Dynamic Detection of AsyncTask Related Defects

Qing Liu[1,3], Linjie Pan[1,3], Baoquan Cui[1,3,*], Jun Yan[1,2,3], Jian Zhang[1,3]
[1]State Key Lab. of Computer Science, Institute of Software, CAS
[2]Technology Center of Software Engineering, Institute of Software, CAS
[3] University of Chinese Academy of Sciences
Email: { liuqing, panlj, cuibq, yanjun, zj}@ios.ac.cn

2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS) | 978-1-6654-5813-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/QRS54544.2021.00047

***Abstract*—As a widely used Android asynchronous component, AsyncTask is used to run time-consuming tasks. However, the misuse of AsyncTask will cause defects, i.e., crashes and memory leaks. Based on static analysis, existing approaches cannot accurately detect AsyncTask-related defects and produce many false positives since some paths are not reachable in practice. In this paper, we propose a dynamic detection method based on instrumentation, Monkey execution and log analysis to detect these defects. And we implement a tool AD2Checker based on the proposed method. Our experiment on 19 real-world apps shows that it has found 145 bugs and has no false positives. Moreover, it triggers crashes caused by misuse of AsyncTask.**

Keywords- Android Application; AsyncTask; Dynamic Analysis; Defect Detection;

I. INTRODUCTION

Android apps are the most widely used mobile apps in the market currently. Android is a widely used operating system based on the single thread and event-driven model, where the single thread (also called UI thread or main thread) instantiates GUI components to dispatch events and asynchronous thread (background threads) process time-consuming tasks (i.e., network access, database queries). This mechanism can improve the response speed of Android applications.

To ease asynchronous programming, Android provides several packaged asynchronous components such as IntentService [31], HandlerThread, AsyncTask [22] and existing studies show such packaged asynchronous components are indeed widely used in Android apps, and AsyncTask accounts for the majority due to its simplicity of use [5].

AsyncTask is mainly suitable for short operations, and five callback functions are provided to ease asynchronous programming [22]. Programmers invoke the *execute()* function to start an asynchronous thread and invoke the *cancel()* function to try to end the background thread. The execution of AsyncTask is usually time-consuming. After it is executed, the state of the UI or Activity may have been changed or even destroyed and at this time, the callback of AsyncTask is trying to update the UI and there will be a problem; or AsyncTask is holding the reference of Activity all time which will cause a memory leak; or the Activity has been discarded but the task in the AsyncTask was not terminated in time, then a waste of resources will happen.

Based on these, AsyncChecker [5] proposes five misuse patterns (i.e., Strong Reference, NotCancel, NotTerminate, EarlyCancel, RepeatStart) to describe these defects and uses static analysis to detect them. Although it has found a lot of real errors, there are still some complex scenarios that cannot be accurately detected and some false positives will appear. There are many reasons. For example, it has performed static analysis but does not have strict symbolic execution and constraint solving, so some unreachable paths are still regarded as bug points; in addition, partial inter-process analysis was carried out to detect bugs, so that some real path information was discarded.

To reduce false positives in AsyncChecker, we propose a dynamic method to detect the misuse of asynchronous component AsyncTask and implement a tool called AD2Checker to automatically detect these errors.

AD2Checker first implements instrumentation containing the execution information of the program and the log printing statements of AsynTask related operations based on the Soot [3] framework and the intermediate representation of Jimple in some points such as the Activity lifecycle methods and the AsyncTask operation methods, addressing several bug patterns mentioned in AsyncChecker, and then writes the instrumented code back into the APK, packs it and signs to obtain a new APK.

AD2Checker uses a script to drive Monkey [1], which is a command-line tool developed by Google that can be run in an emulator or on an actual device sending a pseudo-random user event stream to the system, to automatically run the instrumented APK on the Android device.

Log information is collected for error pattern matching. We have collected enough information to distinguish different object instances and different execution paths to ensure the accuracy of matching. Our method detects 145 bugs on 19 apps in F-Droid [2]. Although it is less than AsyncChecker (215), they are real defects and will not cause additional judgment work for developers. On the other hand, our experiments also triggered crashes caused by AsyncTask misuses, which are the primary concern of app developers and static analysis cannot achieve this effect.

In summary, we make the following contributions in this paper.

1. **Methodology:** We propose a dynamic detection method based on instrumentation and log analysis to

* Corresponding author

2693-9177/21/$31.00 ©2021 IEEE
DOI 10.1109/QRS54544.2021.00047

check AsyncTask-related defects. And then, we check defects in logs according to the detection rules.

2. **Tool:** We develop a tool named AD2Checker based on the proposed detection method.
3. **Evaluation:** We verify AD2Checker's effectiveness in 19 real-world apps. The experiment shows that AD2Checker provides more detailed and readable reports without false positives and triggers some crashes caused by AsyncTask misuse.

II. BACKGROUND

A. Dynamic Detection

Dynamic analysis technology generally uses the dynamic execution information of the program for analysis. This kind of technology does not require the source code of the program, but it needs to run the program in advance and collect the execution sequence, and then analyze it. We can use dynamic analysis techniques to detect the security, energy and performance of Android applications [27, 28, 29]. Our work instruments the applications to collect logs and analyzes the defects of AsyncTask according to the detection rules [5]. Instrumentation [25, 26] and Log analysis [12, 13, 14, 15, 16] are common in dynamic analysis technology. Program instrumentation is inserting specific probe code in a location of the program and maintaining the integrity of the source program. The probe code usually dynamically monitors the program and records runtime data, including path coverage information, function call information and so on. Log analysis is extracting the characteristic information of the program by mining logs. The information includes user request flow, timing information, etc.

