Brahmastra: Driving Apps to Test the Security of Third-Party Components

Ravi Bhoraskar[1,2], *Seungyeop Han*[2], *Jinseong Jeon*[3], *Tanzirul Azim*[4],
Shuo Chen[1], *Jaeyeon Jung*[1], *Suman Nath*[1], *Rui Wang*[1], *David Wetherall*[2]

[1] *Microsoft Research*
[2] *University of Washington*
[3] *University of Maryland, College Park*
[4] *University of California, Riverside*

Abstract

We present an app automation tool called Brahmastra for helping app stores and security researchers to test third-party components in mobile apps at runtime. The main challenge is that call sites that invoke third-party code may be deeply embedded in the app, beyond the reach of traditional GUI testing tools. Our approach uses static analysis to construct a page transition graph and discover execution paths to invoke third-party code. We then perform binary rewriting to "jump start" the third-party code by following the execution path, efficiently pruning out undesired executions. Compared with the state-of-the-art GUI testing tools, Brahmastra is able to successfully analyse third-party code in 2.7× more apps and decrease test duration by a factor of 7. We use Brahmastra to uncover interesting results for two use cases: 175 out of 220 children's apps we tested display ads that point to web pages that attempt to collect personal information, which is a potential violation of the Children's Online Privacy Protection Act (COPPA); and 13 of the 200 apps with the Facebook SDK that we tested are vulnerable to a known access token attack.

1 Introduction

Third-party libraries provide a convenient way for mobile application developers to integrate external services in the application code base. Advertising that is widely featured in "free" applications is one example: 95% of 114,000 popular Android applications contain at least one known advertisement library according to a recent study [22]. Social media add-ons that streamline or enrich the user experience are another popular family of third-party components. For example, `Facebook Login` lets applications authenticate users with their existing Facebook credentials, and post content to their feed.

Despite this benefit, the use of third-party components is not without risk: if there are bugs in the library or the way it is used then the host application as a whole becomes vulnerable. This vulnerability occurs because the library and application run with the same privileges and without isolation under existing mobile application models. This behavior is especially problematic because a number of third-party libraries are widely used by many applications; any vulnerability in these libraries can impact a large number of applications. Indeed, our interest in this topic grew after learning that popular SDKs provided by Facebook and Microsoft for authentication were prone to misuse by applications [30], and that applications often make improper use of Android cryptography libraries [20].

In this paper, we present our solution to the problem of *third-party component integration testing at scale*, in which one party wishes to test a large number of applications using the same third-party component for a potential vulnerability. To be useful in the context of mobile app stores, we require that a successful solution test many applications without human involvement. Observe that it is *not* sufficient to simply test the third-party library for bugs in isolation. This is because vulnerabilities often manifest themselves due to the interaction of the application and the third-party component. Thus our focus is to develop tools that enable testers to observe *in situ* interactions between the third-party component and remote services in the context of a specific application at runtime.

We began our research by exploring automated runtime analysis tools that drive mobile UIs (e.g., [5, 23, 26]) to exercise the third-party component, but quickly found this approach to be insufficient. Although these tools are effective at executing *many* different code paths, they are often unable to reach *specific* interactions deep within the applications for a number of reasons that we explore within this paper. Instead, our approach leverages the structure of the app to improve test hit rate and execution speed. To do this, we characterize an app by statically building a graph of its pages and transitions between them. We then use path information from the graph to guide the runtime execution towards the third-

party component under test. Rather than relying on GUI manipulation (which requires page layout analysis) we rewrite the application under test to directly invoke the callback functions that trigger the desired page transitions.

We built Brahmastra to implement our approach for Android apps. Our tool statically determines short execution paths, and dynamically tests them to find one that correctly invokes a target method in the third-party library. At this stage, behavior that is specific to the library is checked. Because our techniques do not require human involvement, Brahmastra scales to analyze a large number of applications. To show the benefits of our approach, we use our tool for two new studies that contribute results to the literature: 1) checking whether children's apps that source advertisements from a third-party comply with COPPA privacy regulations; and 2) checking that apps which integrate the Facebook SDK do not have a known security vulnerability [30].

From our analysis of advertisements displayed in 220 kids apps that use two popular ad providers, we find that 36% apps have displayed ads whose content is deemed inappropriate for kids—such as offering free prizes, or displaying sexual imagery. We also discover that 80% apps have displayed ads with landing pages that attempt to collect personal information from the users, such as name, address, and online contact information—which can be a violation of the Children's Online Privacy Protection Act [6]. Apart from creating an unsafe environment for kids, this also leaves the app developers vulnerable to prosecution, since they are considered liable for all content displayed by their app.

For our analysis of a vulnerability in third party login libraries, we run a test case proposed by Wang et al. [30] against 200 Android apps that bundle Facebook SDK. We find that 13 of the examined apps are vulnerable.

Contributions: We make two main contributions. The first is Brahmastra, which embodies our hybrid approach of static and dynamic analysis to solve the third-party component integration testing problem for Android apps. We discuss our approach and key techniques in §4 and their implementation in §5. We show in §6 that our techniques work for a large fraction of apps while existing tools such as randomized testing (Monkey) often fail. We have made the static analysis part of Brahmastra available at `https://github.com/plum-umd/redexer`.

Our second contribution is an empirical study of two security and privacy issues for popular third-party components. We find potential violations of child-safety laws by ads displayed in kids apps as discussed in §7; several apps used in the wild display content in potential violation of COPPA due to the behavior of embedded components. We find that several popular Android apps are vulnerable to the Facebook access token attack as discussed in §8; A Facebook security team responded immediately to our findings on 2/27/2014 and had contacted the affected developers with the instructions to fix.

2 Background

As our system is developed in the context of Android, we begin by describing the structure of Android apps and support for runtime testing.

Android app structure: An Android app is organized as a set of pages (e.g., Figure 1) that users can interact with and navigate between. In Android, each page is represented by an `activity` object. Each activity class represents one kind of page and may be initialized with different data, resulting in different `activity instances`. We use the terms page and activity instance interchangeably. Each page contains various GUI elements (e.g., buttons, lists, and images), known as `views`. A view can be associated with a callback function that is invoked when a user interacts with the view. The callback function can instantiate a new activity by using a late binding mechanism called `intent`. An intent encapsulates the description of a desired action (e.g., start a target activity) and associated parameters. The main `activity` (or the first page) of an app, defined in its manifest file, is started by the application launcher by passing a `START` intent to it.

For example, in Figure 1, clicking the "Done" button on activity `A1` invokes its event handler, which calls a callback function defined by the app developer. The callback constructs an intent to start activity `A2` with necessary parameter `P12`. The `Activity Manager` then constructs an instance of `A2`, and starts it with `P12` as parameters. We refer to the documentation of Android internals for more details [2].

Automated dynamic analysis: Recent works have used a class of automation tools, commonly called a Monkey, that, given a mobile app binary, can automatically execute it and navigate to various parts (i.e., states) of the app. Examples include PUMA [23], DECAF [25], AppsPlayground [26], A3E [14], and VanarSena [27]. A Monkey launches the app in a phone or an emulator, interacts with it by emulating user interactions (e.g., clicking a button or swiping a page) to recursively visit various pages, and performs specific tasks (e.g., checking ad frauds in the page or injecting faults) on each page.

