Leak canary android studio

I recently came across this comment in a post: Leaks would be really annoying for Canaries or Android Studios, most of the time identified by LeakCanary doesn't show up in profilers/memory/memory leaks, I wonder if LeakCanary is showing false positives or Android Studios is missing positive. That's a good question, let's dig into the code and find this out! Before answering the question false positive leak from Android Studio, we need to talk about where the idea of false positive leaks comes from: Android Studios. This warning was originally a longer description of activities and fragment instances that could be caused by memory leaks. For activities, these instances have been deleted, but they are still referenced. For fragments, there is no valid FragmentManager, but it is still a referenced instance. These instances can contain fragments that have been created but are not yet in use. This article provides more insight into false positive leaks: In certain situations, filter can generate false positives: fragments are created, but they have not yet been used. The fragment is being cached, but not as part of the fragment transaction. The idiom is ambiguous, but false positive leaks seem to apply only to fragments. Android Studio Leak Filtering analyzes heaps when you click the Android Studio Dump and Dump Heap icons. Instance leaks are displayed by enabling the Activity/Fragment Leak filter, which updates the bottom panel to show only leaky instances. Filtering is performed by the ActivityFragmentLeak instance filter: const val FRAGFMENT_MANAGER_FIELD_NAME = mFragmentManager /** * If the mFragmentManager field is null, the fragment instance is determined to potentially leak. This indicates that the * instance is in its initial state. This can mean that the instance has been deleted or has just started to be *initialized, but not before it is linked to an activity. * The latter gives us false positives, but it's not uncommon to add them to *FragmentManager unless the user makes a piece in advance. */ Personal Fun Leaks Potential Fragments (Instance Objects): Boolean {Returns Are ValidDepthWithAnyField (Instance, {FRAGFMENT_MANAGER_FIELD_NAME == It}, {It =null}} } } /*** Make sure the instance has a valid depth * Personal fun is validDepthWithAnyField (instance object, Names at: > onVal: (Any?) -> Boolean: Boolean { Foot Depth = inst.depth Return Depth != 0 & Depth != Int.MAX_VALUE & inst.fields.any { onName (it.fieldName) & Val (it.value) } } } (e.g., Fragment Manager) So, if that mFragmentManager field is null and the fragment can be reached through a strong reference, the fragment is considered leaky (that is, what valid depth means in the code above). If you create a fragment instance but don't add it, you'll see a warning because it's reported as a leak. False positive leaks. Unlike Android Studios, LeakCanary doesn't see the mFragmentManager field for all fragment instances in the LeakCanary memory, how to spot fragment leaks. LeakCanary connects to the Android life cycle to automatically detect when fragments are destroyed and rubbish needs to be collected. These destroyed objects are passed to ObjectWatcher, which holds weak references to them. (Activities fragmented) { val support fragmentation manager = activity.supportFragmentManager supportddancyments.registerFragmentLifecycleCallbacks (object: FragmentManager.FragmentLifecycleCallbacks() { Redefine fun (fm: fragment manager), Fragment: { objectWatcher.watch (sculpture, Receive fragments#onDestroy() Callback } }, true) } Wait 5 seconds, and if the weak references held by ObjectWatcher are not cleared, the observed object remains and potentially leaks. LeapCanary dumps java heaps into .hprof files (heap dumps), and then BreakCanary uses sharks to parse .hprof files and find objects that have been retained in the heap dump. And this is where the leak is different from CanaryAndroid Studios: it looks for custom weak references made (see Keyed Weak) and Kid confirms his reference field in the Weak Reference Finder: finds all objects tracked by /** KeyedWeakReference, i.e. objects passed to ObjectWatcher.watcher.watcher.watcher.watcher.watcher.watcher.watcher.watcher.watcher.watcher.watcher.watcher.watcher.watcher. */ Object Kid WeakLepper Finder: Find LeakEd Object {Find OverrideForking ofJectdis (Graph: Heapgraph):<Long> {Return Graph.findClassByName (Leak Canary). Keyed WeakReference) .map {weakRef-> Weak Ref[java.lang.ref.Reference, see]!! . Value .asObjectId} .filter {objectId -> objectId != ValueHolder.NULL_REFERENCE } }} } that means only leak canary actually leaking surface fragments. Why leaks report more leaks Now we've established