Android Multitasking Mechanism: Formal Semantics and Static Analysis of Apps

Jinlong He[1,3], Taolue Chen[2,4], Ping Wang[1,3], Zhilin Wu[1,5], Jun Yan[1,3]

[1] State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, China
[2] Birkbeck, University of London, UK
[3] University of Chinese Academy of Sciences, China
[4] State Key Laboratory for Novel Software Technology,
Nanjing University, China
[5] Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, China

Abstract. In this paper we formalize the semantics of the Android multitasking mechanism and develop efficient static analysis methods with automated tool supports. For the formalization, we propose an extension of the existing Android Stack Machine model to capture all the core elements of the mechanism, in particular, the intent flags used in inter-component communication. For the static analysis, we consider the configuration reachability and stack boundedness problem, designing new algorithms and developing a prototype tool TaskDroid to fully support automated model construction and analysis of Android apps. The experimental results show that TaskDroid is effective and efficient in analyzing Android apps in practice.

1 Introduction

Android, a mobile operating system developed by Google, features over 2 billion monthly active users and over 80% of the share of the global mobile operating system market.[6] The Google Play store, Google's official pre-installed app store on Android devices, has supplied 2 million apps since 2016.[7] Multitasking is a fundamental mechanism of the Android operating system. Its unique design, via activities, back stacks and task stacks, greatly facilitates organizing user sessions through tasks, and provides rich features such as handy application switching, background app state maintenance, and smooth task history navigation (using the "back" button) [14]. Although the Android multitasking mechanism has substantially enhanced user experiences of the Android system and promoted personalized features in app design, it is notoriously complex and difficult to understand. As a witness, it constantly baffles app developers and has become a common topic of question-and-answer websites.[8] In addition, such a complex mechanism is plagued by serious security concerns, e.g., GUI phishing and hijacking attacks, denial of service attacks, and privilege leakage [3,14,16].

[6] `https://expandedramblings.com/index.php/android-statistics/`

[7] `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`

[8] For instance, `https://stackoverflow.com/questions/3219726/`

The Android multi-tasking mechanism, despite its importance, had not been systematically studied until very recently. Lee *et al.* formalized the operational semantics of the Android multitasking mechanism [8,9]. Independently, we introduced a formal model, i.e., Android Stack Machine (ASM), to capture the fundamental aspects of the multitasking mechanism [6], where the first step was made towards static analysis of Android apps based on the ASM model. It appears that, despite these initiatives, much more studies are needed to understand the multitasking mechanism, and to utilize it to design, analyse, test and verify Android apps. For instance, the operational semantics [9] is lengthy and hard to grip, while the ASM model [6], being more succinct and accessible, is incomplete, as intent flags, an important and pervasive feature of the multitasking mechanism, were not taken into account. More importantly, static analysis of the multitasking behavior of apps and its supporting tools, which is the focus of the current paper, are largely missing (with the exception that a static analysis tool was developed in [8], but was specialised for detecting activity injection attacks while did not fit for general-purpose analysis). This is in contrast to the large body of work on the static analysis of the other aspects of Android apps.

Contributions. The current paper aims to deepen the understanding of the Android multitasking mechanism via formalization of its semantics, and to develop effective and efficient approaches to the general-purpose static analysis of the multitasking behavior of Android apps.

For the formal semantics of the multitasking mechanism (Section 3), we significantly extend the ASM model [6]. The most pronounced extension lies in the introduction of intent flags which are pervasive for Android inter-component communication but which were ignored by the original ASM model. Our improvements over the operational semantics [9] are as follows: (1) We formalize the semantics for Android 7.0/8.0, which is—interestingly and perhaps surprisingly—different from that of Android 6.0. (The semantics for Android 7.0 and 8.0 also have slight differences.) In particular, we identify the notion of *real activity* which plays an essential role in allocating newly launched activities into respective tasks (referred to as the task allocation mechanism). In contrast, the semantics given before [8,9] is for Android 6.0 and uses a different and simpler tasking allocation mechanism. To the best of our knowledge, this is the first time that the discrepancies of the multitasking mechanism for different Android versions are thoroughly studied and formalized. (2) The semantics we give is more succinct and structured. Instead of an explicit enumeration which takes tens of pages [9], we organize and group different cases, leading to a much shorter and more accessible description with underlying principles identified which greatly facilitate the understanding. (3) We validate the semantics against the actual behavior of the Android system by designing a diagnosis app and conducting exhaustive experiments. All the experimental data are made publicly available to encourage reproductivity (cf. the full version [7]).

For static analysis (Section 4), we consider some of the most fundamental problems based on the extended ASM model. In particular, we consider configuration reachability analysis, which is arguably the cornerstone of any automated analysis of this kind. By this analysis, one can determine whether a particular configuration of the app can be reached by interacting with the mobile phone users and/or possible interaction with other (potentially malicious) apps. It is not very difficult to envisage that most existing security vulnerabilities related to the multitasking mechanism can be reduced to such an analysis. We also consider the stack boundedness analysis. In general, app developers may be interested in checking whether there is a sequence of user actions which can force the height of some task(s) to grow unboundedly. If this were the case, there would be a security risk that the app may crash or even lead to rebooting of mobile devices, if a user or a malicious app interacts with the app by following the sequence. The stack boundedness analysis is used to detect such a vulnerability.

To carry out such analysis, we build ASM models from Android apps by first constructing the call graphs and control flow graphs based on the soot tool [15], then applying control and data flow analysis (Section 4.1). We give new, practical algorithms to solve the configuration reachability and stack boundedness problems (Section 4.2). For the configuration reachability problem, we reduce the problem to the reachability problem of finite state machines, by imposing a (specified) constant bound on the height of tasks. The latter problem can be solved by off-the-shelf symbolic model checkers (e.g., nuXmv [4]) efficiently. For the stack boundedness problem, the algorithm searches witness cycles of transitions for each task along which its back stack may run unbounded with the involvement of other tasks.

To evaluate our approaches, we implement a prototype tool TaskDroid and carry out extensive experiments on over 4, 000 apps, which are either open-source apps from F-Droid or apps downloaded from app markets, e.g., Google Play (Section 5). The experimental results show that our approaches are effective and efficient in analysing the apps in practice. Remarkably, TaskDroid enables us to detect that 29 apps from F-Droid are stack unbounded, and our experiments confirm that the stack unboundedness poses a real threat (Section 5.2): The 29 stack-unbounded F-Droid apps, when being fed into the Monkey tool to simulate the detected witness cycles for hundreds or thousands of times, exhibit black screen, app crash, or even rebooting of mobile devices.

Related work. We discuss the related work from the following three perspectives.

Android GUI models. Some models addressing GUI activities of Android apps have been proposed. *Activity transition graphs* [2] were probably the first model to represent Android GUI activity transitions, but they are essentially a syntactic model without addressing the semantics sufficiently. *Window transition graphs* [18] can represent the possible GUI activity (window) sequences and their associated events and callbacks, and thus can capture how the events and callbacks modify the task stack. However, this model addresses neither the launch modes other than "standard" nor task affinities. Labeled transition graphs with stack and widget (LATTE [17]) consider the effects of launch modes on the task stack, but not those of task affinities. Essentially, it provides a finite-state abstraction of the behavior of the task stack. The ASM model [6] is the basis of the current work, but its was oversimplified for the purpose of formalizing the semantics of Android multi-tasking mechanism.

Static analysis. Static analysis for Android apps has been thoroughly studied, and we refer the readers to [10], which provides a systematic literature review involving 124 research papers published during the period for 2011-2015. More recently, [13] investigated the problem of composite constant propagation, which was able to infer Android inter-component communication values, and developed a tool called IC3. Our model construction may use IC3 but we choose not to do mainly because: (1) IC3 is unable to discover the *indirect* activation between activities in general. (For instance, if the activity A calls a function of the non-activity class C in which the activity B is started, then IC3 does not conclude that B can be started by A, since it will ignore the function call from A to C.) (2) IC3 analyses more information than what the ASM model needs making it less efficient for the purpose of ASM model generation. Finally, we mention recent work which exploits neural networks or probabilistic models to improve the precision and scalability of static analysis [12,20]. On a different matter, [19] introduced a launch-mode aware context-sensitive activity transition analysis, but did not consider task affinities or intent flags.

Security related to multi-tasking mechanism. Various work has identified potential security vulnerabilities related to the android multitasking mechanism, which has become one of the strong motivations to provide a complete formalization. [14] firstly reported task hijacking attacks, which means "malware reside side by side with the victim apps in the same task and hijack the user sessions of the victim apps." [8] analyzed the activity injection vulnerability referring to "inject malicious activities into a victim app's activity stack to hijack user interaction flows." As discussed in the introduction, our formalization provides several improvements over this work. Static analysis was also considered there, but was restricted to the detection of activity injection vulnerabilities. [16] recognized that the multitasking mechanism could give additional privilege to apps, which can be exploited by attackers. The authors analyzed the system/app conditions that can enable privilege leakage and identified new end-to-end attacks where attackers can actively interfere with victim apps to steal sensitive information. [11] introduced TDroid, an approach to detecting app switching attacks, which combines both static and dynamic analysis.

2 Android multitasking mechanism

This section provides an overview of the Android system mainly from an UI perspective focusing on the multitasking mechanism. For the purpose of this paper, an Android app can be considered as a collection of activities.[9] An activity is an instance of the `android.content.Activity` class, and provides GUI on screen [1]. A *task*, as a logical notion, is a collection of activities that users interact with

[9] In this paper, activities as viewed as atomic objects, and thus sub-components (e.g., fragments, `https://developer.android.com/guide/components/fragments`) contained in activities are simply ignored.

when performing a certain job. The running activities of a task are managed by Android as a stack in the order that each activity is opened. Such a stack is usually referred to as a *back stack*. (Unfortunately the terminologies in literature are not necessarily consistent, for instance, [6] use back stack differently.) In the sequel, we will usually identify a back stack and the task it belongs to. In a task there are two distinguished activities, i.e., the "root activity" which is the one sitting at the bottom, and the "top activity" which is the topmost activity. Note that in Android, activities from different apps can stay in the same task, and activities from the same app can enter different tasks.

The Android system may have multiple tasks: one *foreground* task and possibly several *background* tasks. They are organized as a stack as well, which is referred to as a *task stack* [8] (aka. activity stack [14]). The foreground task, as expected, sits on the top of the task stack. When a task comes to the *foreground*, its top activity is displayed on the device screen. When an activity finishes, it is popped from the back stack. If the back stack is not empty, the new top activity is displayed on the screen. Otherwise, the task itself finishes in which case it is popped from the task stack. We mention that, the Home screen comes to the foreground when a user presses the Home button (in this case the task stack will be emptied) or when the task stack becomes empty. The task stack is the central data structure for Android multi-tasking mechanism, and we are mostly interested in its evolution in response to activity activation. When an activity is started, there are three basic attributes which determine the resulting task stack: *launch mode, task affinity*, and *intent flags*. All the activities of an app, as well as their launch modes and task affinities, are defined in the *manifest file* (AndroidManifest.xml). Differently, intent flags are set by caller activities to declare how to activate target activities by calling startActivity() or startActivityForResult() with the intent flags as its arguments. The launch mode attribute specifies one of four modes to launch an activity: standard, singleTop, singleTask, and singleInstance, with standard being the default. The task affinity attribute specifies to which task the activity prefers to belong. By default, all the activities from the same app have the same affinity (i.e., all activities in the same app prefer to be in the same task). However, one can modify the default affinity of the activity. Android allows a great degree of flexibility: activities defined in different apps can share a task affinity whilst activities defined in the same app can be assigned with different task affinities.

Android supports inter-component communication via *intents*. An intent is an asynchronous message that activates activities. Android provides 21 intent flags related to activities, but only part of them may govern activity activation. Intent flags are set by caller activities to declare how to activate target activities and are passed to startActivity() or startActivityForResult() as their arguments.

3 Formalization

In this section, we provide a formalization of the semantics of the Android multitasking mechanism. We focus on the evolution of the task stack when an activity is launched. For this purpose we adapt and extend the ASM model [6]. For $k \in \mathbb{N}$, let $[k] = \{1, \cdots, k\}$.

Following the overview of Section 2, we shall concentrate on the launch mode, the task affinity, and the intent flags when an activity is launched. There are four launch modes in Android: "standard" (STD), "singleTop" (STP), "singleTask" (STK) and "singleInstance" (SIT). For the task affinity, we note that its default value is the package name of the app (i.e., when it is not specified explicitly). However, we find that Android system exhibits unexpected behavior when it is an empty string. Our formal semantics takes special care of this which has not been addressed before, to the best of our knowledge. Furthermore, Android provides 21 intent flags related to activities[10], namely, the flags whose names start with FLAG_ACTIVITY. Among these 21 intent flags, we consider the following seven that are commonly used in Android apps,

- FLAG_ACTIVITY_NEW_TASK (NTK),
- FLAG_ACTIVITY_CLEAR_TOP (CTP),
- FLAG_ACTIVITY_SINGLE_TOP (STP),
- FLAG_ACTIVITY_CLEAR_TASK (CTK),
- FLAG_ACTIVITY_MULTIPLE_TASK (MTK),
- FLAG_ACTIVITY_REORDER_TO_FRONT (RTF),
- FL AG_ACTIVITY_TASK_ON_HOME (TOH).

The rest will not be addressed in this paper. We remark that some flags, i.e., NTK, CTP, STP, can be modeled by launch modes, as mentioned in [6]. However, CTK, MTK, RTF, and TOH cannot be captured.

3.1 The extended ASM model

Let $\mathcal{F} = \{\mathsf{NTK}, \mathsf{CTP}, \mathsf{STP}, \mathsf{CTK}, \mathsf{MTK}, \mathsf{RTF}, \mathsf{TOH}\}$ denote the set of intent flags, $\mathscr{B}(\mathcal{F})$ denote the set of formulae $\phi = \bigwedge\limits_{F \in \mathcal{F}} \theta_F$, where $\theta_F = F$ or $\neg F$.

Definition 1 (Android stack machine). *An* Android stack machine *(ASM) is a tuple* $\mathcal{A} = (\mathsf{Sig}, \Delta)$*, where*

- $\mathsf{Sig} = (\mathsf{Act}, \mathsf{Lmd}, \mathsf{Aft}, A_0)$ *is the* activity signature*, where*
  - Act *is a finite set of activities,*
  - $\mathsf{Lmd} : \mathsf{Act} \to \{\mathsf{STD}, \mathsf{STP}, \mathsf{STK}, \mathsf{SIT}\}$ *is the launch-mode function,*
  - $\mathsf{Aft} : \mathsf{Act} \to [m] \cup \{0\}$ *is the task-affinity function, where* $m = |\mathsf{Act}|$*,*
  - $A_0 \in \mathsf{Act}$ *is the* main *activity,*
- $\Delta \subseteq (\mathsf{Act} \cup \{\rhd\}) \times \mathsf{Inst}$ *is the transition relation, where* $\mathsf{Inst} = \{\mathsf{back}\} \cup \{\alpha(A, \phi) \mid \alpha \in \{\mathsf{start}, \mathsf{finishStart}\}, A \in \mathsf{Act}, \phi \in \mathscr{B}(\mathcal{F})\}$ *such that* $(A, \mathsf{back}) \in \Delta$ *for each* $A \in \mathsf{Act}$*. (Intuitively, the* back *button can be pressed in any time) and for each transition* $(\rhd, inst) \in \Delta$*, it holds that* $inst = \mathsf{start}(A_0, \bigwedge\limits_{F \in \mathcal{F}} \neg F)$*.*

Intuitively, $\rhd$ denotes an empty task stack, $\mathsf{Aft}(A) = 0$ denotes the task affinity of A being the empty string, back denotes the pop action, $(A, \mathsf{start}(B, \phi))$ denotes the action that the activity B is started with some intent flags satisfying ϕ, and

[10] https://developer.android.com/reference/android/content/Intent#flags

$(A, \mathsf{finishStart}(B, \phi))$ is the same as $(A, \mathsf{start}(B, \phi))$, except that the activity A is popped after starting B. For convenience, we usually write $(A, \alpha(B, \phi)) \in \Delta$ as $A \xrightarrow{\alpha(\phi)} B$, where A is the *caller* activity, and B is the *callee* activity.

Remark 1. The main differences wrt [6] are: introducing intent flags, removing control states, and assuming that back actions are always enabled.

3.2 Semantics of ASM

We first discuss briefly how the core concepts such as tasks, task stack, and configurations are formalized. In general, a *task* is encoded as a word $S = [A_1, \cdots, A_n] \in \mathsf{Act}^+$ which denotes the content of its back stack, where n is called the *height* of S. A *task stack* is encoded as a non-empty sequence $\rho = ((S_1, A_1), \cdots, (S_n, A_n))$, where for each $i \in [n]$, S_i is a task and A_i is the real activity of S_i. The *real activity*[11] of a task is the activity which was pushed into the task—as the bottom activity—when the task is created. For any activity A, we refer to an A-task as a task whose real activity is A. The tasks S_1 and S_n are called the top and the bottom task respectively. (Intuitively, S_1 is the foreground task.) The symbol ε is used to denote the empty task stack. The *affinity of a task* is defined as the affinity of its real activity.