In Figure 1, a Monkey may be able to visit the sequence of states `A1` $\rightarrow$ `A2` $\rightarrow$ `A3`$\rightarrow$ `A4` if it knows the right UI actions (e.g., type in mother's name and select "Due Date" in `A1`) to trigger each transition. However, if Monkey clicks a button in `A3` other than "Account", the app would navigate to a different activity. If the goal of testing is to invoke specific methods (e.g., Facebook login as shown in the example), then without knowing the structure of the app, a Monkey is likely to wander around

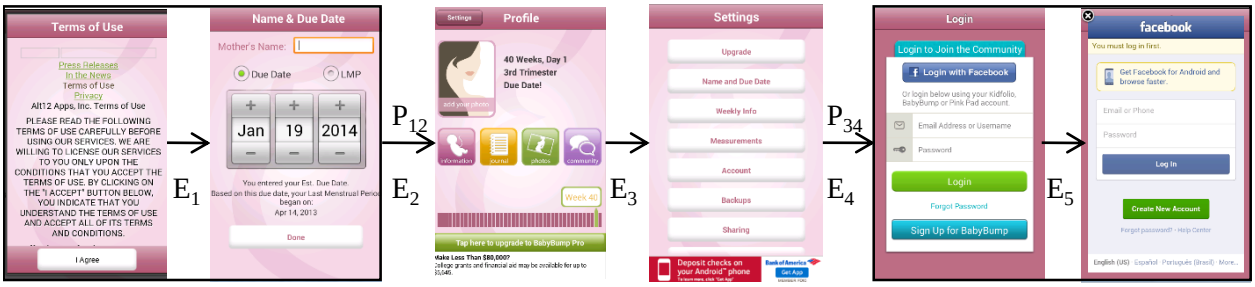

Figure 1: Activity sequences of `com.alt12.babybumpfree` that invoke Facebook single sign-on window in the forth activity (`A4`): Clicking "I Agree" (`E1`) then clicking "Done" (`E2`) opens up `A2` with the parameter, `fromLoader : tru` (`P12`). Clicking "Settings" (`E3`) in `A2` opens up the settings activity, `A3` and then clicking "Account" (`E4`) opens up the login activity, `A4` with the parameter, `WHICHCLASS : com.alt12.babybumpcore.activity.settings.Settings`. Finally, clicking "Login with Facebook" (`E5`) opens up the sign-on window within the same activity, `A4`.

many activities until it reaches `A4`, if it ever does.

3 Problem and Insights

Our goal is to develop the ability to automatically and systematically test a large set of mobile apps that embed a specific third-party component for a potential vulnerability associated with the use of that component. This ability would let app store operators rapidly vet apps to contain security vulnerabilities caused by popular libraries. It would let component developers check how apps use or misuse their interfaces. It would also let security researchers such as ourselves empirically assess vulnerabilities related to third-party libraries.

A straightforward approach is to use existing Monkeys. Unfortunately, this approach does not work well: it often fails to exercise the target third-party component of the app under test. Although recent works propose techniques to improve various types of *coverages*, computed as the fraction of app activities or methods invoked by the Monkey, coverage still remains far from perfect [26, 14, 13, 25, 27]. Moreover, in contrast to traditional coverage metrics, our success metric is binary for a given app indicating whether the target third-party component (or a target method in it) is invoked (i.e., *hit*) or not (i.e., *miss*). Our experiments show that even a Monkey with a good coverage can have a poor hit rate for a target third-party component that may be embedded deep inside the app. We used an existing Monkey, PUMA that reports a $> 90\%$ activity coverage compared to humans [23], but in our experiments it was able to invoke a target third-party component only in 13% of the apps we tested (see §6 for more details). On a close examination, we discovered several reasons for this poor hit rate of existing Monkeys:

R1. Timeout: A Monkey can exhaust its time budget before reaching the target pages due to its trial-and-error search of the application, especially for apps with many pages that "blow up" quickly.

R2. Human inputs: A Monkey is unable to visit pages that are reached after entering human inputs such as login/password, or gestures beyond simple clicks that the automated tester cannot produce.

R3. Unidentified elements: A Monkey fails to explore clickable UI elements that are not visible in the current screen (e.g., hidden in an unselected tab) or are not activated yet (e.g., a "Like" button that is activated only after the user registers to the app) or are not identified by underlying UI automation framework (e.g., nonstandard custom control).

R4. Crashes: By stressing the UI, a Monkey exacerbates app crashes (due to bugs and external dependencies such as the network) that limit exploration.

Note that, unlike existing Monkeys, our goal is not to exhaustively execute all the possible code paths but to execute *particular code paths* to invoke methods of interest in the third-party library. Therefore, our insight is to improve coverage by leveraging ways how third party components are integrated with application code base. These components are incorporated into an app at the activity level. Even if the same activity is instantiated multiple times with different contents, third-party components typically behave in the same way in all those instantiations. This allows us to restrict our analysis at the level of *activity* rather than *activity instances*. Further, even if an app contains a large number of activities, only a small number of them may actually contain the third-party component of interest. Invoking that compo-

nent requires successfully executing any and only one of those activities.

Using this insight, our testing system, Brahmastra, uses three techniques described below to significantly boost test hit rate and speed compared to a Monkey that tries to reach all pages of the app.

Static path pruning: Brahmastra considers only the "useful" paths that eventually invoke the target third-party methods and ignores all other "useless" paths. In Figure 1, Brahmastra considers the execution path `A1` → `A2` → `A3`→ `A4` for exploration and ignores many other paths that do not lead to a target activity, `A4`.

Such useful paths need to be identified statically *before* dynamic analysis is performed. The key challenges in identifying such paths by static analysis arise due to highly asynchronous nature of Android apps. We discuss the challenges and our solution in §4.

Dynamic node pruning: Brahmastra opportunistically tries to start from an activity in the middle of the path. If such "jump start" is successful, Brahmastra can ignore all preceding activities of the path. For example, in Figure 1, Brahmastra can directly start activity `A3`, which can lead to the target activity `A4`.

Dynamic node pruning poses several challenges — first, we need to enable jump-starting an arbitrary activity directly. Second, jump starting to the target activity may fail due to incorrect parameters in the intent, in which case we need to find a different activity that is close to the target, for which jump start succeeds. We discuss these in detail in next section.

Self-execution of app: Brahmastra rewrites the app binary to automatically call methods that cause activity transitions. The appropriate methods are found by static analysis. In Figure 1, instead of clicking on the button with label "Done" in `A1`, Brahmastra would invoke the `onClick`() method that would make the transition from `A1` to `A2`. The advantage over GUI-driven automation is that it can discover activity-transitioning callbacks even if they are invisible in the current screen.

In summary, our optimizations can make dynamic analysis fast by visiting only a small number of activities of an app. More importantly, they also improve the test hit rate of such analysis. Faster analysis helps to avoid any timeouts (**R1**). Dynamic node pruning can bypass activities that require human inputs (**R2**). In Figure 1, Brahmastra can jump to `A3` and bypass `A1` that requires selecting a *future* due date. Intent-driven navigation helps Brahmastra to make transitions where a Monkey fails due to unidentified GUI elements (**R3**). Finally, visiting fewer activities reduces the likelihood of crashes (**R4**). We quantitatively support these claims in §6.

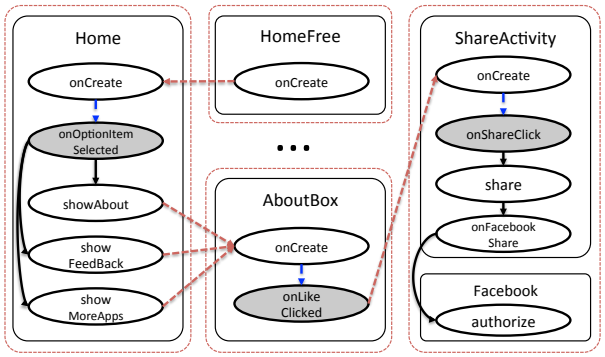

Figure 2: A simplified call graph of `ch.smalltech.battery.free` that shows multiple transition paths composed of multiple activities. Boxes and ovals represent classes and methods. Solid edges correspond to synchronous calls; (red) dotted edges indicate activity transitions; and (blue) dashed edges represent implicit calls due to user interactions. Three different paths starting from `Home.onOptionItemSelected`() reach `AboutBox.onCreate`() and share the remaining part.