that leaks don't have false positive leaks like Android Studios, Let's go back to the main observation: Most of the time leaks identified in the leak canary do not appear in profilers/memory/memory leaksIn the report is a leak document: LeakCanaria automatically detects leaks to the following objects: ViewModel instances where destroyed activity instances destroy fragment instances deleted ViewModel instances do not detect leaks for fragmented view instances by Android Studio, or viewModel instances are deleted. The former is actually a common cause of leaks: add fragment instances to the backstack without clearing the view field of the fragment.onDestroyView() (more on in this StackOverflow answer). If the conclusion is that the leak cannari reports a leak trail, then there is definitely a leak. Leak canary is always right! Unfortunately, we can't blame false positive leaks on small yellow algae as an excuse not to fix leaks. If your app is slow on some devices or in Android Studio's Instant Run feature, click </Long>Random operation can leak memory. When does a memory leak occur? One of the biggest points of Java is the Garvage Collector (GC). The Savage Collector runs periodically to identify and remove unused objects. A memory leak occurs when an object is not used by the app, but the invisible collector cannot recognize it. This results in unexpected results that remain in unused memory, reducing the amount of memory available to your app. How to detect memory leaks? There are several ways to do this. We will cover some of them in this article. But first, let's write code that can generate memory leaks. We will create an application with two activities. Activity A has a button to open Activity B. In addition, you create a single class that contains static references to the context. Class Manager { Private Static Manager Us Instance; Private Context Context; Static Manager getInstance (Context Context) { If (Us Instance = null) { Our Instance = New Administrator (Context); } Returns Our Instance; } Personal Manager (Context) { This.context = Context; } } Activity A Manager.getInstance will call this singleton; Monitor in Android Studio Provides a handy tool for profiling the performance of your app when it detects memory leaks using Android Studio, one of which is a memory monitor. You can use a memory monitor to detect memory leaks (:1) by using the following steps: Run the app on a mobile device or emulator.2. Open Android monitor (press Cmd + 6+ 6 on Windows on

Mac or Alt). At the top of the memory monitor, click Start Allocation Tracking.3 Use the app around the area where you suspect a memory leak may occur. In our example, we move on to other activities.4. Click Start GC to force the collection of the garvage collector to collect and remove objects that are not currently in use. You can see a slight decrease in the memory graph.5. Click on the dump Java heap so that Android Studio can generate a .hprof file containing heap snapshots. The easiest way to check for memory leaks is to use Analyzer Task 6. Open the Analyzer Actions tab, and at the top of this tab, click Run. Under the Analyzer Actions tab, the Analysis Results tab displays the leaked activity. Leak Using canary leakage canary to detect memory leaks is a library created in a square and is a very quick way to detect memory leaks. Leak Canary can detect memory leaks on longer runs because you don't have to connect your device to Android Studio and monitor your app for long periods of time. Leak Canary will send you a notification whenever there is a memory leak. The integration of leak canaries is really easy. All you have to do is to add the following to the application level build.gradle file: dependency { debugcompil 'com.squareup.breakcanary:breakcanary-android:1.5.1' releasecompil 'com.squareup.likeanary:breakcanary-android-no-op:1.5.1' t-compile } Then add the following code to the application class: @Override public void () { super.onCreate (); (LeakCanary.isInAnalyzerProcess (this)) { // If this process is dedicated to leaking cannies for heap analysis. // You should not start the application in this process. LeakCanary.install (this); General app isying code... } And that's it, you can go. Detecting the possibility of leaks with an inferno is a static analyzer tool created by Facebook that helps you find possible null pointer exceptions and resource leaks, as well as note reach, missing lock guards, and concurrency race conditions. It is a good tool to add to the CI to overcome all possible bugs. It's also open source, so tool development is still evolving. You can find more information about inference in the QuickStart article. See: Document. See: