu window.

1.3 AppsPlayground Overview

This section gives a broad view of Playground. We begin with describing the overall architecture of Playground followed by the different components involved in brief.

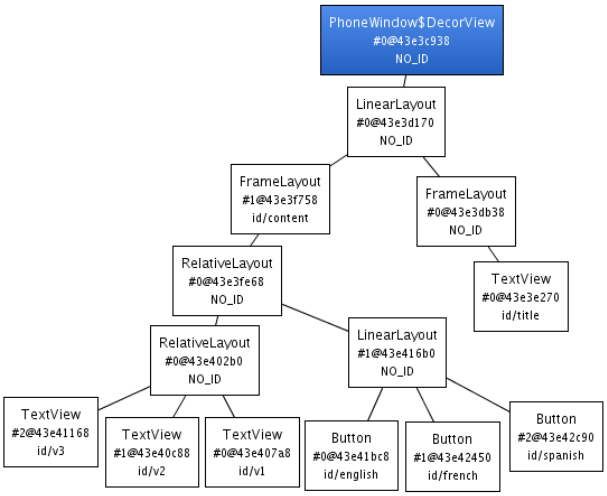


Fig. 1.2 The GUI hierarchy for the window in Figure 1(a)

1.3.1 Overall Architecture

We seek to design a general framework for automatic dynamic analysis for smartphone applications. Playground is built as a virtual machine environment. Specifically, it repurposes the Android emulator, available with the Android SDK, for the dynamic analysis environement. Built on Qemu [27], the emulator emulates an ARM machine and provides support for a few features available on a real phone, such as telephony.

A virtualized environment is essential to providing scalability required for malware analysis. For example, every analysis can use a fresh snapshot of the environment without affecting the analyses of other samples; this is not feasible when using real devices. However, different from a few past approaches [39], we do not employ virtual machine introspection, a technique in which the virtual machine (VM) guest is run unmodified and any analysis tools run outside the VM, analyzing its physical memory to get information from inside the virtual machine. This approach while complicated, allows the analysis tools to be strictly more privileged than the analyzed environment.

In the case of Android however, apps typically run as unprivileged users and hence introspection is not actually required. Even for known attacks that try to get root privileges, signatures may be developed for identifying the attack and safely recording it before the privilege escalation actually completes. For apps requiring root (through su), these arguments do not apply; however, the number of such apps is low and the number of rooted devices is also significantly smaller. Furthermore, the complexity of introspection also hinders in the retrieval of GUI information or sending events from outside the emulator.

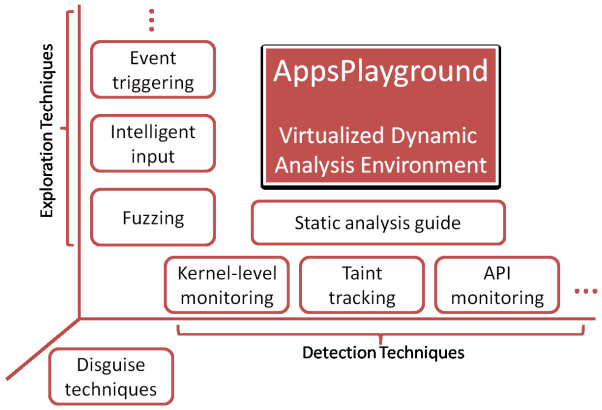


Fig. 1.3 Architectural overview of Playground analysis framework. The figure illustrates the components incorporated in to the framework organized in different dimensions. Playground is a modular framework and allows adding new components in the depicted dimensions or possibly adding altogether new dimensions.

Figure 1.3 shows the architecture of Playground. Playground has several components comprising multiple detection techniques, multiple automatic exploration techniques, and techniques to make analysis environment appear like a real phone. All these components work independently of each other and integrate together in a plugin-able manner. A static analysis component guides the execution in several of the former components for efficient and effective analysis. We next briefly discuss the components listed in the figure.

1.3.2 Playground Components

Detection techniques are the components that actually provide the detection of a possibly malicious functionality while a sample is being tested. The detection techniques that we include are taint tracing for information leakage detection, based on TaintDroid; sensitive API monitoring, such as monitoring for the SMS API; and kernel-level monitoring for detection of root exploits. Disguise techniques are those that make the environment appear like a real device; these include the use of realistic phone identifiers, keeping realistic data in phone databases, and so on.

Automatic exploration techniques help in automatically increasing code coverage of the application code. Without automatic code coverage, it is likely that much of the code in an application will not be executed. Playground simulates events, such as location change and sms received, to trigger code in event receivers (primarily broadcast receivers). To explore the app GUI, we use fuzz testing and intelligent black-box execution. Since fuzz testing simply sends in a stream of random inputs, it may be described as a random walk on the state space. Given the ability to restart

from the start state any number of times, it can eventually explore any finite connected state space. Applications that do not need any meaningful text to be filled in have a small state space consisting of screen taps and drags. Fuzz testing can deal with such applications quite well without any knowledge of their interaction model. On the other hand if some meaningful texts such as login credentials are required, fuzz testing cannot enter in the right input, and fails. For such cases, we need intelligent execution, which heuristically determines what data has to be entered in. Furthermore, since fuzz testing is random, it may sometimes fail to explore some states. Intelligent exploration however deterministically explores states that it can model.

Intelligently driving the user interface of smartphone applications presents several challenges:

- *Modeling the GUI.* In order to intelligently exercise the user interfaces of applications, a representation of the program flow must be abstracted from the GUI. The closeness of this approximation to the actual program flow determines the completeness of the automation algorithms.
- *Efficient exploration strategy.* Even simple applications can have a very large (if not infinite) number of unique program states based on user input (e.g., a counter). Practical testing of applications requires an efficient exploration strategy with the ability to effectively discover distinct and useful states and remove redundant states.
- *Context determination.* Applications often have text fields that require special values. Leaving them empty or filling in garbage can limit application exploration. A few real world examples follow.

  – *Login credentials.* Unless a correct username and password is supplied in the correct fields, the exploration of the application will be seriously limited.
  – *Cities and zip codes.* Application functionality depending on zip codes and cities entered in input fields will likely fail in the presence of random input.
  – *Duplicate input fields.* Applications occasionally require the user to enter the same information in two text fields for consistency checks, e.g., passwords, PINs, and Email addresses.
  – *Input format.* Fields such as Email addresses and phone numbers are occasionally required to be entered in a specific format before the application will accept the input.
  – *Dates.* A future date may not work when a past date is expected. An application which asks for date of birth may not move forward if a date is in the past but is one that does not indicate the user is now over 13.