B. AsyncTask and Its Misuse

Due to the simplicity of use, AsyncTask [22] accounts for the majority of the six Android asynchronous classes. AsyncTask provides five callback functions to ease asynchronous programming, including *onPreExecute()*, *doInBackground()*, *publishProgress()*, *onProgressUpdate()* and *onPostExecute()*. The *doInBackground()* function is used to run time-consuming tasks, so it works in the asynchronous thread. The remaining 4 callback functions all work in the UI thread; *onPreExecute()* is used to do some preparation before starting the asynchronous thread; *publishProgress()* and *onProgressUpdate()* are used to update the progress of the asynchronous thread in real time; *onPostExecute()* is used to return the results of asynchronous thread to the UI thread. Once an asynchronous thread is started, the callback functions will be invoked in order by Android System. However, the running of AsyncTask's time-consuming tasks will be affected by user events (i.e., rotate the screen, click the back button, etc.), so developers should pay attention to these issues when programming.

AsyncChecker [5] has summarized 5 defects caused by the misuse of AsyncTask, as Table I shows. ***StrongReference*** means that if the AsyncTask is declared as a non-static internal class of the Activity, the object of AsyncTask will hold a strong reference to the object of Activity through the GUI component. Once the Activity is destroyed, the memory occupied by the object cannot be released, which will cause memory leak. ***NotCancel*** means that if the Activity is destroyed while the AsyncTask is still running without invoking *cancel()*, the AsyncTask cannot return the result to the GUI component of Activity, which will cause the program to crash. The above two defects are all caused by the incorrect handling of the interaction between AsyncTask and its external call components (e.g., Activity). ***NotTerminate*** means that if the *cancel()* is invoked and the *isCanceled()* is not invoked in the loop body of the asynchronous thread, the asynchronous thread will not be terminated in time and the energy waste will occur. ***EarlyCancel*** means that the asynchronous thread is terminated before being started, which will cause wrong state. The above two defects are all caused by the incorrect way to terminate the asynchronous threads. ***RepeatStart*** means that the same instance object of AsyncTask cannot be executed repeatedly, otherwise, an exception will be thrown.

For the convenience of description, we give BugPatterns abbreviation mapping, and most of the abbreviations will be used in the subsequence. This mapping is also shown in Table I.

TABLE I. BUG PATTERN AND ITS ABBREVIATION

BugPattern	Abbreviation
StrongReference	**SR**
NotCancel	**NC**
NotTerminate	**NT**
EarlyCacncel	**EC**
RepeatStart	**RS**

C. Motivating Example

As shown in Figure 1, when a file does not exist, the file is created and the AsyncTask operation is performed. When *onResume()* is called multiple times, the file will be created at most once. At this time, the *execute()* method of *asyncTask* will not be called repeatedly, the RS error does not exist at this time. However, *AsyncChecker* cannot detect the status change after the file is created, and mistakenly believes that the *if*-module in *onResume()* will be called repeatedly together, resulting in a false positive to the RS.

If we dynamically track related calls of *asyncTask*, the *execute()* will not be executed repeatedly, thereby avoiding false positives.

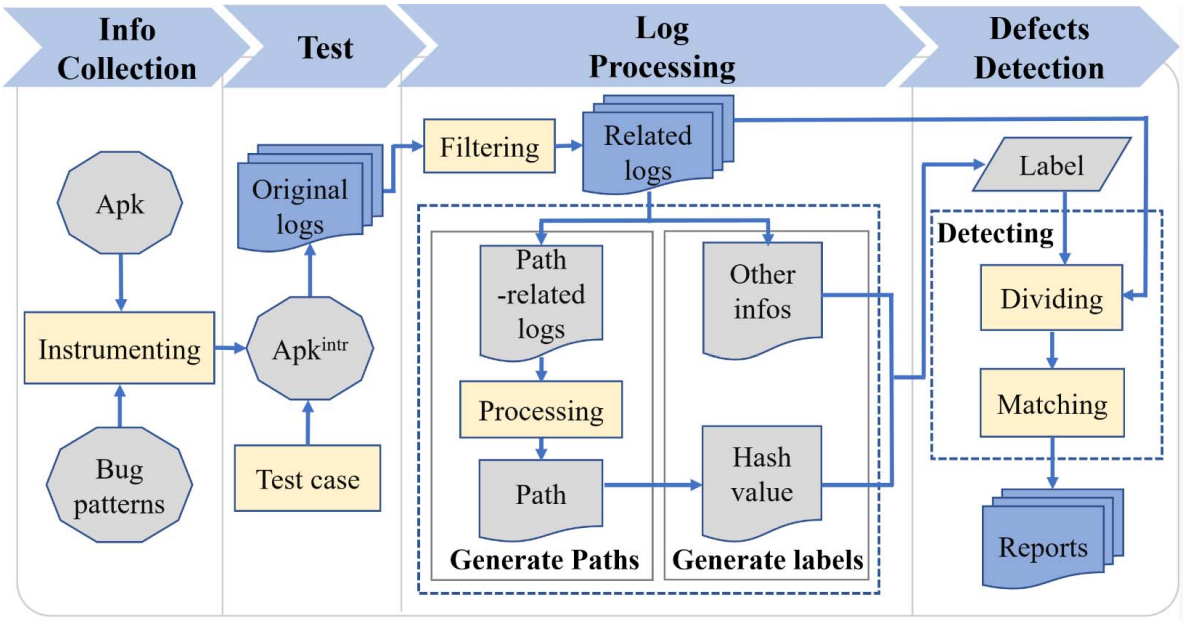

Figure 2. Overview of the Detection Approach

```
public class MyActivity extends Activity{

    private AsyncTask asyncTask = null;
    private File file = null;
    …
    protected void onResume(){
    {
        If (!file.exist()){
            file.createNewFile();
            asyncTask.execute();
        }
    }
    …
}
```

