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

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

Diyu Wu
UNSW Sydney, Australia
diyuwu@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.

CCS CONCEPTS

• **Security and privacy → Software and application security**;

KEYWORDS

Android GUI Security, Static Analysis, Dynamic Analysis

ACM Reference Format:
Jie Liu, Diyu Wu, and Jingling Xue. 2018. TDroid: Exposing App Switching Attacks in Android with Control Flow Specialization. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238188

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238188

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. FuzzDroid [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 goodware 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, possibly until a ransom has been paid.

2.3 Intermediate Representation (IR)

Given an app, TDroid works on its Jimple IR, a three-address bytecode 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 $s_1 \rightarrow^d s_2$ if s_2 is data-dependent on s_1 and $s_1 \rightarrow^c s_2$ 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

```
1  public class Notifications extends Service {
2    public void onCreate() {
3       super.onCreate();
4       checkStatus();}
5    private void checkStatus() {
6      int bk_type = -1;
7      String pack_name = "";
8      for (int j = 0; j < Config.BKLIST.length; j++)
9        if (getInstalledPkgNames().contains(Config.BKLIST[j])){
10         bk_type = j;
11         pack_name = Config.BKLIST[j];
12       }
13     Intent bkMain = new Intent(this, BKMain.class);
14     bkMain.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
15     bkMain.putExtra("PKGN", pack_name);
16     if (bk_type == 0){
17       bkMain.putExtra("BK", "0");
18       startActivity(bkMain);
19     }
20     ...
21     else if (bk_type == 4){
22       bkMain.putExtra("BK", "4");
23       startActivity(bkMain);
24     }
25 }}
26 public class BKMain extends Activity {
27   protected void onCreate(Bundle savedInstanceState) {
28     super.onCreate(savedInstanceState);
29     Intent bkIntent = getIntent();
30     if(bkIntent != null){
31       Logger.i("uninstall", bkIntent.getStringExtra("PKGN"));
32       switch (bkIntent.getStringExtra("BK")) {
33         case "0": setContentView(R.layout.kb_main_activity);
34                 return;
35         ...
36         case "4": setContentView(R.layout.sh_main_activity);
37                 return;
38       }
39 }}}
```

Figure 1: The malware app *Bankun*.

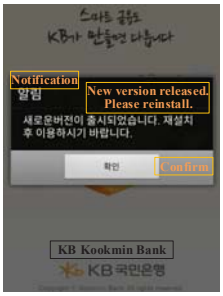

Figure 2: *Bankun*'s fake UI for software update (`bk_type=0`).

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 $\mathcal{I}_{18}$. 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 his own activities.

For $\mathcal{I}_{18}$, *Bankun* is activated by a call to `Notifications.onCreate()` (lines 2 – 4), a non-user-driven callback. Thus, $\mathcal{I}_{18}$ 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 $\mathcal{I}_{18}$ (Table 1), the opened activity can be the top activity. Thus, $\mathcal{I}_{18}$ is a suspicious call.

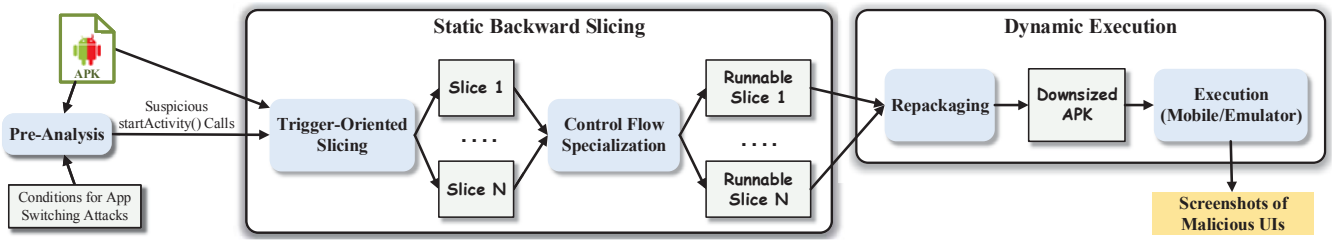

Figure 3: A schematic overview of the components in TDroid.