  In all these cases, Playground must infer from the context present around text fields what should be filled in. We note in most cases, these inputs are validated by remote servers and so even symbolic execution cannot help determine correct values for them.
  Finally, Playground includes a light-weight static analysis component that guides the rest of the analysis at run time. It is primarily used to prioritize the applica-

tions for analysis, and to identify the APIs used in the application so that they may be triggered.

1.4 Detection Techniques

In this section we discuss the various detection techniques that are included in Playground. Other techniques may be included as needed.

Taint tracing. Playground uses taint tracing to track privacy-sensitive information leakage. We have integrated a slightly modified version of TaintDroid [10], an open-source, high-performance taint-tracing system for Android. We note that TaintDroid works only for Dalvik bytecode only. Native code taint-tracing would likely require dynamic binary instrumentation or VM instrospection. We currently do not use such methods for native code taint-tracing; these methods result in a typical slowdown of 10x to 30x for the code and hence are not very attractive from the performance perspective.

Sensitive API monitoring. Playground monitors a few system APIs for detecting possibly malicious functionality. The SMS API is one of the most exploited API in Android. Malicious apps use it to send text messages to premium rate numbers without user's awareness. Playground can record the destination and content of the SMS messages sent by an app. Similarly, Playground monitors the Java reflection API to record method calls and field accesses through reflection as some of these may be indicative of obfuscated codes typical in malware. Playground also monitors dynamic bytecode loading and can inform the analyst of which bytecodes (contained in a .dex file) were loaded. We note that monitoring reflection and bytecode loading APIs is done for application code only. Framework code is trusted and so need not be monitored. The differentiation is done on the basis of class loaders; in Android the class loaders for application code are always different from the class loader that loads the framework code.

Kernel-level monitoring. We also provide kernel-level tracking to identify known root-exploits. Our method of identification of root exploits is based on vulnerability conditions and is thus immune to code polymorphism. We observe that known root exploits such as rageagainstthecage, exploid, and gingerbreak, all have signatures that can easily be used in dynamic analysis without raising too many false alarms:

- Rageagainstthecage/Zimperlich. These attacks fork RLIMIT_NPROC (the maximum allowable) number of processes for a UID (the UID associated with the malicious app) and then make zygote (a system daemon) spawn another process for that user. The zygote daemon typically uses setuid system call to change the UID to the app's uid. However, since this UID already has as many processes as are allowed, setuid fails, and the app gets a process with root privileges. We observe that this attack can be detected simply by monitoring if the number of processes for a user comes close to the maximum allowed.
- Exploid (CVE-2009-1185). This exploit is based on a vulnerability in the init, in which init does not check the origin of NETLINK messages. Untrusted code

may thus be registered and get called later. For this vulnerability to happen, a neccessary condition is that the app code must send a NETLINK message later. We can use this as our signature.

- Gingerbreak (CVE-2011-1823). This exploits a vulnerability in the vold daemon in Android, again requiring untrusted code to send NETLINK messages to vold. Hence our signature here is similar to that for exploid.

We note that the above three are representative examples. In general we can encode conditions for any vulnerability in code; the checks will be inserted in the crtical path that leads to the given vulnerability. We note that the OS used for analysis need not actually be vulnerable for the vulnerability conditions to get triggered. Hence, attacks for vulnerabilities in multiple versions of Android may be detected on the same version. Moreover, attacks that would normally not succeed in the emulator may also be detected.

1.5 Disguise Techniques

Playground adopts a number of measures to make the analysis environment appear realistic. It provides real-looking phone identifiers to the app. These identifiers include the phone number, IMEI, IMSI, Android ID and so on. We also modify the build.prop (a file that contains several properties about the system) properties to match a real device. In a similar vein, we can also modify identifiers that relate to Qemu and other virtualization-related features.

Furthermore, we provide realistic data on the device, such as contacts, SMS, pictures, files on SDCard, and so on. We also provide additional libraries such as the Google Maps library, which is available on real devices. In addition Google apps (a set of Google proprietary apps available on a majority of Android devices) may also be provided though we do not provide them at this moment. Data from sensors such as GPS is also made to appear realistic. Currently, we do not support all sensors. Support for microphones is partial while we do not have any support for accelerometers.

We note that evasion of virtualized environments has long been an issue. Even if the above problems are fixed, there will always be evasion techniques based on timing (virtual devices run slower) and Qemu fingerprinting, for example [28]. These problems are general to all dynamic analysis systems.

1.6 Automatic Exploration Techniques

We discuss here the techniques used for automatic exploration in Playground. The next two subsections describe event triggering and intelligent execution. Fuzz testing being almost a trivial technique is skipped from discussion here. Currently, Playground does not use any symbolic execution, which appears to be a good option for

state space exploration of an app. We note that there are presently no effective symbolic execution solutions for interactive applications such as those involving GUI. Even projects developed around symbolic execution use random walks or fuzz testing to explore the GUI parts of the applications [33]. Symbolic execution can however be used to make event triggering better. For example, SMS messages received from only certain numbers may trigger some code in the application; symbolic execution could be used to construct the right kinds of messages here. We plan to include symbolic execution into Playground as a future work.

1.6.1 Event Triggering

Several API elements in Android are event based. Applications may register some code to be triggered whenever an event happens. There are specific events raised by the system when, for example, an SMS is received, the device location changes, the system completes a reboot, a call is received or is hung up, and so on. Sensitive events are guarded by permissions, which an app must declare statically and get approved for at the time of installation. Many malicious applications have been found to register for specific events [41].

Based on the permissions declared by the application, we raise specific events in the system. For example, if an application contains the BOOT_COMPLETED permission, Playground artificially raises the reboot completed event (note that we use VM snapshots only; booting the VM will be much more time consuming). This triggers the app's code that was registered with this event. However, artificially raising important events may cause system inconsistencies as well. This happened with the reboot completed event. We correct some of the framework code so that it would react to this event only once. Other events are handled similarly.

