All about Activity Injection: Threats, Semantics, and Detection

Sungho Lee
KAIST, Korea
eshaj@kaist.ac.kr

Sungjae Hwang
LG Electronics, Korea
sungje.hwang@lge.com

Sukyoung Ryu
KAIST, Korea
sryu.cs@kaist.ac.kr

***Abstract*—Android supports seamless user experience by maintaining activities from different apps in the same activity stack. While such close inter-app communication is essential in the Android framework, the powerful inter-app communication contains vulnerabilities that can inject malicious activities into a victim app's activity stack to hijack user interaction flows. In this paper, we demonstrate activity injection attacks with a simple malware, and formally specify the activity activation mechanism using operational semantics. Based on the operational semantics, we develop a static analysis tool, which analyzes Android apps to detect activity injection attacks. Our tool is fast enough to analyze real-world Android apps in 6 seconds on average, and our experiments found that 1,761 apps out of 129,756 real-world Android apps inject their activities into other apps' tasks.**

I. Introduction

Smartphones have changed many facets of everyday life immensely. People read books, listen to music, browse websites, purchase goods, and send emails via smartphones without restrictions of locations or time. Researchers report that more than a half of smartphone users have used phones for online banking apps and for monitoring their health conditions [1]. Thus, smartphones maintain various kinds of sensitive data, which makes the security of smartphones very important.

Compared to desktop apps, smartphone apps provide diverse functionalities with seamless user experience by heavily interacting with other apps. For example, the Gallery app uses the Email app to email pictures rather than having its own email functionality. While users may think that a single app manages pictures and sends emails, the actual implementation consists of two independent apps using inter-app communication. To support frequent transitions between apps, Android provides powerful multitasking by managing app activities [2] in a stack called a *task* [3], and it refers the task to find an activity to display. The task is an essential component to support user interaction transitions smoothly by maintaining activities from different apps in the same activity stack.

However, the powerful inter-app communication contains vulnerabilities that may break the sandbox environment of Android. Even though each Android app resides in its own security sandbox, the ability of starting another app introduces a vulnerability that can inject malicious activities into the victim app's activity stack to hijack user interaction flows. The injected malicious activities can run in the context of a vulnerable victim app, and thus can launch various attacks including phishing [4].

Recently, researchers reported security issues in task behaviors with a proof-of-concept implementation of *task hijacking attacks* [4]. They define these attacks as "malware [that] reside side by side with the victim apps in the same task and hijack the user sessions of the victim apps" task hijacking. While the task hijacking attacks include true malicious actions, they identify too many "normal" actions as vulnerable; among the most popular 10,985 apps from Google Play, they identified 93.9% of them as vulnerable, because they consider as vulnerable as well even such cases where users intentionally select particular activities. More importantly, because they used a simplified task state transition model considering only two activities per app and one instantiation per activity, the model does not capture the actual task and activity behaviors correctly, which leads to missing true malicious attacks.

In this paper, we present a class of critical task hijacking attacks, *activity injection attacks*. To reduce the amount of false alarms from identified task hijacking attacks, we define attacks that inject malicious activities into normal tasks as activity injection attacks. To understand the root cause of the attacks, we formally specify the operational semantics of activity activation. The formalized semantics captures every task behavior described in the Android developers reference [5] without any constraints to limit the scope of task behaviors. As a showcase of activity injection attacks, we developed a malware targeting the Facebook app. Our malware injects a malicious activity into the task of Facebook so that Android displays the malicious activity when users execute Facebook by clicking the Facebook icon. This kind of attacks is extremely powerful because it does not require any permission, and it is difficult for users to detect them because the attacks are launched when the users click legitimate apps' icons.

Based on our thorough understanding of the activity activation semantics, we developed a static analyzer that detect activity injection attacks in Android apps. Our experiments show that the tool is fast enough to analyze real-world Android apps in 6 seconds on average. We found that 1,761 apps out of 129,756 Android apps have possible injection attacks.

The contributions of this paper include the following:

- We present *the first formal semantics of the Android activity activation mechanism*. Considering all the status of activity attributes and flags for inter-app communication, we specify all possible task behaviors rigorously and exhaustively.

978-1-5386-2684-9/17/$31.00 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

- We identify *the root causes of activity injection attacks*. The formal semantics specifies all possible ways of activity injection attacks, and we show an example malware that injects a malicious activity into a benign app's task.
- We developed *a static analysis tool that detects possible activity injection attacks* based on the formal semantics. The tool is efficient enough to detect possible activity injection attacks in a large number of Android apps.
- We found and analyzed *a number of possible activity injection cases in real-world Android apps*.

In the rest of this paper, we first provide a brief background about how Android apps communicate with other apps by sharing activities (Section II). After formally specifying the activity activation mechanism in Android and showing its security vulnerability with a sample malware (Section III), we present a tool that detects activity injection attacks automatically (Section IV). We show that our tool can detect activity injection attacks from real-world Android apps efficiently (Section V). We discuss related work (Section VI) and conclude (Section VII).

II. Background

A. Android App Overview

Android apps reside in their own security sandboxes within an Android system [6]. Android treats each app as a different user so that each app executes in isolation from others. It provides a secure environment because each app has access only to its components with permission. An app has its own manifest file (`AndroidManifest.xml`) that declares components and required device features for permission. An Android app consists of *app components* that can be invoked individually. An app component is one of four types: *activities*, *services*, *content providers*, and *broadcast receivers*. Each type serves a different role and has a different lifecycle. For example, an activity denotes a single screen with a user interface like one for reading and one for writing emails. Android apps communicate with others via *intents*. An intent is an asynchronous message that activates activities, services, and broadcast receivers. Intents can start one component from another even in a different app. For example, Gallery can start an activity in Email to send emails by calling `startActivity()`. To share data with others and to access system services, Android apps can request permission to access them.

B. Activities and Tasks

Among four types of components, we focus on activities. An activity is an instance of `android.content.Activity` [2]. An app consists of multiple activities; when a user launches an app, one designated "main" activity starts and it can start another activities. The Android system manages running activities in a stack called a *back stack*, and it calls a pair of a name and a back stack a *task*. When obvious, we use back stacks and tasks interchangeably. The main activity is pushed at the bottom of the task as the "root activity." Whenever a new activity starts, the previous activity is stopped, the new activity is pushed onto the task, and the user focus is with the new activity. When the user presses the Back button, the current activity is popped from the task and the previous activity resumes. The topmost activity in the task is called a "top activity" and it is the screen that a device displays.

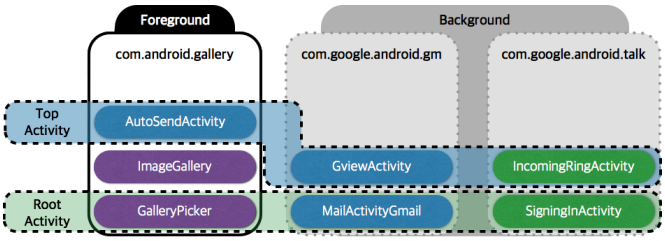


Fig. 1. When Gallery `com.android.gallery` starts an activity to email a picture, the `AutoSendActivity` activity of Email `com.google.android.gm` is pushed onto the task of Gallery.

Figure 1 illustrates that Gallery `com.android.gallery`, Email `com.google.android.gm`, and a chat app `com.google.android.talk` are executing on an Android device. Each app has its own task; Gallery receives user interaction in the foreground, while the other Email and chat apps are suspended in the background, waiting to be resumed. Each task has its own root activity and top activity. When Gallery is launched, the main activity `GalleryPicker` is pushed onto the task of Gallery as its root activity. When a user navigates images, Gallery starts the `ImageGallery` activity and pushes it onto its task suspending the `GalleryPicker` activity, which makes `ImageGallery` a new top activity. Then, a user emails a picture by starting `AutoSendActivity`, which becomes a new top activity. Note that `AutoSendActivity` is an activity of Email; Gallery uses an activity of Email to email pictures rather than having its own email functionality.

The Android system provides various ways to manage activities using attributes in the `<activity>` manifest element and flags in the intent that are passed to the `startActivity()` function. For example, the `launchMode` attribute in the `<activity>` manifest element specifies one of four modes to launch an activity: `standard`, `singleTop`, `singleTask`, and `singleInstance`. The default mode is `standard`, and the manifest file may define launch modes of activities. A `standard` or `singleTop` activity can be instantiated multiple times leading to duplicated activities in a task. In contrast, an activity with the `singleTask` or `singleInstance` launch mode should be instantiated only once. Furthermore, an activity with the `singleInstance` launch mode is always the root activity of a task. While a `singleTask` activity can contain other `standard` or `singleTop` activities in its task, a `singleInstance` activity does not contain any other activities in its task. It is the only activity in its task; if it starts another activity, that activity is assigned to a different task. Because the `singleTask` and `singleInstance` launch modes provide a different interaction model from most other apps, they are not appropriate for most apps.

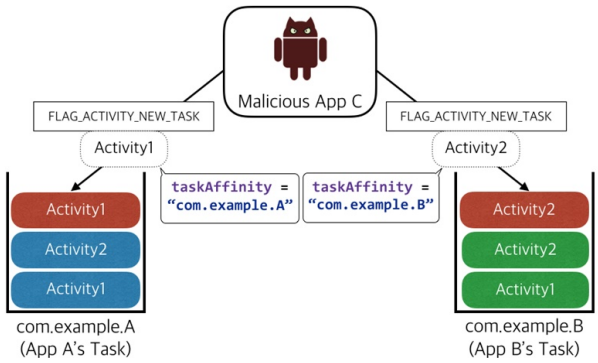


Fig. 2. The malicious app `C` injects its malicious activities into the tasks of victim apps `A` and `B` by specifying `taskAffinity` of the malicious activities with the victim apps' package names and setting the `FLAG_ACTIVITY_NEW_TASK` intent flag.

C. Activity Injection Attacks

Ren *et al.* [4] reported that the Android multitasking features allow *task hijacking attacks*: letting "malware reside side by side with the victim apps in the same task and hijack the user sessions of the victim apps." Because their definition of task hijacking attacks is too broad, they conservatively identify normal actions as vulnerable, which leads to impractically many false positives. For example, 93.9% of the most popular 10,985 apps from Google Play are identified as vulnerable, because they take activities of other apps to their tasks, which is a normal inter-app communication in Android.

In this paper, we focus on *activity injection attacks*, which inject activities of one app into another app's task. Unlike task hijacking attacks that consider activities in different apps' tasks, activity injection attacks focus on injection cases for phishing. For example, Figure 2 shows two tasks for the apps `A` and `B`. Every app has its own task and, by default, the name of the task is the app's package name; `A`'s activities are stored in the task `com.example.A`, `A`'s package name, and `B`'s activities are stored in the task `com.example.B`. While each app maintains its own task, it is possible for adversaries to inject their malicious activities into other legitimate app's task.

As we discuss in the next section, there are 220 ways to make activity injection attacks. One such a way is to first specify the special attributes `taskAffinity` and `launchMode` of a malicious activity in the `<activity>` manifest element with a victim app's package name and `standard`, respectively. Then, to call the `startActivity()` function for the malicious activity with the `FLAG_ACTIVITY_NEW_TASK` intent flag set. Because the Android system refers `taskAffinity` to decide where to put activities, adversaries can use the legitimate app's package name for `taskAffinity` of the malicious activity. Figure 2 depicts that a malicious app `C` injects its malicious activities into the tasks of `A` and `B` by specifying `taskAffinity` with the victim apps' package names and setting the `FLAG_ACTIVITY_NEW_TASK` intent flag. Note that activity injection attacks do not require any permission.

TABLE I
FOUR KINDS OF THE `launchMode` ATTRIBUTE

`launchMode`	**Description**
`standard`	Make a new instance whenever this activity is activated.
`singleTop`	Make a new instance only when the top activity of the target task is not an instance of this activity.
`singleTask`	Make a new instance when there does not exist any instance of this activity in any task.
`singleInstance`	Make a new instance when there does not exist any instance of this activity in any task. It is always the single activity in the task.

III. ACTIVITY ACTIVATION MECHANISM

In this section, we describe the overview of the activity activation mechanism, and specify the mechanism as an operational semantics, which spells out all possible task behaviors rigorously and exhaustively. Using the semantics, we present the threats of activity injection attacks with an example malware that injects a malicious activity into Facebook app's task.

A. Activity Activation Overview

1) Tasks and Task Stack: The Android system may have multiple tasks and one task stack. A task is a logical component for showing a job to users, and a task stack is a stack for managing the current foreground task and previous background tasks. A task has its own back stack to manage activities. When the task comes to the foreground, the top activity in its back stack is displayed on the device screen. When an activity finishes, the Android system pops it from the back stack of the task. If the back stack is not empty, the new top activity is displayed on the screen. Otherwise, the task itself finishes. The task stack behaves similarly with back stacks but it manages tasks. The top task in the task stack is the foreground task, which interacts with users. When a task finishes, the Android system pops it from the task stack, and the new top task, if any, is displayed on the screen. If the task stack is empty, the Home screen comes to the foreground. Also, when the Home screen comes to the foreground, the task stack is always empty.

Tasks may be alive after being popped from the task stack. The task stack manages which task to display after finishing the foreground task. When a user presses the Home button, the task stack becomes empty and the Home screen comes to the foreground. Then, all live tasks popped from the task stack are still pending, and they may come back to the task stack when a user selects them from the overview screen [7].

2) Activity Activation Properties: Developers can specify how to activate target activities using various Android properties. Two kinds of properties mainly govern the activity activation mechanism: the `launchMode` attribute in the `<activity>` manifest element and intent flags used to start activities. While `launchMode` is specified in the manifest file for an activity class, intent flags are set by caller activities to declare how to activate target activities by calling `startActivity()` with the intent flags as its arguments.

Table I describes four kinds of `launchMode`. Developers may set only one of them for each activity, and the

```
1  Intent i = new Intent(CallerActivity.this,
2                        TargetActivity.class);
3  i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
4  startActivity(i);
```

(a) Java code for activation of `TargetActivity`.

```
1  <activity android:name=".TargetActivity"
2            android:taskAffinity="com.facebook.katana"
3            android:launchMode="standard">
4  </activity>
```

(b) `TargetActivity`'s `taskAffinity` and `launchMode`.

Fig. 3. Code snippet for activity activation

$$
\begin{array}{lcll}
s & \in & \textit{String} & \text{task name, } \texttt{taskAffinity} \text{ attribute} \\
a & \in & \textit{ActivityInstance} & \text{activity instance} \\
A & \in & \textit{ActivityClass} & \text{activity class} \\
t & \in & \textit{Task} = \textit{String} \times \textit{BackStack} & \\
l & \in & \textit{LaunchMode} = \{ \texttt{standard}, \texttt{singleTop}, \texttt{singleTask}, \texttt{singleInstance} \} & \\
F & \subseteq & \textit{IntentFlag} = \{ \texttt{FLAG\_ACTIVITY\_CLEAR\_TASK}, \texttt{FLAG\_ACTIVITY\_CLEAR\_TOP}, \texttt{FLAG\_ACTIVITY\_MULTIPLE\_TASK}, \texttt{FLAG\_ACTIVITY\_NEW\_TASK}, \texttt{FLAG\_ACTIVITY\_REORDER\_TO\_FRONT}, \texttt{FLAG\_ACTIVITY\_SINGLE\_TOP}, \texttt{FLAG\_ACTIVITY\_TASK\_ON\_HOME} \} & \\
\alpha & ::= & \epsilon \mid (a,l)::\alpha & \in \textit{BackStack} \\
\beta & ::= & \epsilon \mid t::\beta & \in \textit{TaskStack} \\
\gamma & ::= & \epsilon \mid t::\gamma & \in \textit{TaskPool} \\
C & ::= & \texttt{HomeButton} & \\
 & \mid & \texttt{BackButton} & \\
 & \mid & (a,l).\texttt{startActivity}(A,l,s,F) &
\end{array}
$$

Fig. 4. Domains, stacks, and commands

default `launchMode` is `standard`. While the started target activity is put on the task of the caller activity in most cases, developers can designate a different task for the target task. For example, Figure 3(a) shows Java code that activates `TargetActivity` from `CallerActivity` and Figure 3(b) shows `taskAffinity` and `launchMode` of `TargetActivity` defined in `AndroidManifest.xml`. In the Java code, an `Intent` object is created with a caller activity and a callee activity, and `FLAG_ACTIVITY_NEW_TASK` is set to the intent object. Because `taskAffinity` of `TargetActivity` is `com.facebook.katana` and `launchMode` is `standard`, `TargetActivity` is injected to the task of `com.facebook.katana` when `startActivity` is invoked with the intent object. Since the default value of `taskAffinity` is the package name of the app, an activity is not injected to other task when `taskAffinity` of the activity is not specified.

Among 31 intent flags, 19 flags contain `ACTIVITY` in their names [8] but only 7 flags listed in *IntentFlag* in Figure 4 actually manage the activity activation mechanism. For example, if the `FLAG_ACTIVITY_NO_HISTORY` flag is set, the Android system does not show the activity on the overview screen, and as soon as the user navigates away from it, the activity finishes. Thus, the flag does not affect the activity activation mechanism in any ways.

Because a caller activity's `launchMode` also affects activity activation, the number of ways to activate activities is:

$$
\begin{aligned}
&\#\text{ of caller activity's } \texttt{launchMode} \times \\
&\#\text{ of target activity's } \texttt{launchMode} \times \\
&\#\text{ of all combinations of intent flags} = \\
&2 \times 4 \times ({}_0C_7 + {}_1C_7 + {}_2C_7 + {}_3C_7 + {}_4C_7 + {}_5C_7 + {}_6C_7 + {}_7C_7) = 1024
\end{aligned}
$$

While the activity activation mechanism behaves differently for all 4 kinds of `launchMode` for target activities, it behaves differently for only 2 cases for caller activities: whether a caller activity's `launchMode` is `singleInstance` or not. If a caller activity's `launchMode` is `singleInstance`, the activity activation mechanism behaves as if the intent flags of the target activity contain `FLAG_ACTIVITY_NEW_TASK`. For example, if a caller activity's `launchMode` is not `singleInstance`, the `FLAG_ACTIVITY_MULTIPLE_TASK` intent flag is ignored if one of `FLAG_ACTIVITY_NEW_TASK` or `FLAG_ACTIVITY_NEW_DOCUMENT` is not also set [8]. However, if a caller activity's `launchMode` is `singleInstance`, `FLAG_ACTIVITY_MULTIPLE_TASK` is not ignored even when `FLAG_ACTIVITY_NEW_TASK` is not also set. The Android document just describes that `singleInstance` does not allow other activities to be in the same task.

Among 1024 ways to activate activities, we can remove infeasible or redundant ones by considering dependencies between `launchMode` and intent flags. Some flags should be used together, some flags implicitly imply other flags, and some flags should not be used with other flags or some `launchMode`. When conflicting flags are used together, Android selects one of them and ignores the others. When flags are used without other required flags, Android ignores them. Figure 5 shows all dependencies between 4 `launchMode` attributes of target activities and 7 intent flags. Circle nodes denote `launchMode` and box nodes denote intent flags. Three types of edges denote dependencies between nodes: 1) solid *must* edges denote that the property of the *from* node should be used with the property of the *to* node, 2) dashed *imply* edges denote that the from node implies the to node, and 3) dotted *ignore* edges denote that the to node is ignored when it is used with the from node. Because each activity can have only one `launchMode`, no dependency exists between `launchMode` attributes. We built the dependency graph via thorough analysis of the Android documentation [6] and our own tests with Android devices. Using the dependency graph, we removed 791 out of 1024, leaving only 233 combinations to consider. Our manual inspection showed that among 233, 220 ways can inject activities to other tasks if a target activity's `taskAffinity` does not refer to the caller activity's task.

B. Semantics

To provide a formal ground to understand the semantics of various activity activation possibilities, we specify its operational semantics. Figure 4 defines domains, stacks, and representative commands to describe the operational semantics. We write A for an activity class and a for an activity instance. A task t is a pair of its name and its back stack (s, α). A back

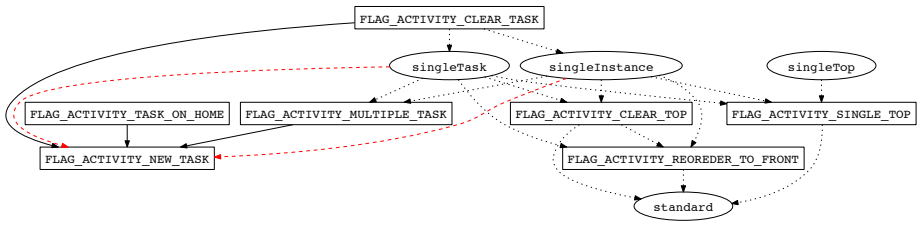


Fig. 5. Dependencies between `launchMode` attributes of target activities and intent flags

$$\frac{\star = \mathit{getTaskWAct}(\gamma, A) \qquad a' = \mathit{new}(A) \qquad (s,\alpha) = \mathit{getTask}(\gamma, s) \qquad \gamma' = \mathit{removeTaskTP}(\gamma,(s,\alpha)) \qquad \beta' = \mathit{removeTaskTS}(\beta,(s,\alpha)) \qquad t = (s,(a',\texttt{singleTask})::\alpha)}{\langle\gamma,\beta\rangle \vdash (a,l).\texttt{startActivity}(A,\texttt{singleTask},s,\varnothing) \rightarrow \langle t::\gamma', t::\beta'\rangle}$$