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 $\mathcal{I}_{18}$, we compute some backward slices affecting $\mathcal{I}_{18}$. 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 $\mathcal{SC}^{\texttt{BKMain}}(\mathcal{I}_{18}) = \{18, 33, 36\}$, 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 $\mathcal{SC}^{\texttt{BKMain}}(\mathcal{I}_{18})$ 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 $\mathcal{SC}^{\texttt{BKMain}}(\mathcal{I}_{18}) = \{18, 33, 36\}$, 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 $\mathcal{I}_{18}$, `bk_type == 0` in line 16 will be specialized with a Boolean constant to ensure that $\mathcal{I}_{18}$ 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 $\mathcal{SC}^{\texttt{BKMain}}(\mathcal{I}_{18}) = \{18, 33, 36\}$, with the initial slice being $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18}) = \mathcal{SC}^{\texttt{BKMain}}(\mathcal{I}_{18})$. At this stage, no branch predicates are specializable yet. Let us start from lines 33 and 36. As $32 \rightarrow^c 33$ and $32 \rightarrow^c 36$, line 32 is added to $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$. 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 $30 \rightarrow^c 32$, line 30 is added to $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$. 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 $17 \rightarrow^d 32$ is a data dependence (through the intent object associated with $\mathcal{I}_{18}$), line 17 is simply added to $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$. As $16 \rightarrow^c 17$ and $16 \rightarrow^c 18$, line 16 is added to $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$. Now, $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18}) = \{16, 17, 18, 30, 32, 33, 36\}$. 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 $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$. Note that the statements that define the values used at specializable predicates are ignored.

Finally, $13 \rightarrow^d 17$, $13 \rightarrow^d 18$, $29 \rightarrow^d 30$, and $29 \rightarrow^d 32$. So we obtain $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18}) = \{13, 16, 17, 18, 29, 30, 32, 33, 36\}$, with line 16 as the only predicate flagged as specializable.

```
Notifications.checkStatus:
8  label1: if true goto label3;
9          if true goto label2;
12 label2: goto label1;
13 label3: Intent bkMain = new Intent(this, BKMain.class);
16         if false goto label4;
17         bkMain.putExtra("BK", "0");
18         this.startActivity(bkMain);
19         return;
21 label4: if true goto label5;
24 label5: return;
BKMain.onCreate:
29         Intent bkIntent = getIntent();
30         if bkIntent == null goto label1;
32         switch(bkIntent.getStringExtra("BK")){
33         case "0": setContentView(R.layout.kb_main_activity);
34         return;
36         case "4": setContentView(R.layout.sh_main_activity);
37         return;
38         default: goto label1;};
39 label1: return;
```

Figure 4: The (simplified) slice for $\mathcal{I}_{18}$ in Soot's Jimple IR after control flow specialization. The statements in $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$ are highlighted in ■ with line 16 marked in ■. The statements in $\overline{\mathcal{T}^{\texttt{BKMain}}}(\mathcal{I}_{18}) \setminus \mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$ appear in the white background.

(2) Control Flow Specialization. Figure 4 displays the slice $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$ in Soot's Jimple IR, consisting of the statements highlighted in the red background (■). The branch predicate in line 16 (■) has been specialized so that line 18 is always triggered. This slice is not yet runnable, as some jump statements for ensuring its correct execution are missing.

Instead of complex algorithms for adding missing jumps [3, 26], TDroid adopts a simple yet effective approach. For every method

m in $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$, there are two steps. First, we add all missing control-flow statements in m to $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$ to obtain a runnable slice: $\overline{\mathcal{T}^{\texttt{BKMain}}}(\mathcal{I}_{18}) = \{8, 9, 12, 13, 16-19, 21, 24, 29, 30, 32-34, 36-39\}$. In Figure 4, the added statements in $\overline{\mathcal{T}^{\texttt{BKMain}}}(\mathcal{I}_{18}) \setminus \mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$ 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.

```
Notifications.checkStatus:
8   label1: if true goto label3;
9           if true goto label2;
12  label2: goto label1;
13  label3: Intent bkMain = new Intent(MainActivity.ctx, BKMain.class);
L1          bkMain.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
16          if false goto label4;
17          bkMain.putExtra("BK", "0");
18          MainActivity.ctx.startActivity(bkMain);
19          return;
21  label4: if true goto label5;
24  label5: return;
BKMain.onCreate:
L2          super.onCreate(savedInstanceState);
L3          ReachReport.reportTargetReached();
29          Intent bkIntent = getIntent();
30          if bkIntent == null goto label1;
32          switch(bkIntent.getStringExtra("BK")){
33          case "0": setContentView(R.layout.kb_main_activity);
34          return;
36          case "4": setContentView(R.layout.sh_main_activity);
37          return;
38          default: goto label1;};
39  label1: return;
```

