InfeRS: Static Memory Leak Detection in Android Apps with Facebook Infer

Alon Grinshpoon
ag3848@columbia.edu

Jacob Sachs
jss2273@columbia.edu

Department of Computer Science, Columbia University

Abstract— **Android apps are written in Java and garbage-collected, so they have no traditional memory leaks. However, in certain cases, applications can drop object references. This memory cannot be reclaimed, resulting in a leak. These leaks occur as a result of Android's memory saving architecture. Over time, these leaks may result in slowdowns or crashes in an application.**

In this paper, we explore how these leaks occur in Android and what can be done to preemptively prevent them. Specifically, we test Facebook's Infer application for performing static analysis to detect memory leaks. In the cases where certain leaks are not detected, we extend Infer, and examine the results.

For this paper, we have developed an Android app that implements memory leak scenarios and created special Infer-based checkers to help developers detect these scenarios.

I. Introduction

Android applications do not encounter traditional memory leak issues, due to the garbage collection capabilities of the Android Runtime (ART). However, Android apps can still leak memory due to dropped references within Java classes. When an app switches between Activities, usually by switching screens, inactive Activities classes will usually be garbage collected, freeing their memory for further use. Yet if the app maintains references to inactive Activities, they cannot be garbage collected. These ”lost objects” continue to consume memory, leading to slowdowns and crashes over time.

There are several programming patterns indicative of this lost object behavior. Additionally, there exist several robust static analysis tools for Android applications. We will explore the ability of Facebook's Infer analyzer to detect these memory leak patterns, and will extend the analyzer where possible. We will also compare the performance of static analysis to LeakCanary, a leading dynamic analysis tool.

II. Approach

There are several antipatterns within Android development that commonly cause memory leaks. These have to do with static activities, static views, inner classes, and anonymous classes. The first step of the project will be to create generalized abstractions for these antipatterns.

Once we have determined the detection criteria and identified the memory-leak-provoking patterns, we will develop a simple Android app that implements all patterns. This app will be used as a code example test case and reference point for Infers performance. Our reference app will help us evaluate the needed extensions we are required to add to Infer in order to capture the different types of leaks..

We will then test the extension in action. We will run the new checker on our leaky reference app and on an open source Android application, and see if we can encounter any dropped reference memory leaks. We will compare this analysis to dynamic analysis run with LeakCanary.

III. Novelty Claim

Memory leaks of undetermined origin can be found effectively with dynamic analysis. However, static analysis is still extremely important within the development lifecycle. This specific type of memory leak is not adequately covered by existing static analysis tools. Currently, the leading way for tracking down these types of issues ahead of time is with a heap dump in Android studio, or another IDE. Static analysis is automatic and creates less additional work for the developer.

IV. Memory Leak Patterns

While there are several different ways that an Android app can leak memory, each one of these programming errors boils down to a single, fundamental problem: the incorrect preservation of a reference to an Activity once its lifecycle has completed. We will investigate four of the most likely manifestations of this problem [1].

A. Static Activities

In a static Activity leak, a static variable remains loaded in memory for the entire runtime of the app. If a static class object holds a reference to the Activity instance, and the object is not cleared before the Activity shuts down, this Activity will not be garbage collected. This results in a memory leak of the Activity.

The code pattern that creates this leak is shown in figure 1. Declaring a static variable inside the class definition of an Activity, followed by setting the static variable to reference the running instance of the Activity, means that the static variable will always hold a reference to the Activity. If such a reference is absolutely necessary, developers may use a WeakReference object to wrap the reference to the Activity.

```
public class StaticActivityLeak extends Activity
{
    static Activity activity;

    @Override
    public void onCreate() {
        leakMemory();
    }
    private void leakMemory() {
        activity = this;
    }
}
```

Fig. 1. Example of a static Activity memory leak

This will indicate to the Android Runtime that the Activity should still be garbage collected, despite the reference [2].

B. Static Views

A static View leak is very similar to a static Activity leak. In this type of leak, a View maintains a reference to its Context, which also happens to be an Activity. In the Android lifecycle, some Views remain unchanged and take a long time to instantiate. This means that recurring Activities may keep such Views loaded in memory for a quick restore.

```
public class StaticViewLeak extends Activity
{
    static View view;

    @Override
    public void onCreate() {
        leakMemory();
    }
    private void leakMemory() {
        view = this.findViewById(R.id.button);
    }
    @Override
    public void onDestroy() {
        // view = null;
    }
}
```

Fig. 2. Example of a static View memory leak

While this makes sense from a design perspective, static View references create a persistent reference chain to an Activity, resulting in the same type of memory leak as before. Thus, developers should be careful to explicitly clear the reference to the instantiated view during an onDestroy() call (e.g., uncomment the nullifying line in figure 2).