A task is called the *main task* of the task stack if it is the first task that was created when launching the app. Note that the current task stack may *not* contain the main task, since it may have been popped out from the task stack. This notion is introduced since the semantics of ASM is also dependent on whether the task stack contains the main task or not.

A *configuration* of $\mathcal{A}$ is a pair $= (\rho, \ell)$, where $\rho = ((S_1, A_1), \cdots, (S_n, A_n))$ with $S_i = [B_{i,1}, \cdots, B_{i,m_i}]$ for each $i \in [n]$ and $B_{i,j} \in \mathsf{Act}$ for $j \in [m_i]$, moreover, $\ell \in [n] \cup \{0\}$. We require (ρ, ℓ) to satisfy that if $\ell \in [n]$, then $A_\ell = A_0$. Intuitively, ℓ is the index of the main task. (If $\ell = 0$, then ρ contains no main task.) Let $\mathsf{Conf}_{\mathcal{A}}$ denote the set of configurations of $\mathcal{A}$. The *initial* configuration of $\mathcal{A}$ is $(\varepsilon, 0)$. The *height* of ρ is defined as $\max\limits_{i \in [m]} |S_i|$, where $|S_i|$ is the height of S_i. By convention, the height of ε is defined as 0.

We use the relation $\xrightarrow{\mathcal{A}}$ which comprises the quadruples

$$((\rho, \ell), \tau, i, (\rho', \ell')) \in \mathsf{Conf}_{\mathcal{A}} \times \Delta \times \{0, 1, 2\} \times \mathsf{Conf}_{\mathcal{A}}$$

to formalize the semantics of $\mathcal{A}$.

Auxiliary functions and predicates. To specify the transition relation precisely and concisely, we define the following functions and predicates. Here (ρ, ℓ) is a configuration with $\rho = ((S_1, A_1), \cdots, (S_n, A_n))$ and $B \in \mathsf{Act}$.

- Let $S = [B_1, \cdots, B_{m'}]$ be a task, then $\mathsf{Top}(S) = B_1$ and $\mathsf{Btm}(S) = B_{m'}$.
- $\mathsf{TopTsk}(\rho) = S_1$, $\mathsf{TopAct}(\rho) = \mathsf{Top}(\mathsf{TopTsk}(\rho))$.
- $\mathsf{Push}((\rho, \ell), B) = ((([B] \cdot S_1), A_1), (S_2, A_2), \cdots, (S_n, A_n)), \ell)$.

[11] The name is inherited from the Android system.

- $\mathsf{MvAct2Top}((\rho,\ell),B) = ((([B]\cdot S_1'\cdot S_1''),(S_2,A_2),\cdots,(S_n,A_n)),\ell)$, if $S_1 = S_1'\cdot[B]\cdot S_1''$ with $S_1' \in (\mathsf{Act}\setminus\{B\})^*$.
- $\mathsf{ClrTop}((\rho,\ell),B) = (((S_1'',A_1),(S_2,A_2),\cdots,(S_n,A_n)),\ell)$ if $S_1 = S_1'\cdot S_1''$ with $S_1' \in (\mathsf{Act}\setminus\{B\})^*B$.
- $\mathsf{ClrTsk}((\rho,\ell)) = ((([],A_1),(S_2,A_2),\cdots,(S_n,A_n)),\ell)$.
- Let $i \in [n]$, then $\mathsf{MvTsk2Top}((\rho,\ell),S_i) =$

  $$(((S_i,A_i),(S_1,A_1),\cdots,(S_{i-1},A_{i-1}),(S_{i+1},A_{i+1}),\cdots,(S_n,A_n)),\ell'),$$

  where ℓ' is defined as follows: if $\ell = 0$, then $\ell' = 0$; if $\ell = i$, then $\ell' = 1$; if $i+1 \le \ell \le n$, then $\ell' = \ell$; if $1 \le \ell \le i-1$, then $\ell' = \ell+1$.
  [Note that ℓ' is the simply the new position of the main task.]
- $\mathsf{NewTsk}((\rho,\ell),B) = ((([B],B),(S_1,A_1),\cdots,(S_n,A_n)),\ell')$, where $\ell' = 0$ if $\ell = 0$, and $\ell' = \ell+1$ otherwise.
- $\mathsf{GetRealTsk}(\rho,B) = S_i$ such that $i \in [n]$ is the *minimum* index satisfying $A_i = B$ if such an index i exists; $\mathsf{GetRealTsk}(\rho,B) = *$ otherwise.
- $\mathsf{GetTsk}(\rho,B) = S_i$ such that $i \in [n]$ is the *minimum* index satisfying $\mathsf{Aft}(A_i) = \mathsf{Aft}(B)$, if such an index i exists; $\mathsf{GetTsk}(\rho,B) = *$ otherwise.
- Let $i \in \{1,2\}$ and $S_i = [B_1,\cdots,B_{m'}]$. Then

  $$\mathsf{RmAct}((\rho,\ell),i) = \begin{cases} (0,(((S_1,A_1),\cdots,(S_{i-1},A_{i-1}), \\ \quad (S_{i+1},A_{i+1}),\cdots,(S_n,A_n)),0)) & \text{if } m'=1 \text{ and } \ell = 0 \text{ or } i, \\ (0,(((S_1,A_1),\cdots,(S_{i-1},A_{i-1}), \\ \quad (S_{i+1},A_{i+1}),\cdots,(S_n,A_n)),\ell)) & \text{if } m'=1 \text{ and } 1 \le \ell \le i-1, \\ (0,(((S_1,A_1),\cdots,(S_{i-1},A_{i-1}), \\ \quad (S_{i+1},A_{i+1}),\cdots,(S_n,A_n)),\ell-1)) & \text{if } m'=1 \text{ and } i+1 \le \ell \le n, \\ (i,(((S_1,A_1),\cdots,(S_{i-1},A_{i-1}),([B_2,\cdots,B_{m'}],A_i), \\ \quad (S_{i+1},A_{i+1}),\cdots,(S_n,A_n)),\ell)) & \text{if } m' > 1. \end{cases}$$

  Intuitively, $\mathsf{RmAct}((\rho,\ell),i) = (i',(\rho',\ell'))$, where (ρ',ℓ') is obtained from (ρ,ℓ) by removing the top activity of S_i from ρ, and $i' = 0,1,2$ denotes the position of the task S_i in ρ'. (In particular, $i' = 0$ denotes that the task S_i disappears in ρ'.)

Transition relation. For readability, we write $((\rho,\ell),\tau,i,(\rho',\ell')) \in \xrightarrow{\mathcal{A}}$ as $(\rho,\ell) \xrightarrow[\tau,i]{\mathcal{A}} (\rho',\ell')$. Intuitively, (ρ,ℓ) is the current configuration, (ρ',ℓ') is the configuration obtained after executing the transition rule τ, and $i = 0,1,2$ corresponds to the cases that the top task of ρ is absent in ρ', remains to be the top task of ρ', or becomes the task immediately below the top task of ρ', respectively.

For $\tau = (A,\mathsf{back})$ such that $\mathsf{TopAct}(S_1) = A$,

- if S_1 contains at least two activities, then $(\rho,\ell) \xrightarrow[\tau,1]{\mathcal{A}} (\rho',\ell')$ with $\rho' = ((S_1',A_1),(S_2,A_2),\cdots,(S_n,A_n))$, where S_1' is obtained from S_1 by removing the top activity A from S_1;

– otherwise, we have $(\rho, \ell) \xrightarrow[\tau,0]{\mathcal{A}} (\rho', \ell')$ with $\rho' = ((S_2, A_2), \cdots, (S_n, A_n))$ and $\ell' = \ell - 1$ if $\ell > 1$ and 0 otherwise.

For $\tau = \triangleright \to \mathsf{start}(A_0, \bigwedge_{F \in \mathcal{F}} \neg F)$, if (ρ, ℓ) is the initial configuration $(\varepsilon, 0)$, we have $(\rho, \ell) \xrightarrow[\tau,0]{\mathcal{A}} (([A_0], A_0), 1)$. (Here 0 is used because there is no top task in ρ.)

In the sequel, we first present the semantics for $\tau = A \xrightarrow{\mathsf{start}(\phi)} B$, which will be followed by the semantics for $\tau = A \xrightarrow{\mathsf{finishStart}(\phi)} B$.

Suppose $\rho = ((S_1, A_1), \cdots, (S_n, A_n))$ for some $n \geq 1$. Let $A = \mathsf{TopAct}(\rho)$.

<u>**Transition rules for $\tau = A \xrightarrow{\mathsf{start}(\phi)} B$**</u>

We distinguish two cases, i.e., $\phi \models \neg\mathsf{TOH}$ or $\phi \models \mathsf{TOH}$.