Figure 5: A repackaged version of Figure 4 with all the changes shown in the blue background ( ).

3.2.3 Dynamic Execution. We repackage $\overline{\mathcal{T}^{\texttt{BKMain}}}(\mathcal{I}_{18})$ 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     new Notifications().checkStatus();
7 }}
```

Figure 6: The launcher activity of the downsized APK for $\mathcal{I}_{18}$.

(1) Repackaging. In Figure 4, `checkStatus()` is the entry method to activate $\mathcal{I}_{18}$. 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 $\mathcal{I}_{18}$ 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., `BroadcastRecevier.onReceive()`), and content providers (e.g., `ContentProvider.onCreate()`).

Bankun in Figure 1 can be activated to run 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).

Receiver Type of startActivity()	launchMode	taskAffinity	Intent Flags (FLAG_ACTIVITY_*)
Non-Activity	*	*	NEW_TASK
Activity	*	*	NEW_TASK & CLEAR_TASK
Activity	singleTop	*	NEW_TASK & MULTIPLE_TASK
Activity	standard	*	NEW_TASK & MULTIPLE_TASK
Activity	singleTask	*	CLEAR_TASK
Activity	singleInstance	*	*
Activity	singleTop	Non-default	NEW_TASK
Activity	standard	Non-default	NEW_TASK
Activity	singleTask	Non-default	*

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 $\mathcal{I}_{18}$ 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, $\mathcal{I}_{18}$ 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 dependences, 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 $\mathcal{I}$, our static analysis comes into play first. "Trigger-Oriented Slicing" first obtains a set $\mathcal{T}(\mathcal{I})$ of slices for $\mathcal{I}$, one per malicious activity that may be opened at $\mathcal{I}$ and "Control Flow Specialization" then expands it into a set $\overline{\mathcal{T}}(\mathcal{I})$ 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 $\mathcal{I}$, 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, $\mathcal{I}_{18}$ is suspicious.

5.2 Trigger-Oriented Slicing

Given a suspicious `startActivity()` call $\mathcal{I}$, Algorithm 1, named TOSlicer, generates a set $\mathcal{T}(\mathcal{I})$ of slices interprocedurally, one slice $\mathcal{T}^t(\mathcal{I})$ for each class type t associated with the malicious activities opened at $\mathcal{I}$. 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 $\mathcal{I}$ (line 3). For each class type representing suspicious activities (lines 4 and 5), we then set up its slicing criterion $\mathcal{SC}^t(\mathcal{I})$ (line 9) and call InterSlicer to compute interprocedurally a backward slice starting from all the points in $\mathcal{SC}^t(\mathcal{I})$, one at a time (lines 10 – 11). At this stage, all branch predicates are not specializable yet (lines 7 – 8). To set up $\mathcal{SC}^t(\mathcal{I})$, we include not only $\mathcal{I}$ but also all the UI-drawing calls (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 $\mathcal{T}^t(\mathcal{I})$ 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. $\mathcal{T}_m$ is the current slice for m (line 16). Let $s_1, \cdots, s_n$ be the n successors of s in the CFG of m, denoted G_m. Let $P(s_i)$ be the set of basic blocks, confined within $\mathcal{T}_m$, reachable from s_i, which includes s_i itself, in G_m. We know that s satisfies Property 1 when the following two conditions are met. (1) There exists only one unique $1 \leqslant i \leqslant n$ 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' in $P(s_i)$ (directly or indirectly if s' is a call site). Then s does not also read from a. This can be done based on the def-use chains (for local variables) and alias information (for field accesses).

Algorithm 1: Trigger-oriented slicing.