C. Inner Classes

In Android code, additional classes can be defined within an encompassing Activity class. This is often done to improve readability, or to create a logical abstraction of encapsulation. Inner classes have access to all the same variables as their outer classes, including a reference to the outer Activity. Thus, if we maintain a static reference to an inner class, we are going to get a memory leak.

Inner classes need not be completely avoided, as the abstraction of encapsulation is often useful. Therefore, if the developer wishes to declare instances of an inner class, they must be instantiated as non-static.

```
public class InnerClassLeak extends Activity
{
    class InnerClass {};
    static InnerClass innerClass;

    @Override
    public void onCreate() {
        leakMemory();
    }
    private void leakMemory() {
        innerClass = new InnerClass();
    }
}
```

Fig. 3. Example of an inner class memory leak

D. Anonymous Classes

The last type of memory leak we will examine occurs as a result of anonymous classes. Like inner classes, anonymous classes also hold a reference to the Activity in which they were created. Anonymous classes are often used to perform background work within an activity, such as AsyncTasks, Handlers, Threads, and TimerTasks. Notably, these classes call methods with the name convention of doInBackground() or run(). In this case, if this background work persists, the anonymous class will continue to hold the reference to the outer Activity, so this Activity will not be garbage collected.

```
public class AnonymousClassLeak extends Activity
{
    @Override
    public void onCreate() {
        leakMemory();
    }
    private void leakMemory() {
        new AsyncTask<Void, Void, Void>() {
            @Override
            protected Void doInBackground(...)
            {
                while(true);
            }
        }.execute();
    }
}
```

Fig. 4. Example of an anonymous class memory leak

The way to avoid this type of memory leak is somewhat more subtle, as all of these background tasks are global to the state of the app. We can perform the same operations, but we must do them with a nested static class, rather than an anonymous class. Nested static classes do not maintain references to outer class instances, so our background tasks will no longer hold Activity references while they execute.

V. Understanding Leaks in Android Development

In order to tackle the issue of memory leaks in the Android framework, we first had to familiarize ourselves with the components that create an Android application, specifically with the Activity class. By understating how an activity is created and destroyed we were able to comprehend the best way to implement a memory-leaking reference app. Our app had to fully cover the memory leak antipatterns that a developer may encounter in a simple and straight forward implementation.

A. The Android Activity Lifecycle

An activity is the single screen in Android, similar to a window or frame in Java UI. The Activity class provides a group of callback methods that allow the activity to understand that its state has changed. For example, an Activity will want to understand that the system is creating, stopping, or resuming the activity, or destroying the entire process in which the activity resides. These states are the components that define the Android lifecycle, which is controlled by seven methods: onCreate(), onStart(), onResume(), onPause(), onStop(), onRestart(), and onDestroy(). Each of these methods describes how the activity will behave at the different corresponding states. Within each lifecycle callback method, a programmer can define how the activity works when the user or system interacts with the activity. A memory leak will occur if the method onDestroy() finishes and some reference to the destroyed Activity still exists.