Fig. 6. Activity injection semantics used in the sample malware

```
Stack #70:
  Task id #329
    TaskRecord{93e6018 #329 A=com.facebook.katana U=0 sz=2}
    Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.facebook.katana/
com.facebook.nodex.startup.splashscreen.NodexSplashActivity (has extras) }
      Hist #1: ActivityRecord{7e92f9b u0 com.example.malicious/.MaliciousActivity t329}
        Intent { flg=0x10400000 cmp=com.example.malicious/.MaliciousActivity }
        ProcessRecord{98d28b9 31389:com.example.malicious/u0a91}
      Hist #0: ActivityRecord{1e96bac u0 com.facebook.katana/.FacebookLoginActivity t329}
        Intent { act=android.intent.action.VIEW flg=0x4000000 cmp=com.facebook.katana/.FacebookLoginActivity bnds=[86,524][995,1433] (has extras) }
        ProcessRecord{41c300c 31237:com.facebook.katana/u0a90}
  Task id #330
    TaskRecord{13b8a80 #330 A=com.example.malicious U=0 sz=1}
    Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10000000 cmp=com.example.malicious/.StartActivity }
      Hist #0: ActivityRecord{86d9497 u0 com.example.malicious/.StartActivity t330}
        Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10000000 cmp=com.example.malicious/.StartActivity }
        ProcessRecord{98d28b9 31389:com.example.malicious/u0a91}
```

Fig. 7. Task stack showing that a malicious activity is injected in the task of the legitimate Facebook app.

stack is a sequence of activities; an activity is a pair of its activity instance and its launch mode (a,l). The task stack and task pool are sequences of tasks. To represent the activity activation concisely, we focus on three commands: pressing the Home button, pressing the Back button, and starting an activity. The $(a,l).\texttt{startActivity}(A,l',s,F)$ command denotes that the caller activity is (a,l), the target activity is A with the `launchMode` attribute l' and the task name s from its `taskAffinity`, and the intent flags are F.

The judgments are of the form $\langle\gamma,\beta\rangle \vdash C \rightarrow \langle\gamma',\beta'\rangle$, which denotes that when the current task pool is γ and the current task stack is β, evaluation of the command C results in the new task pool γ' and the new task stack β'. Thus, $\langle\gamma,\beta\rangle \vdash \texttt{HomeButton} \rightarrow \langle\gamma,\epsilon\rangle$ specifies that whenever a user presses the Home button, the task pool remains the same but the task stack becomes empty. All the behaviors of pressing the Back button are:

$$\frac{\langle\gamma,\epsilon\rangle \vdash \texttt{BackButton} \rightarrow \langle\gamma,\epsilon\rangle}{\langle\gamma,(s,\alpha)::\beta\rangle \vdash \texttt{BackButton} \rightarrow \langle\gamma',\beta\rangle} \quad \alpha = (a,l)::\epsilon \qquad \gamma' = \mathit{removeTaskTP}(\gamma,(s,\alpha))$$

$$\frac{}{\langle\gamma,(s,\alpha)::\beta\rangle \vdash \texttt{BackButton} \rightarrow \langle(s,\alpha')::\gamma',(s,\alpha')::\beta\rangle} \quad \alpha = (a,l)::\alpha' \qquad \alpha' \neq \epsilon \qquad \gamma' = \mathit{removeTaskTP}(\gamma,(s,\alpha))$$

The first specifies that if a user presses the Back button when the task stack is empty, the display remains the same without any changes in the task pool or in the task stack. The second describes that when the top task on the task stack has only one activity, the Back button removes the task from both the task pool and the task stack. The third describes that when the top task on the task stack has more than one activities, the Back button removes the top activity from the task both in the task pool and the task stack. Figure 12 in Appendix describes all the helper functions. For the last command, the semantics has 239 rules. We can split them into 4 cases depending on the target activity's `launchMode`. Figure 12 shows one case when the target activity's `launchMode` is `singleTask`. Due to the space limitation, we describe the remaining rules and helper functions in a companion report [9].

C. Implementation of Malware

Activity injection attacks can steal users' private data like phishing attacks. Since launching benign apps can run malicious activities, it is difficult for users to notice malicious activities. For example, if a malware disguises an activity as a login page of a benign app and injects it to the app, it can steal users' id and password information. To demonstrate such attacks, we developed a malware targeting the Facebook app. Among 239 rules for the `startActivity` function call in our operational semantics, 180 rules specify possible activity injection attacks. While we can make malicious apps utilizing any of such rules, we developed a malware using the semantics described by the rule in Figure 6 to show its practicality.

Figure 7 shows a snapshot of the task stack when we first run the legitimate Facebook app, execute our malware, and let the malware inject a malicious activity in the task of Facebook. The task stack #70 contains two tasks: #329 for

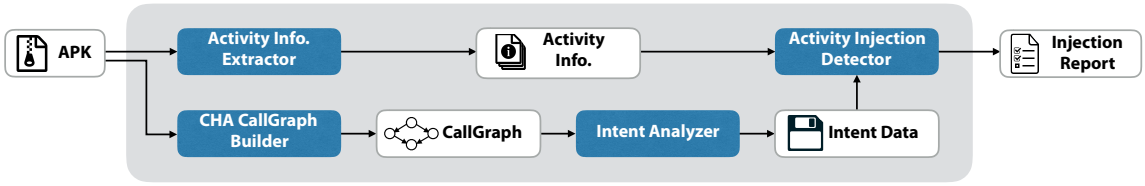
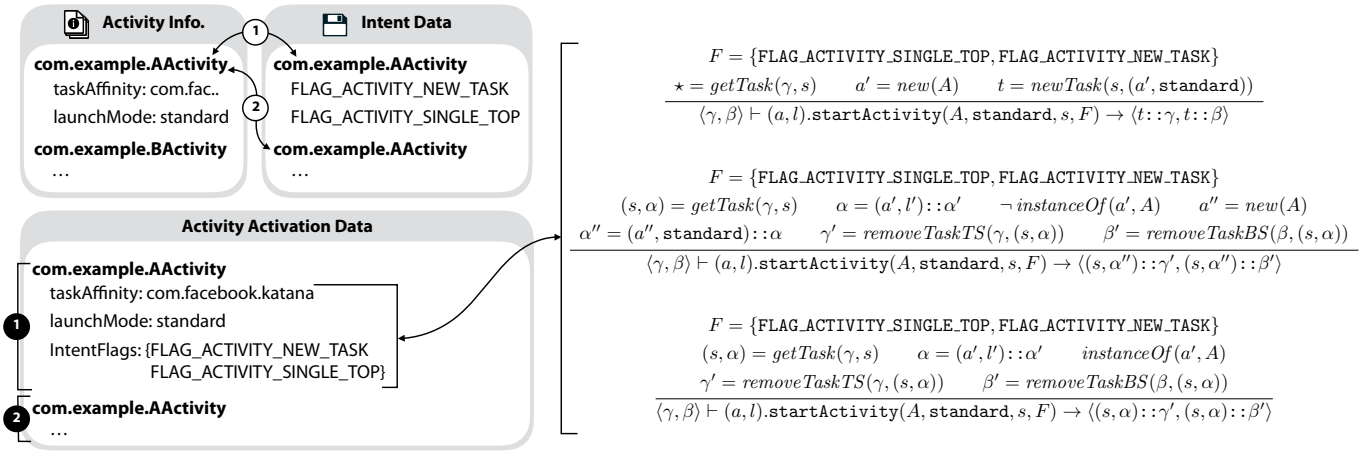


Fig. 8. Overall structure of the static analysis tool



$$\frac{F = \{\texttt{FLAG\_ACTIVITY\_SINGLE\_TOP}, \texttt{FLAG\_ACTIVITY\_NEW\_TASK}\} \quad \star = getTask(\gamma, s) \quad a' = new(A) \quad t = newTask(s, (a', \texttt{standard}))}{\langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{standard}, s, F) \rightarrow \langle t :: \gamma, t :: \beta \rangle}$$

$$\frac{F = \{\texttt{FLAG\_ACTIVITY\_SINGLE\_TOP}, \texttt{FLAG\_ACTIVITY\_NEW\_TASK}\} \quad (s, \alpha) = getTask(\gamma, s) \quad \alpha = (a', l') :: \alpha' \quad \neg\, instanceOf(a', A) \quad a'' = new(A) \quad \alpha'' = (a'', \texttt{standard}) :: \alpha \quad \gamma' = removeTaskTS(\gamma, (s, \alpha)) \quad \beta' = removeTaskBS(\beta, (s, \alpha))}{\langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{standard}, s, F) \rightarrow \langle (s, \alpha'') :: \gamma', (s, \alpha'') :: \beta' \rangle}$$

$$\frac{F = \{\texttt{FLAG\_ACTIVITY\_SINGLE\_TOP}, \texttt{FLAG\_ACTIVITY\_NEW\_TASK}\} \quad (s, \alpha) = getTask(\gamma, s) \quad \alpha = (a', l') :: \alpha' \quad instanceOf(a', A) \quad \gamma' = removeTaskTS(\gamma, (s, \alpha)) \quad \beta' = removeTaskBS(\beta, (s, \alpha))}{\langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{standard}, s, F) \rightarrow \langle (s, \alpha) :: \gamma', (s, \alpha) :: \beta' \rangle}$$

Fig. 9. Example for Activity Injection Detector mechanism

Facebook (`com.facebook.katana`) and #330 for the malware (`com.example.malicious`). The task of Facebook contains two activities: #0 for launching a login activity of the Facebook app (`com.facebook.katana/.FacebookLoginActivity`) and #1 for the malicious activity injected by the malware (`com.example.malicious/.MaliciousActivity`).

Our malware injects the malicious activity when the screen is off, because it will be displayed on the screen when the injection occurs. Once the malicious activity is injected into the Facebook task, it immediately calls the Home activity so that the screen shows the Home screen when the screen is turned on again. Because the malicious activity is the top activity of the Facebook task, it will be displayed when a user touches the Facebook icon[1]. In order to launch the activity injection attack, we assume that our malware is installed on a victim's mobile device (similarly for [10], [11], [12], [13], [4]). Note that injecting a malicious activity does not require any permission; our malware requires only the INTERNET permission to send users' private data to the attacker's server.

The root cause of this threat is that the Android system supports the activity injection feature without any protection policy against malicious apps. One of the possible mitigations is a system-level access control that prohibits apps from injecting activities into other target apps' task without authorizations of the targets. We are discussing this issue with the Android security team.

IV. Activity Injection Detector

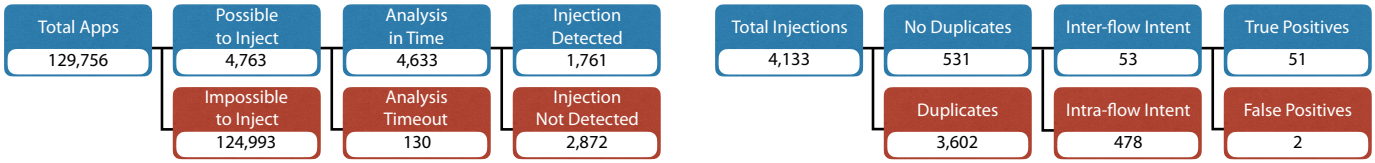
Based on the formal semantics of activity activation, we developed a static analyzer that detects possible activity injection attacks in Android apps. We implemented it on top of WALA [14], an open-source static analysis framework. The prototype implementation of the tool is publicly available[2].

The overall structure of the tool is illustrated in Figure 8. From a given Android apk archive, Activity Info. Extractor builds information about the activities in the app; it disassembles the resource files of the app using apktool [15] and extracts `launchMode` and `taskAffinity` of the activities from the `AndroidManifest.xml` file. Also, CHA CallGraph Builder takes the apk and constructs its call graph via Class Hierarchy Analysis (CHA) [16], which is often less precise than pointer analysis but much faster, since it analyzes only types rather than values. Then, Intent Analyzer extracts intent data that activate activities from the call graph. The intent data contain the target activity names of the intent and the set of intent flags added to the intent. Finally, Activity Injection Detector takes the activity information with intent data, and produces a report of possible activity injections. Using the activity information, intent data, and the activity activation semantics, it analyzes possible activity injection cases.

Similar to FlowDroid [17], Intent Analyzer tracks intent data flows via a forward analysis from intent object creation sites to the `startActivity` method, and detects object aliases via a backward analysis from the locations where intent data are

[1]http://plrg.kaist.ac.kr/doku.php?id=research:material

[2]https://github.com/SunghoLee/AIDetector



(a) Analysis results in terms of apps

(b) Analysis results in terms of injections

Fig. 10. Analysis results of 129,756 Android apps

set to object fields. While Android supports both *explicit* and *implicit* intents, where the former uses concrete target activities to activate and the latter uses string values to denote target activities, our tool analyzes only explicit intents. Because analyzing the targets of implicit intents requires a precise string analysis, the tool does not consider implicit intents. Similarly, the tool does not analyze flows involving built-in method calls and static fields. Even though the prototype implementation of the tool does not address such features that are not the main interests of this paper, we believe that we can easily integrate other analyzers that can analyze them.

In addition, the tool covers about a half of the activity injection rules in the operational semantics. While one can inject a target activity without the `FLAG_ACTIVITY_NEW_TASK` flag if a caller activity's `launchMode` is `singleInstance`, because `singleInstance` is not applicable to most applications [18], the tool does not consider `launchMode` of caller activities.

Figure 9 illustrates how Activity Injection Detector works. It takes two inputs: Activity Info. with `taskAffinity` and `launchMode` for each activity, and Intent Data with intent flags for each activity activation. Because an activity may be activated multiple times, Intent Data in the figure contains multiple entries for `com.example.AActivity`. It merges two inputs into Activity Activation Data and matches each entry with the operational semantics rules. For example, because the first entry specifies that `launchMode` of a target activity is `standard` and the intent flags are `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_SINGLE_TOP`, one of three rules on the right side may be applicable. Note that no matter which rule is applied, the target activity a' always exists in the task pointed by `taskAffinity` at the activity activation time. Thus, the tool detects `com.example.AActivity` as an injected activity.

V. Evaluation

In this section, we evaluate our tool using 129,756 Android apps collected from AndroZoo [19] with timeout of 5 minutes. We performed all the experiments on a Linux x64 machine with 4.0GHz Intel Core i7 CPU and 8GB memory.

A. Experiment Results

The overall results of our experiments are shown in Figure 10. As Figure 10(a) presents, we first identified apps that may be open to activity injection. When an app has an activity, which contains `taskAffinity` that is different from the package name of the app, it is open to activity injection. Otherwise, it is not possible to inject activities to such apps. Out of 129,756 Android apps, 4,763 apps, only about 3.7% of the total apps, are open to activity injection. Among 4,763 apps, our tool could finish analysis of 97.3%; it did not finish analysis of only 130 apps in 5 minutes. Finally, the tool detected activity injection cases from 1,761 apps, which amounts to about 1.4% of the total apps.

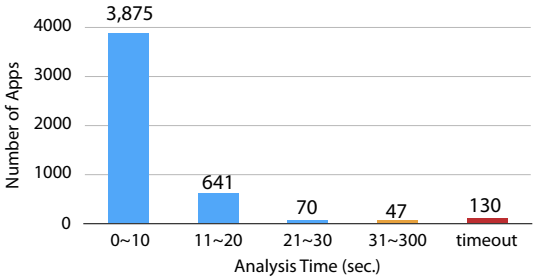


Fig. 11. Analysis time of 4,763 Android apps

From 1,761 apps, the tool detected 4,133 possible activity injection cases. As Figure 10(b) presents, among 4,133 injection cases, only 531 cases are unique and the remaining 3,602 cases are duplicated ones. We identified injection cases using their activity names. For example, when the tool reports that `com.example.AActivity` is injected multiple times to a single app or multiple apps, we count it as one injection case. To evaluate the precision of the tool, we partitioned 531 unique injection cases into the ones with relatively complex data flows and the others. More specifically, we identified such cases that require inter-procedural data flows to detect the injection cases as complex data flows, which amounts to 53 injection cases. We manually investigated them, and found that only two cases are false positives and the remaining 51 cases are all true positives. Thus, the false positive rate of the tool is only 3.8%.

The tool reported a very low false positive rate even though its CHA-based call graph construction produces imprecise call graphs in general, thanks to simple usage patterns of intents. Usually, intent objects use only simple data flows as shown by the number of intra-procedural data flows in Figure 10(b). Our manual inspection revealed that most inter-procedural intent data flows are through static methods that do not affect the imprecision of graph construction. While CHA-based analyses are generally imprecise for code patterns with complex class inheritance relations, most apps do not use complex class hierarchy for activity activation. Thus, the simple call graph construction did not harm the precision of our tool much.

Figure 11 summarizes the analysis time of 4,763 apps. The tool can identify the apps not open to activity injection in 1 or 2 seconds. The X-axis denotes analysis time, and the Y-axis denotes the number of apps. The tool analyzed 4,586 apps,

TABLE II
ACTIVITIES INJECTED FROM MULTIPLE APPS

Activity Name	# of Apps
com/startapp/android/publish/AppWallActivity	1,279
com/startapp/android/publish/list3d/List3DActivity	1,254
com/cmcm/picks/PicksLoadingActivity	40
com/widdit/lockScreen/activities/TermsAndConditionsActivity	30
com/applift/playads/PlayAdsActivity	12
com/zdworks/android/zdclock/ui/alarm/AlarmActivity	6
org/chromium/BackgroundActivity	6
com/igexin/getuiext/activity/GetuiExtActivity	6
ccc71/at/activities/at_ui_progress	6
com/csipsimple/ui/incall/InCallMediaControl	6
org/saturn/stark/interstitial/comb/activity/NativeAdActivity	5
com/tencent/mm/ui/transmit/TaskRedirectUI	5

about 96.3% of 4,763 apps, in 30 seconds, and the average analysis time is 6 seconds except for timeout apps. The overall analysis time shows that our tool detects possible activity injection attacks in a large number of real-world Android apps.

We measured the unsoundness effects of our restrictions in analysis of the target apps. As we discussed in Section IV, we intentionally ignored analysis of data flows via static fields and built-in methods, and `launchMode` of caller activities for the analysis performance. We investigated how many apps have such features, and found that the tool may miss activity injection cases in 287 apps, about 6.0% of 4,763 apps.

B. Case Studies

We discuss three representative cases of activity injection.

1) Activities injected from multiple apps: We observed that some activities are included in multiple apps, and injected to another task. We summarize the activities injected from five or more apps in Table II. Among them, five activities are for mobile advertising platforms: `AppWallActivity` and `List3DActivity` are for StartApp [20], `PicksLoadingActivity` is for Appodeal [21], `PlayAdsActivity` is for AppLift [22], and `GetuiExtActivity` is for igexin [23]. Mobile advertising platforms use different tasks from their integrated apps so that they can logically separate their advertisement activities from the apps' activities. Two activities are from app development frameworks: `BackgroundActivity` is from Cordova [24] and `TaskRedirectUI` is from Tencent [24]. Both frameworks are widely used for Android app development. Three activities are included in libraries: `TermsAndConditionsActivity` is in Homebase [25], which supports personalized lock screen of mobile phone, `at_ui_progress` is in 3C Toolbox [26], which supports other apps' paid functionalities for free, and `InCallMediaControl` is in CSipSimple [27], which is an open-source session initiation protocol software. We found that `AlarmActivity` is injected from 6 apps, but all the apps have the same package name. Because Google Play does not allow duplicated package names, we believe that they are the same app possibly crawled from different app stores by AndroZoo. Finally, we failed to find any information about `NativeAdActivity`, although the activity name implies a mobile advertising platform.

2) Suspicious activity injection cases: To understand the reported activity injection cases in more detail, we collected "suspicious" cases where `taskAffinity` of the activities are the names of existing applications' packages, libraries, or their subdomains. Out of 531 unique injection cases, we collected 12 such cases. The collected cases are highly suspicious, but the other cases may be malicious as well.

We summarize the cases in Table III. Among 12 targets, Umeng is a mobile advertising platform library and the others are applications. For example, we found that the app `com/netngroup/point` has an activity injection to the task `com/umeng/community` of the library Umeng. Interestingly, two cases inject activities into Android built-in apps' tasks: `com/tencent/android/qqdownloader` injects an activity into the task of the Android package installer, and `com/lookout` injects an activity into the task of Google Chrome, the default browser of Android. Thus, when users launch such built-in apps, they may use injected activities instead of the built-in apps' activities. Because the built-in apps are installed on every Android device and they are highly sensitive to the device security, such injected activities may open the gate to security threats.

a) Redundant usage of `launchMode` *and intent flags:* In some injection cases, an intent object has a redundant intent flag for activity activation: `FLAG_ACTIVITY_NEW_TASK` is unnecessary when `launchMode` of a target activity is `singleTask`, because the target activity is always injected into a task pointed by the `taskAffinity`. However, we observed that 73 out of 531 activity injection cases have both `launchMode` and the flag. Since the activity activation mechanism is complex and difficult to understand from the Android document, redundant or misused intent flags may be discovered in various real-world Android apps. We believe that our formal semantics would be helpful for developers to understand the activity activation mechanism clearly.

VI. RELATED WORK

Recently, various attacks for Android apps have been reported. They leverage vulnerabilities in inter- and intra-app communication [28], content providers [29], permission systems [30], push message services [31], OAuth [32], hybrid web app frameworks [33], dynamic code loading [34], Android public resources [13], vendor customization [35], [36], [37], the WebView API [38], and advertisement libraries [39].

Among various Android security issues, we focus on the Android UI security, which has been extensively studied. Leveraging the ADB daemon, third-party apps can steal users' private data by stealthily taking screen shots [40], [41]. Chen *et al.* [10] reported the UI state inference attack, which exploits the public side channel to demonstrate activity hijacking attacks. Roesner and Kohno [42] demonstrated security problems of embedded user interfaces in Android. Niemietz and Schwenk [43] reported the UI Redressing attack, which leverages special characteristics of the toast view. Kraunelis *et al.* [44] reported the Masquerade attack using the accessibility framework in Android. Luo *et al.* [45] showed launching

TABLE III
SUSPICIOUS INJECTION CASES, NAMES OF THEIR TARGET APPS OR LIBRARIES, AND THEIR TARGET TASKS

App with Activity Injection	Target App or Library	Target Task
com/dianxinos/dxhome	91PandaHome2	com/nd/android/pandahome2/manage/shop/ThemeShopMainActivity
com/tencent/android/qqdownloader	Android package installer	com/android/packageinstaller
com/duapps/antivirus	Caller ID Recorder	com/whosthat/aftercall/tips
com/jushine/chouse	Days Money Book	com/bora/chouse/ImageViewActivity
cn/opda/a/phonoalbumshoushou	DU Speed Booster & Cleaner	com/dianxinos/optimizer/task/sevenkey
com/jiubang/goscreenlock	GO Launcher	com/gau/go/launcherex
com/lookout	Google Chrome	com/android/chrome
com/j3gha/mailclient	K-9 Mail	com/fsck/k9/activity/setup/Prefs
com/emoji/input/gif/theme/keyboard	PIP Launcher	cm/theme/wallpaper/launcher/locksceen
com/csms/activities	Power Assistant 4.1.2	com/zhimahu/null
com/netngroup/point	Umeng	com/umeng/community
com/goldbean/yoyo	WeChat	com/tencent/mm

Touchjacking attacks using the WebView API. Felt and Wagner [46] analyzed phishing attacks on mobile devices.

The most closely related work to ours is Taskhijacking attacks [4], which lead users into malicious activities sharing benign apps' tasks. Taskhijacking attacks include all the cases where multiple apps share a task, many of which are normal inter-app communication, but we focus on activity injection attacks where apps inject an activity into another app's task. Also, our work significantly improved the previous work in that we specified all the semantic behaviors formally and exhaustively without any restrictions that the previous work made to build transition diagrams via testing and, we detected possible activity injection attacks in real-world Android apps.

Researchers have tried to analyze Android apps that utilize Inter-Component Communication (ICC). SCanDroid [47] is one of the first static analyzers for Android apps. Based on a formal constraints system for a core ICC language, SCanDroid collects constraints for each component, and solves the collected constraints altogether considering all the caller and callee relations to analyze data flows between components. ComDroid [48] can detect seven kinds of potential vulnerabilities in Android ICC, and it specifically focuses on implicit intents. SmartDroid [49] uses both a static analysis and a dynamic analysis. During static analysis, it analyzes intent data flows to construct Activity Call Graphs (ACGs) that present caller and callee activities. Using the graph, it automatically triggers UI events of each activity to detect sensitive behaviors executed by UI interactions. Epicc [50] reduces ICC to an instance of the Inter-procedural Distributive Environment (IDE) problem, and it detects ICC vulnerabilities via solving the IDE problem with lower false positive rates than SmartDroid. Apposcopy [51] performs taint analysis on Inter-Component Call Graph (ICCG) using intent information to detect privacy leakage in Android app. AmanDroid [52] constructs Inter-component Data Flow Graph (IDFG) via a pointer analysis, and models various Android APIs to extract intent information and to link data flows between components. DroidSafe [53] reports sensitive information leakages in Android apps. It computes the values of intent objects via a string analysis, finds the target components of the intent object values, and replaces the inter-component calls with explicit call instructions to the target components. IC3 [54] constructs inter-procedural Data Dependence Graph (DDG) and tracks data flows on DDGs to collect intent information. Based on IC3, IccTA [55] instruments Android apps' bytecode to replace inter component calls with explicit call instructions, and tracks taint data to detect sensitive data leakages.

The main difference between the above approaches and ours is the analysis purpose. While the existing approaches focus on analysis of control and data flows between components, we focus on efficient analysis of the activity activation mechanism. Unlike other approaches, we specified a formal semantics for the activity activation mechanism, and our tool considers `launchMode`, intent flags, and the semantics to detect activity injection attacks that no other tools could detect.

VII. CONCLUSION

The Android multitasking features are extremely powerful and useful, but, at the same time, they are vulnerable to activity injection attacks. Because there are hundreds ways to launch activities, it is very difficult for developers to understand the activity activation behaviors clearly. More importantly, the complex behaviors contain various ways to inject malicious activities to legitimate apps. To alleviate the problem, we formally specify the activity activation semantics exhaustively, identify possible places for activity injection attacks, and implement a sample malware to show that the activity injection attacks are realistic and dangerous. Based on the formal semantics of activity activation, we developed a static analyzer to detect activity injection attacks in Android apps. The tool is fast enough to analyze real-world Android apps in 6 seconds on average, and it is precise enough to have only 3.8% false positive rate. Our experiments showed that 1,761 out of 129,756 Android apps inject their activities into other apps' tasks, and our manual investigation revealed that 12 apps inject their activities into the tasks of Android apps including built-in apps' tasks. We believe that the formal semantics of the activity activation mechanism would help developers to understand the complex mechanism clearly, and the tool would be useful in detecting activity injection attacks in a large number of Android apps efficiently.

ACKNOWLEDGMENT

This work received funding from National Research Foundation of Korea(NRF) (Grant NRF-2017R1A2B3012020).

Helper function	Type	Description
new	$\mathit{ActivityClass} \to \mathit{ActivityInstance}$	returns an instance of a given activity class
newTask	$\mathit{String} \times (\mathit{ActivityInstance} \times \mathit{LaunchMode}) \to \mathit{Task}$	creates a new task with a given task name, activity instance, and its launch mode
removeTaskTP	$\mathit{TaskPool} \times \mathit{Task} \to \mathit{TaskPool}$	removes a given task from a given task pool
removeTaskTS	$\mathit{TaskStack} \times \mathit{Task} \to \mathit{TaskStack}$	removes a given task from a given task stack
removeActsUntil	$\mathit{BackStack} \times (\mathit{ActivityInstance} \times \mathit{LaunchMode}) \to \mathit{BackStack}$	removes activity instances on top of and including a given activity instance from a given back stack
getTask	$\mathit{TaskPool} \times \mathit{String} \to \mathit{Task} \cup \{\star\}$	returns $\star$ for absence or a task with a given name from a given task pool
getActivity	$\mathit{BackStack} \times \mathit{ActivityClass} \to \mathit{ActivityInstance} \cup \{\star\}$	returns $\star$ for absence or an instance of a given activity class from a given back stack
getTaskWAct	$\mathit{TaskPool} \times \mathit{ActivityClass} \to \mathit{Task} \cup \{\star\}$	returns $\star$ for absence or a task that contains an instance of a given activity class from a given task pool

Target activity's `launchMode` is `singleTask`:

$$\frac{\star = \mathit{getTaskWAct}(\gamma, A) \quad a' = \mathit{new}(A) \quad \star = \mathit{getTask}(\gamma, s) \quad t = \mathit{newTask}(s, (a', \texttt{singleTask}))}{\langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{singleTask}, s, \varnothing) \to \langle t :: \gamma, t :: \beta \rangle}$$

$$\frac{\begin{array}{c}\star = \mathit{getTaskWAct}(\gamma, A) \quad a' = \mathit{new}(A) \quad (s, \alpha) = \mathit{getTask}(\gamma, s) \\ \gamma' = \mathit{removeTaskTP}(\gamma, (s, \alpha)) \quad \beta' = \mathit{removeTaskTS}(\beta, (s, \alpha)) \quad t = (s, (a', \texttt{singleTask}) :: \alpha)\end{array}}{\langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{singleTask}, s, \varnothing) \to \langle t :: \gamma', t :: \beta' \rangle}$$

$$\frac{\begin{array}{c}(s, \alpha) = \mathit{getTaskWAct}(\gamma, A) \quad \gamma' = \mathit{removeTaskTP}(\gamma, (s, \alpha)) \quad \beta' = \mathit{removeTaskTS}(\beta, (s, \alpha)) \\ a' = \mathit{getActivity}(\alpha, A) \quad \alpha' = \mathit{removeActsUntil}(\alpha, (a', \texttt{singleTask})) \quad t = (s, (a', \texttt{singleTask}) :: \alpha')\end{array}}{\langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{singleTask}, s, \varnothing) \to \langle t :: \gamma', t :: \beta' \rangle}$$

$$\frac{F = \{\texttt{FLAG\_ACTIVITY\_CLEAR\_TASK}\} \quad \star = \mathit{getTaskWAct}(\gamma, A) \quad a' = \mathit{new}(A) \quad \star = \mathit{getTask}(\gamma, s) \quad t = \mathit{newTask}(s, (a', \texttt{singleTask}))}{\langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{singleTask}, s, F) \to \langle t :: \gamma, t :: \beta \rangle}$$

$$\frac{\begin{array}{c}F = \{\texttt{FLAG\_ACTIVITY\_CLEAR\_TASK}\} \quad \star = \mathit{getTaskWAct}(\gamma, A) \quad a' = \mathit{new}(A) \\ t = \mathit{getTask}(\gamma, s) \quad \gamma' = \mathit{removeTaskTP}(\gamma, t) \quad \beta' = \mathit{removeTaskTS}(\beta, t) \quad t' = \mathit{newTask}(s, (a', \texttt{singleTask}))\end{array}}{\langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{singleTask}, s, F) \to \langle t' :: \gamma', t' :: \beta' \rangle}$$

$$\frac{\begin{array}{c}F = \{\texttt{FLAG\_ACTIVITY\_CLEAR\_TASK}\} \quad t = \mathit{getTaskWAct}(\gamma, A) \quad \gamma' = \mathit{removeTaskTP}(\gamma, t) \\ \beta' = \mathit{removeTaskTS}(\beta, t) \quad a' = \mathit{new}(A) \quad t' = \mathit{newTask}(s, (a', \texttt{singleTask}))\end{array}}{\langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{singleTask}, s, F) \to \langle t' :: \gamma', t' :: \beta' \rangle}$$

$$\frac{F = \{\texttt{FLAG\_ACTIVITY\_TASK\_ON\_HOME}\} \quad \langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{singleTask}, s, \varnothing) \to \langle \gamma', t :: \beta' \rangle}{\langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{singleTask}, s, F) \to \langle \gamma', t :: \epsilon \rangle}$$

$$\frac{\begin{array}{c}F = \{\texttt{FLAG\_ACTIVITY\_CLEAR\_TASK}, \texttt{FLAG\_ACTIVITY\_TASK\_ON\_HOME}\} \quad F' = \{\texttt{FLAG\_ACTIVITY\_CLEAR\_TASK}\} \\ \langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{singleTask}, s, F') \to \langle \gamma', t :: \beta' \rangle\end{array}}{\langle \gamma, \beta \rangle \vdash (a, l).\texttt{startActivity}(A, \texttt{singleTask}, s, F) \to \langle \gamma', t :: \epsilon \rangle}$$

Fig. 12. Operational semantics for starting an activity when the target activity's `launchMode` is `singleTask`.

REFERENCES

[1] A. Smith and D. Page, "U.S. smartphone use in 2015," http://www.pewinternet.org/files/2015/03/PI_Smartphones_0401151.pdf, 2015.

[2] "Android Activity," http://developer.android.com/reference/android/app/Activity.html.

[3] "Task and Back Stack," http://developer.android.com/guide/components/tasks-and-back-stack.html.

[4] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards discovering and understanding task hijacking in Android," in *SEC*, 2015.

[5] "Android Developers Reference," http://developer.android.com/reference/packages.html.

[6] "Introduction to Android," http://developer.android.com/guide/index.html.

[7] "Overview screen," http://developer.android.com/guide/components/recents.html.

[8] "Android Intent," http://developer.android.com/intl/ko/reference/android/content/Intent.html.

[9] S. Lee, S. Hwang, and S. Ryu, "Operational semantics for the android activity activation mechanism," http://plrg.kaist.ac.kr/doku.php?id=research:material#technical_reports, 2017.

[10] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: UI state inference and novel android attacks," in *SEC*, 2014.

[11] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, "Tapprints: Your finger taps have fingerprints," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, 2012.

[12] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[13] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from Android public resources," in *CCS*, 2013.

[14] IBM, "T.J. Watson libraries for analysis," http://wala.sourceforge.net/wiki/index.php/Main_Page, 2006.

[15] "Apktool," https://ibotpeaches.github.io/Apktool/, 2010.

[16] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *European Conference on Object-Oriented Programming*. Springer, 1995, pp. 77–101.

[17] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[18] A. developer, "activity element," https://developer.android.com/guide/topics/manifest/activity-element.html#aff, 2017.

[19] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 468–471.

[20] "Startapp," http://www.startapp.com, 2017.

[21] "Appodeal," https://www.appodeal.com, 2017.

[22] "Applift," http://www.applift.com, 2017.

[23] "igexin," http://www.igexin.com, 2017.

[24] "Apache cordova," https://cordova.apache.org, 2017.

[25] "Homebase sdk," http://www.widdit.com/home/, 2017.

[26] "3c toolbox," http://www.3c71.com/android/?q=node/2442, 2017.

[27] "Csipsimple," https://github.com/r3gis3r/CSipSimple, 2017.

[28] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, 2011.

[29] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in Android applications," in *NDSS*, 2013.

[30] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *SEC*, 2011.

[31] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han, "Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services," in *CCS*, 2014.

[32] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," in *CCS*, 2014.

[33] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks," in *NDSS*, 2014.

[34] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications," in *NDSS*, 2014.

[35] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on Android security," in *CCS*, 2013.

[36] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in Android device driver customizations," in *S&P*, 2014.

[37] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *NDSS*, 2012.

[38] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *ACSAC*, 2011.

[39] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in Android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.

[40] C.-C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to milk your Android screen for secrets," in *NDSS*, 2014.

[41] S. Hwang, S. Lee, Y. Kim, and S. Ryu, "Bittersweet adb: Attacks and defenses," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015.

[42] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *SEC*, 2013.

[43] M. Niemietz and J. Schwenk, "UI redressing attacks on Android devices," in *Black Hat Abu Dhabi*, 2012.

[44] J. Kraunelis, Y. Chen, Z. Ling, X. Fu, and W. Zhao, "On malware leveraging the Android accessibility framework," in *Proceedings of the EAI Endorsed Transactions on Ubiquitous Environments*, 2015.

[45] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on web in Android, iOS, and windows phone," in *Proceedings of the 5th International Symposium on Foundations and Practice of Security (FPS)*, 2012.

[46] A. P. Felt and D. Wagner, "Phishing on mobile devices," in *Proceedings of Workshop on Web Security and Privacy (W2SP)*, 2011.

[47] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated security certification of Android applications," Department of Computer Science, University of Maryland, College Park, Tech. Rep. CS-TR-4991, 2009.

[48] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 239–252.

[49] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2012, pp. 93–104.

[50] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Proceedings of the 22nd USENIX security symposium*, 2013, pp. 543–558.

[51] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 576–587.

[52] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1329–1341.

[53] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*. Citeseer, 2015.

[54] K. O. Elish, D. Yao, and B. G. Ryder, "On the need of precise inter-app icc classification for detecting android malware collusions," in *Proceedings of IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, 2015.

[55] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 280–291.