4 Design

Brahmastra requires as input: a test application binary; the names of target methods to be invoked within the context of the application; and the plug-in of a specific security analysis to run once the target method is reached. Our system is composed of three parts:

1. *Execution Planner* statically analyzes the test app binary and discovers an execution path to invoke the target third-party method.
2. *Execution Engine* receives execution paths from the Planner and launches the test app in one or multiple emulators and automatically navigates through various pages according to the execution path.
3. *Runtime Analyzer* is triggered when the test app invokes the target method. It captures the test app's runtime state (e.g., page content, sensors accessed, network trace) and runs the analysis plug-in.

4.1 Execution Planner

The job of the Execution Planner is to determine: (1) the activities that invoke the target third-party method; and (2) the method-level execution paths that lead to the target activities. To accomplish these tasks, we statically analyze the app binary to construct a *call graph* that encompasses its activities and interactions that cause activity transitions.

Constructing call graph: A call graph is a graph where vertices are methods and edges are causal relationship between method invocation. More precisely, there exists

```
ImageButton b = (ImageButton)
              findViewById(R.id.b1);
b.setOnClickListener(new OnClickListener() {
 public void onClick(View v) {
   ...
 }});
```

Figure 3: Example of a programmatic handler registration. `onClick()` is bound to `setOnClickListener()`

an edge from method m_1 to m_2 if m_1 invokes m_2. Based on how m_2 is invoked by m_1, there are three types of edges: (1) synchronous edges, if m_1 directly calls m_2, (2) asynchronous edges, if m_1 invokes m_2 asynchronously, and (3) activity transition edges, if m_1 starts an activity that automatically calls m_2. Figure 2 depicts a call graph of one real app.

While synchronous edges can be identified easily by scanning the app binary code, discovering other edges can be difficult. To find activity transition edges, we rely on the fact that one activity can start another activity by generating an intent and passing it to the `startActivity()` method. We perform constant propagation analysis [12] so as to track such intent creations and detect activity transitions. We also conduct class hierarchy analysis [19] to conservatively determine possible receiver types for dynamic dispatch, where the target call sites depend on the runtime types of the receivers.

To discover asynchronous edges, we need to consider all the different ways asynchronous methods can be invoked by a mobile app:

1. Programmatic handler registrations: These are callbacks explicitly bound to methods (e.g., event handler of GUI elements) within the code. Figure 3 shows an example.
2. XML-based handler registrations: These are callbacks specified in the layout or resource XML files. Figure 4 shows an example.
3. Lifetime methods: These are methods provided by the underlying framework that automatically make transitions to other methods on specific events. Examples are splash screens and message boxes that transition to next activities after a timeout or after user acknowledgment, respectively.

To discover the first and third types, we use constant propagation analysis to trace callbacks attached to various event handlers. To handle the second case, we parse layout XML files corresponding to each activity to figure out the binding between UI elements and callback methods.

Efficient call graph computation: A call graph can be extremely large, thus computing the entire call graph can

```
// layout/about_box_share.xml
<Button android:id="@id/mShareFacebook"
  style="@style/ABB_Black_ShareButton" ... />
<Button android:id="@id/mShareTwitter"
  style="@style/ABB_Black_ShareButton" ... />
// values/styles.xml
<style name="ABB_Black_ShareButton ... >
  <item name="android:onClick">onShareClick</item>
</style>
// ch.smalltech.common.feedback.ShareActivity
public void onShareClick(View v){
  // different behavior depending on argument v
}
```

Figure 4: Example of a XML-based handler registration observed from `ch.smalltech.battery.free`. Two buttons share the `onShareClick` callback. The binding between `onShareClick` and `setOnClickListener` of each button can be determined through layout and styles XML files.

be very expensive. For example, the app shown in Figure 1 declares 74 activities in the manifest; we find at least 281 callbacks over 452 registering points; and its call graph is composed of 1,732 nodes and 17,723 edges. To address this, we use two optimizations to compute a *partial* call graph that includes target methods and the start activity methods. First, we exclude system's static libraries and other third-party libraries that are not related to the target methods. Second, we search transition paths backwards on call graph. We pinpoint call sites of target methods while walking through bytecodes. We then construct a partial call graph, taking into accounts component transitions via intent and bindings between views and listeners. Finally, starting from the call sites, we build backward transition paths, until public components including the main activity are reached. If failed, partial paths collected at the last phase will be returned.

Determining target activity(s): Given the call graph and a target method, we determine the activities that invoke the method as follows. From the call graph, we can identify the activity boundaries such that all methods within the same boundary are invoked by the same activity. Since an activity can be started only through an activity transition edge in the call graph, any maximal connected component whose edges are either synchronous or asynchronous define the boundary of an activity. In Figure 2, bigger rectangles denote the activity boundaries. Given the boundaries, we identify the activities that contain the target method.

Finding activity transition paths: Reaching a target activity from the start activity may require several transitions between multiple activities. For example, in Figure 2, navigating from the start activity (`HomeFree`) to a target activity (`ShareActivity`) requires three transi-

tions. This implies that Brahmastra requires techniques for automatic activity transitions, which we describe in the next subsection. Second, a target activity may be reachable via multiple transition paths. While the shortest path is more attractive for fast exploration, the path may contain blocking activities and hence not executable by Brahmastra. Therefore, Brahmastra considers all transition paths (in increasing order of their length); if execution of a short path fails, it tries a longer one.

Given the call graph G, the Planner computes a small set P of acyclic transition paths that the Execution Engine need to consider. P includes a path if and only if it terminates at a target activity without a cycle and is not a suffix of any other path in P. This ensures that P is useful, complete (i.e., Execution Engine does not need to consider any path not in P), and compact. For instance Figure 5 shows one out of three paths contained in P.

```
     HomeFree;.onCreate
---> Home;.onCreate
-#-> Home;.onOptionsItemSelected
---> Home;.showAbout
---> AboutBox;.onCreate
-#-> AboutBox;.onLikeClicked
---> ShareActivity;.onCreate
-#-> ShareActivity;.onShareClick
---> ShareActivity;.share
---> ShareActivity;.onFacebookShare
```

Figure 5: An example path information for `ch.smalltech.battery.free` Dashed arrows stand for explicit calls or activity transition, whereas arrows with a hash tag represent implicit invocations, which are either callbacks due to user interactions or framework-driven callbacks, such as lifecycle methods.

P can be computed by breadth-first traversals in G, starting from each target activity and traversing along the reverse direction of the edges.

4.2 Execution Engine

The useful paths P produced by the Execution Planner already give an opportunity to prune exploration: Brahmastra considers only paths in P (and ignore others), and for each path, it can simply navigate through its activities from the beginning of the path (by using techniques described later). Exploration can stop as soon as a target method is invoked.

Rewriting apps for self-execution: One might use a Monkey to make activity transitions along useful paths. Since a Monkey makes such transitions by interacting with GUI elements, this requires identifying mapping between GUI elements and transitioning activities and interact with only the GUI elements that make desired transitions.

We address this limitation with a technique we develop called *self execution*. At a high level, we rewrite app binaries to insert code that automatically invokes the callbacks that trigger desired activity transitions, even if their corresponding GUI elements are not visible. Such code is inserted into all the activities in a useful path such that the rewritten app, after being launched in a phone or an emulator, would automatically make a series of activity transitions to the target activity, without any external interaction with its GUI elements.

Jump start: Brahmastra goes beyond the above optimization with a node pruning technique called "jump start". Consider a path $p = (a_0, a_1, \ldots, a_t)$, where a_t is a target activity. Since we are interested only in the target activity, success of Brahmastra is not affected by what activity a_i in p the execution starts from, as long as the last activity a_t is successfully executed. In other words, one can execute any suffix of p without affecting the hit rate. The jump start technique tries to execute a suffix — instead of the whole — useful path. This can improve Brahmastra's speed since it can skip navigating through few activities (in the prefix) of a useful path. Interestingly, this can also improve the hit rate of Brahmastra. For example, if the first activity a_0 requires human inputs such as user credentials that an automation system cannot provide, any effort to go beyond state a_0 will fail.

