Detecting Resource Utilization Bugs Induced by Variant Lifecycles in Android

Yifei Lu
lyf@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology,
Software Institute, Nanjing University
Nanjing, China

Minxue Pan*
mxp@nju.edu.cn
State Key Laboratory for Novel Software Technology,
Software Institute, Nanjing University
Nanjing, China

Yu Pei
csypei@comp.polyu.edu.hk
Department of Computing,
The Hong Kong Polytechnic University
Hong Kong, China

Xuandong Li
lxd@nju.edu.cn
State Key Laboratory for Novel Software Technology,
Nanjing University
Nanjing, China

ABSTRACT

The lifecycle models of Android components such as Activities and Fragments predefine the possible orders in which the components' callback methods will be invoked during app executions. Correspondingly, resource utilization operations performed by Android components must comply with all possible lifecycles to ensure safe utilization of the resources in all circumstances, which, however, can be challenging to achieve. In response to the challenge, various techniques have been developed to detect resource utilization bugs that manifest themselves when components go through *common* lifecycles, but the fact that Android components may execute their callback methods in uncommon orders, leading to variant component lifecycles, has largely been overlooked by the existing techniques.

In this paper, we first identify three variant lifecycles for Android Activities and Fragments and then develop a technique called VALA to automatically detect bugs in Android apps that are induced by the variant lifecycles and may cause resource utilization errors like resource leaks and data losses. In an experimental evaluation conducted on 35 Android apps, a supporting tool for the VALA technique automatically detected 8 resource utilization bugs. All the 8 bugs were manually confirmed to be real defects and 7 of them were reported for the first time.

KEYWORDS

Android applications, variant lifecycles, resource utilization bugs, static analysis

1 INTRODUCTION

Android has been the unquestionable world leader of the mobile operating systems market in the past few years [29, 30], and Android apps are becoming more and more indispensable for many people's everyday life and work. To develop an Android app that always functions as expected in all circumstances, however, is never an easy task. One important difference between Android apps and traditional desktop applications is that the lifecycle models of Android components such as activities and fragments are predefined by the platform architecture. The lifecycle model of a component not only stipulates how the component transits through different states in response to various user and system events but also defines a set of callback methods, or *entrypoints* [**? ?**], that will be invoked automatically by the Android framework to handle events triggered on the component during app executions. Correspondingly, to program a new Android component essentially boils down to overriding those entrypoints with appropriate implementations.

Although the concept of lifecycle model is straightforward to understand, to organize necessary operations into various entrypoints so that they always correctly implement the required functionalities is challenging in practice, largely because different component states and input events may lead to distinct entrypoint execution sequences, or distinct lifecycles, in various app executions, while the correct implementation of most functionalities demands that the relevant operations are performed in specific orders. For example, one key aspect of safe and efficient resource utilization in Android apps is to always properly release the allocated resources when they are no longer needed, no matter how users interact with the app or how the states of the app's activities and fragments change. In view of the challenge, various techniques have been developed in the past few years to dynamically [10, 27, 28] or statically [9, 15, 40] detect resource utilization bugs in Android apps due to non-conformity with the Android component lifecycles. These techniques concern the common lifecycles of Android components and have been applied to detect real bugs in popular Android apps.

However, the execution orders of the component entrypoints (and the operations they perform in turn) may deviate from the common cases under specific circumstances, resulting in variant lifecycles for those components. The variant lifecycles have largely been overlooked by the existing techniques, partly because they take place rarely, and partly because the descriptions of the uncommon entrypoint execution orders are scattered at multiple locations of the Android documentation.

Failing to take the deviations into account significantly impairs the effectiveness of existing techniques in the face of variant lifecycles. On the one hand, since the chance of accidentally running an app under those specific circumstances is very slim, existing techniques that are based on dynamic analysis are unlikely to be

*Corresponding author.

successful in triggering variant lifecycles; On the other hand, state-of-the-art static-analysis-based techniques cannot effectively discover those bugs either because, to the best of our knowledge, all of them derive app executions from Android component lifecycle models, but none of their models includes information about the uncommon entrypoint execution orders. Moreover, naively incorporating the uncommon entrypoint execution orders into the techniques' lifecycle models will most likely be counterproductive, since the resultant model will significantly increase not only the costs for analyzing the apps but also the number of false positives produced.

In this work, we first manually analyze the Android API reference to identify three scenarios under which the entrypoints defined within Android activities and fragments will be executed in uncommon orders, and then propose a technique, called VALA (**VA**riant **L**ifecycle **A**nalyzer), to automatically detect bugs in Android apps that are caused by the three related variant lifecycles and will lead to resource utilization errors like resource leak errors and data loss errors. Given that the resource utilization code causing the errors can be scattered across different entrypoints and guarded by specific conditions, VALA applies complex static program analysis techniques, instead of simple search algorithms based on pattern matching, to achieve higher effectiveness. For the analysis to be efficient, we design VALA on the basis of two considerations. First, we refrain from conducting an expensive, global analysis of all the entrypoints of Android components, since the differences between common and variant lifecycles concern only a few types of entrypoints. Second, we focus on resources referenced by member fields of Android components in the analysis. Because we target resource utilization bugs caused by variant lifecycle entrypoints and entrypoints are functions without user-defined parameters, resources accessible across multiple entrypoints of a component are typically referenced by member fields.

We have developed a tool, also named VALA, to support the easy application of the VALA technique. Taking an Android app, a group of resource utilization operations (RUOs), and a set of resource utilization requirements as the input, VALA first checks its activities and fragments to find out whether any of them may go through variant lifecycles and thus risk containing resource utilization bugs that we are looking for. Then, for each component that does face the risk, VALA performs a context-sensitive data-flow analysis to gather possible execution sequences of the RUOs in relevant entrypoints of the component. In the end, VALA concatenates the operation sequences gathered from entrypoints in orders stipulated by the variant lifecycles, and it reports a resource utilization bug if a resultant operation sequence violates the resource utilization requirements. In an experimental evaluation conducted on 35 Android apps, VALA detected 8 resource utilization bugs due to variant lifecycles. All the 8 bugs were manually confirmed to be real defects, and 7 of them were reported to the developers of the corresponding apps for the first time. Besides, the average time VALA needed to analyze each app was less than half a minute. Such results suggest that VALA is both effective and efficient in detecting resource utilization bugs in Android apps that are induced by variant lifecycles.

The contributions this paper makes are as the following:

(1) We identify an important class of bugs that are caused by variant lifecycles of Android components, and we manually analyze the official Android documentation to spot three scenarios under which Android activities and fragments may go through variant lifecycles.
(2) We develop a technique called VALA that applies context-sensitive data-flow analysis to collect possible sequences of RUOs from component entrypoints and detects resource utilization bugs based on the gathered operation sequences and the resource utilization requirements.
(3) We implement the VALA technique into a tool with the same name and make it publicly available online*. We empirically evaluate the effectiveness and efficiency of VALA on 35 real-world Android apps. The experimental results suggest VALA is both effective and efficient in detecting resource utilization bugs induced by variant lifecycles in Android apps.

The rest of this paper is organized as the following. Section 2 introduces the background and motivation of the work. Section 3 elaborates how the VALA technique detects resource utilization bugs caused by Android variant lifecycles step by step. Section 4 reports on the experiments we conducted to evaluate VALA and the results of the experiments. Section 5 reviews recent work that is closely related to ours. Section 6 concludes the paper.

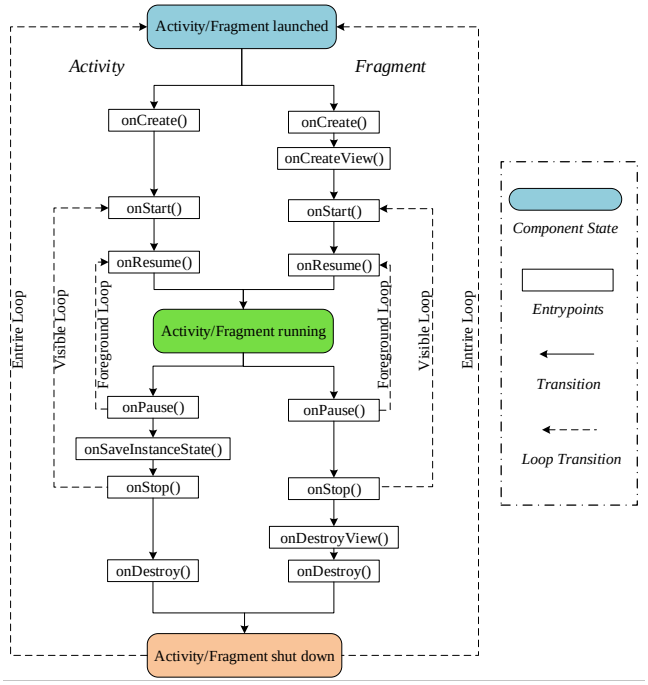


Figure 1: A basic lifecycle model for Android activities and fragments. States like "Activity/Fragment started" and "Activity/Fragment paused" have been omitted for simplicity.