```
   Input  : I
   Output : T(I)
1  Function TOSlicer ()
2      T(I) = ∅;
3      SetofActTypes = set of types of the activities opened at I;
4      foreach t ∈ SetofActTypes do
5          if I is suspicious with respect to t (Section 4) then
6              T^t(I) = ∅;
7              foreach branch predicate s in CG do
8                  s.spec = false;
9              SC^t(I) = {I and all the UI drawing calls reachable in the
                          lifecycle callbacks of class t in CG};
10             foreach s ∈ SC^t(I) do
11                 T^t(I) ∪= InterSlicer (s, T^t(I));
12             T(I) ∪= {T^t(I)};
13     return T(I);
14 Procedure InterSlicer (s, T^t(I))
15     Let m be the method containing statement s;
16     T_m = {s'' is contained in m | s'' ∈ T^t(I)};
17     worklist = {s};
18     while worklist ≠ ∅ do
19         s = worklist.remove();
20         T_m.add(s);
21         foreach s' ∉ T_m in the CFG of m do
22             if s' →^d s ∧ (s is not a predicate ∨ !s.spec) then
23                 worklist.add(s');
24             else if s' →^c s then
25                 worklist.add(s');
26                 if s' satisfies Property 1 then
27                     s'.spec = true;
28         foreach s' ∈ {s'' ∈ T_m | s''.spec = true} do
29             if s' does not satisfy Property 1 then
30                 s'.spec = false;
31                 worklist.add(s');
32     T^t(I) ∪= T_m;
33     if any statement in T_m use any of m's parameters then
34         SetOfCallSites = set of call sites for m in CG;
35         foreach s' ∈ SetOfCallCites do
36             T^t(I) ∪= InterSlicer(s', T^t(I));
37     return T^t(I);
```

In our example, the class type of the activities opened at $\mathcal{I}_{18}$ is `BKMain`. Given that $\mathcal{SC}^{\texttt{BKMain}}(\mathcal{I}_{18}) = \{18, 33, 36\}$, the final slice $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$ computed is given in Section 3.2.2, where line 16 is specializable. Thus, $\mathcal{T}(\mathcal{I}_{18}) = \{\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})\}$.

5.3 Control Flow Specialization

Given a slice $\mathcal{T}^t(\mathcal{I})$, Algorithm 2 transforms and expands it into a runnable slice $\overline{\mathcal{T}^t}(\mathcal{I})$ by performing two flavors of control flow specialization. In our example, Figure 4 demonstrates the transition from $\mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$ to $\overline{\mathcal{T}^{\texttt{BKMain}}}(\mathcal{I}_{18})$ for $\mathcal{I}_{18}$.

In lines 2 – 3, we specialize every specializable branch predicate identified in $\mathcal{T}^t(\mathcal{I})$ to always execute the only branch contained

Algorithm 2: Control flow specialization.