1.6.2 Intelligent Execution

Playground intelligently drives the user interface of a smartphone application by dynamically defining and exploring a model created from window and widget features. We extract features from displayed user interfaces to iteratively define a model that approximates the application's logic. For example, when an application launches, it displays a window with one or more buttons. When a button is selected, a new window appears. The transitions between windows are captured by this model. Note that this approach is based on the intuition that smartphone applications are highly interactive and that the resulting model provides a good approximation of the application's logic states.

Figure 1.4 presents an overview of the intelligent execution module. For every iteration, Playground checks if focus has changed to a different window. To avoid redundant exploration, a window equivalence module uses heuristics to determine

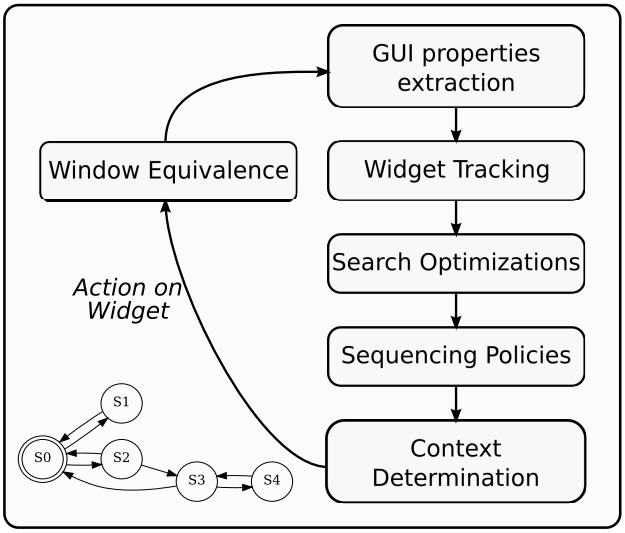


Fig. 1.4 Overview of the intelligent execution module of Playground

if the newly displayed window is similar to previously viewed windows. If so, the window is merged with an existing state. Playground then extracts features relevant to driving the GUI. These include widgets containing texts, editable text fields, buttons, scroll containers and so on. It then creates associations between the current features and those retrieved earlier using widget tracking (why this is needed is discussed below). A few search optimizations are applied next to prune the search space. Next, Playground uses sequencing policies to determine the next GUI action (such as select a button, scroll down, fill text fields). Text fields are filled using heuristics defined by the context determination module. The current iteration is completed with the performance of an action. The rest of this section describes the various modules shown in Figure 1.4 in greater detail.

Despite our focus on Android, many aspects of our design here are more generally applicable to other platforms. Subsequent descriptions are therefore made using platform-independent terminology.

1.6.2.1 Widget Tracking

When navigating windows, widgets may disappear and later reappear. Failure to identify a widget when it reappears may result in concluding identical states or events to be different and hence redundant exploration. For example, consider a window with buttons *A* and *B*. Upon pressing button *A*, the window closes. To complete the exploration, the window is re-opened. The problem would be trivial if the each widget has a unique identifier. This is unfortunately not true for Android.

Playground tracks widgets similar to the way a human user might. We have identified the following widget properties for widget tracking. (1) *Text associated with a widget.* Widgets often have some text associated with them which is made visible to the user, e.g., a text label on a button. In many conditions, this text is sufficient to uniquely identify the widget. However, not all widgets have associated text. Additionally, multiple widgets may have the same text. (2) *Image associated with a widget.* GUI layouts often use widgets containing an image. In such cases, the image can uniquely identify the widget (we modified Android framework for exporting image identifiers which could be hashes of images, their resource names, and so on.) (3) *Position within the window.* Combined with the previous previous, the location of the widget on the screen is a useful indicator. Finally, (4) *Position in the GUI hierarchy.* Widgets often have fixed chains of ascendents. A button, for example, will always have the same chain of ascendents in a window. The user perceives this in terms of the relative positioning of widgets.

1.6.2.2 Sequencing Policies

Each window can contain many widgets that allow input events. In addition to buttons, a window can contain editable text boxes, check boxes, spinners, etc. The result of selecting a button can be directly influenced by interaction with other widgets. Check-boxes can enable/disable other widgets. Finally, scrollable container widgets hide other widgets from the user. Exercising every possible sequence of widget interaction is infeasible. So, we have to arrange the order of event execution in the most meaningful way.

The sequence of interaction with widgets in a window requires consideration. Based on observation, we classify GUI input events into two groups: (*a*) those that input parameters or variables into the app, such as inputting text into an editable text box or a spinner, and (*b*) those that provide actions, such as buttons. First, widgets that accept input variables should be acted upon before action widgets. Second, widgets that are contained within a scrollable container are acted upon before scrolling the container. Third, contents of the scrollable container and the container itself are exercised before acting upon widgets outside the container, except when this is in conflict with the first rule. This design choice follows the intuition that the widgets outside the scrollable widget (if present) are often the control buttons such as “OK”, “Submit”, and “Cancel”.

Note that the choice of these policies has important ramifications. If the behavior of a widget depends on another widget, Playground may not be able to trigger the entire set of behaviors. While we discuss this problem within a single window, it is easy to see such problems would also arise across windows.

1.6.2.3 Search Optimizations

For the sake of practicality, we heuristically prune redundant navigation paths where possible. For items organized as a list or a grid, we explore the items up to a threshold. In addition to reducing exploration time, a threshold is sometimes necessary to achieve program termination. For example, an Android list may dynamically expand and thereby go infinitely deep. We also put a threshold on the number of times the same widget may be interacted with (interacting with the same widget more than once may be required to completely explore the states that this widget leads to).

1.6.2.4 Window Equivalence

When exploring an application, a window is often invoked several times with different parameters. For example, consider an address book application. One window displays a list of contacts. When a contact is selected, an "edit contact" window is opened. On selecting different contacts, the resulting window will be similar, but not identical. Similar windows often correspond to the same application functionality and underlying code. Playground reduces the search space by annotating such equivalent windows.

Playground uses window equivalence heuristics to determine if the current window state is sufficiently similar to a previously visited window state. For our Android implementation, we leverage the correspondence between activity components and window design. That is, our heuristic classifies all windows belonging to the same activity component as equivalent. GUI Ripper [23] also used window titles to determine window equivalence.