*https://bit.ly/3DMSUXB.

2 BACKGROUND AND MOTIVATION

In this section, we first make a brief introduction to a basic lifecycle model for Android activities and fragments, and then illustrate through examples how RUOs that work fine with common lifecycles may cause bugs in the face of variant lifecycles.

2.1 A Basic Lifecycle Model for Android Activities and Fragments

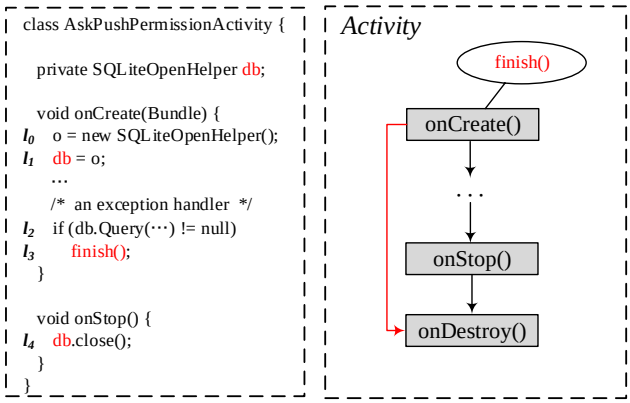
Activities and fragments are two important and frequently-used types of Android components. As essential building blocks of Android apps, an **Activity** abstracts a single screen that interacts with users, whereas a **Fragment** abstracts a modular section of an activity. Activities and fragments are all lifecycle-aware, in the sense that their lifecycle models are predefined by the Android platform architecture. Particularly, the lifecycle model of an activity or a fragment stipulates how the component transits through different states in response to various user and system events. The lifecycle model also defines a set of callback methods, or entrypoints, which the Android framework will automatically invoke to handle events triggered on the component. By overriding the entrypoints with appropriate implementations, Android developers program the desired functionalities into activities and fragments.

Figure 1 gives a basic lifecycle model for Android activities and fragments, with states like "Activity/Fragment started" and "Activity/Fragment paused" ("started" and "paused" for short) being omitted for simplicity reasons. The figure shows, e.g., that an activity's entrypoints `onCreate`, `onStart`, and `onResume` are usually invoked in sequence during the activity's transition from state "launched" to state "running", while four other entrypoints are usually invoked in order when the activity transits from state "running" to state "shut down".

The basic model also highlights three key loops in common lifecycles [27], including i) the *entire* loop that contains all state transitions and entrypoint invocations from `onCreate` through `onDestroy`, ii) the *visible* loop that contains state transitions and entrypoint invocations from `onStart` through `onStop`, and iii) the *foreground* loop that contains state transitions and entrypoint invocations from `onResume` through `onPause`. For instance, a configuration change from landscape orientation to portrait orientation due to the rotation of the device may trigger an execution of the entire loop, turning off and on the screen may trigger an execution of the visible loop, while covering part of a foreground activity or fragment with a GUI element and then discarding that element may trigger an execution of the foreground loop.

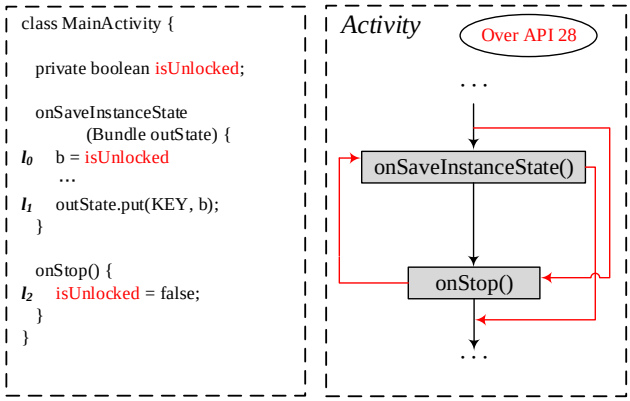
2.2 Motivating Examples

Figure 2 illustrates how a variant lifecycle may cause a resource leak error during the execution of `AskPushPermissionActivity` from app *GmsCore*. Particularly, as shown in Figure 2a, the activity declares a member field `db` to hold a helper object through which it can operate the underlying database, and as done in the common practice, the activity instantiates the helper object in entrypoint `onCreate` and releases the helper object in entrypoint `onStop`. Such implementation would work perfectly fine when the activity goes through its common lifecycles, where entrypoint `onStop` is surely



(a) Code snippet (b) Entrypoint execution order

Figure 2: A resource leak error caused by variant lifecycles



(a) Code snippet (b) Entrypoint execution order

Figure 3: A data loss error caused by variant lifecycles

invoked, causing the helper object to be released, before the activity shuts down.

However, due to the invocation to method `finish` in its entrypoint `onCreate`, the activity may go through a variant lifecycle during its execution, as shown in Figure 2b. More concretely, according to the Android API reference [6], calling method `finish` on an Activity will cause (1) the Activity's entrypoint `onDestroy` to be executed directly and then (2) the Activity to be destroyed. Since `AskPushPermissionActivity` invokes method `finish` in its entrypoint `onCreate`, executing the method invocation will effectively destroy the activity without executing entrypoint `onStop`, which will cause a resource leak error since the helper object is not properly released.

Figure 3 illustrates how another variant lifecycle may cause a data loss error during the execution of `MainActivity` from app *To-Do List*. As shown in Figure 3a, the activity stores the value of its member field `isUnlocked` in entrypoint `onSaveInstanceState`, and it also sets the value of `isUnlocked` to `false` in entrypoint `onStop`. When the activity goes through common lifecycles summarized in Figure 1, entrypoint `onSaveInstanceState` is always

executed before entrypoint `onStop`, and therefore the actual value of `isUnlocked` to be saved into `outState` can be either `true` or `false`, depending on the activity's internal state, which is most likely also the behavior developers expected for the activity.

The execution order of entrypoints `onSaveInstanceState` and `onStop`, however, will be different for apps targeting more recent Android versions. Specifically, starting from Android 9 (or API version 28), entrypoint `onStop` is called before entrypoint `onSaveInstanceState`, as shown in Figure 3b. Such change will cause a data loss error to `MainActivity`, since `isUnlocked` will always be set to `false` before its value is stored. Since data loss errors do not always cause problems that are directly visible to users, they can be hard to notice.

The two examples clearly demonstrate that, if not appropriately handled, variant lifecycles may lead to resource utilization errors. Such variant lifecycles, however, have largely been overlooked by Android developers and researchers, and we can see two main reasons for that. First, the variant lifecycles are not clearly stated in the official guidelines to Android component lifecycle model [7] and can be hard to notice. Actually, the fact that calling method `finish` on an activity will cause the execution of that activity's entrypoint `onDestroy` and then the destruction of the activity is only explained in the document of methods `onCreate` and `onDestroy`, but not in that of method `finish`, probably because the main purpose of `finish` is to destroy an activity, rather than to alter the common activity lifecycle. Second, to understand that implementations like the one shown in Figure 2a are faulty, programmers need to not only be aware of the impact of method `finish` on activity and fragment lifecycles but also relate that impact to the utilization of resources in those components, but the relation can be missed easily.