Note that directly executing an activity $a_i, i > 0$, without navigating to it from the start activity a_0, may fail. This is because some activities are required to be invoked with specific intent parameters. In such cases, Brahmastra tries to jump start to the previous activity a_{i-1} in the path. In other words, Brahmastra progressively tries to execute suffixes of useful paths, in increasing order of lengths, until the jump start succeeds and the target activity a_t is successfully reached or all the suffixes are tried.

Algorithm 1 shows the pseudocode of how execution with jump start works. Given the set of paths, the algorithm first generates suffixes of all the paths. Then it tries to execute the suffixes in increasing order of their length. The algorithm returns true on successful execution of any suffix. Note that Algorithm 1 needs to know if a path suffix has been successfully executed (line 9). We inject lightweight logging into the app binary to determine when and whether target methods are invoked at runtime.

4.3 Runtime Analyzer

Runtime Analyzer collects various runtime states of the test app and makes it available to custom analysis plug-ins for scenario-specific analysis. Runtime states include UI structure and content (in form of a DOM tree) of the current app page, list of system calls and sensors invoked by the current page, and network trace due to the current page. We describe two plug-ins and analysis results in

Algorithm 1 Directed Execution

```
1: INPUT: Set of useful paths P from the Planner
2: OUTPUT: Return true if execution is successful
3: S ← set of suffixes of all paths in P
4: for i from 0 to ∞ do
5:    S_i ← set of paths of length i in S
6:    if S_i is empty then
7:       return false
8:    for each path suffix p in S_i do
9:       if Execute(p) = true then
10:         return true
11:
12: return false
```

```
1 // { v3 →this }
2 new-instance v0, Landroid/content/Intent;
3 // { v0 →Intent(), ... }
4 const-class v1, ...AboutBox;
5 // { v1 →Clazz(AboutBox), ... }
6 invoke-direct {v0, v3, v1}, ...Intent;.<init>
7 // { v0 →Intent(AboutBox), ... }
8 invoke-virtual {v3, v0}, ...;.startActivity
9 // { ... }
```

Figure 6: An example bytecode of activity transition excerpted from `ch.smalltech.battery.free`. Mappings between bytecode represent data-flow information, which shows what values registers *must* have. Only modified or newly added values are shown.

later sections.

5 Implementation of Brahmastra

We implement Brahmastra for analyzing Android apps, and use the tool to perform two security analyses which we will describe in §7 and §8. This section describes several highlights of the tool, along with practical challenges that we faced in the implementation process and how we resolved them.

5.1 Execution Planner

Static analyses for constructing a call graph and finding transition paths to target methods are performed using Redexer [24], a general purpose bytecode rewriting framework for Android apps. Redexer takes as input an Android app binary (APK file) and constructs an in-memory data structure representing `DEX` file for various analyses. Redexer offers several utility functions to manipulate such `DEX` file and provides a generic engine to perform data-flow analysis, along with call graph and control-flow graph.

For special APIs that trigger activity transitions, e.g., `Context.startActivity()`, we perform constant propagation analysis (see Appendix A for details) and identify a target activity stored inside the intent. Figure 6 depicts example bytecode snippets that create and initialize an intent (lines 2 and 6), along with the target activity (line 4), and starts that activity via `startActivity()` (line 8). Mappings between each bytecode show how we accumulate data-flow information, from empty intent through class name to intent with the specific target activity. We apply the same analysis to bindings between views and listeners.

5.2 App Rewriting

We use the Soot framework [29] to perform the bytecode rewriting that enables self execution. Dexpler [7] converts an Android app binary into Soot's intermediate representation, called Jimple, which is designed to ease analysis and manipulation. The re-writing tool is composed of Soot's class visitor methods and an Android XML parser. Given an app binary and an execution path, the rewriter generates a rewritten binary which artificially invokes a callback method upon the completion of the exercising the current activity, triggering the next activity to be launched. The inserted code depends on the type of the edge found by the Planner (Recall three kinds of asynchronous edges described in §4.1). For programmatic and XML-based registrations, the rewriter finds the `view` attached to it — by parsing the activity code, and the manifest respectively — and invokes the appropriate UI interaction on it after it has completed loading. Lifetime methods are invoked by the Android framework directly, and the rewriter skips code insertion for these cases. In other cases, the rewriter inserts a timed call to the transition method directly, to allow the activity and any dependencies of the method to load completely.

5.3 Jump Start

Jump start requires starting an activity even if it is not defined as the `Launcher` activity in the app. To achieve that, we manipulate the manifest file of the Android app. The `Intent.ACTION_MAIN` entry in the manifest file declares activities that Android activity manager can start directly. To enable jump start, we insert an `ACTION_MAIN` entry for each activity along the path specified, so that it can be started by the Execution Engine. Manifest file also declares an intent filter, which determines the sources from which an activity may be started, which we modify to allow the Execution Engine to launch the activity. The Engine then invokes desired activity by passing an intent to it. We use the Android Debug Bridge (ADB) [4] for performing jump start. ADB allows us to create an intent with the desired parameters and target, and then passes it to the Android Activity Manager. The activity manager in turn loads the appropriate app data and invokes the specified activity. Starting the (jump started) activ-

ity immediately activates self execution from that activity onwards.

6 Evaluation of Brahmastra

We evaluate Brahmastra in terms of two key metrics: (1) hit rate, i.e., the fraction of apps for which Brahmastra can invoke any of target methods, and (2) speed, i.e., time (or number of activity transitions) Brahmastra takes to invoke a target method in an app for which Brahmastra has a hit. Since we are not aware of any existing tool that can achieve the same goal, we compare Brahmastra against a general Android app exploration tool called, PUMA [23]. This prototype is the best-performing Monkey we were able to find that is amenable to experimentation. In terms of speed and coverage, PUMA is far better than a basic "random" Monkey. PUMA incorporates many key optimizations in existing Monkeys such as AppsPlayground [26], A3E [14], and VanarSena [27] and we expect it to perform at least on a par with them.

6.1 Experiment Methodology

Target method: For the experiments in this section, we configure Brahmastra to invoke authentication methods in the Facebook SDK for Android.[1] We choose Facebook SDK because this is a popular SDK and its methods are often invoked only deep inside the apps. Using the public documentation for the Facebook SDK for Android, we determined that it has two target methods for testing. Note that apps in our dataset use the Facebook SDK version 3.0.2b or earlier[2]

```
Lcom/facebook/android/Facebook;->authorize
Lcom/facebook/android/Facebook;->dialog
```

Figure 7: Target methods for evaluation

Apps: We crawled 12,571 unique apps from the Google Play store from late December 2012 till early January 2013. These apps were listed as 500 most popular free apps in each category provided by the store at the time. Among them, we find that 1,784 apps include the Facebook SDK for Android. We consider only apps that invoke the authentication method—Over 50 apps appear to have no call sites to Facebook APIs, and over 400 apps use the API but do not invoke any calls related to authorization. We also discard apps that do not work with our tool chain, e.g., crash on the emulator or have issues with `apktool` [1] since our analysis depends on the disassembled code of an apk file. This leaves us with 1,010 apps.

[1] `https://developers.facebook.com/docs/android/login-with-facebook`

[2] The later version of Facebook SDK was released in the middle of data collection and appears to use different methods to display a login screen. However, we find that almost no apps in our data set had adapted the new version yet.