1.6.2.5 Context Determination

As previously discussed, applications often have text fields that must be filled with appropriate values to lead them to the right states. Playground searches for keywords in the hints and the widget IDs (developers often tend to give descriptive IDs to widgets which often convey the purpose of those widgets) associated with editable text boxes and in the visible text labels next to them. For example, the string "Email" may appear immediately to the left of a text box, indicating that it should be filled in with an Email address.

Determining the keyword rules requires empirical investigation. We analyzed the string resources of over 500 Android applications to determine which strings application developers use for particular fields. To do this, we first extracted all of the strings an application's string resource file. We then converted the strings into a canonical form (lowercase, de-hyphenated). Next, we sorted the strings of all applications by frequency. The result was used to manually classify the strings into various semantic buckets, e.g. email, name, and phone. Finally we coded keyword based rules for each semantic bucket. Our final specification included rules

for email, address, date, phone number, password, username, cancel, and ok, among several others. The approach of automatically filling in text fields has also been used for web form completion [14, 29]. These techniques are more sophisticated and include self-learning. We plan to integrate these techniques into Playground.

Our strategy for addressing account sign-up and sign-in follows from the keyword rules approach for context determination. Sometimes, an application requiring sign-in will also include a window to sign-up for the service. The sign-up window will contain text input fields for Email, username, and password. By identifying these fields, Playground can automatically sign up for an account if a sign up option is available from within the app. Currently, Playground always uses the same Email address, username, and password; subsequent tests of an application will automatically sign in by filling in the same credentials. In future, Playground may also be able to identify if it could not successfully log in. A human tester can then create an account which Playground can use to automatically test at least future versions of the application.

1.7 Static Analysis Guide

Any dynamic analysis environment can benefit greatly from a preliminary static analysis. Playground incorporates a light-weight static analysis component, which identifies the permissions and the APIs used in the app, the libraries (ad and analytics, and social networking, for example). Such information helps Playground do the following:

- Rank the applications according to the associated risks. Applications that have sensitive permissions such as those for sending or receiving text messages, or the use of APIs for dynamic code loading may be easily identified with our static analysis. These applications, in general, carry more risk of being malicious or having unwanted, grayware functionality and hence can be prioritized by Playground for analysis. This functionality is largely similar to that provided by other systems such as MAST [6].
- Triggering events and exercising the sensitive APIs. As discussed in Section 1.6.1, if an application has permissions for and registers event receivers for system events, like text message received, or boot completed, we try to generate such events to trigger application behavior on these events.

In future, a heavier static analysis may be used to prove non-trivial properties about applications, such as the absence of privacy leaks and so on, so that functionality of some of the dynamic analysis components need not be exercised.

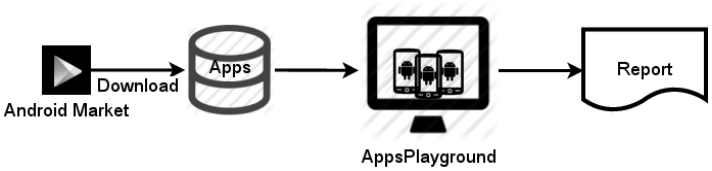


Fig. 1.5 Overview of the platform used for findings and evaluation in Sections 1.9 and 1.10. Several free apps were collected from the official Android Market and analyzed with Playground. Playground itself, as described earlier in Section 1.8, is implemented using the Android emulator, many instances of which execute concurrently on the analysis server.

1.8 Implementation

We have implemented the Playground analysis framework. The implementation is done over the standard Android emulator that comes with the Android SDK. We modify the Android source code to integrate TaintDroid and to insert hooks for API level monitoring. Kernel modifications are made to provide kernel-level monitoring. Furthermore, disguise measures are implemented by changing the appropriate identifiers and data, either directly in the Android source code or by adding files on the disk images and changing the content of the standard databases (such as contacts). Minor changes were required to the Android source for doing event triggering and fuzz testing. Intelligent execution interfaces with the window manager in Android to retrieve window and widget properties from the system. We use the ViewServer/HierarchyViewer for the interface. Changes are made to the code of many standard widgets so that required widget properties may be retrieved. We further modified related code to make retrieval of properties faster than in the original code.

Apart from the guest (Android) side, Playground also has a host side, written in over 3,000 lines of Java code. The host side implements the algorithms for intelligent execution, and also handles the dispatch of apps to multiple emulators for parallel testing and the logging of information received from the detection techniques running inside the emulator.

1.9 Findings

To show the effectiveness of Playground, we conduct some small-scale and a large-scale experiment. Our first experiment tries to automatically derive the results obtained in the TaintDroid paper. The second experiment is conducted on a set of 3,968 apps downloaded from the Android Market in November 2010. Finally, we also test Playground on real, known malware to evaluate the effectiveness of Playground at detecting malware.

For taint tracing in our experiments, we tracked device identifiers and location information leaks. By device identifiers we mean any strings that may be used to identify a particular device. Android ID is an identifier on Android available to any app without requesting any special permission. IMEI is an identifier available on all GSM phones. IMSI is associated with the SIM card and identifies a user on the cellular network. The ICC-ID is also specific to a SIM card. Access to IMEI, IMSI, ICC-ID, and WiFi Mac address requires special permissions.

Table 1.1 Private Information Leaks Detected

Number of applications	3968
Information type	Number of applications leaking
GPS	212
Android ID (AID)	581
IMEI	329
IMSI	91
Phone number	63
ICC-ID	3
WiFi MAC address	4
All types	946
At least one ID	844
At least one non-AID ID	442
GPS with at least one ID	120

1.9.1 Small-Scale Validation

To validate the effectiveness of Playground in helping discover privacy leaks, we used Playground to drive the same set of applications as that studied in the original TaintDroid paper. The TaintDroid researchers had to manually explore the applications but we attempt to achieve the same detection automatically here. Out of thirty total applications, we had to exclude nine because they were now obsolete and non functional or would not run properly on the Android emulator. Of the rest we were able to reproduce the exact findings from the manual tests conducted by the TaintDroid authors except in two cases (Wisdom Quotes Lite, Traffic Jam) where location leaks were not detected. In one other case (Babble) however, we detected an additional location leak which was not found in the original TaintDroid experiments. Such discrepancies are possible due to non deterministic behavior of applications which has been witnessed by others also [13]. Moreover, we also detected several leaks of Android ID which was not being tracked in the TaintDroid paper. This experiment thus conclusively establishes the effectiveness of Playground at automatically detecting privacy leaks.