Figure 1. Motivating Example

III. METHODOLOGY

In this part, we will introduce our method. First, an overview of our approach is introduced. Next, each module will be described in detail.

A. Overview

Figure 2 shows an overview of our approach and we implement a tool named AD2Checker based on the approach.

The basic work-flow of this approach is detecting AsyncTask related defects based on log analysis. Given Android APK and misuse patterns, AD2Checker first performs instrumentation based on the information we need, mainly including log output statements. The instrumented code is written back to the APK, which will be re-signed so that it can run correctly on the Android device. Then, AD2Checker configures automated testing tool (such as Monkey [1]) to drive the operation of this APK, the generated logs are collected by Logcat. Next, AD2Checker processes the collected logs to generate paths and labels. In this step, we first dynamically generate paths to the target functions according to the ExecutionStack; and then, we extract the hash value of Activity to generate the label of each log sequence. Finally, the log sequence is grouped according to the label and matched automatically to report the bug results.

B. Instrumentation

For those AsyncTask operations, log print statements are needed to be inserted into corresponding positions to collect program execution information and we implemented an instrumentation module to automatically insert them. Our hypothesis is the same as the traditional dynamic test, believing that the inserted log information will not cause additional impact on the bug.

Table II describes the inserted log information and location point, as well as the bug patterns served. The entry and exit of all methods are inserted into the log statement, used to identify that a method is called and exited. The reason for adding the exit identifier is to distinguish whether the method call sequence is nested call or sequential execution, such as ABC or A(B(C)). The signature of a method represents its unique identifier. In Java, the signature of a method is mainly composed of the full class name, method name, parameter type list, and return value type. All types include package names if they are reference types.

The *hashCode()* method is easy to understand in Java. It represents the hash value of an object instance, which is usually the only one that is not easy to collide. When instrumenting, the *hashCode()* method of the *Object* class can be directly called by an instance of any reference type, because all reference types inherit from the *Object* class. Developers usually do not deliberately rewrite this method. It

is easy to know that whether the *hashCode()* method has been rewritten by scanning the specific class, we only need to uniquely identify the instantiated object, so calling the default *hashCode()* is enough. Here, we pay more attention to the hash value of Activity and AsyncTask instance objects. Some of the life cycle methods of the former represent the entry of the program because Android does not have a direct *main()* entry, and the latter represents the object we are concerned about. The sequence begins with the Activity lifecycle methods and ends with operations of the AsyncTask object, which represents the execution path of the program.

```
public class MyActivity extends Activity{
    …
+    public void onDestroy(){
+    {
+        super. onDestroy();
+    }
     …
}
```

Figure 3. Example of Adding *onDestroy()* Method

The *onDestroy()* method represents that the Activity is destroyed. This is part of the information we collect to determine whether the AsyncTask object has a strong reference or the *cancel()* method is executed before the activity is destroyed. Some Activity classes do not rewrite *onDestroy()*, which will affect our instrumentation and judgment. Therefore, we need to check all Activity classes. If a class does not override the *onDestroy()* method of the class, we need to insert one and implement its body containing the call to the parent class *onDestroy()* to make the Activity life cycle complete. As shown in Figure 3, *super.onDestroy()* is invoked.

```
public class MyActivity extends Activity{
    …
    public type methodName(){
    {
        //…
         asyncTask = …;
+        StringBuider sb = new StringBuider();
+        StackTraceElement[] trace = Thread
                                .currentThread().getStackTrace();
+        for (StackTraceElement e : trace){
+            sb.append(e.getClassName());
+            sb.append(e.getMethodName());
+            sb.append("_");
+        }
+        sb.append(asyncTask.hashCode()+"/execute");
+        Log.i("execute", sb.toString());
        asyncTask.execute(); // an AsyncTask Operation
    }
    …
}
```

Figure 4. Example of Adding ExecutionStack

ExecutionStack represents the stack information of program execution. It is a stack sequence of method execution. The bottom of the stack represents the entry method and the top of the stack represents the currently executed method. Each stack element in the stack contains the signature information of the method. Java provides the corresponding API to obtain this information. What we have to do is to traverse the stack and concatenate the method names to quickly locate a certain operation of the AsyncTask object. We use *StringBuilder* as a buffer container and finally convert it to a string for printing, as shown in Figure 4.

Separators, such as '#' and '_', will be added between different messages to facilitate log analysis and string segmentation for matching.

C. Execution

Monkey will be used to run the APK after instrumentation. It is a command-line tool used to randomly send time series to the device. It is very simple to use, but we have done extra work to make it more effective for our work.