Meanwhile, existing techniques that detect resource utilization bugs in Android apps due to non-conformity with the Android component lifecycle model are less likely to be effective in the face of variant lifecycles. On the one hand, activities and fragments only go through variant lifecycles under specific conditions, while techniques that are based on dynamic analysis can hardly run into app states satisfying those conditions by chance. For example, the invocation to method `finish` in Figure 2a is part of an exception handler and it is guarded by an condition, making the chance of actually triggering the invocation with existing dynamic analysis based techniques very slim. On the other hand, techniques that are based on static analysis rely on Android component lifecycle models to derive app behaviors, but none of their models contains information about variant lifecycles, and to accurately model the interplay between entrypoints and RUOs is non-trivial. For instance, while the invocation to `finish` in `onCreate` will always cause `AskPushPermissionActivity` to go through a variant lifecycle, no resource leak error would occur if the helper object is instantiated after the call to `finish`. To avoid raising false alarms in such a situation, precise, control- and data-dependent analysis of the entrypoints is needed, which, however, is challenging.

3 THE VALA TECHNIQUE

In this section, we explain in detail how we identify variant Android component lifecycles, what RUOs we consider in this work, and how the VALA technique detects resource utilization bugs caused by those variant lifecycles step by step. Note that, while resource utilization errors may also occur inside individual entrypoints or in common Android component lifecycles, VALA is not devised to detect those errors, and several techniques have been proposed to effectively reveal those errors [9, 14, 34].

Table 1: Basic information about the Activity and Fragment classes in various Android libraries.

Library	Type	Root Class	#SubCls	#Mthd
basic	Activity	android.app.Activity	9	372
	Fragment	android.app.Fragment	4	167
support	Activity	android.support.v4.app.FragmentActivity	1	94
	Fragment	android.support.v4.app.Fragment	25	567
androidx	Activity	androidx.activity.ComponentActivity	2	206
	Fragment	androidx.fragment.app.Fragment	32	623
Overall	-	-	73	2029

We will use the following notations in the explanation. Given an Android app P, let A and G be the set of activities and fragments defined in P, respectively, $C = A \cup G$, while F and EP be the set of all member fields and entrypoints defined in C, respectively. Functions $\theta : C \rightarrow 2^F$ and $\psi : C \rightarrow 2^{EP}$ map each component in C to the set of member fields and entrypoints defined in the component, respectively. Functions $\theta' : F \rightarrow C$ and $\psi' : EP \rightarrow C$ map each member filed and entrypoint defined in an activity or fragment to its containing component.

3.1 Variant Lifecycles of Android Activities and Fragments

We first performed a comprehensive investigation into the Android API reference to manually gather information about variant lifecycles of Android activities and fragments, including the conditions under which activities and fragments will go through variant lifecycles and the entrypoints whose execution orders will be different in those variant lifecycles. We gathered the information from the Android API reference, instead of the other materials about Android, because the API reference is well-maintained, up-to-date, and supposed to be thorough in the sense that it should contain all the information that developers need to know to correctly program and interact with activities and fragments.

To the best of our knowledge, three libraries are widely used in implementing Android activities and fragments, namely the *basic* library introduced at the very beginning of the platform, the *support* library that provides a compatibility layer to make Android development against multiple API versions easier, and the *androidx* library that also provides backward compatibility across Android releases and aims to supersede the *support* library. While the *support* library is no longer maintained, and Google recommends that the *androidx* library should be used instead in future developments, we include the library in this study because many time-honored Android apps still rely on it. Table 1 lists for each library (Library) and each type of component (Type), i.e., Activity or Fragment, the root class (Root), the number of its subclasses (#SubCLs), and the total number of APIs defined in those subclasses (#Mthd).

Table 2: Types of variant lifecycles for Android activities and fragments.

ID	Component	Condition	Variation	Relevant Entrypoints
Skip1	Activity	Method `finish` is called in `onCreate`	Entrypoint `onDestroy` will be invoked and the activity will be destroyed immediately after `onCreate` returns, skipping entrypoints like `onStop`.	`onCreate`, `onDestroy`, `onStop`
Skip2	Fragment	`setRetainInstance(true)` is called to retain the Fragment instance	Entrypoints `onDestroy` and `onCreate` will be skipped, while the other entrypoints will still be executed, in the following entire loop.	`onCreate`, `onDestroy`, `onCreateView`
Swap	Activity	App is running on Android 9.0 or later	Entrypoint `onSaveInstanceState` will be executed after, instead of before, `onStop`.	`onSaveInstanceState`, `onStop`

Table 3: Sample resource types and their utilization operations considered by VALA.

RESOURCE CLASS	REQUEST OP.	RELEASE OP.
android.database.sqlite.SQLiteOpenHelper	new	close
com.google.android.exoplayer2.ExoPlayerFactory	newInstance	release
org.apache.http.impl.client.DefaultHttpClient	connect	close
android.hardware.SensorManager	registerListener	unregisterListener
java.util.logging.FileHandler	new	close
io.reactivex.disposables.CompositeDisposable	new	dispose
DATA STORE CLASS	**SAVE OP.**	**EDIT OP.**[†]
android.database.sqlite.SQLiteDatabase	insert	`p.f=q`
android.os.Bundle	putSize	`p.f=q`
android.content.SharedPreferences.Editor	putBoolean	`p.f=q`

[†] Here, `p` is an alias of `this` activity or fragment, while `f` is a member field of the activity or fragment, and the save operation stores the value of `f`.

To collect the variant lifecycles, we went through the following process: We first studied the Android official guides and collected all the mentioned lifecycle transitions. We then searched all methods' descriptions in the Android API references of the three libraries for mentions of lifecycle transitions. Those transitions that are not listed in the official guides but mentioned in the API references imply variant lifecycles. In total, three types of variant lifecycles were successfully identified from the API references. Table 2 lists, for each variant lifecycle type, its ID, the type of Android components it concerns (Component), the condition under which it will be triggered (Condition), its impact on entrypoint execution order (Variation), and the entrypoints that are relevent to it (Relevant Entrypoints). The three variant lifecycles are independent of the underlying Android libraries used, and we have seen instances of variant lifecycles of types Skip1 and Swap in Section 2.2. Previous research has studied the existence of implicit control flow transitions due to invocations of callback methods in the Android framework [3, 4]. But the callback methods considered there are communicated to the Android framework by using *registration* methods and are not strictly related to Android component lifecycles.

3.2 Resource Utilization Operations

Since the three types of variant lifecycles influence the entrypoint execution orders differently, the possible types of resource utilization errors they may cause are also different. In this work, we consider mainly two kinds of such bugs. For variant lifecycles that may cause the executions of certain entrypoints to be skipped, i.e., those of types Skip1 and Skip2, resource leak errors may occur if entrypoints responsible for releasing the allocated resources are skipped. For variant lifecycles that may cause the executions of entrypoints to be swapped, i.e., those of type Swap, data loss errors may occur if the data to be stored is modified in entrypoint `onStop`. Since VALA aims to detect bugs caused by variant lifecycles, we are only concerned with RUOs directly or indirectly performed by entrypoints here. With this design, the requirements for correct resource utilization that we want to fulfill across entrypoints, even in the face of variant lifecycles, can essentially be summarized as the following. First, all allocated resources should be properly released. Second, all modifications to app data should be faithfully saved.

We collected in total 51 pairs of resource *request*/*release* operations from previous studies [14, 24] and frequently-used third-party libraries [12, 26]. We also gathered from a previous work [28] 60 operations that are often invoked in entrypoint `onSaveInstanceState` to store app data. Since the purpose of entrypoint `onSaveInstanceState`, as suggested by its name, is to save the state of `this` component instance, we consider an assignment to a field of `this` activity or fragment as an edit operation if the field is part of the instance state and therefore should be saved. For example, if `this` activity has a member field named `x` and `SharedPreferences.Editor.putBoolean(key, x)` is a save operation performed by `onSaveInstanceState`, assignment `this.x = q` in entrypoint `onStop` will be considered as an edit operation associated with the save operation. Table 3 lists some of the resource request/release operation pairs and some data save/edit operations. The full list of RUOs that VALA handles is publicly available for download at https://bit.ly/2Y0uycy.

Overall, VALA is concerned with four types of RUOs, namely *REQ*, *REL*, *SAV* and *EDT*, for request, release, save, and edit operations, respectively. Let $\mathcal{T} = \{REQ, REL, SAV, EDT\}$ be the set of RUO types.