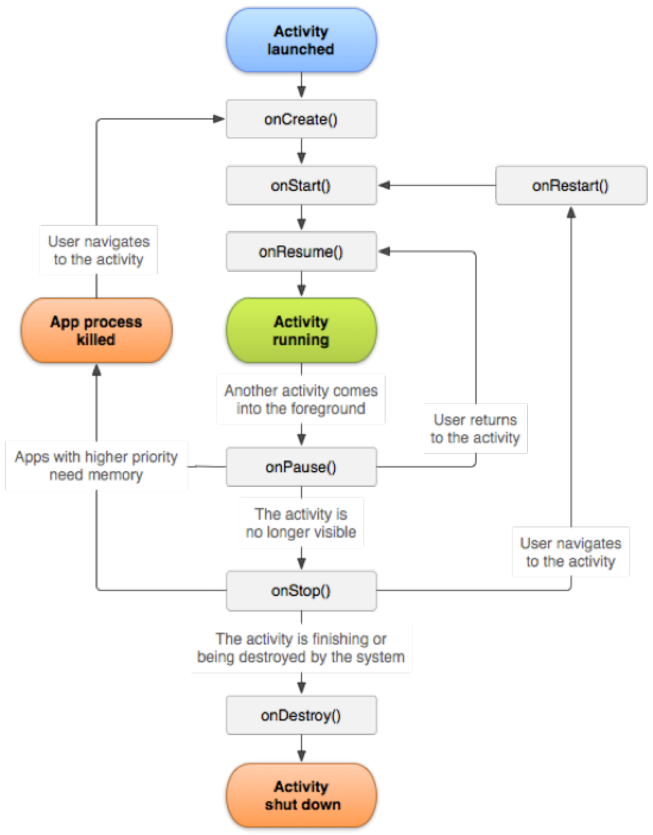

Activity launched
onCreate()
onStart()
onRestart()
onResume()
User navigates to the activity
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Fig. 5. A simplified illustration of the activity lifecycle

B. Developing a Reference app

In our reference app we developed four classes that correspond to the four memory leak anti-patterns. Each class extends the Androids Activity class and leaks memory according to its pattern. After opening the app and going through a splash screen, a user is introduced to a menu in which one can choose to invoke a leaky activity. Once the user opens one of the activities, it will like memory after it is destroyed. Destroying an activity in our app can be performed by switch the phone orientation. This process destroys the current activity and reload a new activity. An about screen is also included in the app and is accessible by pressing the upper right corner of the menu screen. LeakCanary, a leading dynamic analyzer on which we will elaborate later, is also fully integrated in our app. The source code of our app was committed to the Infer project as example code for a memory- leaking Android app. The APK installation file of our app was submitted alongside this paper. The app was developed with Android SDK 7.1.1 (Nougat, API Level 25, rev 3) and tested on a Nexus 5X phone running Android 7.1.1. The minimum supported Android version is Android 4.0 IceCreamSandwich.

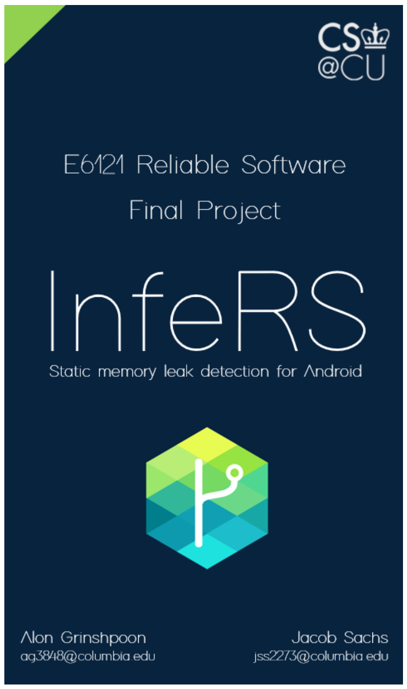


Fig. 6. The about screen of the reference app

VI. Existing Infer Capabilities

Infer is a static analysis tool which, given Java or C/C++/Objective-C source code, is able to produce a list of potential bugs and patterns that may result in memory leaks [3]. By design, Infer reports null pointer exceptions and resource leaks in Android and Java code. After a successful Infer run, a user is presented with a short report regarding the bugs found. One may explore Infers reports in more details by running the command inferTraceBugs from the same directory the original test was running.

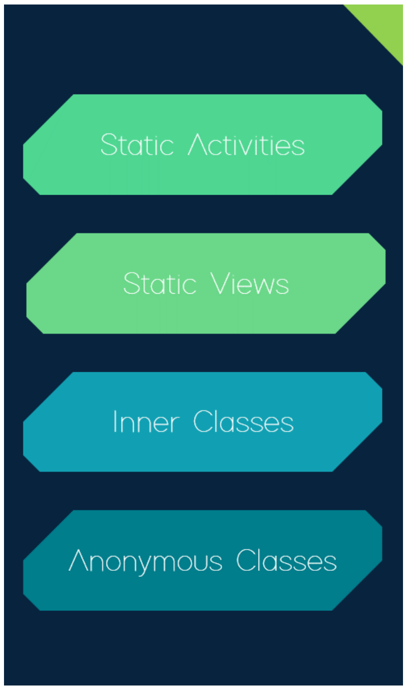


Fig. 7. The menu screen of the reference app

An Infer run is separated into two main phases: the capture phase and the analysis phase. In the capture phase, source files are converted into Infers internal intermediate language. This translation process is similar to compilation, as Infer captures the compilation commands in order to perform the translation. Infer stores the intermediate files in the results directory that is reachable by the user.