```
   Input  : T^t(I)
   Output : T̄^t(I)
1  Function CFSpec ()
2      foreach branch predicate s in T^t(I) s.t. s.spec = true do
3          Specialize it to execute its unique branch in T^t(I);
4      T̄^t(I) = T^t(I);
5      foreach method m, contained in T^t(I), in CG do
6          Add the missing control flow statements of m to T̄^t(I);
7          foreach each thus added branch predicate s do
8              if s represents a branch predicate for a loop then
9                  Specialize s to skip its associated loop body;
10             else if s reaches (along any of its branches, say, b) a
                       statement of m in T^t(I) then
11                 Specialize s to execute b;
12             else
13                 Specialize s to execute any of its branches;
14     return T̄^t(I);
```

in $\mathcal{T}^t(\mathcal{I})$. In our example, `bk_type == 0` in line 16 is specializable, forcing $\mathcal{I}_{18}$ 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 $\mathcal{T}^t(\mathcal{I})$ so that $\overline{\mathcal{T}^t}(\mathcal{I})$ is runnable. In Figure 4 (for $\mathcal{I}_{18}$), all such newly added control flow statements in $\overline{\mathcal{T}^{\texttt{BKMain}}}(\mathcal{I}_{18}) \setminus \mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$ are shown in the white background. For every newly introduced branch predicate, which is guaranteed not to control any statement in $\mathcal{T}^t(\mathcal{I})$ by construction, it is specialized depending on which of the three cases it falls into (lines 7 – 13). In Figure 4 (for $\mathcal{I}_{18}$), $\overline{\mathcal{T}^{\texttt{BKMain}}}(\mathcal{I}_{18}) \setminus \mathcal{T}^{\texttt{BKMain}}(\mathcal{I}_{18})$ 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 $\overline{\mathcal{T}^t}(\mathcal{I})$, we will repackage it into an APK file. There are four tasks, illustrated using our example.

First, we create the launcher activity, `MainActivity`, for $\overline{\mathcal{T}^t}(\mathcal{I})$. The one for $\overline{\mathcal{T}^{\texttt{BKMain}}}(\mathcal{I}_{18})$ is given in Figure 6. Let $m_1, \cdots, m_n$ be the predecessor-less methods (regardless of their access modifiers) backwards reachable from $\mathcal{I}$ 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 $\overline{\mathcal{T}^{\texttt{BKMain}}}(\mathcal{I}_{18})$, 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 $\mathcal{I}$, we replace it by a context object of type `Context` stored in a global variable, `MainActivity.ctx`, as demonstrated for $\mathcal{I}_{18}$ in Figure 6 (line 5) and Figure 5 (lines 13 and 18). For $\overline{\mathcal{T}^t}(\mathcal{I})$, an opened activity of type t can be the top activity. Thus, we simply set the attribute of the intent associated with $\mathcal{I}$ as NEW_TASK (e.g., line L1 for $\mathcal{I}_{18}$ 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 $\overline{\mathcal{T}^t}(\mathcal{I})$, we examine each of the lifecycle callbacks in class t. We add each missing super call (e.g., line L2 for $\mathcal{I}_{18}$ in Figure 5) to $\overline{\mathcal{T}^t}(\mathcal{I})$.

Finally, we inject code into the lifecycle callback `onCreate()` of class t for $\overline{\mathcal{T}^t}(\mathcal{I})$, the entry method for the opened activity of type t, in order to capture the malicious UI shown. For $\mathcal{I}_{18}$, 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 $\mathcal{I}_{18}$ 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 `startActivity()` call are tracked. For example, in Section 3.2.2, $17 \rightarrow^{d} 32$ is found by linking the use of "BK" at `getStringExtra()` in `BKMain.oncreate()` (line 32) with its definition in `putExtra()` in `Notifications.onCreate()` (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.

TDroid 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 (3.20GHz Intel Xeon(R) CPU) and 256GB 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% ($\frac{978}{1062}$) of the 1062 suspicious apps to completion, with 82.7% ($\frac{878}{1062}$) 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).

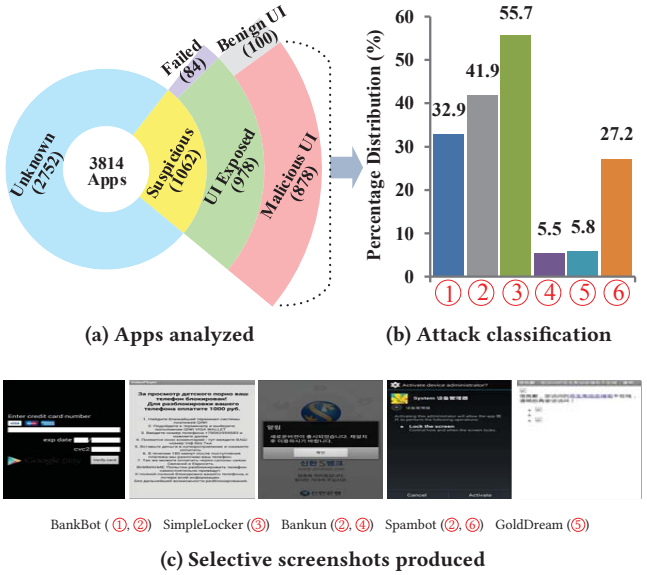

Figure 7: The TDroid results. The app switching attacks fall into six categories: ① sensitive data theft, ② phishing (UI spoofing), ③ loss of availability, ④ malware infection, ⑤ suspicious webviews, and ⑥ admin privilege escalation.