We first use batch scripts instead of direct command lines to automatically combine with re-signing and renaming and Logcat log collecting instead of manually modifying the scripts every time, which can save a lot of time and is not easy to make mistakes. Secondly, the static analysis helped us get a list of activities that contain AsyncTask operations. *Logcat* will export the generated log information instrumented before from the device. When instrumenting, we use the *android.util.Log* related output method in the Android SDK instead of the common *System.out* or *System.err* in Java as print statement. Two parameters (TAG and Information) are provided in it and the former can directly distill the information we have inserted, so as to provide purer AsyncTask-related execution information for log analysis. In order to collect crash information, we sometimes need to extract the whole log in the target application containing the crash execution stack information, which is captured and output by the Android system.

When sending a sequence, these activities can be directly started and those without AsyncTask operations will not be started.

D. Log Analysis

Logcat collects all the information we instrumented. Then the logs are grouped by the hash value of the AsyncTask object, and the hash value of the same value is grouped into one. Thus, we get a sequence of AsyncTask operations with execution path information. This sequence contains the activity life cycle method information of the entry, and the corresponding *onDestroy()* method log information is obtained according to the Activity hash value. Then add the *onDestroy()* log information to the AsyncTask execution sequence ordered by timestamp. Next, we define a Bit Vector Container and standard error template vector for each misuse pattern to match. A Bit Vector Container is a vector of several dimensions, and each dimension

TABLE II. POSITION AND CONTENT OF INSTRUMENTATION

Position		Content Inserted	Bug Pattern
Method Entry & Exit		this.hashcode + MethodSignature + "out"/"in"	All
AsyncTask Operation	execute() Invoke Point	this.Hashcode+ ExecutionStack+ async.hashCode+ StringOf(InvokeStmt)	All
	cancel() Invoke Point		NC/NT/EC
	isCanceled() Invoke Point		NT
	Reference Assign Point in AsyncTask Constructor	this.hashcode + StringOf(AssignStmt)	SR
	onPostExecute() Entry	this.hashcode + MethodSignature + "onPostExecute"	NC
	onProgressUpdate() Entry	this.hashcode + MethodSignature + "onProgressUpdate"	NC
	doInBackground() Entry	this.hashcode + MethodSignature + "doInBackground"	NT/NC
Activity class		onDestroy() method and its implementation	SR/NC
onDestroy() in Activity class		this.hashcode + MethodSignature + "onDestroy"	SR/NC

TABLE III. BUG PATTERN AND ITS BIT VECTOR

Bug Pattern	Vector Container				Error Template Vector
	1st Dimension	2nd Dimension	3rd Dimension	4th Dimension	
SR	Assign Context to Field Invoke	execute() Invoke	onDestroy() Entry	/	<1,1,1>
NC	execute() Invoke	cancel() Invoke	onDestroy() Entry	onPostExecute() Exit	<1,0,1,1>
NT	execute() Invoke	isCanceled() Invoke	doInBackground() Exit	/	<1,0,1>
EC	cancel() Invoke	execute() Invoke	/	/	<1,1>
RS	execute() Invoke	execute() Invoke	/	/	<1,1>

represents a key AsyncTask operation or *onDestroy()* log information. The Bit Vector Container has an initial state with 0 in every bit. Traverse each AsyncTask operation sequence sequentially and like a sliding window, and the corresponding key operation bit will be assigned 1 if it appears, and the previous position bit will stay as 0 if the previous bit does not appear when the next dimension appears. The entire Vector Container starts with the first-dimensional matching and ends with the last-dimensional matching. If the last dimension does not appear when reaching the end of the sequence, set all the previously undecided parts to 0. The standard matching template vector for each misuse pattern represents the misuse pattern. The obtained vector container is XORed with the standard template, and 0 means that it matches the error pattern.

Table III shows the dimensions of the vector container corresponding to each pattern and the standard error template. EC and RS have two dimensions, SR and NT have three dimensions, and NC has four dimensions. Symbol '/' means there is no dimension. The 0 in the standard error template represents no occurrence and 1 represents occurrence. Figure 5 is an example of NC error template matching. The vector container has four dimensions. In the Dimension layer of Figure 5, the first dimension is the *execute()* invoke point, the second dimension is the *cancel()* invoke point, the third dimension is the *onDestroy() Entry*, and the fourth dimension is the *onPostExecute()*. The error standard template of NC is <1,0,1,1>, which means it is a sequence that includes the *execute()*, does not include the *cancel()*, and ends with the *onDestroy()* and the *onPostExecute()*. If the vector container is XORed with <1,0,1,1>, the result is 0, which means NC appears. The principle of other error pattern matching is the same way.

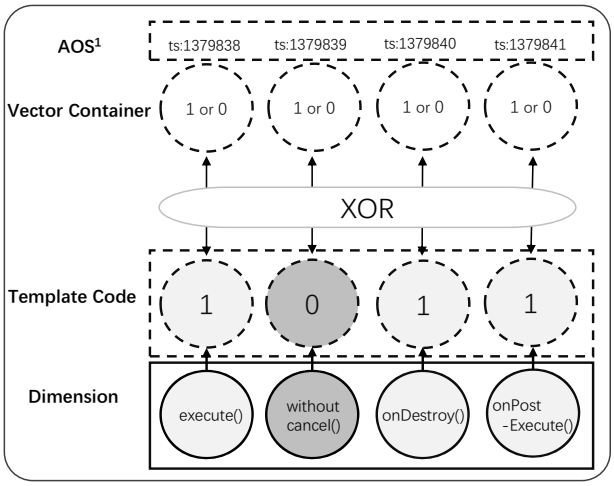

Figure 5. Vector Container Example in NC
[1]AOS is AsyncTask Operation Sequence.