Case $\phi \models \neg\mathsf{TOH}$

$\boxed{\text{CASE } \mathsf{Lmd}(B) = \mathsf{SIT}}$

- if $\mathsf{GetRealTsk}(\rho, B) = S_j$ for some $j \in [n]$, then
  - if $j = 1$, then $i = 1$ and $(\rho', \ell') = (\rho, \ell)$,
  - if $j \neq 1$, then $i = 2$ and $(\rho', \ell') = \mathsf{MvTsk2Top}((\rho, \ell), S_j)$,
- if $\mathsf{GetRealTsk}(\rho, B) = *$, then $i = 2$ and $(\rho', \ell') = \mathsf{NewTsk}((\rho, \ell), B)$.

$\boxed{\text{CASE } \mathsf{Lmd}(B) = \mathsf{STK}}$

- if $\mathsf{GetRealTsk}(\rho, B) = S_j$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_j$, then $i = 1$ if $j = 1$, and $i = 2$ otherwise. Moreover,
  - if $\phi \models \neg\mathsf{CTK}$, then
    - if $B \notin S_j$, then $(\rho', \ell') = \mathsf{Push}(\mathsf{MvTsk2Top}((\rho, \ell), S_j), B)$,
    - if $B \in S_j$, then $(\rho', \ell') = \mathsf{Push}(\mathsf{ClrTop}(\mathsf{MvTsk2Top}((\rho, \ell), S_j), B), B)$,
  - if $\phi \models \mathsf{CTK}$, then $(\rho', \ell') = \mathsf{Push}(\mathsf{ClrTsk}(\mathsf{MvTsk2Top}((\rho, \ell), S_j)), B)$,
- if $\mathsf{GetTsk}(\rho, B) = *$, then $i = 2$ and $(\rho', \ell') = \mathsf{NewTsk}((\rho, \ell), B)$.

$\boxed{\text{CASE } \mathsf{Lmd}(B) = \mathsf{STD}}$

- if $\mathsf{Lmd}(A) \neq \mathsf{SIT}$ and $\phi \models \neg\mathsf{NTK}$, then $i = 1$ and
  - if $\phi \models \mathsf{STP} \vee \mathsf{RTF} \vee \mathsf{CTP}$ and $\mathsf{TopAct}(\rho) = B$, then $(\rho', \ell') = (\rho, \ell)$,
  - if $\phi \models \neg\mathsf{STP} \wedge \neg\mathsf{RTF} \wedge \neg\mathsf{CTP}$, or $\phi \models \mathsf{STP} \wedge \neg\mathsf{RTF} \wedge \neg\mathsf{CTP}$ and $\mathsf{TopAct}(\rho) \neq B$,
    or $\phi \models \mathsf{RTF} \vee \mathsf{CTP}$ and $B \notin \mathsf{TopTsk}(\rho)$, then $(\rho', \ell') = \mathsf{Push}((\rho, \ell), B)$,
  - if $\phi \models \mathsf{RTF} \wedge \neg\mathsf{CTP}$ and $B \in \mathsf{TopTsk}(\rho)$, then $(\rho', \ell') = \mathsf{MvAct2Top}((\rho, \ell), B)$,
  - if $\phi \models \mathsf{CTP}$ and $B \in \mathsf{TopTsk}(\rho)$, then $(\rho', \ell') = \mathsf{Push}(\mathsf{ClrTop}((\rho, \ell), B), B)$,
- if $\phi \models \mathsf{NTK} \wedge \mathsf{MTK}$, or $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\phi \models \mathsf{MTK}$, then $i = 2$ and $(\rho', \ell') = \mathsf{NewTsk}((\rho, \ell), B)$,
- if $\phi \models \mathsf{NTK} \wedge \neg\mathsf{MTK}$, or $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\phi \models \neg\mathsf{MTK}$, then
  - if $\mathsf{GetRealTsk}(\rho, B) = S_j$, then $i = 1$ if $j = 1$, and $i = 2$ otherwise. Moreover,
    - if $\phi \models \neg\mathsf{CTP} \wedge \neg\mathsf{CTK}$, then

- · if $j \neq \ell$, or $\phi \models \mathsf{STP}$ and $\mathsf{Top}(S_j) = B$, then
$$(\rho', \ell') = \mathsf{MvTsk2Top}((\rho, \ell), S_j),$$
- · otherwise, $(\rho', \ell') = \mathsf{Push}(\mathsf{MvTsk2Top}((\rho, \ell), S_j), B)$,

∗ if $\phi \models \mathsf{CTP} \wedge \neg\mathsf{CTK}$, then
- · if $B \not\in S_j$, then $(\rho', \ell') = \mathsf{Push}(\mathsf{MvTsk2Top}((\rho, \ell), S_j), B)$,
- · otherwise,$(\rho', \ell') = \mathsf{Push}(\mathsf{ClrTop}(\mathsf{MvTsk2Top}((\rho, \ell), S_j), B), B)$,

∗ if $\phi \models \mathsf{CTK}$, then $(\rho', \ell') = \mathsf{Push}(\mathsf{ClrTsk}(\mathsf{MvTsk2Top}((\rho, \ell), S_j)), B)$,

- if $\mathsf{GetRealTsk}(\rho, B) = *$ and $\mathsf{GetTsk}(\rho, B) = S_j$, then $i = 1$ if $j = 1$, and $i = 2$ otherwise. Moreover,
  ∗ if $\phi \models \neg\mathsf{STP} \wedge \neg\mathsf{CTP} \wedge \neg\mathsf{CTK}$, or $\phi \models \mathsf{STP} \wedge \neg\mathsf{CTP} \wedge \neg\mathsf{CTK}$ and $\mathsf{Top}(S_j) \neq B$, then $(\rho', \ell') = \mathsf{Push}(\mathsf{MvTsk2Top}((\rho, \ell), S_j), B)$,
  ∗ if $\phi \models \mathsf{STP} \wedge \neg\mathsf{CTP} \wedge \neg\mathsf{CTK}$ and $\mathsf{Top}(S_j) = B$, then $(\rho', \ell') = \mathsf{MvTsk2Top}((\rho, \ell), S_j)$,
  ∗ if $\phi \models \mathsf{CTP} \wedge \neg\mathsf{CTK}$, then
    · if $B \not\in S_j$, then $(\rho', \ell') = \mathsf{Push}(\mathsf{MvTsk2Top}((\rho, \ell), S_j), B)$,
    · otherwise, $(\rho', \ell') = \mathsf{Push}(\mathsf{ClrTop}(\mathsf{MvTsk2Top}((\rho, \ell), S_j), B), B)$,
  ∗ if $\phi \models \mathsf{CTK}$, then $(\rho', \ell') = \mathsf{Push}(\mathsf{ClrTsk}(\mathsf{MvTsk2Top}((\rho, \ell), S_j)), B)$,
- if $\mathsf{GetTsk}(\rho, B) = *$, then $i = 2$ and $(\rho', \ell') = \mathsf{NewTsk}((\rho, \ell), B)$.

$\boxed{\text{CASE } \mathsf{Lmd}(B) = \mathsf{STP}}$

The semantics is adapted from the case $\mathsf{Lmd}(B) = \mathsf{STD}$ by assuming $\phi \models \mathsf{STP}$ (cf. the full version [7] for details).

Case $\phi \models \mathsf{TOH}$

We then consider the transition rules $\tau = A \xrightarrow{\mathsf{start}(\phi)} B$ with $\phi \models \mathsf{TOH}$. It turns out that we can largely reuse the semantic definitions of the case that $\phi \models \neg\mathsf{TOH}$. Namely, let $\tau' = A \xrightarrow{\mathsf{start}(\phi')} B$ where ϕ' is obtained from ϕ by replacing TOH with $\neg\mathsf{TOH}$. (The behavior of τ' is fully prescribed before, viz, $(\rho, \ell) \xrightarrow[\tau', i]{\mathcal{A}} (\rho', \ell')$ where $\rho' = ((S'_1, A'_1), \cdots, (S'_{n'}, A'_{n'}))$.) Then we have that

– if $i = 1$, then $(\rho, \ell) \xrightarrow[\tau, i]{\mathcal{A}} (\rho', \ell')$,

– if $i = 2$, then $(\rho, \ell) \xrightarrow[\tau, 0]{\mathcal{A}} (((S'_1, A'_1)), \ell'')$, and $\ell'' = 1$ if $\ell' = 1$; 0 otherwise.

Note that if $i = 2$, then due to the effect of TOH, all the tasks in ρ', except the top one, will be removed.

