TDroid: Exposing App Switching Attacks in Android with Control Flow Specialization

Jie Liu
UNSW Sydney, Australia
jie.liu@cse.unsw.edu.au

Diyu Wu
UNSW Sydney, Australia
diywu@cse.unsw.edu.au

Jingling Xue
UNSW Sydney, Australia
jingling@cse.unsw.edu.au

ABSTRACT

The Android multitasking mechanism can be plagued with app switching attacks, in which a malicious app replaces the legitimate top activity of the focused app with one of its own, thus mounting, e.g., phishing and denial-of-service attacks. Existing market-level defenses are still ineffective, as static analysis is fundamentally unable to reason about the intention of an app and dynamic analysis has low coverage.

We introduce TDroid, a new market-level approach to detecting app switching attacks. The challenge lies in how to handle a plethora of input-dependent branch predicates (forming an exponential number of paths) that control the execution of the code responsible for launching such attacks. TDroid tackles this challenge by combining static and dynamic analysis to analyze an app without producing any false positives. In its static analysis, TDroid transforms the app into runnable slices containing potentially app switching attacks, one slice per attack. In its dynamic analysis, TDroid executes these slices on an Android phone or emulator to expose their malicious GUIs. The novelty lies in the use of a new trigger-oriented slicing technique in producing runnable slices so that certain input-dependent branch predicates are specialized to execute always some fixed branches.

Evaluated with a large set of malware apps, TDroid is shown to outperform the state of the art, by detecting substantially more app switching attacks, in a few minutes per app, on average.

1 INTRODUCTION

The Android multitasking mechanism has been plagued by severe security risks [2]. We introduce a new program analysis tool, TDroid, deployable at the market level for detecting app switching attacks [8] (before they hit the users). In such an attack, a malicious app can replace the legitimate top activity of the focused app with one of its own (opened by typically calling startActivity()), thereby mounting, e.g., phishing and denial-of-service attacks.

Challenge. Consider a malware app that runs in the background, keeping track of the device status. Once the device has reached a specific state, determined by an input-dependent branch predicate involving some environment variables, the malware will execute some malicious code for launching an app switching attack. Such attacks can be launched without requiring any permissions, for all apps installed on a compromised device under all Android versions.

The challenge lies in how to handle a plethora of input-dependent branch predicates (forming an exponential number of paths) that control the execution of the code responsible for launching such attacks.

State of the Art. To mitigate app switching attacks (as a special case of GUI attacks), several on-device defenses have been proposed, including [8, 14, 57] (by adding a security indicator showing the app identity being interacted with) and [41] (by raising an alert dialog, deployable only in rooted devices). According to [52], however, such passive defenses are only marginally effective, yet at the expense of negatively impacting system performance and user experience.

To prevent app switching attacks, market-level defenses, which can be used during the vetting process, are preferred. Static analysis, dynamic analysis, and their hybrids have all been tried, but with only limited success in handling input-dependent branch predicates.

For static analysis, we are only aware of [8], which can detect app switching attacks. Their tool, referred to here as StaDef, scans an app for the code for obtaining the information about the focused app and the code for starting a new activity. If the former reaches the latter, then the app is flagged as suspicious. As a static analysis, StaDef can only report warnings about possible app switching attacks in a suspicious app, whose code must still be inspected by a human before any final decision can be made.

For dynamic analysis, general-purpose GUI testing tools, such as Google’s Monkey [15] (randomized), SAPIENZ (search-based) [37], and STOAT [47] (model-based) can also be used for detecting app switching attacks (with malicious GUIs visually inspected and confirmed by a human analyst). However, these tools have low coverage, as evaluated in Section 6, as many input-dependent branch predicates guiding malicious code may become true only after GUI testing (e.g., when a targeted victim app is subsequently installed).

Static and dynamic analysis can be used together. FuzzZDroid [40], which combines symbolic-execution-enhanced static...
analysis and dynamic fuzzing to expose the malicious behaviors in an app, can be used for detecting app switching attacks. However, its symbolic execution mechanism cannot generate the test inputs required to satisfy all the input-dependent branch predicates for two reasons. First, all such predicates form an exponential number of paths to be explored. Second, its fuzzing technique only models some simple APIs, but a malware app can, for example, inspect the device status by using many different APIs. Thus, the chances for reaching some malicious code are still low (as evaluated later).

**Our Solution.** We present TDroid, a market-level approach that combines static and dynamic analysis to detect app switching attacks. In its static analysis, TDroid identifies suspicious startActivity() calls leading to such attacks in an app and transforms the app into a number of runnable slices containing potentially app switching attacks, one slice per attack. In its dynamic analysis, TDroid simply executes these runnable slices on an Android device or emulator to expose the malicious calls hidden in these slices. A human analyst will flag a call (and the underlying app) as malicious if its exposed UI (screenshot) is malicious (as discussed in Section 3). Therefore, no false positives are produced.

The key novelty of TDroid lies in its use of a new trigger-oriented slicing technique for building runnable slices that allow app switching attacks to be exposed. Traditional slicing [22, 54], which reasons about data and control dependences, does not apply; since all the input-dependent branch predicates that guide some malicious code cannot be sliced away. In contrast, TDroid will specialize such predicates so that the guided malicious code is forced to be always executed.

We have evaluated TDroid with a set of 3814 malware apps labeled with “Bank”, “Ransom” or “Fraud” in the Android Malware Dataset [53]. These apps do not have the ground truth about app switching attacks. Statically, TDroid finds 3075 suspicious startActivity() calls in 1062 suspicious apps. Dynamically, TDroid exposes UIs in 978 out of 1062 apps, with 878 apps containing app switching attacks, confirmed by a human analyst. TDroid is fast, by spending an average of 160.5 seconds per app for the 1062 apps that are both statically and dynamically analyzed.

TDroid outperforms the state of the art substantially. As a static analysis tool for finding app switching attacks, StaDef [8] reports only 1642 suspicious startActivity() calls in 770 suspicious apps (also reported by TDroid), with 63 suspicious calls missed by TDroid. Note that such pure static error reports are not useful unless a human analyst actually reads the code of these 770 apps. In addition, we have also compared TDroid with four representative general-purpose analysis tools, which can also be used for detecting app switching attacks, using 20 malware apps (with one from each distinct malware family), containing 58 suspicious startActivity() calls identified by TDroid. Given a budget of 3 hours per app, Google’s Monkey [15], Sapienz [37], Stoat [47] and FuzzDroid [40] have achieved their recall rates (percentage hit rates for these calls) as 8.62%, 10.34%, 12.07% and 18.97%, respectively. In contrast, TDroid’s recall rate is 94.83%.

We have also evaluated TDroid using a set of goodwill apps to confirm its effectiveness as a market-level tool.

In summary, this paper makes the following contributions (with the artifact for reproducing all our results downloadable at https://tdroidtool.github.io/):

- a novel market-level security defense, TDroid, for defeating app switching attacks via a hybrid analysis;
- a new trigger-oriented slicing technique for constructing runnable slices containing app switching attacks; and
- a dynamic analysis for executing runnable slices on Android devices or emulators to expose malicious GUIs.

## 2 BACKGROUND

### 2.1 Android Components

Android apps are constructed from four types of components, Activity (a window containing UI components), Service (an operation running in the background), Broadcast Receiver (a listener responding to a system or app announcement), and Content Provider (a component managing a set of data shared with other apps). Three of the four component types, Activity, Service, and Broadcast Receiver, are activated by an asynchronous message called an **intent**.

Activity lies at the heart of the Android programming framework due to its event-driven nature. An activity acts as a container consisting of different GUI elements (e.g., views and text boxes), through which, users interact with an app for transitions between different activities. Conceptually, an app executes along the activity transition paths and other callbacks are sprawled out of them.

### 2.2 App Switching Attacks

Such attacks aim to steal the focus of the top, i.e., foreground app [8]. This is achieved when a malicious app replaces the legitimate top activity with one of its own. App switching attacks can be exploited for different purposes. As a UI spoofing attack (discussed in Section 3), a malware app waits until a specific victim app has been installed and then triggers an app switch with a fake UI requesting software update in order to lure users to download another malware app (e.g., a malicious version mimicking the victim app). As a denial-of-service attack, a malware app continuously monitors the device status to ensure that it is the top app and triggers an app switch otherwise, until a ransom has been paid.

### 2.3 Intermediate Representation (IR)

Given an app, TDroid works on its Jimple IR, a three-address byte-code IR, constructed by SOOT [51]. The following types of control flow statements exist: (1) jumps, including goto, return, and throw (treated as return), (2) two-way branches for if statements, and (3) switches, i.e., multi-way branches (which are not desugared into two-way if statements). The CFG (Control Flow Graph) of a method is constructed in the standard way, except for its explicit representation of jump statements and switch statements. For convenience, a basic block in a CFG consists of one single statement.

Let $s_1$ and $s_2$ be two statements in the CFG of a method. We write $s_1 \rightarrow^e s_2$ if $s_2$ is data-dependent on $s_1$ and $s_1 \rightarrow^f s_2$ if $s_2$ is control-dependent on $s_1$.

## 3 MOTIVATING EXAMPLE

Figure 1 shows an example abstracted from **Bankun** [55], one of the most widespread malware families [1]. **Bankun** masquerades as Google Play. Once executed, it checks to see if the device has installed any of the five major, legitimate Korean banking apps. If
The branch predicate inline 16 (line 21) is input-dependent, guiding with a fake UI requesting software update for the victim. BKMain lines 13–25, for an installed banking app in device in the background. In lines 8 – 12, the device is checked is similar and thus elided (lines 20 and 35).

and 4) is given but the code for the other three (identified as 1 – 3) generality, the code for attacking two banking apps (identified as 0 device information from the compromised devices. Without loss of switching attack to display a fake UI requesting software update a particular banking app is installed, Bankun will perform an app switching attack to display a fake UI requesting software update to lure users to download a rogue version of the same banking app. In addition, Bankun also steals SMS messages, contacts and device information from the compromised devices. Without loss of generality, the code for attacking two banking apps (identified as 0 and 4) is given but the code for the other three (identified as 1 – 3) is similar and thus elided (lines 20 and 35).

The Service, named Notifications (lines 1 – 25), monitors the device in the background. In lines 8 – 12, the device is checked for an installed banking app in Config.BKLIST, with the last one selected as the victim (recorded in bk_type and pack_name). In lines 13 – 25, startActivity() is called to start an activity of type BKMain with a fake UI requesting software update for the victim. The branch predicate in line 16 (line 21) is input-dependent, guiding the malicious code in lines 17 – 18 (lines 22 – 23). In lines 32 – 38, the activity BKMain draws a fake UI for the victim targeted, with different UIs for different banking apps. Figure 2 shows the fake UI for the banking app (identified as 0), requesting a newer but a rogue version to be downloaded.

3.1 Existing Approaches
The market-level solutions [8, 15, 37, 40, 47], reviewed in Section 1 and evaluated in Section 6, are ineffective for vetting Bankun. With pure static analysis, StaDEF [8] cannot detect app switching attacks in Bankun, since there are the startActivity() calls for making an app switch (lines 18 and 23) but not any code for checking the focused app. Even if Bankun is flagged as suspicious, a human analyst must still read the code to know why it is malicious.