IV. IMPLEMENTATION

We have implemented the AsyncTask misuse detection approach mentioned above into a tool called AD2Checker. It is written by Java based on the Soot [3] framework and Jimple intermediate representation. As Figure 2 shows, there are four modules in AD2Checker, including Information Collection Module, Test Module, Log Processing Module and Defects Detection Module.

Information Collection Module inserts extra instructions into APK and writes the instrumented code back into APK. Test Module is based on the Monkey [1] tool; it implements some scripts to resign and rename the APK to a new one and drive Monkey to run for distilling log information. Log Processing Module first extracts AsyncTask related information and path information in logs, and then the logs with the same hash value of the AsyncTask will be grouped into one. Thus, we get a sequence of AsyncTask operations with execution path information. Defects Detection Module matches bug patterns in the extracted sequences. Each path information will be matched by each bug pattern to see if the match is successful. Once the match is successful, a bug will be recorded, along with its path information. AsyncTask objects with the same path and the same hash value will be merged to reduce duplication.

V. EVALUATION

We apply AD2Checker on 19 real-world apps to show its effectiveness. Some instrumentation statistics are provided as supplementary explanations. Finally, we compare AD2Checker with existing tools AsyncChecker [5] and the result shows that our tool has a higher precision without false positives.

A. Benchmark and Experimental Setup

We collected 19 real-world applications mentioned in AsyncChecker to compare with it. The applications in the benchmark include music player applications, picture processing applications and game applications, etc., and the scale of applications are evenly distributed, which can illustrate the representativeness of the benchmark. Table IV lists the detailed information of these experimental instances.

The first column of Table IV denotes the name of apps. The following four columns show the numbers of its classes (#C), Activities (#AC), AsyncTasks (#AS), and methods (#M). The numbers of classes and methods illustrate the sizes of apps. The numbers of Activities and AsyncTasks show the attributes related to AsyncTask. The experiments have been performed on a mobile smartphone with 1.82GHz CPU and 3GB RAM.

TABLE IV. EXPERIMENTAL INSTANCES

App Name	#C	#AC	#AS	#M
taskbar	1341	21	3	9398
Jamendo	224	13	7	1146
osrshelper	639	9	4	4725
fixme	45	4	6	233
reddinator	2651	36	29	22224
gallery	2161	16	9	14830
syncthingandroid	5002	28	7	36433
commander	1047	13	6	7890
Upm	190	14	10	994
mileage	419	37	5	2620
anki	3259	38	11	25023
d00r	3062	5	6	19889
richapp	1636	14	4	12898
library	1507	9	7	12012
encapsulation	2099	9	4	17086
blinkenlights	247	13	2	1654
easy_xkcd	1192	3	2	8970
Kiss	5710	24	8	43677
wikipedia	4585	31	12	31745

We set the number of input events for each application to 200,000 for Monkey. In order to ensure the normal execution of the application and prevent the execution path from being truncated, we add the parameter "-ignore crashes" when matching rules bug delete it when crash collecting. At the same time, in order to meet the environmental conditions of asynchronous work, we sang the time interval for sending the next event. The average test time required for each application is about 20 minutes.

Monkey's decision time for each input operation is short, and its random test method can cover most of the accessible codes. We calculated the coverage rate of Monkey in 19 real-world applications. It can reach a coverage rate of 0.419 for Activity components and 0.505 for AsyncTask components. For the login page that needs personal information, we enter the legal information in advance so that the following pages behind the login page can be triggered.

To evaluate the effectiveness of our approach, we raise several research questions as follows.

- RQ1. How accurate is our work in the instrumenting module?

- RQ2. How effective is *AD2Checker* on real-world apps?
- RQ3. How accurate is *AD2Checker* compared with the existing tools?

B. RQ1- Accuracy of the Instrumenting Module

In this subsection, we will discuss how the instrumentation improves the accuracy of the collected information and impact the size of applications and instrumentation times.

We have done the following improvements in instrumentation. We found that (1) The *onDestroy()* method is not overridden in some Activities of the application; (2) there are no return statements in some functions; since our dynamic method requires these two types of information. These two problems will affect the accuracy of collected information. Therefore, we checked these two types of defects and inserted them in correct positions to improve the accuracy of information collection.

We have collected instrumentation information on 19 apps as Table V shows. The first column denotes the names of apps. The second column denotes the time of instrumentation. The last two columns denote the number of Activities that have not overridden *onDestroy()* functions and the number of functions without return statements.

As it can be seen, there are 280 Activities without *onDestroy()* method in 19 applications from 337 Activities totally, each application has 15 Activities on average; and there are 7 methods in 19 applications, which do not have return statements. We added the *onDestroy()* method to 83%(280/337) of Activities.

By instrumenting the detailed information, we improve the accuracy of collected logs about program execution. We analyzed the increment of application size and time of instrumentation. The average instrumenting time on 19 applications is 38s. This is an acceptable overhead.