Transition rules for $\tau = A \xrightarrow{\mathsf{finishStart}(\phi)} B$

We now consider $\tau = A \xrightarrow{\mathsf{finishStart}(\phi)} B$. Intuitively, $A \xrightarrow{\mathsf{finishStart}(\phi)} B$ specifies that B is started with the intent flags ϕ followed by the termination of A. Let $\tau' = A \xrightarrow{\mathsf{start}(\phi)} B$ and $(\rho, \ell) \xrightarrow[\tau', i]{\mathcal{A}} (\rho', \ell')$, with $\rho' = ((S'_1, A'_1), \cdots, (S'_{n'}, A'_{n'}))$. Then the semantics of $\tau = A \xrightarrow{\mathsf{finishStart}(\phi)} B$ is defined as follows.

– If $i = 0$ (this may happen when $\phi \models \mathsf{TOH}$), then $(\rho, \ell) \xrightarrow[\tau,0]{\mathcal{A}} (\rho', \ell')$.

– If $i = 2$, then $(\rho, \ell) \xrightarrow[\tau,i']{\mathcal{A}} (\rho'', \ell'')$, where $(i', (\rho'', \ell'')) = \mathsf{RmAct}((\rho', \ell'), 2)$.

– If $i = 1$, then
  - if $|S'_1| = |S_1| + 1$ (in this case, the top activity of S'_1 is B, and the top second is A), then let S''_1 obtained from S'_1 by removing the second activity from the top, and ρ'' obtained from ρ' by replacing S'_1 with S''_1, then we have $(\rho, \ell) \xrightarrow[\tau,1]{\mathcal{A}} (\rho'', \ell')$,
  - if $|S'_1| = |S_1|$, then $(\rho, \ell) \xrightarrow[\tau,i']{\mathcal{A}} (\rho'', \ell'')$, where if $\mathsf{Top}(S'_1) = A$, then $(i', (\rho'', \ell'')) = \mathsf{RmAct}((\rho', \ell'), 1)$, otherwise (in this case, $\phi \models \mathsf{RTF}$, $\mathsf{Top}(S'_1) = B$, and the top second activity of S'_1 is A), $i' = 1$, $\ell'' = \ell'$, and ρ'' is obtained from ρ' by removing from S'_1 the top second activity,
  - if $|S'_1| < |S_1|$, then $(\rho, \ell) \xrightarrow[\tau,1]{\mathcal{A}} (\rho', \ell')$.

3.3 High-level descriptions

We now present some high-level description which would facilitate the understanding of the semantics.

Task allocation mechanism. One of the main elements of the semantics of ASM is the *task allocation mechanism*, namely, to specify, when an activity is launched, to which task will it be allocated. Via extensive experiments, we identify a crucial notion, i.e., real activity of tasks, in Android 7.0 and 8.0, which plays a pivotal role in such a mechanism.

Generally speaking, for an activity B which is not to land on the top task, the following three steps will apply: (1) If there is any task whose real activity is B, then B will be put on the task; (2) Otherwise, if there is any task whose real activity has the same task affinity as B, then B will be put on the task (3) Otherwise, a new task is created to hold B. In the first two cases, if there are multiple instances, the first occurrence starting from the top task will be selected. Note that, due to the CTK flag, the bottom activity of a task may *not* be the real activity of the task.

Dependencies between launch modes and intent flags. For transitions $A \xrightarrow{\alpha(\phi)} B$, the launch modes of A, B and the intent flags in ϕ may depend on each other. The dependency can exhibit in the following three forms: n *subsumes* n', i.e., n' is ignored if n co-occurs with n', (2) n *enables* n', i.e., n' takes effect if n co-occurs with n', (3) n *implies* n', i.e., if n' subsumes (resp. enables) n'', then n subsumes (resp. enables) n'' as well. We summarize these dependencies in Figure 1, where the solid lines represent the "subsume" relation, the dashed lines represent the "enable" relation, the dotted lines represent the "implies" relation.

The following properties hold for these relations: (1) the "subsume" and "imply" relations are transitive, (2) the composition of the "imply" relation and the "subsume" (resp. "enable") relation is a subset of the "subsume" (resp. "enable") relation. Moreover, we remark that the two "enable" edges to TOH in Figure 1 are "incomplete" for TOH, in the sense that the two edges do not fully cover the situations where TOH takes effect, viz. the situations where *the launched activity B is not to land on the original top task.*

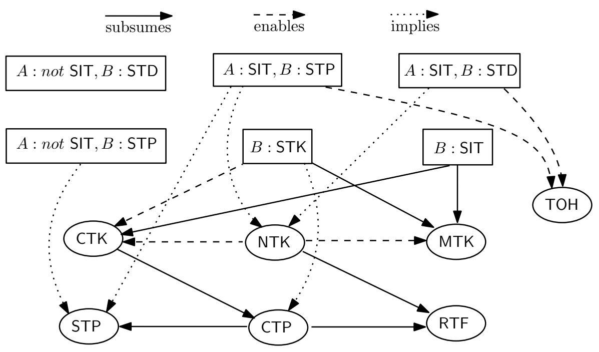

Fig. 1: Dependency graph for launch modes and intent flags in transitions $A \xrightarrow{\alpha(\phi)} B$. The launch modes (resp. the intent flags) are in boxes (resp. circles)

The empty-string task affinity. Intuitively, if the task affinity of some activity A is the empty string, then *the transition rules involving A are the same as if the task affinity of A were different from those of all the other activities.* Formally, suppose $A_1, \cdots, A_{m'}$ are an enumeration of all the activities with the empty-string task affinity, then the semantics of $\mathcal{A}$ is defined as that of $\mathcal{A}'$, where $\mathcal{A}'$ is obtained from $\mathcal{A}$ by setting $\mathsf{Aft}(A_j) = m + j$ for each $j \in [m']$ (recall that $m = |\mathsf{Act}|$).

The differences of the semantics for Android 6.0, 7.0, and 8.0. The semantics of ASM for Android 7.0 and Android 8.0 are almost the same except that: In Android 7.0, in case that $\mathsf{Lmd}(A) \neq \mathsf{SIT}$, $\mathsf{Lmd}(B) = \mathsf{STD/STP}$, B occurs on the top task of ρ, and $\phi \models \neg\mathsf{NTK} \wedge \mathsf{RTF} \wedge \neg\mathsf{CTP}$, the successor configuration (ρ', ℓ') is obtained from (ρ, ℓ) by first clearing the top task, then pushing B into it. Note that here, RTF acts like CTK,[12] although CTK is not enabled and takes no effect (see Figure 1).

The task allocation mechanism of Android 6.0 is considerably different, which is irrelevant to the real activities of tasks and only uses the affinities of tasks. Its semantics can be adapted from that of Android 8.0 and is given in the full version [7]. We remark that the semantics of ASM for Android 6.0 formalized here is essentially the same as that in [8,9] modulo some minor differences. Indeed, we have found that some of transition rules given in [9] are inconsistent to our semantics. For instance, in case $A \xrightarrow{\mathsf{start}(\phi)} B$ where $\mathsf{Lmd}(A) = \mathsf{SIT}$, $\mathsf{Lmd}(B) = \mathsf{STD}$, $\phi \models \mathsf{MTK} \wedge \mathsf{STP}$, and there exists some task with the same affinity as B in the current task stack: according to the 6th rule on page 22 of [9], the task whose affinity is $\mathsf{Aft}(B)$ and which is closest to the top task will be moved to top and no new task will be created, whereas in our semantics, a new task $S' = [B]$ will be created and become the top of the task stack.

[12] It is a confirmed bug of the multitasking mechanism affecting Android 4.4, 7.0 and 7.1.1; see `https://issuetracker.google.com/issues/36986021` for the discussions.

Validation of the formal semantics. To validate that the formal semantics of ASM conforms to the actual behavior of the respective Android versions, we have conducted exhaustive experiments by designing a diagnosis app and comparing, for each case in the definition of the formal semantics, the exhibited behavior of the app against the formal semantics. The details of the experiments can be found in the full version [7].

4 Static analysis of Apps

In this section, we consider static analysis of Android apps. At first, we show how to build the ASM model out of Android apps. Then, we illustrate how to solve the reachability and boundedness problems of ASM.

4.1 From Apps to ASM

We show how to construct an ASM model for an Android app. Recall that an ASM model comprises a signature of activities and a transition relation.

We take the input as either an Android PacKage (apk) file or simply the source code. We extract the manifest file from the source code or by decompiling the apk file. From the manifest file, we can obtain the signature of activities, namely, a list of activities of the app with their launch modes and task affinities, as well as the main activity. In addition, the intent filters, which include actions, categories and data, are also extracted from the manifest file to facilitate the construction of the transition relation.