3.3 Resource Utilization Operation Sequences from Entrypoints

To gather all RUO sequences from an entrypoint, VALA repeatedly applies a context-sensitive data-flow analysis, each time to an individual execution path of the entrypoint for effectiveness. In Android, resources that need to be accessed across entrypoints are typically referenced by member fields since entrypoints are functions that do not have user-defined parameters. Therefore, VALA focuses on one member field of the entrypoint's containing activity or fragment in the analysis of each execution path. To gather

stmt ::= **assign** | **call**
assign ::= $p = q \mid p.f = q \mid p = q.f$ **call** ::= $p = q.m(\bar{r})$
$p, q, r \in Variable, \quad f \in Field, \quad m \in Method$

Figure 4: Syntax of the core language.

an accurate sequence of RUOs performed on a member field, the analysis especially keeps track of the aliasing relation between the member field and other expressions.

More concretely, given an entrypoint $ep \in EP$, VALA first constructs a program structure tree (PST) [17] for ep by unfolding the loops and inlining the non-library methods invoked directly or indirectly by ep. In the PST, nodes are simple statements from P, with the root node corresponding to the entry of ep and the leaf nodes corresponding to the exit of ep, while edges are possible control flows. Let Ω be the set of all paths from the root node to leaf nodes in the tree. Each path $\omega \in \Omega$ can be mapped to an execution of ep, if there do exist appropriate inputs to the entrypoint that can make conditions along the path evaluate to values matching the branches being taken. Then, VALA constructs a *derived* program P_ω from each $\omega \in \Omega$ by collecting the statements along ω. The following symbolic analysis is applied to P_ω. Figure 4 gives the syntax of the core language of P_ω and we will use the language to present the algorithm for the analysis. Note that, due to its specific construction process, P_ω contains neither control structures like branches and loops nor invocations to non-library methods. Instead, program P_ω contains simply a sequence of simple statements to be executed in order. Suppose the first statement of P_ω is at index 0.

During the symbolic analysis of program P_ω targeting a member field $f \in \theta(\psi'(ep))$, a state $\langle idx, \phi, \epsilon \rangle$ of the program contains the index idx of the next statement to execute, the set ϕ of alias expressions of f in the form of access paths [5, 20], and the set ϵ of RUOs performed on f so far. The inference rules in Figure 5 present the operational semantics of the statements in P_ω. The following notations are used in defining the inference rules. Function pre takes a set ϕ of access paths and a access path r as the parameters and returns the set of access paths from ϕ that have r as a prefix. Function $replace$ takes a set ϕ_a of access paths, a access path r, and another set ϕ_b of access paths as the parameters and returns a new set of access paths by taking each access path from ϕ_a and replacing its prefix r, when exists, with each access path in ϕ_b. Function RUO constructs a RUO from a statement. Function $backtrace$ takes a statement index idx and a access path r as the input and returns a pair $\langle \phi, \epsilon \rangle$ as the output, where ϕ is the set of aliasing access paths of r and ϵ is the set of RUOs performed on r, when the program P_ω executes till index idx. For performance reasons, VALA calculates the result of function $backtrace$ by conducting a backward analysis [31].

Particularly, VALA combines taint propagation (i.e., to derive new access paths when the variable on the right-hand side is tainted) and strong update (i.e., to remove access paths from ϕ if they can be mapped to the variable on the left-hand side) techniques from traditional interprocedural taint analysis with a backward on-demand alias analysis [31] to analyze assignment statements. For instance, when the assignment is of form $this.f = q$ ($assign_{2.2}$ in Figure 5), the member field of interest is directed to another (possibly different) object. In this case, VALA first performs backward analysis to find out the collection of access paths to q at that location and the set of RUOs applied to q before the assignment. Then, the tool replaces the RUOs and access paths collected for $this.f$ so far with the sets of operations and access paths returned by the backward analysis, since the previously collected operations and access paths will no longer affect the use of $this.f$ after the assignment. The current assignment statement is also treated as an EDT operation and added to the set.

VALA applies two different strategies in handling calls to library methods, depending on whether the method invocation constitutes a RUO or not. When yes, the tool constructs a RUO from the statement and adds it to the operation set. When no, the tool simply skips the method call, since the call is irrelevant to the type of bugs that we are looking for.

The analysis reaches its end when the last statement in P_ω has been processed. The RUO set in the final state of the analysis then contains all the operations on f that may be perceived outside ep.

Consider the code snippet in Figure 2a for example. A derived program P' for entrypoint `onCreate` may contain statements corresponding to lines $l_0, l_1, \ldots, l_2, l_3$. The analysis of P' w.r.t. field `db` will start from an initial state $\langle 0, \{this.db\}, \emptyset \rangle$. At statement l_0, since the statement is not related to db, the tool will simply change the state to $\langle 1, \{this.db\}, \emptyset \rangle$. At statement l_1, the assignment to db will enlarge ϕ with the local reference `o` and an EDT operation will also be collected. Moreover, a backward analysis will be performed to look for aliases of `o` and operations on these aliases, and the program state will be updated to incorporate the analysis results. As the result, a REQ operation will be constructed from the instantiation statement at l_0, and the program state will become $\langle 2, \{this.db\}, \{RUO(P'[0]), RUO(P'[1])\} \rangle$. If no other RUOs, including assignments, is applied to db in P', the set of RUOs gathered from P' will be $\{RUO(P'[0]), RUO(P'[1])\}$.

Given the set of RUOs gathered from a derived program for an entrypoint, we can naturally turn that set into a sequence based on their execution order. For an entrypoint $ep \in EP_P$, we use $\mathcal{E}_{ep}$ to denote the set of all RUO sequences gathered from the derived programs of ep.

3.4 Detection of Resource Utilization Bugs Induced by Variant Lifecycles

In this section, we elaborate on how VALA detects resource utilization bugs induced by the three types of variant lifecycles.

Considering that the variant lifecycles are only triggered under specific circumstances, e.g., when certain methods are invoked, we extend the mechanism for gathering resource utilization sequences described in Section 3.3 so that invocations to the variant lifecycle-triggering methods are also recorded during the analysis. Particularly, we introduce a new type $NULL$ of RUO, construct $NULL$ operations from invocations to variant lifecycle-triggering methods when identifying RUOs, and add the $NULL$ operations to the resource utlization operation sequences gathered from entrypoints. Take the code snippet in Figure 2a for example. When encountering the invocation to method `finish`, VALA builds a new

$$assign_1: \quad \frac{P[idx] = p = q \qquad \phi' = \phi \setminus pre(\phi, p) \cup replace(pre(\phi, q), q, \{p\})}{\langle idx, \phi, \epsilon\rangle \rightarrow\!\!\!\!\!\!\rightarrow [\![\langle P[idx]\rangle]\!] \Longrightarrow \langle idx + 1, \phi', \epsilon\rangle}$$

$$assign_{2.1}: \quad \frac{P[idx] = p.f1 = q \quad p.f1 \neq this.f \quad backtrace(idx, p) = \langle \phi_1, \epsilon_1\rangle \quad \phi' = \phi \setminus pre(\phi, p.f1) \cup replace(pre(\phi, q), q, \{p.f1\} \cup \phi_1.f1) \quad \epsilon' = \epsilon \cup \epsilon_1 \cup \{RUO(P[idx])\}}{\langle idx, \phi, \epsilon\rangle \rightarrow\!\!\!\!\!\!\rightarrow [\![\langle P[idx]\rangle]\!] \Longrightarrow \langle idx + 1, \phi', \epsilon'\rangle}$$

$$assign_{2.2}: \quad \frac{P[idx] = p.f1 = q \quad p.f1 = this.f \quad backtrace(idx, p) = \langle \phi_1, \epsilon_1\rangle \quad \phi' = \{this.f, q\} \cup \phi_1 \quad \epsilon' = \epsilon_1 \cup \{RUO(P[idx])\}}{\langle idx, \phi, \epsilon\rangle \rightarrow\!\!\!\!\!\!\rightarrow [\![\langle P[idx]\rangle]\!] \Longrightarrow \langle idx + 1, \phi', \epsilon'\rangle}$$

$$assign_3: \quad \frac{P[idx] = p = q.f \qquad \phi' = \phi \setminus pre(\phi, p) \cup replace(pre(\phi, q.f), q.f, \{p\})}{\langle idx, \phi, \epsilon\rangle \rightarrow\!\!\!\!\!\!\rightarrow [\![\langle P[idx]\rangle]\!] \Longrightarrow \langle idx + 1, \phi', \epsilon\rangle}$$

$$call_{1.1}: \quad \frac{P[idx] = p = m(\bar{r}) \quad isRUO(m) \quad \epsilon' = \epsilon \cup \{RUO(P[idx])\}}{\langle idx, \phi, \epsilon\rangle \rightarrow\!\!\!\!\!\!\rightarrow [\![\langle P[idx]\rangle]\!] \Longrightarrow \langle idx + 1, \phi, \epsilon'\rangle}$$

$$call_{1.2}: \quad \frac{P[idx] = p = m(\bar{r}) \quad \neg isRUO(m)}{\langle idx, \phi, \epsilon\rangle \rightarrow\!\!\!\!\!\!\rightarrow [\![\langle P[idx]\rangle]\!] \Longrightarrow \langle idx + 1, \phi, \epsilon\rangle}$$

Figure 5: Operational semantics of the statements in the core language presented as a set of inference rules.

resource utilization operation $RUO(P'[3])$ and add it to the resultant operation sequence.