TABLE V. INFORMATION OF INSTRUMENTATION

App Name	Time(s)	#Without onDestroy()	#Without return Stmt
taskbar	54	15	0
jamendo	15	13	0
osrshelper	32	8	0
fixme	20	3	0
reddinator	14	32	2
gallery	52	13	0
syncthing	46	21	1
commander	16	10	0
Upm	21	14	0
mileage	34	37	0
anki	53	28	1
d00r	23	4	0
richapp	34	12	1
library	27	7	0
encapsulation	16	7	0
blinkenlights	52	10	0
easy_xkcd	42	1	1
Kiss	10	22	1
wikipedia	61	23	0

C. RQ2- Effectiveness of AD2Checker

Table VI shows the detection results of AD2Checker. The first column denotes the names of apps. Column *#SR* and *#NC* denote how many StrongReference defects and NotCancel defects that are checked by AD2Checker respectively. Similarly, the columns *#NT* and *#EC* denote how many NotTerminate defects and EarlyCancel defects that are checked by AD2Checker respectively. The last column *total* denotes how many AsyncTask-related defects are detected in the app. RepeatStart defects did not appear in our experiment.

In this part, we discuss the effectiveness of *AD2Checker* in detecting AsyncTask-related defects. We apply AD2Checker on 19 real-world apps to verify the ability on detecting AsyncTask-related defects. The result shows that AD2Checker can detect 145 defects on 19 real-world apps.

As we can see in Table VI, the total number of defects detected by AD2Checker in 19 applications is 145, and each application has 8 defects on average. It can be seen that these AsyncTask-related defects are common in real-world applications. Among them, Jamendo App has 15 defects, which has the largest number of defects in 19 apps. The reason is that it is an image upload tool that contains many asynchronous components.

TABLE VI. RESULTS OF AD2CHECKER ON REAL-WORLD APPS

App Name	Misuse Pattern				Total
	#SR	#NC	#NT	#EC	
taskbar	1	2	2	0	5
jamendo	5	5	5	1	16
osrshelper	4	3	4	0	11
fixme	2	1	4	0	7
reddinator	2	3	6	0	11
gallery	0	2	5	0	7
syncthingandroid	1	3	3	0	7
commander	2	1	2	0	5
Upm	3	3	7	0	13
mileage	2	1	1	0	4
anki	0	0	9	0	9
d00r	2	5	5	0	12
richapp	1	1	1	0	3
library	1	1	1	1	4
encapsulation	1	0	1	0	2
blinkenlights	1	1	1	0	3
easy_xkcd	2	1	9	0	12
Kiss	1	3	3	0	7
wikipedia	3	2	4	0	9
Total	34	38	73	0	145

Furthermore, SR, NC and NT are the most common in the defects, and EC and RS are rare due to the Execute-related defects that the later one will cause the program to crash. This is similar to the distribution trend of results mentioned in AsyncChecker. Therefore, to test the effectiveness of the tool for these two patterns, we manually injected 15 defects in 5 open-source applications to verify the accuracy of AD2Checker. The results are shown in Table VII. The column *#EC* and *#RS* under the column *Execute-related* denote how many RepeatStart defects and EarlyCancel defects that are checked by AD2Checker. AD2Checker reports 13 defects of 15 and there are 2 defects

have not been reported. The reason for the false negatives is that the related asynchronous components have not been executed, which means the path is difficult to be executed under Monkey. The recall rate is 86%, and there is no false positive.

TABLE VII. MANUAL INJECTION OF EXECUTE-RELATED DEFECTS

	#EC	#RS
TP	5	8
FP	-	-
FN	1	1

If RepeatStart appears, the app will crash. This error will only appear in the code that has never been run or even before release. App after smoking test will be able to avoid this error and therefore, this kind of error will not occur in real applications. So far, whether RepeatStart is a useful pattern of misuse in practice may be discussed again.

TABLE VIII. CRASH LOGS

App Name	Crash Logs
jamendo	java.lang.IllegalArgumentException: View=DecorView@b1cf153[] not attached to window manager
syncthing-android	java.lang.IllegalArgumentException: View=DecorView@69d73d8[] not attached to window manager
d00r	java.lang.IllegalArgumentException: View=DecorView@ece2468[] not attached to window manager
d00r	android.view.WindowManager $BadTokenException: Unable to add window -- token android.os.BinderProxy@8a5f58e is not valid; is your activity running?

Some of the defects we detected can cause real-world apps to crash, which means developers should take these as serious problems. The partial crash logs are listed in Table VIII. We manually analyzed the complete stack information of Crash and confirmed that it was caused by the misuse of AsyncTask. In fact, the first three lines are the most common crashes caused by AsyncTask. *DecorView* is the root view of the Activity. When the Activity is destroyed but the AsyncTask is not terminated or canceled immediately, the UI of the Activity has been destroyed also when the UI tries to update UI after the execution then crash occurs. This is a typical crash caused by NotCancel or NotTerminate.

D. RQ3-Comparison with Existing Tool

In this section, we will compare our tool with existing tool. To the best of our knowledge, there is no dynamic-based analysis tool for detecting AsyncTask related defects. AsyncChecker checks defects of AsyncTask based on static analysis technology and it has already been compared with related work in the article, which obtains considerable experimental results. Therefore, we compared our tool AD2Checker with AsyncChecker on these 19 applications, and will not compare with other tools.

We compared the detection types, the precision of reports, the number of repeats, false positives and false negatives between the two tools. The results are shown in Table IX.

TABLE IX. COMPARED RESULTS OF ASYNCCHECKER

	FP	FN	TP	Total
AsyncChecker	7	33	215	336
AD2Checker	0	103	145	145

In Table IX, we manually compared the results of the two tools, and we counted the defects that AD2Checker could detect but AsyncChecker could not as AsyncChecker's false negatives and vice versa. In fact, AD2Checker's results are all reachable. We consider each defect as AD2Checker's false negative when it is detected by AsyncChecker but it is difficult to judge whether it is reachable. So we get the lower bound of the recall rate is 58.5% (145/(103+145)) and the precision rate is 100% as without false positives，and the corresponding F_1 value is 0.74. Although the F_1 value of AsyncChecker may be higher, developers need to spend a lot of extra time to determine whether it is a false positive in a complex path, and high false positives are often the reason why developers abandon detection tools. Our AD2Checker provides a considerable F_1 value without false positives, which is its contribution. Another contribution of AD2Checker is to provide more details and execution information, which is more readable, as shown in Figure 6.

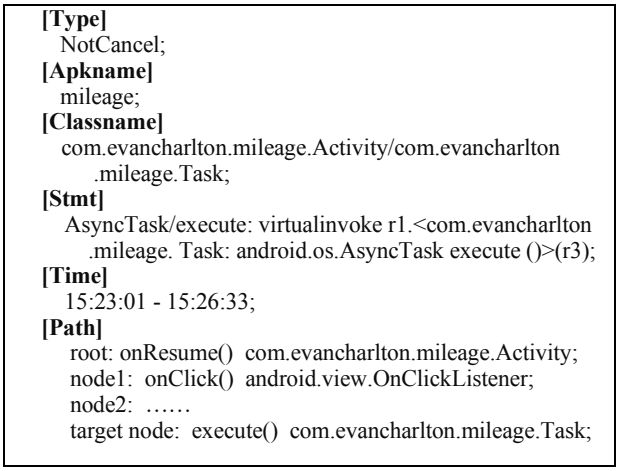

Figure 6. The Report of AD2Checker

Because the coverage of dynamic testing cannot reach 100%, some Activity and AsyncTask components cannot be executed. Therefore, AD2Checker has some false negatives compared to static analysis tool AsyncChecker.

VI. THREAT TO VALIDITY

There are two major threats to the validity of the studies.

The first one lies in the experimental benchmark. We choose apps with one DEX file from the real-world as the

benchmark. Since our instrumentation work is based on the Soot framework [3], Soot cannot repackage apps with multiple DEX files at version 2.3.3. The above problems lead to the failure of instrumentation for multi-DEX apps. However, the scale of our benchmark is an average of 2283 classes for each app, which is close to multi-DEX apps. The latest Soot version is not very stable for multi-DEX instrumentation. In the future, we will use the latest Soot version to do more experiments on multi-DEX apps.

The second one lies in the callback function set we construct. Android does not provide a detailed list of callback functions, and users can customize the callback function according to their needs. Therefore, we have summarized 18 common callback functions according to Android Developer Documentation [30] and we used a set to record them. There are also some callback functions used infrequently and we have not summarized them. Users can add callback functions they want to consider to the set as needed, which is easy to implement.

VII. Related Work

We summarize previous works in Android asynchronous classes, and existing log analysis techniques.

Research on Android Asynchronous Class. *APEChecker* [4] proposes three rules for async programming to reduce errors caused by misuse of asynchronous components but it is not open source. Yu Lin et al. [7] reconstructs AsyncTask into another asynchronous class IntentService. Tang et al. [8] propose a method to generate test cases triggering concurrency errors and the same problem is detected by defining the happens-before relationship model [9]. Since Amandroid's analysis directly handles inter-component control and data flows, [10] can be used as a framework to address security problems that result from interactions among multiple components from either the same or different apps and Portend [11], a tool that not only detects data race but also automatically classifies them based on their potential consequences. Kang et al. [20] researched the impact of background thread scheduling on the response time of user events in Android asynchronous programming and Bouajjani et al. [21] used formal verification method to verify the robustness of concurrent operations (i.e., event conflicts and data competitions). The above works consider the performance and robustness of Android asynchronous programming respectively, while they all analyze the problem at thread-level and do not consider the problem introduced by characteristics of AsyncTask. Existing approaches do not pay more attention to the defects caused by the AsyncTask asynchronous components dynamically.

Log-based analysis. There have been a lot of studies [12, 13, 14, 15, 16, 17, 18, 19] on fault detection based on the correlation analysis between logs and the running status of thread or program or server node. Log-related works include log collection, log parsing, log grouping, log detection and etc. Many researchers process logs to extract various information, including event flow, timing information, etc. Xu et al. [23] mines console logs and applies machine learning techniques to detect anomaly executions. Iprof [24] extracts request ID and timing information from logs to profile request latency and [23] also detects large-scale system problems by mining console logs. These dynamic detection methods have not been used in the detection of AsyncTask related defects.

VIII. Conclusion

In this paper, we propose a dynamic detection method based on instrumentation and log analysis to check AsyncTask-related defects. We have collected a wealth of log printing information to distinguish asynchronous logs. And then, we check defects in logs according to match rules.

To evaluate AD2Checker, we use 19 real-world apps from F-Droid. AD2Checker successfully checks 145 defects. We compare AD2Checker with an existing tool AsyncChecker which is based on static analysis and the result shows that AD2Checker has higher precision and provides more detailed and readable reports without false positives.

Acknowledgment

This work is supported by the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (Grant No. QYZDJ-SSW-JSC036), and the National 973 Program (Grant No. 2014CB340701).

References

[1] Android developers. ui/application exerciser monkey. http://developer.Android.com/tools/help/monkey.html.

[2] F-Droid. 2019. https://f-droid.org.

[3] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The fan framework for Java program analysis: a retrospective. In Cetus Users and Compiler Infastructure Workshop (CETUS 2011), Vol. 15. 35.

[4] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently manifesting asynchronous programming errors in Android apps. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE'18, Montpellier, France, September 3-7. 486–497. https://doi.org/10.1145/3238147.3238170

[5] Linjie Pan, Baoquan Cui, Hao Liu, Jiwei Yan, Siqi Wang, Jun Yan and Jian Zhang. 2020. Static Asynchronous Component Misuse Detection for Android Applications. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE '20, Virtual Event, USA, November 8–13, 2020. 952-963. https://doi.org/10.1145/3368089.3409699

[6] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static analysis of android apps: A systematic literature review. Information & Software Technology 88 (2017), 67–95. https://doi.org/10.1016/j.infsof.2017.04.001

[7] Yu Lin, Semih Okur, and Danny Dig. 2015. Study and Refactoring of Android Asynchronous Programming (T). In

Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE'15, Lincoln, NE, USA, November 9-13. 224–235. https://doi.org/10.1109/ASE.2015.50

[8] Hongyin Tang, Guoquan Wu, Jun Wei, and Hua Zhong. 2016. Generating test cases to expose concurrency bugs in Android applications. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE'16, Singapore, September 3-7. 648–653. https://doi.org/10.1145/2970276. 2970320

[9] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for Android applications. In Proceedings of the 35th ACM-SIGPLAN Symposium on Programming Language Design and Implementation, PLDI'14, Edinburgh, United Kingdom, June 09 - 11. 316–325. https://doi.org/10.1145/2594291.2594311

[10] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 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), 14:1–14:32. https://doi.org/10.1145/3183575

[11] Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data races vs. data race bugs: telling the difference with portend. ACM SIGPLAN Notices 47, 4 (2012), 185–198.

[12] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving Software Diagnosability via Log Enhancement. ACM Trans. Comput. Syst. 30, 1 (2012), 4:1–4:28.

[13] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. 2021. Semi-supervised log-based anomaly detection via probabilistic label estimation. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 1448–1460.

[14] Anunay Amar and Peter C Rigby. 2019. Mining historical test logs to predict bugs and localize faults in the test logs. In Proceedings of the 41st International Conference on Software Engineering. IEEE Press, 140–151.

[15] Edward Chuah, Shyh-hao Kuo, Paul Hiew, William-Chandra Tjhi, Gary Lee, John Hammond, Marek T Michalewicz, Terence Hung, and James C Browne. 2010. Diagnosing the root-causes of failures from cluster log files. In 2010 International Conference on High Performance Computing. IEEE, 1–10.

[16] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2018. Identifying impactful service system problems via log analysis. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 60–70.

[17] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 557–566.

[18] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. 2016. An evaluation study on log parsing and its use in log mining. In Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on. IEEE, 654–661.

[19] Yu Kang, Yangfan Zhou, Hui Xu, and Michael R. Lyu. 2016. DiagDroid: Android Performance Diagnosis via Anatomizing Asynchronous Executions. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE'16, Seattle, WA, USA, November 13-18. 410–421.

[20] Bouajjani A , Emmi M , Enea C , et al. Verifying Robustness of Event-Driven Asynchronous Programs Against Concurrency[C]// European Symposium on Programming. 2017.

[21] Linjie Pan, Baoquan Cui, Jiwei Yan, Xutong Ma, Jun Yan, and Jian Zhang. 2019. Androlic: an extensible flow, context, object, field, and path-sensitive static analysis framework for Android. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019. 394–397.

[22] AsyncTask. 2019. https://developer.android.com/reference/android/os/AsyncTask.

[23] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 117–132.

[24] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In OSDI, Vol. 14. 629–644.

[25] I. Zhukov, C. Feld, M. Geimer, M. Knobloch, B. Mohr, P. Saviankou, Scalasca v2: Back to the Future, in: Proc. of Tools for High Performance Computing 2014, Springer, 2015, pp. 1– 24.

[26] DCov - A Test Coverage Program, The GNU Project, 2020, https://gcc.gnu.org/ onlinedocs/gcc/Gcov.html.

[27] Singh L , Hofmann M . Dynamic behavior analysis of android applications for malware detection[C]// 2017 International Conference on Intelligent Communication and Computational Techniques (ICCT). IEEE Computer Society, 2017.

[28] Junying Huang, Jing Ye, Xiaochun Ye, Da Wang, Dongrui Fan, Huawei Li, Xiaowei Li, Zhimin Zhang:Instruction Vulnerability Test and Code Optimization Against DVFS Attack. ITC-Asia 2019: 49-54.

[29] Li Li, Alexandre Bartel, Tegawend´e F Bissyand´e, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In ICSE, 2015.

[30] Android documentation. 2019. https://developer.android.google.cn/docs.

[31] IntentService. 2019. https://developer.android.com/reference/android/app/IntentService.