In a nutshell, we construct the transition relation by the static analysis of the control- and data-flow of Android apps. Recall that a transition $A \xrightarrow{\alpha(\phi)} B$ contains a caller activity A, a callee activity B, an action α, and intent flags ϕ. It is noteworthy that a caller activity can start a callee activity, and finish itself at the same time by invoking the function finish(). In terms of modeling by an ASM, the action is finishStart if finish() is invoked, and start otherwise. As mentioned before, Android may use intents and the functions startActivity() and startActivityForResult() to activate activities. There are two types of intents: explicit intents and implicit intents. The former sets the name of the callee activity directly, while the latter declares the desired values of actions, categories and data fields. Activities that can be activated by implicit intents will declare intent filters in the manifest file. Android starts the activity that an implicit intent intends to run by matching the parameters of the intent with all the intent filters. If an implicit intent matches several intent filters of different activities, users can pick up which activity to launch.

We locate all the methods invoking functions startActivity() or startActivity-ForResult(), and all the activities accessing these methods, which are the caller activities in these transitions. We then exploit data-flow analysis to identify the sets of possible values of the parameters of the intents. From these values, we then obtain the intent flags directly. For explicit intents, we also obtain the callee activities whereas for implicit intents, we compute the set of callee activities by matching the values of these parameters with the intent filters obtained from the manifest file.[13]

Remark that, in this work, we focus on activities and ignore the other Android application components (e.g., services). Therefore, during the construction of the ASM model, we ignore all the occurrences of startActivity() and startActivityForResult() in the functions related to these components, e.g., "onServiceConnected()".

4.2 Static analysis of ASM

We perform static analysis for Android apps based on the ASM model. We shall focus on two types of analysis, i.e., configuration reachability and stack boundedness analysis, with applications. For simplicity, we restrict our attention to ASMs where the intent flag MTK is absent. This is *not* a severe restriction as the proportion of benchmarks containing the MTK flag is approximately 1% (37/3,245). However, it could tremendously facilitate our analysis because in each configuration of the ASM, the affinities of non-SIT tasks would be distinct.

Configuration reachability. The configuration reachability problem is formally defined as follows: Given an ASM $\mathcal{A}$ and a configuration ρ, decide whether $([A_0], 1) \overset{\mathcal{A}}{\Rightarrow} (\rho, \ell)$ for some ℓ, where $\overset{\mathcal{A}}{\Rightarrow}$ is the reflexive and transitive closure of $\xrightarrow{\mathcal{A}}$. Note here we ignore the components (τ, i) in tuples $(\rho, \ell) \xrightarrow[\tau,i]{\mathcal{A}} (\rho', \ell')$ and take $\xrightarrow{\mathcal{A}}$ as a binary relation over $\mathsf{Conf}_{\mathcal{A}} \times \mathbb{N}$.

In the sequel, we assume that there is a given constant bound $\hbar$ on the heights of tasks in the configurations, and the resulting reachability relation is called $\hbar$-*reachable.* (Evidently, $\hbar$-reachable implies reachable but not vice versa.) This assumption yields a finite, though exponential, state space, and the evolution of configurations can be captured by a finite state machine (FSM). To tackle the exponential state space we resort to the well-known symbolic model checker nuXmv [5] to provide an efficient and scalable analysis.

Our general approach is to translate an ASM $\mathcal{A}$ with a constant bound $\hbar$ (over heights of tasks) to an FSM $\mathcal{M}_{\mathcal{A}}$, the size of which is polynomial in the size of $\mathcal{A}$. Intuitively, the states of $\mathcal{M}_{\mathcal{A}}$ represent the configurations of $\mathcal{A}$ whose heights are bounded by $\hbar$, and the transitions of $\mathcal{M}_{\mathcal{A}}$ simulate the transition rules of $\mathcal{A}$. More technically, since each configuration ρ contains at most $m = |\mathsf{Aft}(\mathsf{Act} \setminus \mathsf{Act}_{\mathsf{SIT}})| + |\mathsf{Act}_{\mathsf{SIT}}|$ tasks and the height of each back stack is bounded by $\hbar$, ρ can be represented by a word of length exactly $m(\hbar + 1)$. In particular, each task is represented by a word of length $\hbar + 1$, where the last letter specifies the real activity of the task. (Dummy symbols $\bot \notin \mathsf{Act}$ are to be appended if the number of tasks in ρ is less than m or the height of a task is less than $\hbar$.)

As per the semantics of ASM, after some transition, a task may emerge to become the top task, which means that in the corresponding simulation, a subword of length $\hbar + 1$ will become the prefix of the new configuration representation. It turns out that, for the translation purpose, this is cumbersome to define, so we adapt the word representation of configurations as follows: an extra "pointer"

[13] cf. https://developer.android.com/guide/components/intentsfilters

word v of length m is introduced where each letter of v refers to a task currently in the configuration via its real activity. The order of the tasks can then be captured by permutations of v. (Note that if the number of tasks is less than m, then the dummy symbol $\perp$ is also used in v.) Generally, the "pointer" word is a word of real activities, possibly followed by multiple $\perp$'s, with a total length m. The detailed encoding is technical and is given in the full version [7].

The configuration reachability problem is fundamental to static program analysis and has various applications. Below we present an example; A further application is given in the stack boundedness section.

Back pattern analysis. The back pattern analysis computes, for a given activity A in $\mathcal{A}$, the set of activities B such that when pressing the back button, the foreground activity can switch from A to B. We shall denote this set by $\mathsf{Act}_{\mathsf{back}}(A)$. Such information is valuable for developers of Android apps, for instance, to validate the multitasking design of the app and to detect unexpected behaviors.

For a given $A \in \mathsf{Act}$, it is not hard to see that we can compute the desired set of activities B by solving for each $B \in \mathsf{Act}$, a slightly more general version of the configuration reachability problem, namely, whether a configuration matching the regular expression $e = A\perp^* B$ is reachable. The nuXmv tool facilitates handling this generalized version of the reachability problem. Furthermore, for each A and $B \in \mathsf{Act}_{\mathsf{back}}(A)$, a path can be generated by nuXmv to witness an occurrence of (some word matching) e.

Stack boundedness. Formally, an ASM $\mathcal{A}$ is said to be *stack-unbounded*, if for every $n \in \mathbb{N}$ there is a configuration (ρ, ℓ) of $\mathcal{A}$ such that $(\epsilon, 0) \overset{\mathcal{A}}{\Rightarrow} (\rho, \ell)$ and the height of ρ is at least n. We will consider a relaxation of stack-unboundedness, i.e., *k-stack unbounded* where k is a (purported small) natural number. Intuitively, an ASM $\mathcal{A}$ is k-stack unbounded if $\mathcal{A}$ is stack-unbounded and the unboundedness is caused by a particular task such that the height of the task is unbounded during the evolution which involves the interplay with at most k other tasks.

We are interested in the *stack boundedness problem* which is to decide whether a given ASM is stack unbounded. While this turns out to be difficult, we hypothesize that, most stack unbounded ASMs are actually k-stack unbounded for a small number k (normally, $k \leq 2$). As a result, as a practical solution, we can appeal to checking k-stack unboundedness for a small k. (See the full version [7] for justification.)

We start with some notations. Let $\mathsf{Act}_{\mathsf{real}}$ be the set of activities $A \in \mathsf{Act}$ such that one of the following conditions holds: 1) $\mathsf{Lmd}(A) = \mathsf{SIT}$, 2) $\mathsf{Lmd}(A) = \mathsf{STK}$, 3) $\mathsf{Lmd}(A) = \mathsf{STD}$ or STP, and A occurs in some transition $B \xrightarrow{\alpha(\phi)} A$ such that $\mathsf{Lmd}(B) = \mathsf{SIT}$ or $\phi \models \mathsf{NTK}$. Intuitively, $\mathsf{Act}_{\mathsf{real}}$ is the set of activities that may occur as a real activity of tasks.

Two activities $A, B \in \mathsf{Act}_{\mathsf{real}}$ are said to *represent different tasks* if one of the following conditions holds: 1) $\mathsf{Lmd}(A) = \mathsf{Lmd}(B) = \mathsf{SIT}$ and $A \neq B$, 2) $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\mathsf{Lmd}(B) \neq \mathsf{SIT}$, 3) $\mathsf{Lmd}(A) \neq \mathsf{SIT}$ and $\mathsf{Lmd}(B) = \mathsf{SIT}$, 4) $\mathsf{Lmd}(A) \neq \mathsf{SIT}$, $\mathsf{Lmd}(B) \neq \mathsf{SIT}$, and $\mathsf{Aft}(A) \neq \mathsf{Aft}(B)$. For each activity

$A \in \mathsf{Act}_{\mathsf{real}}$ such that $\mathsf{Lmd}(A) \neq \mathsf{SIT}$, let $\mathsf{Reach}(\Delta, A)$ denote the least subset $\Theta \subseteq \Delta$ satisfying that $B \xrightarrow{\alpha(\phi)} C \in \Theta$ (where $\alpha = \mathsf{start}$ or $\mathsf{finishStart}$) whenever the following two constraints are satisfied:

- $B = A$ or there exists a transition $A' \xrightarrow{\alpha'(\phi')} B \in \Theta$ (where $\alpha' = \mathsf{start}$ or $\mathsf{finishStart}$),
- $\mathsf{Lmd}(C) \neq \mathsf{SIT}$, and if $\mathsf{Lmd}(C) = \mathsf{STK}$ or $\phi \models \mathsf{NTK}$, then $\mathsf{Aft}(C) = \mathsf{Aft}(A)$.