Let us examine the results in Figure 7 in more detail. Figure 7(a) divides the 878 malicious apps into six categories, with their percentage distributions shown in Figure 7(b) and their representative malicious UIs in Figure 7(c). "Loss of Availability" is the mostly dominant attack category (55.7%), followed by "Phishing" (41.9%) and then "Sensitive Data Theft" (32.9%). Some malware apps (27.2%) exploit an "Admin Privilege Escalation" attack to lure users to give their admin privileges to prevent them from being uninstalled. Some malware apps (5.5%), e.g., *Bankun*, attempt to inject new malware apps into the compromised devices. Others (5.8%) display malicious webviews, with possibly malicious URLs.

To examine TDroid further, we have randomly picked one sample from each of the 20 families containing suspicious `startActivity()` calls. Table 2 shows the 20 malware samples selected (Column 1), our results (Columns 3 – 6), and the types of attacks launched (Column 7). TDroid finds statically 58 suspicious `startActivity` calls and exposes dynamically their associated UIs in 55 of these calls, in under 55 minutes (with an average of about one minute per call). For the remaining 3 calls, their corresponding slices are incomplete (as mentioned above). For one in `FakeAV` and one in `SlemBunk`, some data dependencies through `SharedPreferences` are missing. For another one in `SlemBunk`, its associated intent object is incomplete.

For the 55 `startActivity` calls with exposed UIs, 51 are malicious, and the remaining four, which reside in `FakeDoc`, `FakePlayer`, `Nandrobox`, and `Zitmo`, are benign. Thus, the other 16 apps are malicious, capable of launching app switching attacks.

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

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

APK	MD5	TDroid				Attack Types
		# of Suspicious `startActivity()` Calls	# of Calls with Exposed UIs	# of Malicious Calls	Time (secs)	
Aples	0c9eccbd1f0ed4ed29178f5b2f785b7f	2	2	**2**	76.04	②, ⑥
BankBot	2ba4663a14899854fff429a6d6035cc2	4	4	**4**	117.47	①, ②, ⑥
Bankun	110001afe867f8e1e79790b2336ec96d	5	5	**5**	153.97	②, ④
FakeAV	418b5b165ac8b2d9611a4c7acdff0c8c	5	*4*	**4**	874.02	①, ②
FakeDoc	063846036d6b5d45998879ae2cc142ab	1	1	0	54.16	
FakePlayer	2337a421045aaebe2be6497dab822826	1	1	0	44.49	
Fusob	63c980138792a5fab4fd6e5f1ac6a938	3	3	**3**	95.91	②, ③, ⑤, ⑥
GoldDream	31ff6d0c28cae40d2e97fa7770973fdd	1	1	**1**	149.78	⑤
Gumen	d30e857aaba02275c54b76badb1bc855	1	1	**1**	85.02	②, ④
Jisut	004d0c40db80d291804c44d531bf7b64	2	2	**2**	193.94	③
Koler	1e341a93ea251267b7af6548f49752f5	9	9	**9**	277.43	③, ⑤
Nandrobox	4d871daec6a4270a70f42800f9a932f2	1	1	0	46.62	
Roop	7584538eb727cc7acf57c3b1b55ff8e3	1	1	**1**	157.96	③
SimpleLocker	0b7e470f80205786e00e277fc1d4c224	1	1	**1**	31.63	③
SlemBunk	41c9add2be9f4b04047ab232eb08058f	12	*10*	**10**	460.16	①, ②, ⑥
SmsZombie	0e5dd82f8aeddc76160576c512804930	2	2	**2**	60.03	②, ④
Spambot	0c5b3709f970af266393d0206633dec6	1	1	**1**	136.48	②, ⑥
Svpeng	9a0617349c91000b04a0cf887c63c40e	4	4	**4**	129.26	①, ②
Triada	de52d8cbccdb32dbdf576966031bf2bd	1	1	**1**	103.21	⑥
Zitmo	003bb03ec84313dc73a86af3876a128d	1	1	0	33.97	
Total		**58**	**55**	**51**	3281.55	

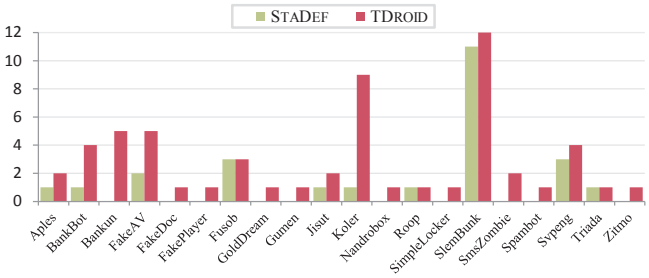

Figure 8: Comparing TDroid and StaDef on detecting suspicious startActivity() calls in the 20 selected malware apps.

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`, `Gumen`, `SimpleLocker`, `SmsZombie` and `Spambot`. (Recall that TDroid finds no attacks in `FakeDoc`, `FakePlayer`, `Nandrobox` and `Zitmo`.)

6.3 RQ3: General-Purpose Detection

We compare TDroid with four representative general-purpose tools, Google's Monkey (randomized) [15], Sapienz (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: Comparing TDroid with Google's Monkey [15], Sapienz) [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).

Tool	% of Suspicious Calls Reached
Monkey	8.62
Sapienz	10.34
Stoat	12.07
FuzzDroid	18.97
TDroid	**94.83**

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 Soot framework. As reflection is handled only partially in Soot, 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. CredMiner [61] proceeds similarly to study the prevalence of unsafe developer credential uses. However, it executes slices in a custom engine, requiring a precise model of the Android OS and libraries. Harvester [39] 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.