Table 1.2 Most common leaking domains. The percentages indicate the proportion of apps which leak the corresponding information.

	# apps	# creators	Android_id	IMEI	IMSI	Phone#	Location
data.flurry.com	265	180	98.1%	2.2%	0	0	14.0%
mobclix.com	152	71	95.4%	68.4%	0	0	12.5%
Google domains	63	58	0	0	0	0	96.8%
localwireless.com	58	1	0	0	100%	0	24.1%
admob.com	51	27	0	0	0	0	90.1%
ad.qwapi.com	45	26	97.8%	2.2%	0	0	13.3%
playgamesite.com	29	2	0	100%	0	0	0
ade.wooboo.com.cn	21	8	100%	0	0	100%	4.7%

1.9.2 Large Scale Measurements

We used Playground to drive 3,968 applications. Our findings are summarized in Table 1.1. We detected 946 applications to be leaking information to Internet, which is 23.8% of total number of applications we evaluate. This is because many free applications likely include third party ads and/or analytics libraries which track unique users based on these identifiers. Among the identifiers, Android ID is the one with least risk, as it can be changed at any time. Other identifiers are permanently associated with either the device or the SIM card. We find that in 52.3% of applications leaking an identifier, there is at least one non Android ID identifier. In 56.7% of instances of location leaks, both an ID and the location information is leaked out. In these cases, the applications can uniquely track the location history of the users. We also found 63 phone number leaks. Since phone numbers are often found on social networking profiles, the privacy implications of tracking are more significant than those of other identifiers.

Analysis of Results: We would like to know the final destinations of information leaks; if the leaks are to advertisement/analytics networks or to developer's own servers. Usually, the applications from a single creator (we obtained the creator information from the Android Market) may share the same set of servers. If applications from multiple creators leak the information to a single destination domain, it is most likely the domain belongs an advertisement/analytics network, or a domain related to third-party libraries used by the applications. We found a total of 392 unique domains. Of these 29 domains relate to at least two creators. These are more likely to be advertisement/analytics networks. The rest of the domains come from single creators and hence are very likely to be domains used by the developers.

In Table 1.2, we show the domains that are related to a large number of unique applications. We also show what information has been leaked to this domain. For example, we find in 98.1% of leaks to data.flurry.com, the Android ID has been leaked. We find most of these are advertisement/analytics networks. localwireless.com and playgamesite.com are however developer sites. We note that AdMob is known to track users on the basis of hashed device identifiers. TaintDroid does not propagate taint through cryptographic hash functions and hence it appears, that none of the identifiers were sent to AdMob.

1.9.3 Analyses on Known Malware

We also analyzed known malware to confirm that Playground is able to detect malware in the wild. We considered three malware samples, FakePlayer, DroidDream, and DroidKungfu. The first one is an SMS trojan that sends SMS messages to premium numbers. The other two are root exploits. Detailed information about the samples may be found in Table 1.3. Following is our experience of analyzing these malware samples with Playground.

Table 1.3 Malware samples used for testing anti-malware tools

Family	Package name	SHA-1 code	Found	Remarks
Fakeplayer	org.me.android-application1	`1e993b0632d5bc6f0741 0ee31e41dd316435d997`	08/2010	SMS trojan
DroidDream	com.droiddream.bowlingtime	`72adcf43e5f945ca9f72 064b81dc0062007f0fbf`	03/2011	Root exploit
DroidKungFu	com.sansec	`4bf050f089a0d44d6865 ff74b75cb7f1706fdcaa`	05/2011	Root exploit

FakePlayer. This malware sample installs as a movie player. On starting the application, the an activity came up momentarily and then closed. On checking the logging done by Playground, we found that this app had sent three text messages to short numbers 3353, 3354, and 3353 in sequence. Each message contained text "798657". The SMS destinations being short would make it highly suspicious that this sample is malware.

DroidDream. On starting the application inside Playground, we did not experience anything suspicious; rather the app crashed. On disassembling the app's code and examining it, it turned out that the app would get stuck on the "phoning home" behavior. Apparently, it tries to connect to a remote server sending private information about the phone, including IMEI and IMSI numbers, but failed when we tested because the remote server did not respond. We removed this "phoning home" behavior (which is a single method call with the name of `postUrl()`), and tested the modified app again. It turns out that this time app did execute the rageagainstthecage exploit. We could see several running processes with this app's UID and finally could also see a root process; the privilege escalation had completed. Next, we checked the logs collected by Playground. The logs showed a huge number of forks and exceeding of a threshold number of processes. The logs thus give sufficient evidence of the rageagainstthecage attack having being attempted.

DroidKungFu. On launching this app inside Playground, the only thing we observed was the "phoning home" behavior, which is quite well documented. The app sent the IMEI, Android version, and phone model out of the phone. While IMEI was explicitly marked to be taint-traced; the Android version and phone model appeared as plain text in the logs as being sent out of the phone. We however did not observe any attempt to gain root privileges. On looking deeper into the code, we

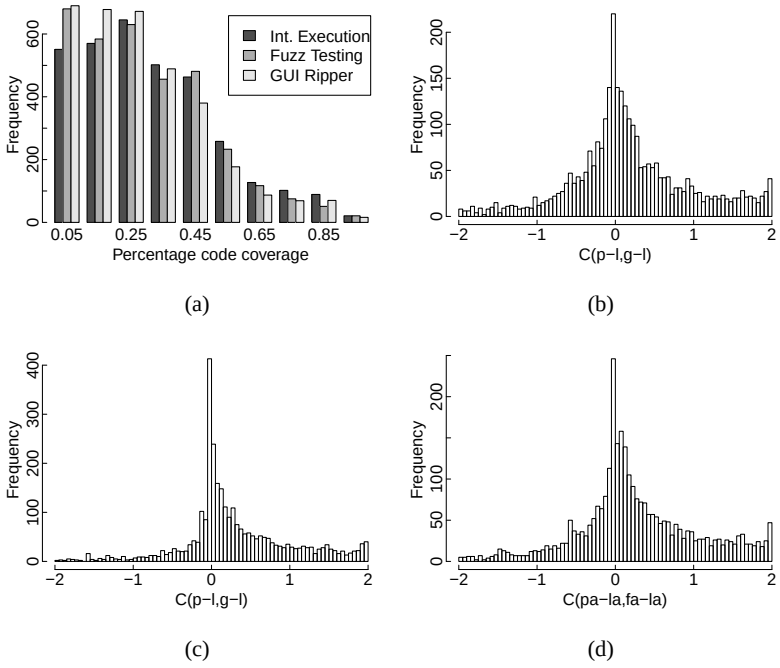


Fig. 1.6 Comparison of Playground and other approaches. (*a*) Percentage code coverage obtained, (*b*) percentage difference in code coverage between intelligent execution and fuzz testing, (*c*) percentage difference in code coverage between intelligent execution and GUI Ripper, and (*d*) percentage difference in code coverage between intelligent execution and fuzz testing, not considering ad library code.