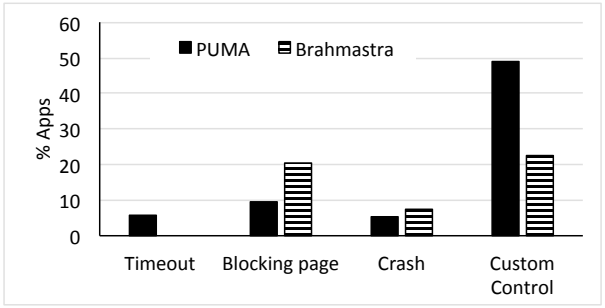

Figure 8: Failure causes of Brahmastra and PUMA.

App execution: In order to determine if Brahmastra or PUMA is able to reach a program point that invokes the target method, we instrument apps. The instrumentation detects when any of the target methods are invoked during runtime, by comparing signatures of executing methods with signatures of target methods. For Brahmastra, we consider only 5 of all paths generated by the Execution Planner. For PUMA, we explore each app for up to 250 steps; higher timeouts significantly increase overall testing time for little gain in hit rate.

6.2 Hit Rate

In our experiments, PUMA was able to successfully invoke a target method in 127 apps (13%). Note that PUMA's hit rate is significantly lower than its activity coverage ($>$ 90% compared to humans) reported in [23], highlighting the difficulty in invoking specific program points deep inside an app. In contrast, Brahmastra was successfully able to invoke a target method in 344 (34%) apps, a 2.7$\times$ improvement over PUMA. A closer examination of our results, as shown in Table 1, reveals that Brahmastra's technique can help circumventing all the root causes for PUMA's poor hit rate as mentioned in §3.

We now investigate why PUMA and Brahmastra sometimes fail to invoke the target method. For PUMA, this is due to the aforementioned four cases. Figure 8 shows the distribution of apps for which PUMA fails due to specific causes. As shown, all the causes happen in practice. The most dominant cause is the failure to find UI controls to interact with, which is mostly due to complex UI layouts of the popular apps we tested. Figure 8 also shows the root causes for Brahmastra's failure. The key reasons are as follows:

Blocking page: Even if jump start succeeds, successive activity transition may fail on a blocking page. Brahmastra fails for 20% of the apps due to this cause. We would like to emphasize that Brahmastra experiences more blocking pages than PUMA only because Brahmastra explores many paths that PUMA does not (e.g., because those paths are behind a custom control that

Case	Apps
R1: Timeout in PUMA, success in Brahmastra	62%
R2: Blocking page in PUMA, success in Brahmastra	48%
R3: Unknown control in PUMA, success in Brahmastra	43%
R4: Crash in PUMA, success in Brahmastra	30%

Table 1: % of apps for which Brahmastra succeeds but PUMA fails due to various reasons mentioned in §3.

PUMA cannot interact with, but Brahmastra can find and invoke the associated callback) and many of these paths contain blocking pages. If PUMA tried to explore those paths, it would have failed as well due to these blocking pages.

Crash: Jump start can crash if the starting activity expects specific parameters in the intent and Brahmastra fails to provide that. Brahmastra fails for 7% of the apps due to this cause.

Custom components: Execution Planner may fail to find useful paths if the app uses custom components[3], which can be used to override standard event handlers, thus breaking our model of standard Android apps. Without useful paths, Brahmastra can fail to invoke the target methods. In our experiments, this happens with 16% of the apps. We leave as future work a task to extend Execution Planner to handle custom components. We find that PUMA also failed 91% on these apps, proving the difficulty of navigating apps with custom components. In fact, PUMA suffers much more than Brahmastra due to custom components.

Improving the hit rate: There are several ways we can further improve the hit rate of Brahmastra. First, 16% failures of Brahmastra come because the static analysis fails to identify useful paths. A better static analysis that can discover more useful paths can improve Brahmastra's hit rate. Second, in our experiments, Brahmastra tried only up to 5 randomly selected useful paths to invoke the target method and gave up if they all failed. In many apps, our static analysis found many tens of useful paths, and our results indicate that the more paths we tried, the better was the hit rate. More specifically, Brahmastra succeeded for 207 apps after considering only one path, and for 344 apps after considering up to five paths. This suggests that considering more paths is likely to improve the hit rate. Additionally, we should select the paths to avoid any nodes or edges for which exploration failed in previously considered paths instead of choosing them randomly. In 72 apps, Brahmastra was unable to find the binding between a callback method and the UI element associated with it, causing it to fall back on a direct invocation of the callback method. A better static analysis can help in this case as well. In 22 apps, Brahmastra deemed a page blocked due to UI elements in the Android SDK (e.g., list elements) whose behavior was not encoded in the instrumentation engine. An engineering effort in special-case handling of these and more views would increase hit rate. We plan to explore such optimizations in future. Finally, PUMA (and other Monkeys) and Brahmastra use fundamentally different techniques to navigate between app pages and it might be possible to combine them in a single system where PUMA is used if Brahmastra fails (or vice versa). In our experiments, such a hybrid approach would give an overall hit rate of 39% (total 397 apps).

[3] http://developer.android.com/guide/topics/ui/custom-components.html

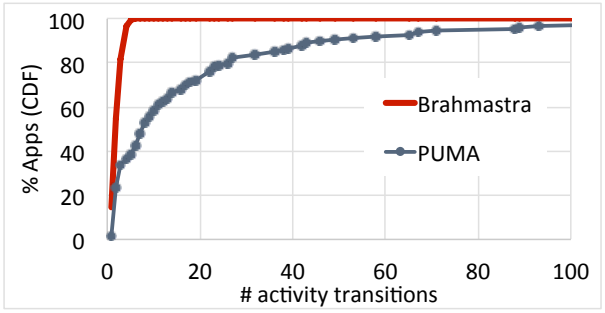

Figure 9: Test speed comparison of Brahmastra and PUMA

6.3 Speed

We use the number of activity transitions required to reach the target activity as a proxy for speed, since the actual time will vary depending on a variety of computational factors (e.g., network speed, device specifications). In Figure 9, we plot the CDF of the number of transitions required to reach the target activity for the apps which are successfully tested by both Brahmastra and PUMA. Since Brahmastra prunes away many unnecessary paths using static analysis, it runs faster than PUMA that suffers from uninformed activity transitions and large fanout in the activity transition graphs. On average, PUMA requires 18.7 transitions per app, while Brahmastra requires 2.5 transitions per app, resulting in 7 fold speedup.

7 Analysis of Ads in Kids Apps

Our first scenario is to use Brahmastra to study whether ad libraries for Android apps meet guidelines for protecting the online privacy of children. We give results for two popular ad components embedded in 220 kids apps. Our analysis shows that 80% of the apps displayed ads with a link to landing pages that have forms for collecting personal information, and 36% apps displayed ads deemed inappropriate to children.

7.1 Motivation and Goals

The Children's Online Privacy Protection Act (COPPA) lays down a variety of stipulations that mobile app developers must follow if their apps are directed at children under 13 years old [6]. In particular, COPPA disallows the collection of personal information by these apps unless the apps have first obtained parental consent.

COPPA holds the app developer responsible for the personal information collected by the embedded third party components as well as by the app's code [6]. Since it is common to see ad components included in free apps, we aim to measure the extent of potentially non-COPPA-compliant ad components in kids apps. Specifically, our first goal is to determine *whether in-app ads or landing pages pointed by these ads present forms that collect personal information*. Although displaying collection forms itself is not a violation, children might type in requested personal information, especially if these websites claim to offer free prizes or sweepstakes. In such cases, if these ads or landing pages do collect personal information without explicit parental consent, this act could be considered as a violation according to COPPA. Since it is difficult to model these legal terms into technical specifications, we only report potential concerns in this section. Our second goal is to test *whether content displayed in in-app ads or landing pages is appropriate for children*. Since this kind of judgement is fundamentally subjective, we show the breakdown of content categories as labeled by human testers.