REFERENCES

[1] 2014. *ASEC Report.* http://download.ahnlab.com/asecReport/ASEC_Report_Vol.55_Kor.pdf.

[2] 2016. *Internet Security Threat Report.* https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf.

[3] Hiralal Agrawal. 1994. On Slicing Programs with Jump Statements. In *Proceedings of the Conference on Programming Language Design and Implementation.*

[4] Matthew Allen and Susan Horwitz. 2003. Slicing Java Programs That Throw and Catch Exceptions. In *Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation.*

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[6] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *Proceedings of the International Workshop on State of the Art in Java Program Analysis.*

[7] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. 2003. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation.*

[8] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. 2015. What the App is That? Deception and Countermeasures in the Android User Interface. In *Proceedings of IEEE Symposium on Security and Privacy (S&P).*

[9] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proceedings of the USENIX Security Symposium.*

[10] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the ACM SIGPLAN International Conference on Object-oriented Programming, Systems, Languages, and Applications.*

[11] F-Droid. 2018. *Free and Open Source App Repository.* https://f-droid.org/.

[12] Adrienne Porter Felt and David Wagner. 2011. Phishing on mobile devices. In *Web 2.0 Security and Privacy.*

[13] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. 2017. Automated Synthesis of Semantic Malware Signatures using Maximum Satisfiability. In *Proceedings of Network and Distributed System Security Symposium (NDSS).*

[14] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J Alex Halderman, Z Morley Mao, and Atul Prakash. 2016. Android UI Deception Revisited: Attacks and Defenses. In *Proceedings of Financial Cryptography and Data Security (FC).*

[15] Google. 2017. *UI/Application Exerciser Monkey.* https://developer.android.com/studio/test/monkey.html.

[16] Neville Grech, George Kastrinis, and Yannis Smaragdakis. 2018. Efficient Reflection String Analysis via Graph Coloring. In *Proceedings of the European Conference on Object-Oriented Programming.*

[17] Joji Hamada. 2014. *Simplocker: First Confirmed File-Encrypting Ransomware for Android.* https://www.symantec.com/connect/blogs/simplocker-first-confirmed-file-encrypting-ransomware-android.

[18] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services.*

[19] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization.*

[20] Mark Harman and Sebastian Danicic. 1998. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance* 10, 6 (1998), 415–441.

[21] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and Detecting Evolution-induced Compatibility Issues in Android Apps. In *Proceedings of the International Conference on Automated Software Engineering (ASE).*

[22] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 12, 1 (1990), 26–60.

[23] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor. In *Proceedings of the Ninth European Conference on Computer Systems.*

[24] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and Precise Taint Analysis for Android. In *Proceedings of the International Symposium on Software Testing and Analysis.*

[25] Ranjit Jhala and Rupak Majumdar. 2005. Path slicing. In *Proceedings of the conference on Programming language design and implementation (PLDI).*

[26] Sumit Kumar and Susan Horwitz. 2002. Better slicing of programs with jumps and switches. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering.*

[27] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the International Conference on Compiler Construction.*

[28] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the International Conference on Software Engineering.*

[29] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the Performance of Flow-sensitive Points-to Analysis Using Value Flow. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering.*

[30] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and Scalable Context-sensitive Pointer Analysis via Value Flow Graph. In *Proceedings of the International Symposium on Memory Management.*