With pure dynamic analysis, Google’s Monkey [15], Sapienz [37], and Stoat [47] will always fail if no targeted banking app in Config.BKLIST is installed yet during testing (in which case, by_type = -1 always). Even if some banking apps in Config.BKLIST are installed, these tools will still fail in exposing the malicious UIs in Bankun if checkStatus() is never triggered.

By combining symbolic-execution-enhanced static analysis and dynamic fuzzing, FuzzDroid [40] has achieved an improved coverage but will still be ineffective for Bankun. As the input-dependent branch predicate in line 9 cannot be modeled accurately, by_type = -1 is expected. Thus, no startActivity() call can be triggered.

3.2 The TDroid Approach
Figure 3 gives an overview of TDroid. Given an APK file, TDroid first performs a static “Pre-Analysis” to look for all suspicious startActivity() calls that may lead to app switching attacks. We focus on startActivity() since it is the most widely used API for starting an activity in goodware and malware [8]. However, our approach can be generalized to handle other activity-starting APIs.

TDroid then handles each suspicious startActivity() call separately. TDroid first constructs its runnable slices in “Static Backward Slicing” and then executes these slices in “Dynamic Execution” to expose the malicious startActivity() call (by exposing malicious UIs). As malicious activities of different class types may be started at the same startActivity() call, different slices as highlighted may be generated, one per malicious activity.

We will focus on the suspicious call in line 18, identified as $I_{18}$. The others in lines 20 and 23 are handled similarly.

3.2.1 Pre-Analysis. The objective is to identify all suspicious startActivity() calls that may lead to app switching attacks. A startActivity() call in an app is suspicious if it satisfies two conditions (Section 4): (1) the app is currently running in the background and (2) the call startActivity() may cause the legitimate top activity to be replaced by one of its own activities.

For $I_{18}$, Bankun is activated by a call to Notifications.onCreate() (lines 2 – 4), a non-user-driven callback. Thus, $I_{18}$ can be triggered when Bankun runs in the background. Then an activity of type BKMain can be opened. Based on the attribute of the intent object (line 14), bkMain, passed to $I_{18}$ (Table 1), the opened activity can be the top activity. Thus, $I_{18}$ is a suspicious call.
3.2.2 Static Backward Slicing. For each suspicious `startActivity()` call, we first apply a novel trigger-oriented slicing technique to generate backward slices with specializable branch predicates for the call. We rely on the following insight to identify heuristically specializable branch predicates.

**Property 1.** Let $S$ be a backward slice computed for a method $m$. Let $p$ be a branch predicate in $m$. If $p$ controls only one single non-empty execution path in $S$, such that $p$ does not read the same memory address that is also read along the path, then $p$ is specializable to always execute that path only.

We then apply a new control flow specialization technique to both (1) add the missing control-flow statements (including jump statements) in the methods contained in these slices and (2) specialize the specializable and missing branch predicates thus added. TDroid is simple and efficient (in terms of generating and running the slices thus obtained as evaluated later).

Given $I_{18}$, we compute some backward slices affecting $I_{18}$. The class type for its associated activity is BKMain (lines 26 – 39). So only one slice will be generated. We start with the slicing criterion $SC_{BKMain}(I_{18}) = \{18, 33, 36\}$, i.e., the three statements identified by their line numbers. In addition to line 18, we have also included lines 33 and 36 since these are where the malicious UIs are drawn in the callback method, BKMain.onCreate(), triggered at line 18, despite the fact that lines 33 and 36 appear lexically after line 18.

(1) Trigger-Oriented Slicing. Traditionally [22, 54], the backward slice computed for $SC_{BKMain}(I_{18})$ consists of lines 6, 8 – 10, 13, 16 – 18, 29, 30, 32, 33, 36. Thus, the input-dependent predicate in line 16 is included. As discussed earlier, applying a GUI testing tool to such a slice will hardly expose the hidden malicious behaviors.

To compute the backward slice for $SC_{BKMain}(I_{18}) = \{18, 33, 36\}$, TDroid proceeds similarly as traditionally, except that certain input-dependent branch predicates are flagged as specializable (Property 1). Note that a predicate that is initially specializable can become non-specializable as the slice grows. For $I_{18}$, bk_type == 0 in line 16 will be specialized with a Boolean constant to ensure that $I_{18}$ is always triggered. As a result, all the statements affecting the definitions of bk_type are no longer in the slice. Trigger-oriented slicing has two benefits. First, it is lightweight, costing 22.2 seconds per app in our evaluation. Second, it improves the code coverage of a subsequent dynamic analysis.

Let us see how TDroid computes the slice from $SC_{BKMain}(I_{18}) = \{18, 33, 36\}$, with the initial slice being $T_{BKMain}(I_{18}) = SC_{BKMain}(I_{18})$. At this stage, no branch predicates are specializable yet. Let us start from lines 33 and 36. As $32 \rightarrow 33$ and $32 \rightarrow 36$, line 32 is added to $T_{BKMain}(I_{18})$. As lines 33 and 36 are in two different branches controlled by line 32, Property 1 is violated. So line 32 is not specializable. As $30 \rightarrow 32$, line 30 is added to $T_{BKMain}(I_{18})$. Line 30 is not specializable by Property 1, as the variable bkIntent used in line 30 is also used in line 32.

Let us now start from line 18. As $17 \rightarrow 32$ is a data dependence (through the intent object associated with $I_{18}$), line 17 is simply added to $T_{BKMain}(I_{18})$. As $16 \rightarrow 17$ and $16 \rightarrow 32$, line 16 is added to $T_{BKMain}(I_{18})$. Now, $T_{BKMain}(I_{18}) = \{16, 17, 18, 30, 32, 33, 36\}$. By Property 1, line 16 is flagged as specializable since it does not use any variable also used in the single path (lines 17 and 18), controlled by line 16, in $T_{BKMain}(I_{18})$. Note that the statements that define the values used at specializable predicates are ignored.