Detection of resource utilization bugs induced by variant lifecycles with VALA essentially involves concatenating the RUO sequences gathered from entrypoints and checking whether any of those resultant sequences may violate the requirements for correct resource utilization. If a violation is detected, VALA reports the error, together with the operation sequences, the Android activity or fragment, the entrypoints, and the variant lifecycle, to the user.

Particularly, to detect resource utilization bugs induced by Skip1 variant lifecycles, VALA looks for a pair $\langle s_1, s_2\rangle$ of RUO sequences, where s_1 and s_2 are from entrypoints `onCreate` and `onStop` of an activity, they both concern the same member field, and they satisfy the following conditions: 1) s_1 contains a *NULL* operation that invokes method `finish`; 2) s_1 contains a *REQ* operation o_1; 3) o_1 is not followed by any *REL* operations in s_1; 4) s_2 contains a *REL* operation o_2; 5) o_1 and o_2 concern the same resource. Here, condition 1 ensures that a Skip1 variant lifecycle is triggered, conditions 2 and 3 ensure that a resource is requested, but not released, in `onCreate`, while conditions 4 and 5 ensure that the resource is released in `onStop`. Given that entrypoint `onStop` will be skipped in a Skip1 variant lifecycle, the resource will not be released in such cases, causing a resource leak error.

To detect resource utilization bugs induced by Swap variant lifecycles, VALA first checks the app's manifest file to determine whether the app's target Android OS version is set to 9.0 or higher. If yes, activities in the app may go through Swap variant lifecycles. VALA then looks for a pair $\langle s_1, s_2\rangle$ of RUO sequences, where s_1 and s_2 are from entrypoints `onStop` and `onSaveInstanceState` of an activity, they both concern the same member field, and they satisfy the following conditions: 1) s_1 contains an *EDT* operation o_1; 2) o_1 is not followed by any *SAV* operations in s_1; 3) s_2 contains a *SAV* operation o_2; 4) o_2 is not preceded by any *EDT* operations in s_2; 5) o_1 and o_2 concern the same resource. Here, conditions 1 and 2 ensure that the member field is edited but not saved in entrypoint `onStop`, while conditions 3, 4 and 5 ensure that the member field is not edited but saved in entrypoint `onSaveInstanceState`. Since entrypoint `onStop` will be executed before `onSaveInstanceState` in a Swap variant lifecycle, a wrong value may be stored for the member field, causing a data loss error. If the app has been configured to run only on Android versions before 9.0, the app is safe from resource utilization bugs caused by Swap variant lifecycles. But if we can still find RUO pairs satisfying these conditions, VALA will issue a warning against the case to draw developers' attention to the risk involved in changing the app's configuration.

To detect resource utilization bugs induced by Skip2 variant lifecycles, VALA first checks if method `setRetainInstance` is invoked with argument `true` on a fragment of the app. If yes, the fragment may go through a Skip2 variant lifecycle. For such a fragment, VALA then looks for a pair $\langle s_1, s_2\rangle$ of RUO sequences, where s_1 and s_2 are from entrypoints `onCreateView` and `onDestroy` of a fragment, they both concern the same member field, and they satisfy the following conditions: 1) s_1 contains a *REQ* operation o_1; 2) o_1 is not followed by any *REL* operations in s_1; 3) s_2 contains a *REL* operation o_2; 4) o_1 and o_2 utilize the same type of resource. Here, conditions 1 and 2 ensure that a resource is requested, but not released, in `onCreateView`, while conditions 3 and 4 ensure that the resource is released in `onStop`. Since entrypoint `onStop` will be skipped in a Skip2 variant lifecycle, the resource will not be released in such a cases, causing a resource leak error.

Two things are worth noting about the design of VALA here. First, VALA only analyzes the utilization of resources referenced directly by member fields of activities and fragments, but in some rare cases, resources may also be referenced by variables contained in those member fields or shared in other ways, and VALA may miss resource utilization bugs in those cases. While those ways of resource sharing may also be modeled and analyzed in VALA, they are far less common in Android apps according to our experience, and the analysis of extra resource sharing forms will significantly increase the application costs of VALA, not in proportion to the enhancement of tool's effectiveness. In the end, our design of VALA trades a small portion of effectiveness for a large improvement to efficiency. Second, while VALA can only detect resource utilization bugs that are induced by the three types of variant lifecycles, the bugs cover quite some different resource types and reflect common problems in the interplay between RUOs and Android component lifecycles. Besides, support for new types of resources, utilization

operations, and variant lifecycles can be easily added into VALA to expand its applicability.

Since for each type of variant lifecycle only a few entrypoints are involved in detecting these the bugs, VALA analyzes only the relevant entrypoints and avoids doing the same on irrelevant entrypoints. Besides, VALA also implements another optimization in bug detection. More concretely, if a component member field is not utilized in any variant lifecycle-relevant entrypoints or it is not of any resource type that VALA supports, the tool will not gather RUO sequences concerning the member field when detecting bugs induced by Skip1 and Skip2 variant lifecycles.

3.5 Implementation

The VALA technique has been developed into a prototype tool with the same name. The VALA tool was built on the *FlowDroid* [2] static analysis framework for Android apps. We reused the class hierarchy relation and control-flow graph generated by *FlowDroid*, but replaced its implementation of the IFDS algorithm with our own propagation functions. Since *FlowDroid* does not fully support the analysis of reflective or native method calls, VALA may miss some method invocations when applied on apps utilizing such advanced language features. However, we do not expect this limitation to seriously impair the effectiveness of VALA since, according to our experience, activity/fragment entrypoints seldom contain reflective or native method invocations. Besides, as observed in our experimental evaluation of VALA (see Section 4), *FlowDroid* may also crash when analyzing some apps, which negatively impacts VALA's applicability.

4 EXPERIMENTAL EVALUATION

In this section, we empirically evaluate the effectiveness and efficiency of VALA. The evaluation aims to address the following research questions:

- **RQ1:** How *effective* is VALA in detecting resource utilization bugs induced by variant lifecycles?
- **RQ2:** How *efficient* is VALA in detecting resource utilization bugs induced by variant lifecycles?
- **RQ3:** How does VALA compare with existing tools in detecting resource utilization bugs induced by variant lifecycles?

4.1 Experimental Setup

Subjects. We selected popular apps from *GitHub* [11] as the subjects of our experiments based on the following criteria:

(1) the app should have at least 50 stars. This criterion helps us exclude projects that are less representative of real-world Android apps, e.g., training projects from novice developers.
(2) the app should have at least one executable artifact and the corresponding source code should be available on *Github*. This criterion is necessary because VALA takes executable APK files as its input and we manually check the source code to confirm whether bugs reported by VALA indicate real problems;
(3) the app should contain at least one activity or fragment that *may* go through one type of the variant lifecycles. That is, at least one activity or fragment in the app should override all the entrypoints relevant to one variant lifecycle type (see Table 2). Naturally, apps without such activities or fragments will never exhibit variant lifecycles and therefore will never contain bugs induced by variant lifecycles. We apply the *FlowDroid* tool on each app to check whether this criterion is satisfied.

Table 4: Basic information about the 35 subject apps.