found that the root exploits were not executed due to some condition checks, which looked for the existence of `/system/xbin/su` and some version checks. Changing either the analysis environment or the app code would allow us to see the attacks being executed. This is a general problem in dynamic analysis that sometimes the environment conditions may not match. Symbolic execution may be of help here.

1.10 Effectiveness of Automatic Exploration

In this section we evaluate and discuss the effectiveness of automatic exploration. To evaluate the effectiveness of Playground, we compare the UI exploration of Playground with GUI Ripper [23]. We use code coverage as the evaluation metric.

We would like to evaluate the effectiveness of different application driving approaches. The metric we decide to use byte code instruction level code coverage. It measures the percentage of instructions actually executed during an application run. To measure this metric, we augmented the Dalvik VM to report the number of

instructions executed. At any point of time when the application is running, we can query the VM to generate an instruction execution report by sending a signal to the corresponding process. These reports may later be processed to compare executions using different approaches as well as to generate the total number of instructions executed. We also extract the total number of instructions in the application's executable offline. The ratio of bytecode level instructions executed to the number of instructions in the application's executable gives the percentage code coverage.

We provide a comparison of three techniques, fuzz testing, intelligent execution, and GUI Ripper. The first two are integrated into Playground while the third is a previous work for exploring the application GUI in a black-box manner. We used Android port of the latter tool for our comparison.

As can be seen in Figure 1.6 (*a*) we obtained low percentage code coverage using any of the automated approaches. The mean is at about 30% for intelligent execution and for other approaches they are a little lower. The low coverage is expected because both the systems treat the application as a black-box. In fact, low coverage is one of the most limiting factors in dynamic analysis. It is also true that many applications may not give close-to-100% coverage. There may be several reasons for this. Applications may have dead code or code which executes only under special circumstances such as special device configurations and so on.

1.10.0.1 Comparison with Fuzz Testing

Fuzz testing is a common approach used to explore the program states for software testing and bug finding by supplying random inputs. Whereas intelligent execution explores program states systematically by building a state machine model of the application, fuzz testing tries to cover code by supplying random input. To compare with intelligent execution on an application, we gave fuzz testing as much time as had been taken by intelligent execution to test that application. We measured code coverage for both the approaches.

Since the total number of instructions is very large, the difference between the two approaches is not highlighted properly. To bring out the difference between any two approaches, we use percentage difference, $C(x,y) = 2(x-y)/(x+y)$. We take a ratio of the difference to the average. Using average in the denominator is a standard technique when none of x or y is a clear reference. Let p be the number of instructions executed by intelligent execution and f be the number of instructions executed by fuzz testing. Also, let l be the number of instructions executed when the application is launched, i.e., without exercising any UI on the application. We measure $C(p-l, f-l)$ and plot it in Figure 6(b). We find that intelligent execution performs better by about 17% in mean. Moreover, the third quartile lies at 54% while the first quartile is at -20%. This implies intelligent execution performs much better on the third quartile than it does worse in the first quartile. We note that code coverage only loosely represents the effectiveness of an approach. Some of the code may execute non deterministically and hence coverage over different runs may vary.

As can be seen in Figure 6(b), it is important to note that both intelligent execution and fuzz testing, as used together in Playground, complement each other. This will be discussed further later on in this section.

1.10.0.2 Comparison with GUI Ripper

To compare Playground's intelligent execution with the algorithm provided by GUI Ripper [23], we wrote an implementation for Android. Using C, p, l as defined earlier and defining g to be the number of instructions executed by this implementation, we plot $C(p-l, g-l)$ in Figure 6(c). Our results show that intelligent execution offers an improvement of nearly 31% in mean and 13% in median. The GUI Ripper algorithm treats window GUI as static; it cannot change during the course of exercising some parts of the GUI. The algorithm also has no capacity to repeatedly act on widgets and reopen windows that have been closed so that it often fails to explore windows completely. Therefore, we see this improvement.

1.10.1 Discussion

While event triggering is undoubtedly needed, it was not clear to us before the experiments how fuzz testing and intelligent execution would help and compare with each other. First, we found that the code coverage at simply launching the applications is only 16% while our automatic exploration techniques of fuzz testing and intelligent execution nearly double the code coverage. Second, intelligent execution cannot work in cases that it does not model; this applies to all the custom-made widgets and, in the current implementation, to web-based GUI, which may also be embedded in apps and which is not handled currently (the process would be similar to handling normal GUI but in a different environment). In such cases, fuzz testing was found help, filling up the limitations of intelligent execution. Such reasons are the primary explanation for the observations made from Figure 6(b).

In Figure 6(d) we show the percentage difference in code coverage between Playground and fuzz testing after removing the code contributed by ad and analytics libraries. We find that Playground offers 25% mean improvement over fuzz testing in covering application code only (with popular ad and analytics libraries excluded). The difference between Playground and GUI Ripper stays almost the same on excluding these libraries. Such improvements in results confirm our hypothesis – most of these libraries include custom and web-based widgets, and so intelligent execution finds it difficult to model them. Note that these libraries are often more trusted than the applications themselves; being few in number it is possible to do a thorough manual analysis on them.