In the analysis phase, the intermediate files results directory are analyzed by Infer. Infer analyzes each function and method separately. In the case that Infer encounters an error while analyzing a function or method, it will stops the analysis for that specific method or function and continue the analysis to other methods or functions. This means that after fixing reported errors, future runs of Infer may result in additional errors reported.

Infer reports errors to the standard output and also saves the results to the results directory. Infer serializes OCaml data structures that contain a control flow graph for each function or method implemented in the source code. By traversing the control flow graph Infer is able to search for bugs and memory leaks. In Android, Infer is able to report if any Context is reachable from a static field. Thus it is able to catch static activity memory leaks in Android since Activity is a subclass of Context (see figure 8).

```
java.lang.Object
   android.content.Context
      android.content.ContextWrapper
         android.view.ContextThemeWrapper
            android.app.Activity
```

Fig. 8. The Activity class hierarchy which originates from Context

Infer is also able to report inner class memory leaks in Android. Since inner class instances maintain a reference to the outer class instance, a static reference to such instances means the existence of a static field that can be used to reach the outer activity. Infer will consider it as a reachable Context and report the error.

Currently, Infer handles neither static view memory leaks nor anonymous class memory leaks. This is due to how these components maintain a reference to their context in Android. While both introduce a persistent reference chain that leaks their context, Infer does not report it as a bug. We plan to extend Fnfer to cover such possible leaks.

The Infer analyzer performs sophisticated interprocedural static analysis. When interprocedural analysis is not needed, one may refer to the Infer:Checkers framework. Checkers can analyze and check for a given property in each method of a given project intra-procedurally instead of inter-procedurally. This means that by extending the Checkers component we can check for memory leaks in each function at a time. Therefore, using checkers suits well as a detection approach for static view memory leaks and anonymous class memory leaks.

A. *Running Infer on the Reference App*

In order to demonstrate the existing and missing components of Infer for static memory leak detection, we ran it on our reference application containing the four main types of memory leak. Infer can be run at build time for Android apps, and detects leaks via build data on a clean build. Running `gradle clean && infer -- gradle build` yields two alerts. The alert for the static activity memory leak can be seen in figure 9.

```
CONTEXT_LEAK
`edu.columbia.cs.infers.Option1Activity.onCreate`:
Leaked edu.columbia.cs.infers.Option1Activity```
38.             // Leak memory
39.             leakMemory();
40. >       }
41.
42.         private void leakMemory(){
```

Fig. 9. The Activity class hierarchy which originates from Context

These two leaks are picked up by Infer's generalized Context leak checker, which picks up on both Activity and InnerClass static memory leaks. However, it misses static View leaks and thread-based anonymous classes memory leaks. These are the leaks we targeted in our extension of Infer.

VII. Modifying Infer

In order to modify Infer, you must first set up a local development environment. Please see appendix A for instructions on quickly building Infer from source. The source for the extensions was submitted alongside this paper.

A. Static View Leaks

To extend Infer to identify static View memory leaks, we leveraged the checkers feature of Infer. In addition to Infer's standard interprocedural checkers, the software also contains intraprocedural analysis, in the form of checkers. These checkers are all based on the Control Flow Graph (CFG) of an application. A checker operates by looking for a given pattern within source code, then iterating over every node within the CFG [4].

To create a new checker, we add the checker to a registration file in the Infer source. We then create a corresponding callback function for the checker. This callback is essentially a pattern matcher which is run on every CFG node. If the callback matches a specified pattern, the checker will alert the user to a potential problem.

For static View leaks, we created a checker called `activity_retains_static_view`. This is based on the `fragment_retains_view` checker, but modified to check all statics within an Activity, rather than fragments. The pattern we are matching is any static View field that is not nulled out in the onDestroy lifecycle function.

If we run `infer -a checkers -- gradle build` on our sample application, we get an error on the onDestroy method of the Activity, since a static View was not nulled out. When running Infer, it will point you to the lifecycle method with the offending memory leak. It is up to the developer to determine where the memory leak is caused, though the Infer error text usually provides enough information.

B. Anonymous Classes

To extend Infer to identify Async Task memory leaks, we again utilized the checkers feature of Infer. This time, we leveraged Infer's `is_anonymous_inner_class` type checker, which performs an evaluation on method strings to determine if they pattern match on an anonymous function. As previously mentioned, anonymous classes should not be used to run background tasks in Android, as the persistent task will also persist a reference to the parent Activity.