ID	App	Cate*	Version	#C	#F	$\#C_V$				$\#F_V$
						Total	Skip1	Skip2	Swap	
S01	Aptoide	Prod.	9.13.3.1	156	1502	28	17	11	2	566
S02	Beedio	Music	1.3.0	17	97	2	0	2	0	36
S03	BiglyBT	Tool	1.2.6	81	392	17	17	0	0	112
S04	BuildmLearn	Tool	2.5.0	46	201	1	1	0	1	15
S05	Campus-Android	Edu	3.4	79	343	8	8	0	0	42
S06	cgeo	Ent.	2020.04.25	65	553	8	8	0	5	176
S07	EasyBudget	Fin.	1.6.4	17	58	2	1	0	2	12
S08	GmsCore	Tool	0.2.10	27	53	1	1	1	0	5
S09	Hentoid	Comic	1.11.5	80	352	17	16	1	0	77
S10	iNaturalistAndroid	Edu	1.18.14	101	1655	16	7	0	15	655
S11	jchat	Com.	2.3.0	116	1061	48	48	0	0	469
S12	kefu	Prod.	1.5	39	280	1	0	0	1	8
S13	Koler	Com.	0.3.1	29	245	3	1	2	0	87
S14	nextcloud	Prod.	3.12.0 RC1	122	1234	2	0	0	2	36
S15	nextcloud_news	News	0.9.9.36	33	325	2	1	1	0	17
S16	nRF Toolbox	Tool	2.8.4	52	332	9	9	0	0	79
S17	ObscuraCam	Video	4.0.1	14	240	1	1	0	1	75
S18	ODK Collect	Prod.	1.26.3	96	473	5	4	1	2	73
S19	OneBusAway	Map	2.7.2	187	829	1	0	0	1	43
S20	open-gpstracker-ng	Travel	2.0.0	4	5	1	0	0	1	2
S21	Resplash	Pers.	v1.3.3	44	411	13	0	13	0	187
S22	S1-Next	Social	2.1.2-56	156	897	91	91	0	0	696
S23	Sensor-Data-Logger	Tool	1.5	28	146	1	0	0	1	15
S24	SkyTube	Social	2.972	47	251	11	2	9	0	65
S25	stock-hawk	Fin.	1.0	11	72	1	0	0	1	14
S26	StoryMaker	Phot.	2.1.6.10	81	634	1	1	0	1	9
S27	talon-twitter-holo	Social	4.17.1	125	971	2	2	0	0	77
S28	To-Do List	Prod.	2.3	18	121	1	1	0	1	39
S29	Trivia hack	Tool	2.5.1	25	119	2	2	0	0	6
S30	Ushahidi	Com.	v3.9	57	290	1	1	0	0	3
S31	wallabag	Prod	2.4.0	35	293	2	1	0	1	72
S32	wikipedia	Book	2.7.50320	94	823	4	0	1	3	90
S33	WordPress	Prod.	14.8-rc-2	241	2078	21	21	0	13	373
S34	Zapp	Video	3.4.0	27	147	1	1	0	0	13
S35	zulip	Com.	1.3.2	27	217	1	0	0	1	14
Overall				2377	17700	326	263	42	55	4258

* Prod.: Productivity; Edu.: Education; Ent.: Entertainment; Fin.: Finance; Com.: Communication; Pers.: Personalization; Phot.: Photography;

We initially gathered from *GitHub* in total 287 apps satisfying the criteria 1 and 2. Among them, 252 apps were excluded because they do not contain activities or fragments that may exhibit variant lifecycles or *FlowDroid* crashed when analyzing them, which violates criterion 3. This leaves us with 35 apps as our subjects. Table 4 lists basic information about the 35 apps. For each app (App), the table gives an ID, its category (Categ), the version used (Version), the total number of activities and fragments defined in it (#C), the total number of fields defined in those activities and fragments (#F), the total number of activities and fragments that may go through variant lifecycles ($\#C_V$|Total), the breakdown of that total number to the three types of variant lifecycles, and the total number of fields defined in those activities and fragments ($\#F_V$). Our subject apps are diversified in terms of their application categories and size (measured in #C and #F). Note, however, that the #C and #F values reported in Table 4 do not directly indicate analysis complexity for VALA since most activities and fragments in those apps will never exhibit variant lifecycles, while VALA's analysis complexity is

Table 5: Experimental results of VALA and Relda2 on the 12 subjects. All times are in seconds.

App	VALA								Relda2			
	#Bug	#FP	$\#C_A$	$\#F_A$	T_{total}	T_{PST}	T_{det}	Mem	#Bug	#FP	T_{total}	Mem
S01	0	0	3	3	40	37	3	873	0(0)	0	30	790
S05	1	0	2	2	11	9	2	493	2(0)	82	123	735
S06	1	0	4	7	15	3	18	791	0(0)	9	101	661
S08	1	0	1	1	3	2	1	514	0(0)	0	17	436
S18	1	0	3	35	15	12	3	697	0(0)	64	618	1280
S23	0	0	1	2	2	1	1	468	0(0)	0	35	692
S24	1	0	1	1	13	12	1	251	0(0)	3	39	666
S25	0	0	1	1	4	3	1	105	0(0)	0	1	133
S27	1	0	1	1	10	8	2	542	7(0)	0	51	658
S28	1	0	1	1	2	1	1	434	0(0)	0	4	251
S32	1	0	1	1	10	7	3	387	3(0)	7	98	669
S33	0	0	1	1	24	22	2	572	0(0)	0	55	866
Overview	8	0	20	56	151	122	29	6127	12(0)	165	1142	7047

dominated by (1) the number of activities and fragments that may exhibit variant lifecycles (measured in $\#C_V$ and $\#F_V$) and (2) how the relevant resources are utilized in those variant lifecycles.

Baseline bug detection technique. To the best of our knowledge, no technique that targets specifically resource utilization bugs induced by variant lifecycles has been developed before. Therefore, we compare VALA with *Relda2* [34, 35], which is an off-the-shelf resource leak detection tool and is closely related to VALA. We answer RQ3 based on results of the comparison.

Environment. All experiments were run on a desktop computer with one 2.6GHz Quad-core processor and 64GB RAM running CentOs 6.10. The OS is required for running *Relda2*.

4.2 Experimental Results

In this section, we report the experimental results and answer the research questions.

4.2.1 RQ1: Effectiveness. In total, VALA found 8 bugs, including 6 resource leak errors and 2 data loss errors, in the 35 apps. In addition to the 8 bugs, VALA also issued 2 warnings against apps *Sensor-Data-Logger* and *Stock-Hawk*. Recall that, VALA issues a warning against an app if, although the app targets Android OS versions before 9.0, it would trigger a data loss error when executed on Android OS version 9.0 or later (Section 3.4). VALA did not find any resource utilization bugs in the remaining 25 apps. Such a high rate of variant lifecycle relevant apps with detected bugs indicates that even experienced developers of popular apps may not realize the complications associated with the variant lifecycles. VALA terminated almost instantly on 23 apps since, although they contain activities and/or fragments satisfying the conditions listed in Table 2, no member fields of those activities and fragments are of the resource or data types that VALA supports. Table 5 lists the remaining 12 apps and, for each app, the number of bugs detected by VALA (#Bug), the number of detected bugs that turned out to be false positives (#FP).

We manually checked the 8 bugs reported by VALA and decided they are all true positives. Furthermore, we submitted all the detected bugs, except the one in app *ODK Collect* since it has been fixed by the programmer, as issues to their corresponding development teams. All the 7 issues were reported for the first time: 4 of those issues were quickly fixed or confirmed as bugs by the developers, while we received no response regarding the other 3 issues, probably because the related projects are no longer maintained. Developers, such as those of app *SkyTube*, admitted in their response that the issue we reported is indeed an interesting bug, and that it is hard to correctly implement resource utilization into the Android lifecycles. Table 6 lists the 8 bugs reported by VALA and, for each bug, the containing app (App), the type of variant lifecycle that induced the bug (VL Type), the error type (Error), its state (State), and the URL of its issue report (Issue).

Table 6: Eight bugs revealed by VALA in the experiments.