Note that runtime observation is critical for this testing, since ads displayed within apps change dynamically depending on the inventory of ads at the time of request.

7.2 Testing Procedure

The testing has two steps. For a given app, we first collect ads displayed within apps and landing pages that are pointed by the ads. Second, for each ad and landing page, we determine: (1) whether they present forms to collect personal information such as first and last name, home address, and online contact as defined in COPPA; and (2) whether their content appears inappropriate to children and if so why.

Driving apps to display ads: We use Brahmastra to automatically drive apps to display ads. In this study, we focus on two popular ad libraries, AdMob and Millennial Media, because they account for over 40% of free Android apps with ads [11]. To get execution paths that produce ads, we use the following target methods as input to Brahmastra:

```
Lcom/google/ads/AdView;-><init>
Lcom/millennialmedia/android/MMAdView;-><init>
```

Collecting ads & landing pages: We redirect all the network traffic from executing test apps through a `Fiddler` proxy [8]. We install the `Fiddler` SSL certificate on the phone emulator as a trusted certificate to allow it to examine SSL traffic as well. We then identify the requests made by the ad libraries to their server component using domain names. Once these requests are collected, we replay these traces (several times, over several days), to fetch ad data from the ad servers as if these requests were made from these apps. This ad data is generally in the form of a JSON or XML object that contains details about the kind of ad served (image or text), the content to display on the screen (either text or the URL of an image), and the URL to redirect to if the ad is clicked upon. We record all of above for analysis.

Analyzing ads & landing pages: We use two methods to characterize ads and landing pages. First, for each landing page URL collected, we probe the Web of Trust (WoT) database [9] to get the "child safety" score. Second, to better understand the reasons why landing pages or ads may not be appropriate for children, we use crowdsourcing (via Amazon Mechanical Turk [3]) to label each ad and landing page and to collect detailed information such as the type of personal information that landing pages collect. As data collected from crowds may include inconsistent labeling, we use majority voting to filter out noise.

7.3 Results

Dataset: We collected our dataset in January 2014. To find apps intended for children, we use a list of apps categorized as "Kids" in Amazon's Android app store[4]. Since apps offered from the Amazon store are protected with DRM and resist bytecode rewriting, we crawled the Play store for apps with the same package name.

Starting from slightly over 4,000 apps in the Kids category, we found 699 free apps with a matching package name in the Play store. Among these, we find 242 apps that contain the AdMob or Millennial Media ad libraries. Using Brahmastra, we were successfully able to retrieve at least one ad request for 220 of these apps (also in January 2014), for which we report results in this section. For the remaining 22 apps, either Brahmastra could not navigate to the correct page, or the app did not serve any ad despite reaching the target page.

Results: We collected ads from each of the 220 apps over 5 days, giving us a total collection of 566 unique ads, and 3,633 unique landing pages. Using WoT, we determine that 183 out of the 3,633 unique landing pages have the child-safety score below 60, which fall in the "Unsatisfactory", "Poor" or "Very Poor" categories. 189 out of the 220 apps (86%) pointed to at least one of these pages during the monitoring period. Note that WoT did not contain child-safety ratings for 1,806 pages, so these

[4]Google Play store does not have a separate kids category.

Info Type	Landing Pages	Apps
Home address	47	58
First and last name	231	174
Online contact	100	94
Phone number	17	15
Total	235	175

Table 2: Personal information collected by landing pages

numbers represent a lower bound. We then used Amazon Mechanical Turk to characterize all 566 ads, and 2,111 randomly selected landing pages out of the 3,633. For each ad and landing page, we asked Amazon Mechanical Turk to check whether they collect personal information (of each type) and whether they contain inappropriate content for children (see Appendix B for the task details). We offered 7 cents (US) per each task (which involves answering various questions for each website or banner ad) and collected three responses per data point. As discussed above, we only counted responses that were consistent across at least two out of three respondents, to filter out noise.

Table 2 summarizes the types of personal information that landing pages ask users to provide as labeled by Amazon Mechanical Turk. We find that at least 80% of the apps in the dataset had displayed ads that point to landing pages with forms to collect personal information. On a manual examination of a subset of these pages, we found no labeling errors. We also found that none of the sites we manually checked attempt to acquire parental consent when collecting personal information. See Appendix B for examples.

Table 3 breaks down child-inappropriate content of the ads displayed in apps as labeled by Amazon Mechanical Turk. Although COPPA does not regulate the content of online services, we still find it concerning that 36% (80 out of 220) of the apps display ads with content deemed inappropriate for children. In particular 26% (58 apps) displayed ads that offer free prizes (e.g., Figure 13), which is considered a red flag of deceptive advertising, especially in ads targeting children as discussed in guidelines published by the Children's Advertising Review [10]. We also analysed the content displayed on the landing pages, and found a similar number of content violations as the ad images.

8 Analysis of Social Media Add-ons

Our second use case is to test apps against a recently discovered vulnerability associated with the Facebook SDK [30]. Our testing with Brahmastra shows that 13 out of 200 Android apps are vulnerable to the attack. Fixing it requires app developers to update the authentication logic in their servers as recommended by [30].

Content Type	Image Ads	Apps
Child exploitation	2	8
Gambling, contests, lotteries or sweepstakes	3	2
Misleading users about the product being advertised	7	16
Violence, weapons or gore	4	5
Alcohol, tobacco or drugs	3	3
Profanity and vulgarity	0	0
Free prize	39	58
Sexual or sexually suggestive content	12	29
Total	62	80

Table 3: Breakdown of child-inappropriate content in ads

8.1 Testing Goal

The Facebook access token vulnerability discussed in [30] can be exploited by attackers to steal the victim's sensitive information stored in vulnerable apps. For instance, if a malicious-yet-seemingly benign news app can trick the victim *once* to use the app to post a favorite news story on the victim's Facebook wall (which is a common feature found in many news apps), then the malicious app can use the access token obtained from the Facebook identity service to access sensitive information stored by *any* vulnerable apps that the victim had interacted with and have been associated with the victim's Facebook account. This attack can take place offline—once the malicious app obtains an access token, then it can send the token to a remote attacker who can impersonate as the victim to vulnerable apps.

Figure 10 gives the steps that allow a malicious application to steal `Victim`'s information from $\texttt{VulApp}_\text{s}$. The fact that the online service ($\texttt{VulApp}_\text{s}$) is able to retrieve the user's information from Facebook only means that the client ($\texttt{MalApp}_\text{c}$) possesses the privilege to the Facebook service, but is not a proof of the client app's identity ($\texttt{MalApp}_\text{c} \neq \texttt{VulApp}_\text{c}$). The shaded boxes in Figure 10 highlight the vulnerability. See [30] for more detail.

Wang et al. [30] manually tested 27 Windows 8 apps and showed that 78% of them are vulnerable to the access token attack. Our goal is to scale up the testing to a large number of Android applications. Note that testing for this vulnerability requires runtime analysis because the security violation assumptions are based on the interactions among the application, the application service, and Facebook service.

8.2 Testing Procedure

The testing has three steps. For a given app, we first need to drive apps to load a Facebook login screen. Second, we need to supply valid Facebook login credentials to observe interactions between the test application and Face-

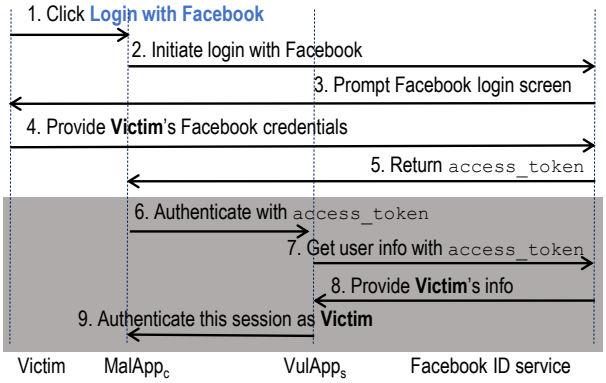

Figure 10: Facebook's access token, intended for authorizing access to `Victim`'s info, is used by `VulApp`$_s$ to authenticate the session as `Victim`. From step 9, `MalApp`$_c$ can steal `Victim`'s sensitive information in `VulApp`$_s$.

```
1  if(oSession.url.Contains("m.facebook.com")) {
2   var toReplace = "access_token=CAAHOi...";
3   ...
4   if(oSession.oResponse.headers.
5    ExistsAndContains("Location", "access_token"))
6   {
7     oSession.oResponse.headers["Location"] =
8      oSession.oResponse.headers["Location"].
9        replace(oRegEx, toReplace);
10    oSession["ui-customcolumn"] = "changed-header";
11  } }
```

Figure 11: A script used to manipulate `access_token`: We only show the first 6 bytes of the access token used in the attack.

book ID service. Third, we need to determine whether the test application misuses a Facebook access token for authenticating a client (steps 7-9) by monitoring network traffic and application behavior after providing a fraudulent access token.

Driving apps to display Facebook login: We use Brahmastra to automatically drive apps to invoke the Facebook SDK's authentication methods shown in Figure 7. Once the authentication methods open the sign-in window, we supply valid Facebook credentials.

Manipulating traffic with MITM proxy: As before, we direct all network traffic through a `Fiddler` proxy. Since Facebook sign-in traffic is encrypted over SSL, we also install a `Fiddler` SSL certificate on the phone emulator to decrypt all SSL traffic.

To manipulate the login, we record an `access_token` from a successful login session associated with another application (and therefore simulating an attacker as illustrated in the steps 1-5 of Fig. 10) and use the script shown in Fig. 11. It runs over HTTP responses, and overwrites an incoming `access_token` with a recorded one.

8.3 Experiments

Dataset: We randomly draw 200 apps from the dataset used in §6 for this testing.

Results: We find that 18 out of 200 apps use a Facebook access token for authentication, and among them 13 apps are vulnerable to a fraudulent access token (72%). 5 apps appear not vulnerable, and show some sort of error message when given a fraudulent access token. The remaining 182 apps use the Facebook SDK merely to post content to the user's wall, and not as an authentication mechanism. We determined this by looking at the network traffic at the login event, and observing that all of it is sent only to Facebook servers.

To understand how widespread the vulnerability is, we look at the statistics for the number of downloads on the Google Play store. Each of the 13 vulnerable apps has been downloaded more than 10,000 times, the median number of app downloads is over 500,000, and the most popular ones have been downloaded more than 10 million times. Further, these 13 apps have been built by 12 distinct publishers. This shows that the problem is not restricted to a few naïve developers. We shared the list of vulnerable apps with a Facebook security team on 2/27/2014 and got a response immediately that night that they had contacted the affected developers with the instructions to fix. The privacy implications of the possessing the vulnerability are also serious. To look at what user data can potentially be exfiltrated, we manually investigated the 13 vulnerable apps. Users of these apps may share a friends list, pictures, and messages (three dating apps); photos and videos (two apps); exercise logs and stats (one app); homework info (one app) or favorite news articles, books or music preferences (remaining six apps). By exploiting the vulnerability, a malicious app could exfiltrate this data.

9 Related Work

Automated Android app testing: A number of recent efforts proposed improvements over Android Monkey: AndroidRipper [13] uses a technique known as GUI ripping to create a GUI model of the application, and explores its state space. To improve code coverage, AndroidRipper relies on human testers to type in user credentials to get through blocking pages. However, despite this manual effort, the tool shows less than 40% code coverage after exploring an app for 4.5 hours. AppsPlayground [26] employs a number of heuristics—by guessing the right forms of input (e.g., email address, zip code) and by tracking widgets and windows in order to

reduce duplicate exploration. It shows that these heuristics are helpful although the coverage is still around 33%. SmartDroid [31] uses a combination of static and dynamic analysis to find the UI elements linked to sensitive APIs. However, unlike Brahmastra, SmartDroid explores every UI element at runtime to find the right view to click. A3E [14] also uses static analysis to find an activity transition graph and uses the graph to efficiently explore apps. We leveraged the proposed technique when building an execution path. However, similarly to the tools listed above, A3E again uses runtime GUI exploration to navigate through activities. In contrast to these works, Brahmastra determines an execution path using static analysis and rewrites an app to trigger a planned navigation, bypassing known difficulties related to GUI exploration.

Security analysis of in-app ads: Probably because only recently COPPA [6] had been updated to include mobile apps[5], we are not aware of any prior work looking into the issues around COPPA compliance of advertisements (and the corresponding landing pages) displayed within apps directed at children. However, several past works investigated security and privacy issues with respect to Android advertising libraries. AdRisk [22] is a static analysis tool to examine advertising libraries integrated with Android apps. They report that many ad libraries excessively collect privacy-sensitive information and expose some of the collected information to advertisers. Stevens et al. examine thirteen popular Android ad libraries and show the prevalent use of tracking identifiers and the collection of private user information [28]. Worse, through a longitudinal study, Book et al. show that the use of permissions by Android ad libraries has increased over the past years [18].

Analyzing logic flaws in web services and SDKs: The authentication vulnerability discussed in §8 falls into the category of logic flaws in web programming. Recent papers have proposed several technologies for testing various types of logic flaws [16, 17, 21]. However, these techniques mainly target logic flaws in *two-party web programs*, i.e., programs consisting of a client and a server. Logic flaws become more complicated and intriguing in multi-party web programs, in which a client communicating with multiple servers to accomplish a task, such as the Facebook-based authentication that we focus in this paper. AuthScan is a recently developed technique to automatically extract protocol specifications from concrete website implementations, and thus discover new vulnerabilities in the websites [15]. In contrast, our goal is not to discover any new vulnerability on a website, but to scale up the testing of a known vulnerability to a large number of apps.

[5]The revision was published on July 2013.

10 Discussion

Limitations: Although Brahmastra improves test hit rates over Monkey-like tools, we discover several idiosyncratic behaviors of mobile apps that challenge runtime testing. Some apps check servers upon launching and force upgrading if newer versions exist. Some apps constantly load content from remote servers, showing transient behaviors (e.g., extremely slow at times). We also have yet to implement adding callbacks related to sensor inputs. Another challenge is to isolate dependent components in the code. We assume that each activity is more or less independent (except that they pass parameters along with intent) and use our jump start technique to bypass blocking pages and to speed up testing. However, we leave as future work a task to statically determine dependent activities to find activities to jump-start to without affecting the program behavior.

Other runtime security testing of mobile apps: As mobile apps are highly driven by user interaction with visual components in the program, it is important to analyze the code behavior in conjunction with runtime UI states. For instance, malicious third-party components can trick users into authorizing the components to access content (e.g., photos) that the users intended to share with the application. Brahmastra can be used to capture visual elements when certain APIs are invoked to check against such click jacking attempts. Brahmastra can also automate the testing to check whether privacy-sensitive APIs are only invoked with explicit user interactions.

11 Conclusion

We have presented a mobile app automation tool, Brahmastra, that app store operators and security researchers can use to test third-party components at runtime as they are used by real applications. To overcome the known shortcomings of GUI exploration techniques, we analyze application structure to discover desired execution paths. Then we re-write test apps to follow a short path that invokes the target third-party component. We find that we can more than double the test hit rate while speeding up testing by a factor of seven compared to a state-of-the-art Monkey tool.