Intelligent execution was primarily useful in cases where user credentials or some meaningful information was required. In fact, for automatic login, we found that in several cases we had received emails on the email account we used for testing from

several services. Playground had automatically created accounts with these services. In particular, we found emails from 34 different services. Some of these are popular social networking, cloud and media services such as Pandora, Dropbox, Last.fm, and Kik Messenger. Most of these related to account registrations while a few were received on supplying email address alone. We note that account registration for most applications is done through web sites. Playground currently cannot work with web pages. Moreover, many account registration routines also have captcha tests. However, once registered, Playground can easily use these credentials for subsequent navigation. A few situations were also related to providing other meaningful inputs such as a city name or a zipcode. For example, the Weather.com app asks for this in the absence of consent to location data access. Exploration is quite stunted if this is not provided.

Intelligent execution is thus specially useful for complex apps, such as those for social networking. In these cases, fuzz testing is usually stuck at the beginning only due to need of login or similar things. It is however, usually after login only, that there is access to the user's databases, files, location and other sensor information.

1.11 Related Work

Dynamic Malware Analysis. Given we are trying to run applications and detect security and privacy breaches, our work naturally falls into the category traditionally known as dynamic malware analysis. For Android two works are quite comparable to our work. DroidScope [39] is a malware analysis framework for Android applications. It is however different from our work in that while we aim to detect malicious or unwanted functionality on a large scale (in thousands of apps), they aim at malware forensics, to provide accurate analysis of apps that are known to be malware. Their analysis does not provide automatic exploration and requires significant manual effort to understand the working of the malware.

Google Bouncer is a tool that screens applications uploaded to the Google Play market for malware. This tool appears to be similar to Playground in that it needs to provide automatic exploration and detection techniques. It is however a closed, proprietary tool and not much is known about it. Researchers [25, 37] have however found that it is poor at disguising techniques and many of the common identifiers may be used to identify the virtual environment.

Strider HoneyMonkey [36] loads webpages in the browser, automatically clicks dialog boxes to allow installation of any binary and then detect if it is malware. CWSandbox [38] and Botlab [15] study malware behavior in monitored environments. All the above works have little or even no interaction with the malware executables being studied. Playground however is designed to work with highly interactive applications. These applications are different from the traditional malware in that the former's execution is primarily driven by interaction.

In the area of smartphones, DroidRanger [42] performed mostly static with some dynamic analysis to identify malicious applications. Our techniques could possibly

enhance their results. Different techniques have been proposed for behavioral identification of malware (e.g. [5]). These techniques can be plugged into Playground to provide more robust identification of malware.

Driving Applications. Any dynamic program analysis approach may be classified as either a black-box or a white-box approach depending on whether it meaningfully uses the program code to do the analysis. For our automatic exploration, we decided to stick to the black-box (or a somewhat gray-box) approach which is far simpler than the white-box paradigms. Approaches like model checking [8] and symbolic and concolic execution [18, 34] would fall into the white-box space. We plan to include symbolic execution in the future in Playground.

GUI Testing. Automatic GUI testing has for long been an intriguing area in software engineering, specifically because of the complexity of event interactions that are possible. Much of the commercially available tools are directed towards capture-playback [3] or towards programmatic descriptions of input and output event sequences [1, 32]. These however do not provide completely automatic solutions to GUI testing. Our task at GUI exploration is obviously very different from what these tools can accomplish. Privacy Oracle [16] however uses capture-playback to its advantage for multiple runs along same paths on application GUI but with slightly perturbed inputs.

GUI testing is often accomplished as model based testing [2], involving coming up with a finite state machine model of the event space that the app provides and subsequent generation and execution of test cases based on that model. Given a model, automatic techniques may be used to come up with test cases [26, 24].

Memon et al. automatically deduce GUI models by exploring the GUI [23, 22]. We face a similar problem of automatically generating an abstract state machine by exploring the application UI. However, we model much more accurately window transitions without assuming a directed-acyclic-graph organization amongst windows (in Android, for example, cycles are possible). More importantly, Memon et al. do not provide abilities to fill contextual text input and do not talk about modules such as widget tracking and sequencing policies which we found crucial for black-box exploration. These advantages do show up in Section 1.10.

Zheng et al. [40] also propose a framework for automatic UI exploration of Android apps. It is a grey-box technique as some static analysis is involved. We can improve our approach by including similar static analysis to guide the dynamic exploration. However, as they also note static analysis is insufficient to analyze all aspects of the UI. Our black-box, yet sophisticated dynamic exploration techniques can help to cover such aspects.

Recently, Choi et al. [7] and Azim and Neamtiu [4] have also developed frameworks systems for automatic GUI exploration of Android applications. The techniques are largely similar to AppsPlayground, as discussed here and in a previous conference paper [30], but also introduce new techniques such as automata learning and targeted exploration. With the published results, it is not certain if these frameworks clearly outperform AppsPlayground.

UI exploration applications in mobile platforms. UI exploration is being leveraged to novel security and non-security applications in mobile devices. Automatic testing and detection of faults and bugs is one application [7, 4, 31]. SMV-Hunter used automatic UI exploration to detect SSL/TLS man-in-the-middle vulnerabilities in Android apps [35]. DECAF [20] uses automatic exploration to detect and characterize ad fraud in Windows Mobile apps. Spade [19] is another automatic exploration framework for Windows Mobile apps. The authors have used it for multiple applications such as app keyword indexing, ad fraud detection, and contextual ad-keyword mining.

1.12 Conclusion and Future Work

This chapter presented AppsPlayground, a tool automatic dynamic analysis of smartphone applications. We integrated a number of detection, exploration, and disguise techniques to come up with an effective analysis environment that may be used to evaluate Android applications on a large scale.

The future directions for Playground include including symbolic execution for systematic exploration of the applications' state space and to make Playground even more stealthy by enhancing the disguise techniques.

Acknowledgements We would like to thank Zhichun Li for his extensive comments through the major part of this project. We are grateful to Patrick Traynor for his helpful comments.