Finally, $13 \rightarrow 17$, $13 \rightarrow 32$, $13 \rightarrow 30$, $13 \rightarrow 30$, $13 \rightarrow 30$, and $13 \rightarrow 30$. So we obtain $T_{BKMain}(I_{18}) = \{13, 16, 17, 18, 29, 30, 32, 33, 36\}$, with line 16 as the only predicate flagged as specializable.

(2) Control Flow Specialization. The statements in $T_{BKMain}(I_{18}) \setminus T_{BKMain}(I_{18})$ appear in the white background.

The statements in $T_{BKMain}(I_{18}) \setminus T_{BKMain}(I_{18})$ appear in the white background.
m in $\mathcal{I}_{I_{18}}$, there are two steps. First, we add all missing control-flow statements in m to $\mathcal{I}_{I_{18}}$ to obtain a runnable slice: $\mathcal{I}_{BKMain}(I_{18}) = \{8, 9, 12, 13, 16 - 19, 21, 24, 29, 30, 32 - 34, 36 - 39\}$. In Figure 4, the added statements in $\mathcal{I}_{BKMain}(I_{18})$ appear in the white background. Second, the newly added branch predicates in lines 8, 9 and 21 are specialized to execute some fixed branches (Section 5.3). For line 8 (corresponding to a loop), the loop body is always skipped. For each if statement in lines 9 and 21, either branch can be specialized as being taken.

3.2.3 Dynamic Execution. We repackage $\mathcal{I}_{BKMain}(I_{18})$ to a downsized APK file to expose its malicious UI by dynamic execution.

1 public class MainActivity extends Activity { 
2     public static Context ctx;
3     protected void onCreate(Bundle bundle) {
4         super.onCreate(bundle);
5         ctx = getApplicationContext();
6     }
7     new Notifications().checkStatus();
8 }

Figure 6: The launcher activity of the downsized APK for $I_{18}$.

(1) Repackaging. In Figure 4, checkStatus() is the entry method to activate $I_{18}$. We make use of the launcher activity in Figure 6 to call checkStatus() (even if it is non-public) on a Notifications object created ourselves (rather than by the Android framework). Thus, in lines 13 and 18 of Figure 4, this (of type Notifications) can no longer be used as a context object. Rather, as shown in the repackaged version in Figure 5, a context object of type Context that is made available in a static field MainActivity.ctx is used instead. L1 is added to force an app switch, since this is possible originally (Section 3.2.1). As the Android framework is not modeled accurately during the slicing, some data and control dependences related to callbacks are missing. L2, as the super call in BKMain.onCreate(), is added. Finally, L3 is added so that screenshots are taken and analyzed.

(2) Execution. We install the downsized APK for $I_{18}$ on an unmodified Android phone or emulator and then execute it to expose the malicious UI in Figure 2. Similarly, the malicious UIs for the other four banking apps can also be exposed.

4 APP SWITCHING ATTACK

A startActivity() call is suspicious (in opening a top activity) if it satisfies two conditions: (1) the app can run in the background, and (2) the app can replace the legitimate top activity with one of its own (to therefore steal the focus to become the focused app).

4.1 Background Running Apps

For a startActivity() call, its containing app can run in the background if the call can be activated by a non-user-driven callback. User-driven callbacks manage user interactions [56], including (1) lifecycle callbacks for activities and UI components (e.g., dialogs and menus), defining some changes to their visible state, run-time events and behaviors, and (2) GUI event handler callbacks, responding to user actions (e.g., clicking a button). Non-user-driven callbacks require no direct user interactions, including the lifecycle callbacks for services (e.g., Service.onCreate()), broadcast receivers (e.g., BroadcastReceiver.onReceive()), and content providers (e.g., ContentProvider.onCreate()).

Banku in Figure 1 can be activated in the background by the non-user-drive callback Notifications.onCreate() (line 2).

4.2 App Switching

An activity that is opened at a startActivity() can become a top activity, determined by three factors reported in [8]: the class type of the Android component from which the call is made, the launchMode attribute for the opened activity, and the flags set for its associated intent object. In this paper, we find that a fourth influencing factor, the taskAffinity attribute of the opened activity, is also relevant.

Table 1: The nine scenarios where startActivity() can open a top activity (where *, as usual, means don’t-care).

<table>
<thead>
<tr>
<th>Receiver Type</th>
<th>launchMode</th>
<th>taskAffinity</th>
<th>Intent Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Activity</td>
<td>*</td>
<td>*</td>
<td>NEW_TASK</td>
</tr>
<tr>
<td>Activity</td>
<td>*</td>
<td>NEW_TASK</td>
<td>NEW_TASK, CLEAR_TASK</td>
</tr>
<tr>
<td>Activity</td>
<td>singleTop</td>
<td>*</td>
<td>NEW_TASK, MULTIPLE_TASK</td>
</tr>
<tr>
<td>Activity</td>
<td>standard</td>
<td>NEW_TASK</td>
<td>NEW_TASK, MULTIPLE_TASK</td>
</tr>
<tr>
<td>Activity</td>
<td>singleTask</td>
<td>*</td>
<td>CLEAR_TASK</td>
</tr>
<tr>
<td>Activity</td>
<td>singleInstance</td>
<td>Non-default</td>
<td>NEW_TASK</td>
</tr>
<tr>
<td>Activity</td>
<td>standard</td>
<td>Non-default</td>
<td>NEW_TASK</td>
</tr>
</tbody>
</table>

As Android’s official documentation does not state clearly when an opened Activity can become a top one, we have developed a tool, as in [8], to explore all possible combinations of the four factors. We restrict ourselves to Android 4.4, one of the most widely distributed versions. However, the basic principle behind applies to other versions. Table 1 gives a total of nine scenarios for an activity to become the top one, with the last three being new.