To create this checker, we again started with the `fragment_retains_view` checker. Instead of checking if a value is a fragment, we now check to see if values inside of an Activity are anonymous functions. If they are, we report an error to the user.

When we run this checker, we get many alerts, and not all of them are actual problematic areas of code. In order fine tune the checker to only alert the developer about anonymous classes which perform background work, we filtered the results by looking only on classes that include function named doInTheBackground() and run(). There as common name conventions for potently memory-leaking classes as AsyncTasks, Handlers, Threads, and Timers. Again, it is up to the developer to make the final call on whether or not these risk areas are true memory leaks.

VIII. Evaluation

A. Testing the New Checkers

With our new checkers, we are looking to catch both the Static View memory leaks, as well as any Anonymous Classes memory leaks. Figures 10, 11, and 12 contain outputs from a run of modified Infer on the dedicated Android application we build for this paper.

```
48.
49.        @Override
50. >      public void onDestroy(){
51.       //view = null;
52.       nonStaticView = null;
```

Fig. 10. Static view memory leaks

```
40.         private void leakMemory(){
41. >           new AsyncTask<Void, Void, Void>() {
42.                 @Override protected Void
                      doInBackground (Void... params) {
43.                     while(true);
```

Fig. 11. Anonymous class with background work memory leak

```
40.       new Thread() {
41.           @Override
42. >         public void run(){
43.               while (true){
44.                   SystemClock.sleep(1000);
...
48.       new Handler().postDelayed(new Runnable(){
49.           @Override
50. >         public void run() {
51.
52.           Intent mainIntent =
                new Intent(SplashActivity.this,
                         MenuActivity.class);
```

Fig. 12. Additional anonymous class memory leak

Our checkers successfully pick up on the retained Static View memory leak, as well as three different anonymous classes memory leaks.

Infer points to the onDestroy lifecycle method for the Static View leak. Our checker reports that the static view reference is not set it to `null` in the onDestroy function. It also disregards non-static views as expected. Uncommenting line 51 in figure 10, solves the static view memory leak and results in the consecutive run of our checker to report no error.

We also catch the doInBackground AsyncTask memory leak seen in figure 11. Because this task is an anonymous inner class, it will hold an Activity reference as long as it persists. Background tasks implemented with a `Thread()` call are also caught by the checker.

We were surprised to see our checkers catch a memory leak in our SplashActivity seen in figure 12. Here, the splash screen uses an anonymous `Handler()` to start the main activity. We should have actually extended the handler with a static private class, rather than declaring an anonymous handler.

It is worth mentioning that Infer can also produce an image of the Control Flow Graph. This can make analysis of Infer warnings much more meaningful, and can allow a developer to make a better determination if a warning is actually a memory leak. The CFG for a Static View leak is included as a PDF with our submitted materials. To generate a CFG, run the following commands:

```
infer -g -a checkers -- gradle build
dot \
  -Tpdf infer-out/captured/Option2*.java*/icfg.dot \
  -o icfg.pdf
open icfg.pdf
```

We used these control flow graph to plan the implementation of our checkers and to make sure they perform as expected.

B. *Testing on a open-source Android app*

After verifying the work of our checkers using our own Android application created for this paper, we decided to test them on a real-world third-party open-source Android app. This would put our checkers to the test and ensure their correctness while proving their necessity in the commercial space of Android development.

Habitica is a popular open-source Android application, which converts a ToDo list into an RPG-style interface [7]. It is a massive project, with over 1200 source files. We decided to run Infer on this project to see what we could find.

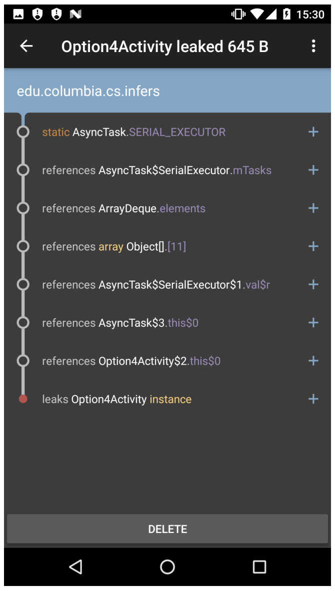
Our new checkers found 26 Static View memory leaks and 17 anonymous class memory leaks. These leaks occur in both small UI classes as well as critical classes such as the in-app message manager. The message manager has several anonymous inner class patterns. If this activity ever goes out of scope, it will leave behind a large number of async tasks. As long as any of these tasks persist, a reference to the original activity will also persist. The full bug report is included in our additional materials.

C. *Comparison to LeakCanary*

The primary point of comparison for a static analysis product like Infer is dynamic analysis. One of the leading dynamic analysis tools for Android applications is LeakCanary [5]. Developed by Square, LeakCanary is a simple and powerful way to execute dynamic analysis on an Android app.

By adding a single line of code to your app's base Application class, LeakCanary will run in the background. If a memory leak occurs while the app is running, LeakCanary will raise an alert and a stack trace. The alert for discovery of an AsyncTask memory leak is displayed in figure 13.

LeakCanary was able to find all of the memory leaks that we tested against with static analysis. In general, a good dynamic analysis tool possesses the same capabilities as a static analysis tool. However, there are important differences in how a developer should rely on these tools during the development lifecycle, which we will discuss below.



Option4Activity leaked 645 B

edu.columbia.cs.infers

```
static AsyncTask.SERIAL_EXECUTOR
references AsyncTask$SerialExecutor.mTasks
references ArrayDeque.elements
references array Object[].[11]
references AsyncTask$SerialExecutor$1.val$r
references AsyncTask$3.this$0
references Option4Activity$2.this$0
leaks Option4Activity instance
```

Fig. 13. An alert and trace of an AsyncTask memory leak

IX. Related Work

A. *LeakCanary*

It is very reaffirming that, with our additions, Infer was able to pick up the same memory leaks as LeakCanary. Being able to rely on static analysis provides many advantages over dynamic analysis. For example, static analysis can be run as a git hook, preventing developers from committing any leak-prone code to the central repository. Static analysis also incurs significantly less overhead than dynamic analysis. No modifications to the code are necessary, and the time it takes to run–in the case of Infer–is only bounded by the project build time.

Despite this, we cannot dismiss the importance of a tool like LeakCanary. Part of the reason that we could use it to confirm our results is that dynamic analysis has a much lower rate of false positives and negatives. With dynamic analysis, memory leaks are detected by simply monitoring Android's memory management. There is no need to pattern match, or to determine if a code pattern will actually generate a leak or not. Static analysis is predictive, but dynamic analysis is responsive. If LeakCanary raises an alert, then a leak has truly occurred.

B. *ErrorProne*

While Infer appears to be the most powerful open source static analysis tool available for Android code, there are a few alternatives. The most promising tool, which we initially considered evaluating, is Google's Error Prone [6]. Error prone is specifically for Java code, and it is build with the developer in mind.

The tool exists as a plugin which can be added to most Java build stacks. This restricts it to being used as part of the development lifecycle. When run against our reference application, Error Prone does not catch any of our memory leaks. This tools really only serves as a linter, catching common Java programming mistakes and typos. It does, however, provide immediate remediation suggestions, pointing to the precise location to make a suggested edit.

Error Prone may have an important position in certain development toolchains. However, if catching memory leaks is important to your development lifecycle, Infer appears to be the best static analysis tool for the job.

X. Conclusion

In this paper we were able to create a way for Android developers to preemptively check for and prevent memory leaks. We were able successfully extend Facebooks Infer framework to support all the discussed memory leaking code pattern.

We managed to identify and implement the memory leak scenarios by creating an Android app dedicated for this paper. This app serves as test cases for our project and example code for how to implement the memory leaks in Android.

We successfully developed Infer-based checkers to help developers detect scenarios not covered by Infer by default. We succeeded in testing our checkers with our reference app and compare its results to LeakCanany, a leading dynamic analyzer. Our static analyzers results were on par with those of the dynamic analyzer, covering all memory leaks in the app.

We also ran our checkers on Habitica, a real-world third-party app that is used by over 2 million users, and detected numerous bugs. We compared our performance with the static analyzer ErrorProne by Google at concluded that our solution is able to better detect memory leaks in Andorid.

We have posted a pull request to Facebook for our project to be integrate into Infer. We hope that our project will help Android developers to avoid slowdowns and crashes in their applications by statically detecting possible memory leaks throughout the development cycle.