References

1. Abbot: `http://abbot.sourceforge.net/`
2. Apfelbaum, L., Doyle, J.: Model Based Testing. In: Software Quality Week Conference, pp. 296–300 (1997)
3. AutoIt: `http://www.autoitscript.com/site/autoit/`
4. Azim, T., Neamtiu, I.: Targeted and depth-first exploration for systematic testing of android apps. In: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, pp. 641–660. ACM, Indianapolis, Indiana, USA (2013). DOI 10.1145/2509136.2509549
5. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, pp. 15–26. ACM, Chicago, Illinois, USA (2011). DOI 10.1145/2046614.2046619
6. Chakradeo, S., Reaves, B., Traynor, P., Enck, W.: Mast: triage for market-scale mobile malware analysis. In: Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks, pp. 13–24. ACM, Budapest, Hungary (2013). DOI 10.1145/2462096.2462100
7. Choi, W., Necula, G., Sen, K.: Guided gui testing of android apps with minimal restart and approximate learning. In: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, pp. 623–640. ACM, Indianapolis, Indiana, USA (2013). DOI 10.1145/2509136.2509552

8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
9. Egele, M., Kruegel, C., Kirda, E., Vigna, G.: PiOS: Detecting Privacy Leaks in iOS Applications. In: ISOC Network and Distributed System Security Symposium (NDSS). San Diego, California, USA (2011)
10. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Vancouver, BC (2010)
11. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: Proceedings of the 20th USENIX Security Symposium. San Francisco, CA (2011)
12. Fuchs, A., Chaudhuri, A., Foster, J.: Scandroid: Automated security certification of android applications. Manuscript, Univ. of Maryland, `http://www.cs.umd.edu/~avik/projects/scandroidascaa` (2009)
13. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: "These aren't the Droids you're looking for": Retrofitting Android to protect data from imperious applications. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011). Chicago, IL, USA (2011). DOI 10.1145/2046707.2046780
14. Huang, Y., Huang, S., Lin, T., Tsai, C.: Web application security assessment by fault injection and behavior monitoring. In: Proceedings of the 12th international conference on World Wide Web, pp. 148–159. Budapest, Hungary (2003). DOI 10.1145/775152.775174
15. John, J.P., Moshchuk, A., Gribble, S.D., Krishnamurthy, A.: Studying spamming botnets using Botlab. In: Proceedings of the 6th USENIX symposium on Networked systems design and implementation, pp. 291–306. USENIX Association, Boston, Massachusetts (2009)
16. Jung, J., Sheth, A., Greenstein, B., Wetherall, D., Maganis, G., Kohno, T.: Privacy oracle: a system for finding application leaks with black box differential testing. In: CCS '08: Proceedings of the 15th ACM conference on Computer and communications security, pp. 279–288. ACM, Alexandria, Virginia, USA (2008). DOI 10.1145/1455770.1455806
17. Kasperskey Lab: First SMS Trojan detected for smartphones running Android. `http://www.kaspersky.com/news?id=207576158` (2010)
18. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). DOI 10.1145/360248.360252
19. Lin, F.X., Ravindranath, L., Nath, S., Liu, J.: Spade: Scalable app digging with binary instrumentation and automated execution
20. Liu, B., Nath, S., Govindan, R., Liu, J.: Decaf: Detecting and characterizing ad fraud in mobile apps. Tech. rep., Technical Report 13-937, University of Southern California (2013)
21. Lookout: Update: Security Alert: DroidDream Malware Found in Official Android Market. `http://blog.mylookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/`
22. Memon, A.: An event-flow model of gui-based applications for testing. Software Testing, Verification and Reliability **17**(3), 137–157 (2007). DOI 10.1002/stvr.v17:3
23. Memon, A., Banerjee, I., Nagarajan, A.: GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. Reverse Engineering, Working Conference on pp. 260+ (2003). DOI 10.1109/WCRE.2003.1287256
24. Memon, A.M., Pollack, M.E., Soffa, M.L.: Hierarchical GUI test case generation using automated planning. IEEE Transactions on Software Engineering **27**(2), 144–155 (2001). DOI 10.1109/32.908959
25. Oberheide, J.: Dissecting android's bouncer (2012). `https://blog.duosecurity.com/2012/06/dissecting-androids-bouncer/`
26. Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S.: Model-based testing for real. International Journal on Software Tools for Technology Transfer (STTT) **5**(2), 140–157 (2004). DOI 10.1007/s10009-003-0128-3
27. Qemu. `http://www.qemu.org`
28. Raffetseder, T., Kruegel, C., Kirda, E.: Detecting system emulators pp. 1–18 (2007)
29. Raghavan, S., Garcia-Molina, H.: Crawling the hidden web. In: Proceedings of the International Conference on Very Large Data Bases, pp. 129–138. Roma, Italy (2001)

30. Rastogi, V., Chen, Y., Enck, W.: Appsplayground: automatic security analysis of smartphone applications. In: Proceedings of the third ACM conference on Data and application security and privacy, pp. 209–220. ACM, San Antonio, Texas, USA (2013). DOI 10.1145/2435349.2435379
31. Ravindranath, L., Nath, S., Padhye, J., Balakrishnan, H.: Automatic and scalable fault detection for mobile applications
32. Robotium: `http://code.google.com/p/robotium/`
33. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: Security and Privacy (SP), 2010 IEEE Symposium on, pp. 513–528. IEEE, Oakland, CA, USA (2010). DOI 10.1109/SP.2010.38
34. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. SIGSOFT Softw. Eng. Notes **30**(5), 263–272 (2005). DOI 10.1145/1095430.1081750
35. Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., Khan, L.: Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In: Proceedings of the 19th Network and Distributed System Security Symposium. San Diego, California, USA (2014)
36. Wang, Y.M., Beck, D., Jiang, X., Roussev, R.: Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In: ISOC Network and Distributed System Security Symposium (NDSS). San Diego, California, USA (2006)
37. Whitwam, R.: Circumventing google's bouncer, android's anti-malware system (2012). http://www.extremetech.com/computing/130424-circumventing-googles-bouncer-androids-anti-malware-system
38. Willems, C., Holz, T., Freiling, F.: Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE Security and Privacy **5**(2), 32–39 (2007). DOI 10.1109/MSP.2007.45
39. Yan, L.K., Yin, H.: DroidScope: Seamlessly Reconstructing the OS and Dalvik. In: Proceedings of USENIX Security Symposium. USENIX Association, Bellevue, WA, USA (2012)
40. Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., Zou, W.: Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In: Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices, pp. 93–104. ACM, Raleigh, North Carolina, USA (2012). DOI 10.1145/2381934.2381950
41. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. Security and Privacy, IEEE Symposium on (2012). DOI 10.1109/SP.2012.16
42. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In: Proceedings of the 19th Network and Distributed System Security Symposium, NDSS '12. San Diego, California, USA (2012)