Consider the startActivity() call $I_{18}$ in Figure 1 that is examined in Section 3. The class type of the receiver object on which startActivity() is called is a Notifications Service. In addition, its associated intent bkMain is set with the flag NEW_TASK. Thus, $I_{18}$ is suspicious (falling into the first case in Table 1), regardless of the class type of any activity that may be opened.
5 THE TDROID DESIGN

We describe our algorithms for realizing the five components in TDROID (depicted in Figure 3). Given an app, TDROID works on its Jimple IR (Section 2), by taking as input (1) the call graph CG for the app, (2) the CFGs of its methods, (3) data and control dependencies, and (4) the alias information (in CG). In Section 6, we will explain how (1) – (4) are obtained in our implementation.

Given an app, “Pre-Analysis” is first performed to find all the suspicious startActivity() calls in CG. For each suspicious call I, our static analysis comes into play first. “Trigger-Oriented Slicing” first obtains a set T(I) of slices for I, one per malicious activity that may be opened at I and “Control Flow Specialization” then expands it into a set T(I) of runnable slices by performing branch predicate specialization. Next, our dynamic analysis takes over. For each runnable slice, “Repackaging” first turns it into a downsized APK file and “Execution” then takes care of executing it on an Android device or emulator to expose the malicious UI.

5.1 Pre-Analysis

For each startActivity() call I, we check to see if it is suspicious by verifying the two conditions in Section 4. Our subsequent algorithms apply to each suspicious call in isolation. In our example discussed in Section 3.2.1, I_{Ba} is suspicious.

5.2 Trigger-Oriented Slicing

Given a suspicious startActivity() call I, Algorithm 1, named TOSlicer, generates a set T(I) of slices interprocedurally, one slice T(I) for each class type t associated with the malicious activities opened at I. For simplicity, we assume the absence of static variables in the app considered. However, global variables can be handled in the standard manner [22].

Let us start with TOSlicer (lines 1 – 13). We first find the set of class types for the opened activities at I (line 3). For each class type representing suspicious activities (lines 4 and 5), we then set up its slicing criterion SC(I) (line 9) and call InterSlicer to compute interprocedurally a backward slice starting from all the points in SC(I), one at a time (lines 10 – 11). At this stage, all branch predicates are not specializable yet (lines 7 – 8). To set up SC(I), we include not only I but also all UI-drawing calls reachable (e.g., calls for modifying the UI contents in Dialog, Activity, or Window) that can be reached from all the lifecycle callbacks of class t in CG.

Let us now consider InterSlicer (lines 14 – 37). InterSlicer computes a backward slice interprocedurally from the statement s, as is done traditionally [22, 54], except that certain branch predicates are flagged as specializable (Property 1). Therefore, it suffices to explain only the parts inside the three blue boxes. In line 22, we ignore all the statements that define the values used at a specializable branch predicate. In lines 26 – 27, we flag a newly added branch predicate as specializable if Property 1 is satisfied. In lines 28 – 31, we recognize that a predicate that was previously specializable is no longer specializable (as T(I) has grown big). Thus, the slicing process must now be restarted from these predicates (line 31).

It is easy to check if s, contained in method m, satisfies Property 1 or not. T_m is the current slice for m (line 16). Let s_1, \ldots, s_n be the n successors of s in the CFG of m, denoted G_m. Let P(s_j) be the set of basic blocks, confined within T_m, reachable from s_j, which includes s_j itself, in G_m. We know that s satisfies Property 1 when the following two conditions are met. (1) There exists only one unique 1 \leq i \leq n such that P(s_i) is non-empty. This can be done by performing a standard control flow reachability analysis on G_m. (2) Let a be a memory address read by any statement s_i in P(s_i) (directly or indirectly if s_i is a call site). Then s does not also read from a. This can be done based on the def-use chains (or local variables) and alias information (for field accesses).

In our example, the class type of the activities opened at I_{Ba} is BKMain. Given that SC_{BKMain}(I_{Ba}) = \{18, 33, 36\}, the final slice T_{BKMain}(I_{Ba}) computed is given in Section 3.2.2, where line 16 is specializable. Thus, T(I_{Ba}) = \{T_{BKMain}(I_{Ba})\}.

5.3 Control Flow Specialization

Given a slice T(I), Algorithm 2 transforms and expands it into a runnable slice T(I) by performing two flavors of control flow specialization. In our example, Figure 4 demonstrates the transition from T_{BKMain}(I_{Ba}) to T_{BKMain}(I_{Ba}) for I_{Ba}.

In lines 2 – 3, we specialize every specializable branch predicate identified in T(I) to always execute the only branch contained...
in $T^t(I)$. In our example, bk_type = 0 in line 16 is specializable, forcing $I_{18}$ to be always executed in Figure 4.

In lines 4–14, we add all the missing control flow statements (including jump statements) contained in the methods in $T^t(I)$ so that $T^t(I)$ is runnable. In Figure 4 (for $I_{18}$), all such newly added control flow statements in $T^{\text{BGMain}}(I_{18}) \setminus T^{\text{BGMain}}(I_{18})$ are shown in the white background. For every newly introduced branch predicate, which is guaranteed not to control any statement in $T^{t'}(I)$ by construction, it is specialized depending on which of the three cases it falls into (lines 7–13). In Figure 4 (for $I_{18}$), $T^{\text{BGMain}}(I_{18}) \setminus T^{\text{BGMain}}(I_{18})$ contains three such branch predicates, in lines 8, 9 and 21 (of the original app in Figure 1), which are specialized, respectively, according to the three cases in that order.

5.4 Repackaging

Given a runnable slice $T^t(I)$, we will repack it into an APK file. There are four tasks, illustrated using our example.

First, we create the launcher activity, MainActivity, for $T^t(I)$. The one for $T^{\text{BGMain}}(I_{18})$ is given in Figure 6. Let $m_1, \ldots, m_n$ be the predecessor-less methods (regardless of their access modifiers) backwards reachable from $I$ in CG. Let $C_i$ be the class where $m_i$ is defined. For each $m_i$, we add "new $C_i$.m_i()" to MainActivity(). For $T^{\text{BGMain}}(I_{18})$, as shown in Figure 6, "new Notifications().checkStatus()" is added.

Second, each object of $C_i$ thus created can no longer request app-level operations such as launching activities. Regardless of the object used for calling startActivity() at $I$, we replace it by a context object of type Context stored in a global variable, MainActivity.ctx, as demonstrated for $I_{18}$ in Figure 6 (line 5) and Figure 5 (lines 13 and 18). For $T^t(I)$, an opened activity of type $t$ can be the top activity. Thus, we simply set the attribute of the intent associated with $I$ as NEW_TASK (e.g., line L1 for $I_{18}$ in Figure 5), so that this is also true for MainActivity.ctx (Table 1).

Third, as the Android framework is not modeled accurately, some data and control dependences may be missing. Given $T^t(I)$, we examine each of the lifecycle callbacks in class $t$. We add each missing super call (e.g., line L2 for $I_{18}$ in Figure 5) to $T^t(I)$.

Finally, we inject code into the lifecycle callback onCreate() of class $t$ for $T^t(I)$, the entry method for the opened activity of type $t$, in order to capture the malicious UI shown for $I_{18}$, this is done by the call in line L3 in Figure 5.

5.5 Execution

For each repackaged APK file, we run it on an Android device or emulator to expose its malicious UI. The screenshot captured for $I_{18}$ in Bankun is given in Figure 2. A human analyst can see clearly that Bankun, which masquerades as Google Play, is malicious.

6 EVALUATION

We have evaluated TDroid using both goodware and malware apps. With goodware, which do not contain app switching attacks, we show that TDroid is both efficient and effective as a market-level vetting tool. With malware, we both confirm this finding and show further that TDroid outperforms the state of the art in detecting substantially more app switching attacks. Below we first discuss our results for goodware briefly and then our results for malware extensively.

With goodware, we have used a total of 85 Android apps from the popular Android app repository F-Droid [11]. To minimize biases towards certain app categories, we selected top 5 apps from the top charts in all 17 app categories (on 23 March 2018). TDroid analyzes each app in 19.2 seconds on average. Statically, TDroid finds 7 suspicious startActivity() calls in 6 apps: MqttPublisher Plugin (1), Kwik EFIS (1), Linphone (1), Little Sir Echo (1), EteSync (1), and Clock (2). Dynamically, TDroid exposes the UIs in 6 of these 7 calls: MqttPublisher Plugin (1), Kwik EFIS (1), Linphone (1), Little Sir Echo (1), EteSync (0), and Clock (2). For the startActivity() call in EteSync, TDroid has failed to expose its UI. In this case, the underlying slice is incomplete since some data dependencies related to IPC (inter-process communication) are missing. The times (in seconds) elapsed on analyzing these seven apps are MqttPublisher Plugin (32.2), Kwik EFIS (35.9), Linphone (73.7), Little Sir Echo (43.3), EteSync (66.7), and Clock (111.6). As benign apps rarely open a top activity when running in the background (as expected), TDroid reports only a few suspicious startActivity() calls. In addition, TDroid can analyze such real-world apps efficiently and effectively.

With malware, we address three research questions (RQs):

• RQ1. Can TDroid detect app switching attacks effectively and efficiently with a hybrid static-dynamic analysis?

• RQ2. Is TDroid more effective than special-purpose tools developed for detecting app switching attacks?

• RQ3. Is TDroid more effective than general-purpose tools that can also be used to find app switching attacks?

We consider a new popular Android Malware Dataset [53]. We started by selecting a set of all the 4171 malware apps from all the 26 families (widely used in [8, 13, 14, 39, 41]) labeled with “Bank”, “Ransom” or “Fraud”. However, 357 apps cannot be decompiled by Dexpler [6] successfully. We have finally settled with a set of 3814 malware apps. Despite the labels, these apps do not have the ground truth about any hidden app switching attacks.
To address RQ1, we show how TDroid can (for the first time) detect which of these 3814 apps definitely contain app switching attacks. To address RQ2, we show that TDroid’s static analysis can discover more suspicious startActivity() calls (and more suspicious apps) than StaDef [8], the only special-purpose tool available, which applies static analysis to detecting app switching attacks. To address RQ3, we show that TDroid can find substantially more app switching attacks than three representative GUI testing tools, Google’s Monkey [15], Sapienz [37], and Stoat [47], as well as a hybrid analysis tool, FuzzDroid [40].

Implementations. We have implemented TDroid in Soot [51] in about 8 kLOC of Java code. Given an app (in an APK file), we first use Dexpler [6] to convert its Dalvik bytecode into Soot’s Jimple IR. We then run FlowDroid [5] on the Jimple code to build a call graph for the app. FlowDroid achieves this by working with Soot’s pointer analysis, Spark, to create a dummy main() consisting of the callbacks found iteratively in the app. We make use of the data and control dependences, alias information, and the CFGs for the methods in the call graph provided by Soot. In particular, the data dependences for the “Extras” stored into the intent for a method are tracked. For example, in Section 3.2.2, startActivity() is found by linking the use of “BK” at line 32 into the definition at line 17.

For StaDef [8], we have implemented it based on the algorithm described by its authors (as it is not open-sourced). For Google’s Monkey [15], Sapienz [37], Stoat [47], and FuzzDroid [40], we use their open-source tools.