Appendix A: Infer Development Environment

These instructions should be carried out on Ubuntu 16.04.2 LTS (Xenial Xerus). These instructions cover both compiling our fork of Infer and testing the checkers on our demo Android application.

```
cd ~
git clone https://github.com/jsachs/infer.git

# Download and install build dependencies:
sudo apt-get install -y autoconf automake build-essential git
sudo apt-get install -y libgmp-dev libmpc-dev libmpfr-dev m4 pkg-config
sudo apt-get install -y python-software-properties unzip zlib1g-dev
apt-get install python-ctypeslib
apt-get install libffi
apt-get install cmake

# Install Java:
sudo add-apt-repository ppa:openjdk-r/ppa
sudo apt-get update
sudo apt-get install openjdk-8-jdk

# Install Opam:
apt-get install opam

# Make sure dependencies are install and up to date:
opam --version
python --version
java --version
gcc --version
autoconf --version
automake --version

# Build the java analyzer:
cd ~/infer/
./build-infer.sh java
export PATH=`pwd`/infer/bin:$PATH

# Test Java Analyzer
infer -- java examples/Hello.java

# Download and install Android 64-bit build dependencies:
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386 lib32z1 libbz2-1.0:i386

# Download and unzip Android studio:
wget dl.google.com/dl/android/studio/ide-zips/2.3.0.8/android-studio-ide-162.3764568-linux.zip
unzip android-studio-ide-162.3764568-linux.zip
cd android-studio/bin/
./studio.sh

# Follow the onscreen wizard to install the latest Android SDK and build tools.

# Add the Android SDK path to the android example project:
cd ~/infer/examples/android_infeRS/

# Create a file called  local.properties  with its content being the following single line,
# connecting the location to the downloaded SDK:
sdk.dir=<location of your Android SDK>

# Run the example:
infer -- ./gradlew build
```

Appendix B: Add a Checker to Infer

These instructions should be carried out on the latest version of Infer.

A. *Register the new checker*

Edit the file /infer/src/checkers/registerCheckers.ml and add your new checker signature call to the Java checkers list. For example:

```
let java_checkers =
    let l =
      [

      MyCheckerClass.callback_my_checker_function , checkers_enabled;

      ] in
```

B. *Creating the checker code*

1) The Checker Module (myCheckerClass.ml): Create a checker file written in OCaml. See http://fbinfer.com/docs/adding-checkers.html for more information.

2) The Checker Module Interface (myCheckerClass.mli): Create an interface file with your checkers signature. For example:

```
val callback_callback_my_checker_function: Callbacks.proc_callback_t
```

C. *Defining the Printed Output*

1) The localisation file: Edit the file /infer/src/IR/Localise.ml to add support for you need checker. For example:

```
let desc_my_checker activity_typ fieldname fld_typ pname : error_desc =
   let problem =
     Printf.sprintf "Activity␣%s␣does␣not␣handle␣field␣%s␣(type␣%s)␣in␣%s␣correctly."
       (format_typ activity_typ)
       (format_field fieldname)
       (format_typ fld_typ)
       (format_method pname) in
   let consequences =
     "Some␣description␣of␣why␣this␣is␣a␣problem." in
   let advice =
     "Some␣suggestions␣on␣how␣to␣solve␣the␣problem." in
   { no_desc with descriptions = [problem; consequences; advice] }
```

2) The localisation interface: Also edit the file /infer/src/IR/Localise.mli to add support for you need checker. For example:

```
val desc_my_checker :
    Typ.t -> Ident.fieldname -> Typ.t -> Procname.t -> error_desc
```