App	Bug			
	VL Type	Error	State	Issue
Campus-Android	Skip1	Leak	Reported	bit.ly/3s146J9
cgeo	Swap	Loss	Fixed	bit.ly/2R7Q4IX
Gmscore	Skip1	Leak	Fixed	bit.ly/3sWVm8e
ODK Collect	Skip2	Leak	Fixed	bit.ly/3t0v4Cd
SkyTube	Skip2	Leak	Confirmed	bit.ly/3mwFerS
talon-twitter-holo	Skip1	Leak	Reported	bit.ly/3mvv9v3
To-Do List	Swap	Loss	Reported	bit.ly/3d1E6ZY
Wikipedia	Skip2	Leak	Fixed	bit.ly/3rV0g4k

Particularly, half of the 6 resource leak errors, i.e., those in apps *Campus-Android*, *Wikipedia*, and *ODK Collect*, are related to class `CompositeDisposable`—a disposable container from a third-party library that obtains and releases multiple other disposables. Such results suggest that, compared with writing all the resource utilization code in activities and fragments from scratch, utilizing resources through third-party libraries poses extra challenges on developers, probably because third-party libraries often document the common behaviors of their APIs but seldom specify how the APIs function under special circumstances. In view of such additional challenges, programmers may opt against relying on third-party libraries in similar situations. In fact, the developers of app *ODK Collect* did replace `CompositeDisposable` with their homemade `Schedulers` for asynchronous work in a later revision.

Another thing worth mentioning is that, while developers of app *Gmscore* confirmed the reported issue is indeed a bug, their fix to the bug was still defective: They just moved the resource releasing operation from `onStop` to `onDestory`, which, however, still cannot guarantee the release operation is always executed since, according to the Android lifecycle model guidelines, entrypoint `onDestroy` can be skipped even in common activity lifecycles. A safer way to correct the error is to override method `finish` and perform the resource releasing operations inside the method. This improper fix to the reported bug highlights the challenge in correctly incorporating RUOs into Android lifecycles and the value of this work.

VALA detected 8 bugs in 35 Android apps. VALA was effective in revealing resource utilization bugs induced by variant lifecycles.

4.2.2 RQ2: Efficiency. Table 5 also lists, for each app on which VALA did not terminate instantly, the numbers of activities/fragments ($\#C_A$) and member fields ($\#F_A$) that VALA actually analyzed, the total detection time with VALA in seconds (T_{total}) and the breakdown of that to time spent in PST construction (T_{PST}) and bug detection

(T_{det}), and the amount of memory used in megabytes (Mem). Recall that VALA implements several optimizations to reduce the number of activities, fragments, entrypoints, and member fields that it has to analyze in bug detection (see Section 3.4). It took VALA 151 seconds in total to finish running on the 12 apps, with an average memory consumption of 510MB. Most of the time was spent on constructing the PST of the apps based on the *Soot* framework, while the analysis and detection time was only 29 seconds, i.e., 19.2% of the total running time. The main reason for VALA's high efficiency and low memory consumption lies in that the analysis in VALA only takes into account variant lifecycle-relevant entrypoints and member fields used in variant lifecycles, which accounts for a small percentage of entrypoints and member fields defined in apps. More concretely, VALA only analyzed 2 activities/fragments and 5 member fields for each app on average, which accounts for 3.2% of all the activities/fragments and 1.4% of all the member fields.

It took VALA less than one minute to complete its analysis on each app. VALA was efficient in revealing resource utilization bugs induced by variant lifecycles.

4.2.3 RQ3: Comparison with Relda2. The detection results produced by *Relda2* on the 12 apps are also included in Table 5.

Relda2 reported 177 resource leaks. However, manual inspection of the detected leaks reveals that 165, or 93%, of them are false positives. The high false positive rate greatly diminishes the usefulness of the tool. Compared with *Relda2*, VALA gathers RUO sequences for only variant lifecycle-relevant entrypoints through a precise context-sensitive interprocedural data-flow analysis, which greatly reduces the false positive rate of its results.

The 12 real resource leaks found by *Relda2* did not overlap with the ones that VALA detected. In fact, none of the resource leaks reported by *Relda2* was related to variant lifecycles, probably because the variant lifecycles were not included in *Relda2*'s lifecycle models for Android activities and fragments. It took *Relda2* 7X longer time to finish running on the 12 apps, but its memory consumption was comparable with that of VALA.

None of the bugs detected by VALA was also revealed by Relda2. VALA complements Relda2 in detecting resource utilization bugs in Android apps.

4.2.4 Threats to Validity. In this section, we discuss possible threats to the validity of our findings and show how we mitigate them.

Construct validity. Threats to construct validity are mainly concerned with whether the measurements used in the experiment reflect real-world situations.

In this work, we manually examined all the bugs reported by VALA and Relda2. Programmers, however, may have different opinions regarding whether a reported bug reflect real defects in apps. To mitigate this risk, we submit 7 bugs reported by VALA to the development teams of the corresponding apps. 4 of those bugs were confirmed by the developers, while no response was received regarding the other 3 bugs, probably because the related projects are no longer maintained.

Internal Threats. Threats to internal validity are mainly concerned with the uncontrolled factors that may have also contributed to the experimental results.

In our experiments, a major threat to internal validity is the possible faults in the implementation of our approach and the integration of external libraries. To address the threat, we review our code and experimental scripts to ensure their correctness before conducting the experiments.

External Threats. Threats to external validity are mainly concerned with whether the findings in our experiment are generalizable for other situations.

A major threat to external validity is that, the apps used in our experiments may not be good representatives of all the Android apps in the market. To mitigate this threat, we selected from a wide range of candidate apps hosted on *GitHub*. The subject apps used in our experiments were popular and include widely used apps like *Wikipedia*, *ODK Collect*, *cgeo* and *WordPress*, which are published on the Google Play Store with millions of installations. Nevertheless, we plan to conduct larger scale experiments to evaluate VALA more thoroughly in the future.

5 RELATED WORK

In this section, we review research studies from two areas that are closely related to VALA.

5.1 Android Lifecycle Modeling

To produce high-quality lifecycle models for Android components has always been an important task in static analyses for Android apps, and a variety of approaches have been proposed so far to tackle that task. Mainstream static analysis techniques like *FlowDroid* [2], *IccTA* [22], *DroidSafe* [13], *AmanDroid* [32], and *Gator* [37, 38], tend to use manually constructed lifecycle models to support their specific analyses of Android components. These models, however, usually do not cover the variant lifecycles studied in this paper due to the modelers' unawareness. Junaid et al. [19] automatically built their lifecycle model via reverse engineering and dynamic tracing, but the resultant model they constructed does not include the variant lifecycles either. Cao et al. [3] and Chen et al. [4] employed automated analyzer or machine learning techniques to automatically identify implicit state transitions in the Android framework. On the one hand, including all these implicit state transitions will produce lifecycle models that are orders of magnitude larger than the manually constructed ones. On the other hand, although Cao et al. [3] showed that, to utilize these implicit transitions in static analyzers such as *FlowDroid* is not extremely expensive, since the implicit state transitions are not accompanied by information regarding the conditions under which they will be triggered (e.g., variant lifecycles of type Skip1 and the corresponding implicit state transitions are only triggered when method `finish()` is invoked in entrypont `onCreate`), analyses solely based on those implicit state transitions most likely will produce results of low accuracy.

5.2 Lifecycle Conformance Guarantee

Another realm of research related to VALA is lifecycle conformance insurance, i.e., ensuring an app's own correctness through reacting appropriately to state changes of Android lifecycle. Resource leak errors and data loss errors are two of main issues in these researches.

To detect resource leaks injected in Android lifecycle, several researches have been conducted with dynamic testing [10, 27, 33], while most choose to use static analysis. Zein et al. [40] present a UML-based model to present lifecycle rules and system resources and a novel algorithm to check whether system resources are requested and released at correct entrypoints based on the model. Guo et al. [14] proposed *Relda*, a lightweight though lifecycle-aware static analysis tool to detect resource leaks based on call graphs with resource requesting/releasing operations added as call nodes. Based on *Relda*, Wu et al. [34, 35] present *Relda2*, which enhances *Relda* in terms of using call graphs of higher precision, employing flow-sensitive detection technique to eliminate false negatives, and speeding up the analysis with multi-threading techniques. There are also many other novel ideals in detecting or fixing such resource utilization bugs [18, 23, 25, 36], however, none of them have ever targeted these bugs induced by variant lifecycles, and certainly, as our evaluation results show.

Particularly, a number of researches focus on data loss errors due to state transitions of app restarting on rotation or app killing on low memory [1, 8, 16, 21, 28, 39]. One of the state-of-the-arts relevant to VALA is *LiveDroid* [9], which identifies only necessary part of the app state that needs to be preserved across app lifecycles, and automatically saves and restores it. *LiveDroid* also performs a precise data-flow analysis on app state preserving entrypoints, such as `onSaveInstanceState()` involved in variant lifecycle Swap. However, *LiveDroid* is not aware of this variant lifecycle while only focus on the whether app state are correctly preserved and restored in app restarting and killing transitions. These transitions are a part of the common lifecycles depicted in Figure 1, which have already drawn great attentions from developers, researches and the Google Android team. Regrettably, compared with the common lifecycles, the resource utilization bugs induced in the variant lifecycles illustrated in this paper, their importance and harmfulness are underestimated or even ignored.

6 CONCLUSION

In this paper, we study variant lifecycles of Android activities and fragments, i.e., lifecycles where execution orders of entrypoints deviate from that in common lifecycles as highlighted in the official Android lifecycle guidelines. We explain why variant lifecycles may cause resource utilization bugs and identify three types of variant lifecycles that Android activities and fragments may go through. We also present the VALA technique that automatically detect resource utilization bugs in Android apps due to these variant lifecycles.

VALA was able to detect 8 bugs in experiments conducted on 35 popular Android apps. All the 8 bugs were confirmed to be real defects, 7 of those bugs were reported for the first time, and none of these bugs were revealed by the state-of-the-art resource leak detection tool *Relda2*. Such results strongly suggest that VALA is highly effective in detection resource utilization bugs induced by variant lifecycles.

In this work, we focused on detecting resource utilization bugs because it is challenging to correctly utilize different kinds of resources in conformity with the Android component lifecycles, and we carefully devised VALA for high accuracy of the detection results to reduce the costs of applying the technique. We plan to research on the detection of other types of bugs due to variant lifecycles via various techniques in the future.

ACKNOWLEDGMENTS

This research is supported by the National Natural Science Foundation of China under Grant No.61972193 and the Hong Kong RGC General Research Fund (GRF) PolyU 152002/18E.

REFERENCES

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of Android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 83–93.
[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM SIGPLAN Notices* 49, 6 (2014), 259–269.
[3] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework. In *NDSS*.
[4] Xiupeng Chen, Rongzeng Mu, and Yuepeng Yan. 2018. Automated identification of callbacks in Android framework using machine learning techniques. *International Journal of Embedded Systems* 10, 4 (2018), 301–312.
[5] Alain Deutsch. 1994. Interprocedural may-alias analysis for pointers: Beyond k-limiting. *ACM SIGPLAN Notices* 29, 6 (1994), 230–241.
[6] Google Developers. 2022. Documentation of Android API references. https://developer.android.com/reference. [online, accessed 27-May-2022].
[7] Google Developers. 2022. Understand the Activity Lifecycle | Android Developers. https://developer.android.com/guide/components/activities/activity-lifecycle. [online, accessed 27-May-2022].
[8] Umar Farooq and Zhijia Zhao. 2018. Runtimedroid: Restarting-free runtime change handling for Android apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 110–122.
[9] Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. 2020. Livedroid: Identifying and preserving mobile app state in volatile runtime environments. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
[10] Dominik Franke, Stefan Kowalewski, Carsten Weise, and Nath Prakobkosol. 2012. Testing conformance of life cycle dependent properties of mobile applications. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 241–250.
[11] Github. 2021. The world's leading software development platform · GitHub. https://github.com. [online, accessed 27-May-2022].
[12] Google. 2022. SimpleExoPlayer (ExoPlayer library). https://exoplayer.dev/doc/reference/com/google/android/exoplayer2/SimpleExoPlayer.html. [online, accessed 27-May-2022].
[13] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of Android applications in droidsafe. In *NDSS*, Vol. 15. 110.
[14] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. 2013. Characterizing and detecting resource leaks in Android applications. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. 389–398.
[15] Noura Hoshieah, Samer Zein, Norsaremah Salleh, and John Grundy. 2019. A static analysis of Android source code for lifecycle development usage patterns. *Journal of Computer Science* 15, 1 (2019), 92–107.
[16] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–15.
[17] Richard Johnson, David Pearson, and Keshav Pingali. 1994. The Program Structure Tree: Computing Control Regions in Linear Time. *SIGPLAN Not.* 29, 6 (1994), 171–185.
[18] Ma Jun, Liu Sheng, Yue Shengtao, Tao Xianping, and Lu Jian. 2017. LeakDAF: An automated tool for detecting leaked activities and fragments of Android applications. In *Proceedings of the 41st Annual Computer Software and Applications Conference*, Vol. 1. IEEE, 23–32.
[19] Mohsin Junaid, Donggang Liu, and David Kung. 2016. Dexteroid: Detecting malicious behaviors in Android apps using reverse-engineered life cycle models. *Computers and Security* 59 (2016), 92–117.
[20] William Landi and Barbara G Ryder. 1992. A safe approximate algorithm for interprocedural aliasing. *ACM SIGPLAN Notices* 27, 7 (1992), 235–248.
[21] Niel Lebeck, Arvind Krishnamurthy, Henry M Levy, and Irene Zhang. 2020. End the senseless killing: improving memory management for mobile operating systems. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 873–887.

[22] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in Android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.

[23] Jierui Liu, Tianyong Wu, Jun Yan, and Jian Zhang. 2016. Fixing resource leaks in Android apps with light-weight static analysis and low-overhead instrumentation. In *Proceedings of the 27th International Symposium on Software Reliability Engineering*. IEEE, 342–352.

[24] Yepang Liu, Lili Wei, Chang Xu, and Shing-Chi Cheung. 2016. DroidLeaks: benchmarking resource leak bugs for Android applications. *arXiv preprint arXiv:1611.08079* (2016).

[25] Jun Ma, Shaocong Liu, Yanyan Jiang, Xianping Tao, Chang Xu, and Jian Lu. 2018. LESdroid: a tool for detecting exported service leaks of Android applications. In *Proceedings of the 26th Conference on Program Comprehension*. 244–254.

[26] ReactiveX. 2022. CompositeDisposable (RxJava Javadoc 3.0.11) - ReactiveX. http://reactivex.io/RxJava/javadoc/io/reactivex/disposables/CompositeDisposable.html. [online, accessed 27-May-2022].

[27] Vincenzo Riccio, Domenico Amalfitano, and Anna Rita Fasolino. 2018. Is this the lifecycle we really want? an automated black-box testing approach for Android activities. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. 68–77.

[28] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 2016. Finding resume and restart errors in Android applications. *ACM SIGPLAN Notices* 51, 10 (2016), 864–880.

[29] statista. 2020. Global smartphone sales to end users from 1st quarter 2009 to 2nd quarter 2018, by operating system. https://www.statista.com/statistics/266219/global-smartphone-sales-since-1st-quarter-2009-by-operating-system/. [online, accessed 27-May-2022].

[30] statista. 2020. Tablet operating systems market share worldwide from 1Q'16 to 2Q'20. https://www.statista.com/statistics/273840/global-market-share-of-tablet-operating-systems-since-2010/. [online, accessed 27-May-2022].

[31] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. ANDROMEDA: accurate and scalable security analysis of web applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*. 210–225.

[32] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. *ACM Transactions on Privacy and Security (TOPS)* 21, 3 (2018), 1–32.

[33] Haowei Wu, Hailong Zhang, Yan Wang, and Atanas Rountev. 2020. Sentinel: generating GUI tests for sensor leaks in Android and Android wear apps. *Software Quality Journal* 28, 1 (2020), 335–367.

[34] Tianyong Wu, Jierui Liu, Xi Deng, Jun Yan, and Jian Zhang. 2016. Relda2: an effective static analysis tool for resource leak detection in Android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 762–767.

[35] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. 2016. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. *IEEE Transactions on Software Engineering* 42, 11 (2016), 1054–1076.

[36] Zhiwu Xu, Cheng Wen, and Shengchao Qin. 2018. State-taint analysis for detecting resource bugs. *Science of Computer Programming* 162 (2018), 93–109.

[37] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25, 4 (2018), 833–873.

[38] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android applications. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. 89–99.

[39] Razieh Nokhbeh Zaeem, Mukul R Prasad, and Sarfraz Khurshid. 2014. Automated generation of oracles for testing User-Interaction features of mobile apps. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*. 183–192.

[40] Samer Zein, Norsaremah Salleh, and John Grundy. 2017. Static analysis of Android apps for lifecycle conformance. In *2017 8th International Conference on Information Technology (ICIT)*. IEEE, 102–109.