We use Brahmastra for two case studies, each of which contributes new results: checking if third-party ad components in kids apps are compliant with child-safety regulations; and checking whether apps that use Facebook Login are vulnerable to a known security flaw. Among the kids apps, we discover 36% of 220 kids apps display ads deemed inappropriate for children, and 80% of the apps display ads that point to landing pages which attempt to collect personal information without parental consent. Among the apps that use Facebook Login, we find that 13 applications are still vulnerable to the Face-

book access token attack even though the attack has been known for almost a year. Brahmastra let us quickly check the behavior of hundreds of apps for these studies, and it can easily be used for other studies in the future—checking whether privacy-sensitive APIs can be invoked without explicit user interaction, discovering visible UI elements implicated in click jacking attempts, and more.

Acknowledgments

This material is based on research sponsored in part by DARPA under agreement number FA8750-12-2-0107, NSF CCF-1139021, and University of Maryland Partnership with the Laboratory of Telecommunications Sciences, Contract Number H9823013D00560002. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] A tool for reverse engineering Android apk files. `http://code.google.com/p/android-apktool/`.

[2] Activity — Android Developers. `http://developer.android.com/reference/android/app/Activity.html`.

[3] Amazon Mechanical Turk. `https://www.mturk.com`.

[4] Android Debug Bridge. `http://developer.android.com/tools/help/adb.html`.

[5] Android Developers, The Developer's Guide. UI/Application Exerciser Monkey. `http://developer.android.com/tools/help/monkey.html`.

[6] Complying with COPPA: Frequently Asked Questions. `http://business.ftc.gov/documents/Complying-with-COPPA-Frequently-Asked-Questions`.

[7] Dexpler: A Dalvik to Soot Jimple Translator. `http://www.abartel.net/dexpler/`.

[8] Fiddler. `http://www.telerik.com/fiddler`.

[9] Web of Trust. `https://www.mywot.com/`.

[10] Self-Regulatory Program for Childrens Advertising, 2009. `http://www.caru.org/guidelines/guidelines.pdf`.

[11] AppBrain, Feb. 2014. `http://www.appbrain.com/stats/libraries/ad`.

[12] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[13] D. Amalfitano, A. R. Fasolino, S. D. Carmine, A. Memon, and P. Tramontana. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the IEEE Conference on Automated Software Engineering (ASE)*, 2012.

[14] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA*, 2013.

[15] G. Bai, J. Lei, G. Meng, S. S. V. P. Saxena, J. Sun, Y. Liu, and J. S. Dong. Authscan: Automatic extraction of web authentication protocols from implementations. In *NDSS*, 2013.

[16] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. Notamper: Automatically detecting parameter tampering vulnerabilities in web applications. In *CCS*, 2010.

[17] P. Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan. Waptec: Whitebox analysis of web applications for parameter tampering exploit construction. In *CCS*, 2011.

[18] T. Book, A. Pridgen, and D. S. Wallach. Longitudinal analysis of android ad library permissions. In *IEEE Mobile Security Technologies (MoST)*, 2013.

[19] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[20] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *CCS*, 2013.

[21] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security*, 2010.

[22] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *WiSec*, 2012.

[23] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-Automation for Large Scale Dynamic Analysis of Mobile Apps. In *Mobisys*, 2014.

[24] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

[25] B. Liu, S. Nath, R. Govindan, and J. Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *USENIX NSDI*, 2014.

[26] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*, 2013.

[27] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. In *Mobisys*, 2014.

[28] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *IEEE Mobile Security Technologies (MoST)*, 2012.

[29] R. Valle-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *IBM Centre for Advanced Studies Conference*, 1999.

[30] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *USENIX Security*, 2013.

[31] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

A Constant Propagation Analysis

We extend the existing constant propagation analysis so as to trace `intents`, UI elements, and listeners. In addition to traditional value types, such as numerical or string constant, we add meta-class, object, and intent sorts, which track class ids, object references, and intent instances, respectively. For instructions that create objects; load class ids; or invoke special APIs such as `Intent.setClass()`, we add their semantics into the data-flow transfer function.

Figure 12 illustrates how we extend data-flow lattice; how we conform to *meet* operation property; and how we define semantics of relevant instructions.

```
1  type lattice = ...
2    | Clazz of string (* const−class *)
3    | Object of string (* instance *)
4    | Intent of string (* Intent for a specific component *)
5    | ...
6  let meet l1 l2 = match l1, l2 with ...
7    | Clazz c1, Clazz c2 when 0 = compare c1 c2 → l1
8    | Object o1, Object o2 when 0 = compare o1 o2 → l1
9    | Intent i1, Intent i2 when 0 = compare i1 i2 → l1
10   | ...
11 let transfer (inn: lattice Map.t) (op, opr) = ...
12   else if OP_NEW = op then (* NEW *)
13   (
14     let dst :: id ::[] = opr in
15     let cname = Dex.get_ty_name id in
16     if 0 = compare cname ''Intent''
17     then Map.add dst (Intent '''') inn
18     else Map.add dst (Object cname) inn
19   ) ...
```

Figure 12: Abbreviated source code of extended constant propagation analysis. Meta-class, object, and intent sorts maintain information as string, and they can be merged only if internal values are identical, hence *must*-analysis. As an example, this shows how to handle opcode `NEW`.

B Examples of Ads in Kids Apps

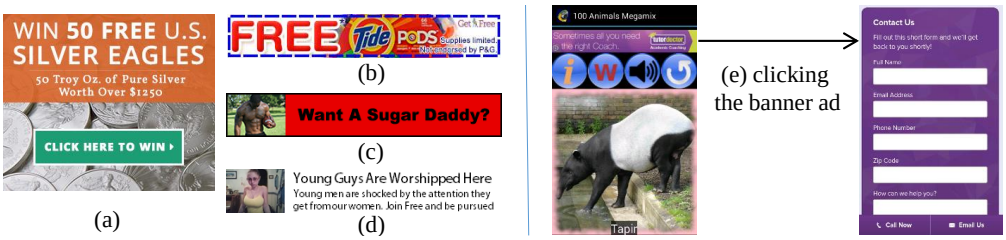

Figure 13: a) and (b) offer a free prize and (c) and (d) are sexually suggestive. (e) shows an example where clicking a banner ad displayed in a kids app opens up a landing page that presents forms to collect personal information.

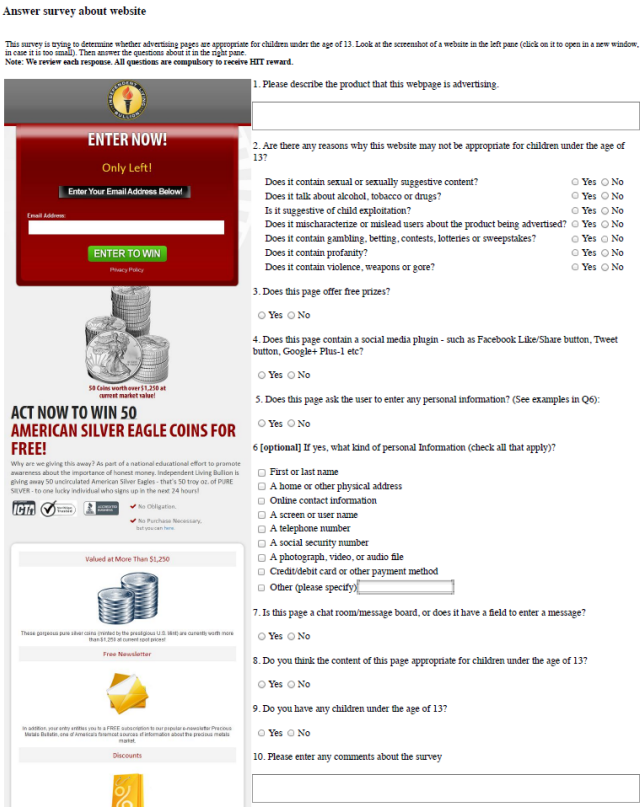

Figure 14: A screenshot of the Amazon Mechanical Turk task that we created to characterize landing pages pointed by ads displayed in kids apps