Intuitively, $\mathsf{Reach}(\Delta, A)$ comprises all the transition rules that can be applied and once applied would retain an A-task as the top task. By abusing the notation slightly, $\mathsf{Reach}(\Delta, A)$ also denotes the graph whose edge set is $\mathsf{Reach}(\Delta, A)$.

$\mathsf{Reach}(\Delta, A)$ can be generalized to the case that $A \in \mathsf{Act}_{\mathsf{real}}$ and $\mathsf{Lmd}(A) = \mathsf{SIT}$, where $\mathsf{Reach}(\Delta, A)$ is regarded as the graph that contains a single node A without edges.

In the rest of this section, we will sketch a procedure to check stack unboundedness for $k = 0$. The underpinning idea is to search, for each $A \in \mathsf{Act}_{\mathsf{real}}$, a *witness cycle*, i.e., a sequence of transitions from $\mathsf{Reach}(\Delta, A)$, the execution of which would force the stack to grow indefinitely.

Formally, a witness cycle is a simple cycle in $\mathsf{Reach}(\Delta, A)$ of the form

$$\mathcal{C} = A_1 \xrightarrow{\alpha_1(\phi_1)} A_2 \cdots A_{n-1} \xrightarrow{\alpha_{n-1}(\phi_{n-1})} A_n$$

where $n \geq 2$ and $\alpha_i = \mathsf{start}$ or $\mathsf{finishStart}$ for each $i \in [n]$ satisfying the following two constraints:

[Non-clearing.] The content of an A-task is *not* cleared when C is executed. Namely, for each $i \in [n-1]$, $\phi_i \models \neg\mathsf{CTP}$, moreover, either $\phi_i \models \neg\mathsf{CTK}$, or $\phi_i \models \neg\mathsf{NTK}$ and $\mathsf{Lmd}(A_{i+1}) \neq \mathsf{STK}$ (intuitively, this means that CTK is not enabled, cf. Figure 1).

[Height-increasing.] The height of the task content is increasing after C is executed. Namely, it is required that $\sum_{i \in [n-1]} \mathit{weight}_{\mathcal{C}}(\tau_i) > 0$, where for each $i \in [n-1]$, $\tau_i = A_i \xrightarrow{\alpha_i(\phi_i)} A_{i+1}$ and $\mathit{weight}_{\mathcal{C}}(\tau_i)$ is defined as follows.

- If $\alpha_i = \mathsf{start}$, then
  - if $\phi_i \models \mathsf{RTF}$, then $\mathit{weight}_{\mathcal{C}}(\tau_i) = 0$,
  - if $\phi_i \models \neg\mathsf{RTF}$, $A_i = A_{i+1}$, and either $\phi_i \models \mathsf{STP}$ or $\mathsf{Lmd}(A_{i+1}) = \mathsf{STP}$, then $\mathit{weight}_{\mathcal{C}}(\tau_i) = 0$,
  - otherwise, $\mathit{weight}_{\mathcal{C}}(\tau_i) = 1$.
- If $\alpha_i = \mathsf{finishStart}$, then
  - if $\phi_i \models \mathsf{RTF}$, then $\mathit{weight}_{\mathcal{C}}(\tau_i) = -1$,
  - if $\phi_i \models \neg\mathsf{RTF}$, $A_i = A_{i+1}$, and either $\phi_i \models \mathsf{STP}$ or $\mathsf{Lmd}(A_{i+1}) = \mathsf{STP}$, then $\mathit{weight}_{\mathcal{C}}(\tau_i) = -1$,
  - otherwise, $\mathit{weight}_{\mathcal{C}}(\tau_i) = 0$.

If a witness cycle exists for some $A \in \mathsf{Act}_{\mathsf{real}}$, the algorithm returns "stack unbounded". Otherwise, if Δ is a directed acyclic graph, then the algorithm returns "stack bounded". Otherwise, the procedure reports "unknown".

The more general cases for $k \geq 1$ are much more technical and involved. We introduce the concept of "virtual transitions" for tasks to capture the situation that the content of a task can be indirectly modified by first jumping off the task and returning to the task later on. When this happens, the procedure adds virtual transitions for each task before checking the existence of witness cycles. The details of the procedure can be found in the full version [7].

As mentioned before, stack unboundedness suggests a potential security vulnerability. As a result, when this is spotted, it is desirable to synthesize a concrete transition sequence so that the developers can, for instance, follow this sequence to test and improve their apps. It turns out that the synthesis can be reduced to the more general version of the configuration reachability problem mentioned in the back pattern analysis. This can be easily incorporated and has been implemented in the tool.

5 Evaluation

We implement the procedures in Section 4 and develop a tool TaskDroid which comprises two modules, APP2ASM and ASMAnalyzer.

The former module builds ASM models from Android apps. The inputs of APP2ASM are either Android PacKage (apk) files or simply source codes. APP2ASM is based on the widely adopted Java bytecode analysis framework soot [15]. APP2ASM uses soot to create call graphs (CG) to represent the calling relationship between functions, and the control flow graphs (CFG) of functions to represent the control flow of the function bodies. APP2ASM includes two submodules, i.e., *Manifest Analyzer* and *Transition Extractor* which generate the signature Sig and the transition relation Δ of the ASM model respectively (cf. Definition 1). Manifest Analyzer extracts the manifest file from the source code or by decompiling the apk file. It then obtains the signature Sig from the manifest file. Moreover, it also gets the intent filters from the manifest file and passes them to the Transition Extractor for further analysis. Transition Extractor constructs the transition relation Δ from the call graph and the control flow graphs of functions (cf. Section 4.1). In order to make the model-building process more efficient, Transition Extractor applies the program slicing technique to extract those statements that are related to the attributes of intent objects corresponding to callee activity classes, intent flags, as well as actions, categories, and data of intent filters.

ASMAnalyzer carries out the static analysis on ASM models. ASMAnalyzer includes two submodules for *Reachability analysis* and *Boundedness analysis* respectively which implement the procedures given in Section 4.2. Note that the Reachability submodule can generate witness paths for reachability. The Boundedness submodule utilises the Reachability module to generate a path starting from the main activity when the ASM is found to be stack unbounded.

Benchmarks. The benchmarks comprise 4,496 apps (apk files) collected from three sources, i.e., the open-source F-Droid repository (`https://f-droid.org/`), the Google Play market, and app market Wandoujia (`https://www.wandoujia.com/`). The statistics of these apps can be found in Table 1. For F-Droid, we use a web crawler to download all the available apps. For Google Play (resp. the app market X), we download the first 500 apps of each of the 32 categories (resp. 14 categories) according to the displaying order. (Note that Google Play disallows direct app-downloading, so we use a third-party website APKLeecher `http://apkleecher.com/`.) Note that we have removed the apps that use fragment components which are not considered in this paper.

Table 1: Statistics of the benchmarks

Source	F-Droid	Google Play	Market Wandoujia
Num. of apps	674(15.0%)	2,068(46.0%)	1,754(39.0%)
Avg. Size	3.1 MB	15 MB	18 MB
Max. Size	158.0 MB	103.9 MB	428.7 MB
Total num. of apps	4,496		

We carry out all experiments on a Linux server with a CPU of Intel® Xeon® Processor E5-2680 v4 at 2.40GHz and 64GB memory.

5.1 Scalability of our approaches

APP2ASM. We evaluate the scalability of APP2ASM on the 4,496 apps, where the timeout is set to 600 seconds. The experimental results are shown in Table 2. Out of these 4,496 apps, there are 1,251 apps that the soot tool fails to handle. The average/maximum time of APP2ASM on these apps is 33.7/599.2 seconds. In the end, APP2ASM outputs 3,245 ASM models to the ASMAnalyzer module.

Table 2: Scalability: APP2ASM

Total num. of apps	Num. of soot-failing apps	Avg. time	Max. time
4,496	1,251	33.7s	599.2s
Num. of ASMs output by APP2ASM			
3,245			

Reachability analysis. We evaluate the performance of the Reachability analysis submodule by carrying out the back pattern analysis on the 3,245 ASMs (generated by the APP2ASM module), where the stack height bound $\hbar$ is set to 4 and the timeout period is set to 60 seconds. The experimental results are shown in Table 3. Only 9 (0.3%) ASMs out of the 3,245 ASMs time out. Furthermore, relatively large ASM models (e.g., with 50 activities and 128 transitions, or 72 activities and 72 transitions) can be handled successfully. The average (resp. maximum) running time is only 0.1 second (resp. 3.3 seconds).

Boundedness analysis. We evaluate the performance of the Boundedness submodule based on the same 3,245 ASM models. The parameter k (i..e., the number of interplaying tasks, cf. Section 4.2) is set to 2 and the timeout is set to 60 seconds. The experimental results are shown in Table 4. On this occasion, no timeout happened and the average (resp. maximum) running time is only 0.01 (resp. 0.4) second.

It is noteworthy that, for the reachability analysis, we hypothesize that the heights of involved tasks are bounded by a small number (i.e., $\hbar \leq 4$). Likewise,

Table 3: Scalability: Reachability (Back pattern)

Avg. size ($\vert\mathsf{Act}\vert, \vert\Delta\vert$) of ASMs	(6.7, 12.0)
Max. size ($\vert\mathsf{Act}\vert, \vert\Delta\vert$) of ASMs	(130, 292)/(63, 677)
Num. of T.O. ASMs	9(0.3%)
Max. size of non-T.O. ASMs	(50, 128)/(72, 72)
Avg. time	0.1s
Max. time	3.3s
Avg. of $\vert\mathsf{Act}_{\mathsf{back}}(A)\vert$	1.7
Max. of $\vert\mathsf{Act}_{\mathsf{back}}(A)\vert$	18

Table 4: Scalability: Boundedness

Total num. of ASMs	3,245
Avg. size ($\vert\mathsf{Act}\vert, \vert\Delta\vert$) of ASMs	(6.7, 12.0)
Max. size ($\vert\mathsf{Act}\vert, \vert\Delta\vert$) of ASMs	(130, 292)/(63, 677)
Num. of T.O. ASMs	0
Num. of stack-unbounded ASMs	989
Avg. time	0.01s
Max. time	0.4s

for the stack-boundedness analysis, we hypothesize that only a small number of tasks are involved ($k \leq 2$). In the full version [7] we empirically justify these hypotheses which give sufficiently precise results for the $\hbar$ and k we have set.

The experimental results demonstrate efficiency and scalability of the model construction and static analysis when applied to real-world Android apps.

5.2 Threat of stack unboundedness

As shown in the preceding section, TaskDroid has discovered that 989 ASMs out of 3,245 ASMs are stack unbounded (cf. Table 4). We investigate whether the stack unboundedness pose genuine threats in practice. Out of those 989 stack-unbounded ASMs, we select apps from F-Droid as examples to evaluate the threat of the stack-unboundedness.[14] We carry out the experiments using Android Emulator[15] to create a virtual device for Nexus 6 (RAM size 512 MB, heap size 16 MB, and Android version 7.1.1). Moreover, we use Monkey[16] and ADB (Android Debug Bridge[17]) tools. The experiments proceed in the following steps: (1) Generate a witness cycle as well as a reachability path for stack unboundedness. (Note that the witness cycle is a segment of the reachability path.) (2) For each activity A in the reachability path, locate the UI widget corresponding to A by reading the source code and locating the occurrence of the intent object corresponding to the activation of A. (3) Find the coordinates of the UI widgets which are used to generate a Monkey script, specifically, a

[14] The experiments need considerable manual work and are very time-consuming, we choose to conduct experiments on the F-Droid apps only as they are relatively small in size. We plan to carry out more extensive experiments in the near future.

[15] `https://developer.android.com/studio/run/emulator`

[16] `http://developer.android.com/tools/help/monkey.html`

[17] `https://developer.android.com/studio/command-line/adb`

sequence of click operations, to simulate the witness cycle. (4) Install the app in the virtual device, simulate the sequence of click operations manually until reaching the witness cycle, then use Monkey to repeatedly run the script (corresponding to the witness cycle). We use ADB to obtain the number of activities in tasks and calculate the number of repetitions of the witness cycle.

The results of the experiments are reported in Table 5. Out of the analysed 101 F-Droid apps, the witness cycles synthesized by TaskDroid can be successfully simulated in 29 apps. After hundreds or thousands of repetitions of the witness cycle, the 29 apps end up with either app crash, or black screen, or even rebooting of device. These suggest that stack-unbounded apps can be potentially harmful to, and thus a vulnerability of, the Android system, highlighting the importance of such an analysis. For the other 72 apps, we were unable to simulate the witness cycles, due to the following reasons: login is required (23 apps), apps crash immediately after launching (14 apps), ASM models are imprecise (35 apps) so the potential threat returned by TaskDroid may be spurious.

Table 5: Threat of stack unboundedness

Abnormal behavior	Num. of apps	Num. of repetitions of the witness cycle		
		Avg.	Min.	Max.
App crash	21	709	66	1418
Black screen	6	1002	228	1213
Device reboot	2	2451	406	4495

6 Conclusion

We have provided a rigorous formalization of the Android multitasking mechanism, which gives a considerably more complete and concise account of the evolution of the Android task stack in relation to activity activation, and highlights the discrepancy between the semantics of different Android versions. Based on the formalized Android stack machine model and its semantics, we have provided new modeling and static analysis methods for Android apps, which have been implemented in a prototype tool TaskDroid. Experiments on large-scale benchmarks confirmed the efficacy and efficiency of our approaches.

Future work includes further improving the precision of the ASM modeling and analysis, more extensive experiments on Android app markets, and in-depth investigations of the decidability and complexity of static analysis.

Acknowledgements. This work is partially supported by the NSFC grants (No. 61672505, 61872340, 61572478), Key Research Program of Frontier Sciences, Chinese Academy of Sciences (No. QYZDJ-SSW-JSC036), Guangdong Science and Technology Department grant (No. 2018B010107004), the Open Project of Shanghai Key Laboratory of Trustworthy Computing (No. 07dz22304201601), and the INRIA-CAS joint research project VIP. Taolue Chen is also partially supported by Birkbeck BEI School Project (ARTEFACT) and an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2018A16).

References

1. Android documentation. `https://developer.android.com/guide/components/activities/tasks-and-back-stack.html`.
2. T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA*, pages 641–660, 2013.
3. A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *SP 2015*, pages 931–948, 2015.
4. R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuxmv symbolic model checker. In *CAV 2014*, pages 334–342, 2014.
5. R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuxmv symbolic model checker. In *CAV 2014*, pages 334–342, 2014.
6. T. Chen, J. He, F. Song, G. Wang, Z. Wu, and J. Yan. Android stack machine. In *CAV 2018*, pages 487–504, 2018.
7. J. He, T. Chen, P. Wang, Z. Wu, and J. Yan. Android multitasking mechanism: Formal semantics and static analysis of apps (Full version). 2019. Available at `https://github.com/LoringHe/TaskDroid`.
8. S. Lee, S. Hwang, and S. Ryu. All about activity injection: threats, semantics, and detection. In *ASE 2017*, pages 252–262, 2017.
9. S. Lee, S. Hwang, and S. Ryu. Operational semantics for the android activity activation mechanism. Technical report, 2017.
10. L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. L. Traon. Static analysis of android apps: A systematic literature review. *Information & Software Technology*, 88:67–95, 2017.
11. J. Liu, D. Wu, and J. Xue. Tdroid: exposing app switching attacks in android with control flow specialization. In *ASE 2018*, pages 236–247, 2018.
12. D. Octeau, S. Jha, M. Dering, P. D. McDaniel, A. Bartel, L. Li, J. Klein, and Y. L. Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *POPL 2016*, pages 469–484, 2016.
13. D. Octeau, D. Luchaup, S. Jha, and P. D. McDaniel. Composite constant propagation and its application to android program analysis. *IEEE Trans. Software Eng.*, 42(11):999–1014, 2016.
14. C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *USENIX Security*, pages 945–959, 2015.
15. R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot — a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research*, 1999.
16. Y. Xiao, G. Bai, J. Mao, Z. Liang, and W. Cheng. Privilege leakage and information stealing through the android task mechanism. In *PAC 2017*, pages 152–163, 2017.
17. J. Yan, T. Wu, J. Yan, and J. Zhang. Widget-sensitive and back-stack-aware GUI exploration for testing android apps. In *QRS 2017*, pages 42–53, 2017.
18. S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static window transition graphs for android. In *ASE 2015*, pages 658–668, 2015.
19. Y. Zhang, Y. Sui, and J. Xue. Launch-mode-aware context-sensitive activity transition analysis. In *ICSE 2018*, pages 598–608, 2018.
20. J. Zhao, A. Albarghouthi, V. Rastogi, S. Jha, and D. Octeau. Neural-augmented static analysis of android communication. In *FSE 2018*, pages 342–353, 2018.