[31] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-inferencing reflection resolution for Java. In *Proceedings of the European Conference on Object-Oriented Programming.*

[32] Yue Li, Tian Tan, and Jingling Xue. 2015. Effective Soundness-Guided Reflection Analysis. In *Proceedings of the International Static Analysis Symposium.*

[33] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program tailoring: Slicing by sequential criteria. In *Proceedings of the European Conference on Object-Oriented Programming.*

[34] Jie Liu, Yue Li, Tian Tan, and Jingling Xue. 2017. Reflection Analysis for Java: Uncovering More Reflective Targets Precisely. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE).*

[35] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the Joint Meeting on Foundations of Software Engineering.*

[36] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering.*

[37] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the International Symposium on Software Testing and Analysis.*

[38] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the International Conference on Software Engineering.*

[39] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques. In *Proceedings of Network and Distributed System Security Symposium (NDSS).*

[40] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments. In *Proceedings of the International Conference on Software Engineering.*

[41] Chuangang Ren, Peng Liu, and Sencun Zhu. 2017. WindowGuard: Systematic Protection of GUI Security in Android. In *Proceedings of Network and Distributed System Security Symposium (NDSS).*

[42] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards Discovering and Understanding Task Hijacking in Android. In *Proceedings of the USENIX Security Symposium.*

[43] Franziska Roesner and Tadayoshi Kohno. 2013. Securing Embedded User Interfaces: Android and Beyond. In *Proceedings of the USENIX Conference on Security.*

[44] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2017. PATDroid: Permission-aware GUI Testing of Android. In *Proceedings of the Joint Meeting on Foundations of Software Engineering.*

[45] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.*

[46] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin Slicing. In *Proceedings of the conference on Programming language design and implementation (PLDI).*

[47] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the Joint Meeting on Foundations of Software Engineering.*

[48] Yulei Sui and Jingling Xue. 2016. On-demand Strong Update Analysis via Value-flow Refinement. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering.*

[49] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Proceedings of the International Static Analysis Symposium.*

[50] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[51] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research.*

[52] Jaap Vermeij. 2016. *Alerting Users on Android: The Effect of an Alert During a GUI Confusion Attack.*

[53] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[54] Mark David Weiser. 1979. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. (1979). PhD thesis.

[55] Daniel Xiang. 2013. *Android.Bankun.* https://www.symantec.com/security_response/writeup.jsp?docid=2013-072318-4143-99&tabid=2.

[56] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android applications. In *Proceedings of the International Conference on Software Engineering*.

[57] Lingyun Ying, Yao Cheng, Yemian Lu, Yacong Gu, Purui Su, and Dengguo Feng. 2016. Attacks and Defence on Android Free Floating Windows. In *Proceedings of the Asia Conference on Computer and Communications Security*.

[58] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*.

[59] Yifei Zhang, Yue Li, Tian Tan, and Jingling Xue. 2018. Ripple: Reflection analysis for Android apps in incomplete information environments. *Software: Practice and Experience* 48, 8 (2018), 1419–1437.

[60] Yifei Zhang, Yulei Sui, and Jingling Xue. 2018. Launch-Mode-Aware Context-Sensitive Activity Transition Analysis. In *Proceedings of the International Conference on Software Engineering*.

[61] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. 2015. Harvesting Developer Credentials in Android Apps. In *Proceedings of the ACM Conference on Security & Privacy in Wireless and Mobile Networks*.