D. Build Infer

Run: `./build-infer.sh java`

Appendix C: InfeRS results on the reference app

```
Found 5 issues

app/src/main/java/edu/columbia/cs/infers/Option2Activity.java:50: error:
      CHECKERS_ACTIVITY_RETAINS_STATIC_VIEW
  Activity edu.columbia.cs.infers.Option2Activity does not nullify static View field view
          (type android.view.View) in onDestroy. If this Activity is destroyed, a reference
           to it will leak through the static View that retains. In general, it is a good
          idea to initialize Views in onCreate, then nullify them in onDestroy.
  48.
  49.         @Override
  50. >       public void onDestroy(){
  51.                 //view = null;
  52.                 nonStaticView = null;

app/src/main/java/edu/columbia/cs/infers/Option4Activity.java:41: error:
      CHECKERS_ACTIVITY_CONTAINS_ANONYMOUS_CLASS
  Activity contains anonymous inner class in doInBackground. If this Activity is destroyed
          , a reference to it will persist if the anonymous class performs background work.
           Use only static nested classes.
  39.
  40.         private void leakMemory(){
  41. >           new AsyncTask<Void, Void, Void>() {
  42.                 @Override protected Void doInBackground(Void... params) {
  43.                     while(true);

app/src/main/java/edu/columbia/cs/infers/Option4Activity.java:42: error:
      CHECKERS_ACTIVITY_CONTAINS_ANONYMOUS_CLASS
  Activity contains anonymous inner class in doInBackground. If this Activity is destroyed
          , a reference to it will persist if the anonymous class performs background work.
           Use only static nested classes.
  40.         private void leakMemory(){
  41.             new AsyncTask<Void, Void, Void>() {
  42. >               @Override protected Void doInBackground(Void... params) {
  43.                     while(true);
  44.                 }

app/src/main/java/edu/columbia/cs/infers/Option5Activity.java:42: error:
      CHECKERS_ACTIVITY_CONTAINS_ANONYMOUS_CLASS
  Activity contains anonymous inner class in run. If this Activity is destroyed, a
          reference to it will persist if the anonymous class performs background work. Use
           only static nested classes.
  40.             new Thread() {
  41.                 @Override
  42. >               public void run(){
  43.                     while (true){
  44.                         SystemClock.sleep(1000);

app/src/main/java/edu/columbia/cs/infers/SplashActivity.java:50: error:
      CHECKERS_ACTIVITY_CONTAINS_ANONYMOUS_CLASS
  Activity contains anonymous inner class in run. If this Activity is destroyed, a
          reference to it will persist if the anonymous class performs background work. Use
           only static nested classes.
  48.                         new Handler().postDelayed(new Runnable(){
  49.                             @Override
  50. >                           public void run() {
  51.                                 // Create an Intent that will start the Menu Activity
  52.                                 Intent mainIntent = new Intent(SplashActivity.this,
          MenuActivity.class);

Summary of the reports

  CHECKERS_ACTIVITY_CONTAINS_ANONYMOUS_CLASS: 4
      CHECKERS_ACTIVITY_RETAINS_STATIC_VIEW: 1
```

As can be seen above, our added checkers were able to detect the issues that Infer ignored: static view memory leak and anonymous class memory leak.

Acknowledgements

We extend our gratitude to Prof. Junfeng Yang for introducing us to the paper's initial idea of static memory leak detection for Android.

We owe a great thanks to the Facebook development team for their development of the Infer application. Additionally, Sam Blackshear (GitHub: @sblackshear) gave us an enormous amount of guidance in getting to know Infer.

References

[1] T. Huzij, Eight Ways Your Android App Can Leak Memory, in Nimbledroid Blog, May 23, 2016, http://blog.nimbledroid.com/2016/05/23/memory-leaks.html.
[2] Android Developer, Weak Refrences, https://developer.android.com/reference/java/lang/ref/WeakReference.html
[3] Infer by Facebook, http://fbinfer.com
[4] Simple intraprocedural checkers in Infer, http://fbinfer.com/docs/adding-checkers.html
[5] LeakCanary, Source Code, https://github.com/square/leakcanary
[6] ErrorProne by Google, http://errorprone.info/
[7] Habitica on GitHub, https://github.com/HabitRPG/habitica-android
[8] E6121 Reliable Software Course Website, Final Project: http://www.cs.columbia.edu/ junfeng/17sp-e6121/hw/hw3.html
[9] Android Developers, The Activity Lifecycle: https://developer.android.com/guide/components/activities/activity-lifecycle.html
[10] JavaPoint, Android Activity Lifecycle: http://www.javatpoint.com/android-life-cycle-of-activity
[11] Nimbledroid Blog, Ways Your Android App Can Leak Memory: http://blog.nimbledroid.com/2016/05/23/memory-leaks.html
[12] Codexpedia, Memory Leaks in Android, http://www.codexpedia.com/android/memory-leak-examples-and-solutions-in-android/
[13] Medium, Memory Leaks in Android, https://medium.com/freenet-engineering/memory-leaks-in-android-identify-treat-and-avoid-d0b1233acc8
[14] Infer Bug Types, http://fbinfer.com/docs/infer-bug-types.html
[15] LeakCanary, FAQ, https://github.com/square/leakcanary/wiki/FAQ
[16] Using LeakCanary, https://www.youtube.com/watch?v=2VKBjlHtKMY
[17] Open-Source Android Apps, https://github.com/pcqpcq/open-source-android-apps
[18] F-Droid, catalogue of FOSS (Free and Open Source Software) Android apps, https://f-droid.org/repository/browse/
[19] OCaml Basic Programming, https://ocaml.org/learn/tutorials/basics.html