StaDef runs on a desktop computer while the app being analyzed runs on an Android emulator. The Android Debug Bridge (adb) is used to facilitate communication between the desktop and emulator. Google’s Monkey [15], Sapienz [37], Stoat [47], and FuzzDroid [40] operate similarly.

Experimental Setup. Our desktop computer runs on a 64-bit Ubuntu 16.04 with 8 cores (3.2 GHz Intel Xeon(R) CPU) and 256 GB RAM. Our Android emulator is a Nexus 5 with the KitKat version (SDK 4.4, API level 19), one of the most widely distributed Android versions. The analysis time of an app is the average of 5 runs.

6.1 RQ1: TDroid’s Effectiveness and Efficiency

Figure 7 depicts our results on analyzing the 3814 malware apps from 26 malware families. Statically, TDroid finds 3075 suspicious startActivity() calls in 1062 suspicious apps spreading across 20 out of the 26 malware families, in an average of 16.4 seconds per app. Dynamically, TDroid has succeeded in running 92.1% \( (\frac{258}{280}) \) of the 1062 suspicious apps to completion, with 82.7% \( (\frac{228}{278}) \) being malicious, confirmed by visual inspection, in an average of 160.5 seconds (including both the static and dynamic analysis times) per app. These results demonstrate again TDroid’s effectiveness and efficiency, making it deployable as a market-level vetting tool.

TDroid failed in executing 84 suspicious apps. For 49 of these 84 apps, their original APK files are faulty (not executable). For the remaining 35 apps, their repackaged APK files are incomplete since their underlying slices are incomplete (due to, e.g., the unsound handling of reflection and the Android framework in Soot).

6.2 RQ2: Special-Purpose Detection

StaDef finds 1642 suspicious startActivity() calls in 770 apps, in 7.1 seconds per app, on average. As discussed earlier, TDroid
TDROID: Exposing App Switching Attacks in Android with Control Flow Specialization    ASE ’18, September 3–7, 2018, Montpellier, France

Table 2: Effectiveness and efficiency of TDROID in handling a set of 20 selected malware samples.

<table>
<thead>
<tr>
<th>APK</th>
<th>MDs</th>
<th>TDROID</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Suspicious startActivity() Calls</td>
<td># of Calls with Exposed Uls</td>
<td># of Malicious Calls</td>
</tr>
<tr>
<td>Aplex</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>BankBot</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Bankun</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>FakeAV</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>FakeDoc</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FakePlayer</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Findo</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>GoldDream</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Gumpen</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Jiaut</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Koler</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Nandroidbox</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Roop</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SimpleLocker</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Slenbugn</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>SmsZombie</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Spambot</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Spveng</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Triada</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zitmo</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>58</td>
<td>53</td>
</tr>
</tbody>
</table>

Figure 8: Comparing TDROID and StaDef on detecting suspicious startActivity() calls in the 20 selected malware apps.

Table 3: Comparing TDROID with Google’s Monkey [15], StaDef [37], Stoat [47], and FuzzDroid [40] in terms of their ability in exposing the malicious UIs for the 58 suspicious startActivity() calls found by TDROID’s static analysis in the 20 selected malware apps (Table 2).

<table>
<thead>
<tr>
<th>Tool</th>
<th>% of Suspicious Calls Reached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monkey</td>
<td>8.62</td>
</tr>
<tr>
<td>StaDef</td>
<td>10.34</td>
</tr>
<tr>
<td>Stoat</td>
<td>12.07</td>
</tr>
<tr>
<td>FuzzDroid</td>
<td>18.97</td>
</tr>
<tr>
<td>TDROID</td>
<td>94.83</td>
</tr>
</tbody>
</table>

Figure 8 shows that TDROID finds 3075 suspicious calls in 1062 apps, in 16.4 seconds per app, on average. StaDef reports 63 suspicious calls missed by TDROID, but only a subset of suspicious apps reported by TDROID. In total, TDROID has found 1242 malicious startActivity() calls in 205 malicious apps, which are not flagged as suspicious by StaDef.

It is important to note again that StaDef, as a pure static analysis tool, is not suitable for vetting apps, since a human analyst must still read the code of all suspicious apps to make a final decision. For the 63 suspicious startActivity() calls reported by StaDef but missed by TDROID, we have inspected their relevant code. Due to obfuscation, it is unclear whether 15 of these calls are malicious or not. Passing them to TDROID is still inconclusive as they try to access a database that could not be reproduced in our emulation environment. The remaining 48 calls are all malicious, capable of launching app switching attacks. TDROID has missed these calls since they are all made from asynchronous tasks started by user-driven callbacks (e.g., onClick), when these malicious apps are initially running in the foreground. Given these calls, TDROID can expose their malicious UIs in 27.0 seconds each.

Figure 8 shows that TDROID is more effective than StaDef in finding suspicious startActivity() calls in the 20 selected malware apps (Table 2). As discussed in Section 6.1, TDROID reports 58 suspicious calls, of which 51 are malicious, in 16 apps. In contrast, StaDef reports only 25 suspicious calls (also reported by TDROID), which are all malicious in 10 apps. In particular, StaDef misses app-switching attacks in the six malicious apps, Bankun, GoldDream, Gummen, SimpleLocker, Smzombie and SpamBot. (Recall that TDROID finds no attacks in FakeDoc, FakePlayer, Nandroidbox and Zitmo.)

6.3 RQ3: General-Purpose Detection

We compare TDROID with four representative general-purpose tools, Google’s Monkey (randomized) [15], StaDef (search-based) [37], Stoat (model-based) [47], and FuzzDroid [40], which all involve executing an app to expose its malicious behaviors. The first three aim for high coverage while the last attempts to reach a particular program point. Therefore, the criterion used here is the recall rate, measured as the percentage hit rate for the suspicious startActivity() detected by TDROID’s static analysis. For these four tools, the per-app budget allocated 3 hours.

Table 3 shows that TDROID is significantly more effective than the state of the art in exposing the malicious UIs hidden at the 58 suspicious startActivity() calls found by TDROID’s static analysis in the 20 selected malware apps (Table 2). This is mainly because these existing tools cannot handle effectively the input-dependent branch predicates, which come in a variety of flavors in
malware apps. For example, many malicious apps, such as Aples, Spambot and GoldDream, launch app switching attacks after some system broadcasts (e.g., reboot completed, network connectivity changed, and device woke up) are sent.

To improve the coverage, i.e., recall rate of Google’s Monkey (8.62%), SAPIENZ [37] and STOAT [47] inject system-level events randomly, but are only marginally more effective (with their recall rates being 10.34% and 12.07%, respectively). To go further, FuzzDroid [40] applies static analysis to model system broadcasts and generates candidate environments with dynamic fuzzing to reach a particular code location. However, due to its incomplete handling of system APIs and event dependencies (among others), FuzzDroid is only relatively more effective (at 18.97%). By applying trigger-oriented slicing, TDroid can avoid evaluating explicitly many input-dependent predicates, achieving a recall rate of 94.83%.

6.4 Limitations
TDroid performs its trigger-oriented slicing for an app on a call graph built by FLOWDROID [5], together with Soot’s spark pointer analysis [27], in the Sooot framework. As reflection is handled only partially in Sooot, the call graph may be incomplete. This can be improved by incorporating a more powerful reflection analysis tool [16, 31, 32, 34, 59]. In addition, FLOWDROID does not support inter-component communication (ICC). Currently, the data dependences for the “Extras” added to the intent objects at startActivity() are tracked. The call graph can also be incomplete this way. This can be improved by adding an ICC analysis [28].

Persistent storage APIs such as SharedPreferences are widely used in Android apps. Therefore, the statements operating on persistent data by these APIs can be distributed throughout an app. Currently, we do not model the data dependencies through persistent storage, resulting in incomplete slices sometimes (Section 6.1).

7 RELATED WORK

GUI Security. Earlier papers show the possibility of launching UI spoofing [8, 9, 12] and denial-of-service [17, 41, 42] attacks in Android by playing tricky maneuvers on the currently active task of the foreground app. In particular, Felt et al. [12] assess the risk of phishing attacks on mobile devices driven by inter-app control transfers. Ren et al. [42] study the task hijacking attacks in which the attacker can let the activities of a malware app reside side by side with those of a victim app in the same task and hijack the user sessions of the victim app. Chen et al. [9] reveal that GUI confidentiality can be breached by a shared-memory side channel. He et al. [21] investigate broadly a number of API-related compatibility issues in Android apps due to API evolution.

Several techniques exist for defeating GUI attacks [8, 41, 43]. Bianchi et al. [8] introduce StaDEF (the static analysis compared with TDroid) and an on-device defense for mitigating GUI attacks (including app switching attacks). Ren et al. [41] also propose an on-device defense for rooted devices.

In this paper, we focus on developing a market-level hybrid analysis for defending against app switching attacks.

Program Slicing. There have been many extensions of Weiser’s slicing technique [54], including thin slicing [46], path slicing [25], and tailoring [33]. There are others focusing on handling jump statements [3], unstructured programs [20], jumps and switches [26], and exceptions [4].

To obtain data dependences required in slicing, the aliasing information in the program can often be computed by applying a pointer analysis algorithm [7, 19, 24, 29, 30, 45, 48–50, 58].

In this paper, TDroid produces runnable slices by considering not only data and control dependences as in the prior work but also control flow specialization in order to expose the malicious UIs easily.

GUI Testing. To improve the coverage of Google’s Monkey [15] through optimizing test sequences, SAPIENZ [37] embraces multi-objective search-based testing and STOAT [47] resorts to automated model-based testing. DYNODROID [35] applies an observe-select-execute principle to generate UI and system inputs. EvoDroid [36] adopts segmented evolutionary testing to improve coverage. PUMA [18] is a programmable UI automation framework for implementing various state-based test strategies. Some other recent work for facilitating GUI testing can also be found in [10, 38, 44, 60].

Despite these advances, GUI testing tools are still poor in test coverage. In contrast, TDroid has significantly improved coverage for detecting app switching attacks by applying a new trigger-oriented slicing technique.

Hybrid Analysis. AppDoctor [23] combines static slicing and dynamic execution to find bugs triggered by user actions. CRED-MINER [61] proceeds similarly to study the prevalence of unsafe developer credentials. However, it executes slices in a custom engine, requiring a precise model of the Android OS and libraries. HARVESTER [59] is designed to extract runtime values from Android apps, by executing a traditional backward slice of an app on an Android device or emulator to log the values of interest, such as some class and method names. FuzzDroid [40], which is compared with TDroid in Section 6.3, combines static analysis and dynamic fuzzing to generate candidate environments to steer an app towards a code location. Both HARVESTER and FuzzDroid need to handle adequately a large number of system APIs in Android.

In this paper, TDroid combines a novel trigger-oriented slicing approach and dynamic execution to find app switching attacks effectively.

8 CONCLUSION
We have introduced a new market-level approach, TDroid, for detecting app switching attacks in Android apps, by combining a trigger-oriented slicing technique and dynamic execution. TDroid is substantially more effective in detecting app switching attacks than the state of the art. In addition, TDroid has two other immediate benefits. First, TDroid can be used to extract dynamic values from Android apps that are of interest to a human analyst, such as SMS messages and reflective call targets. Second, TDroid can help GUI testing tools improve their coverage.

ACKNOWLEDGEMENTS
We wish to thank all the reviewers for their constructive comments on an earlier version of this paper. This research is supported by an ARC grant DP170103956.


