Static Asynchronous Component Misuse Detection for Android Applications

Linjie Pan
State Key Lab. of Computer Science
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China
panlj@ios.ac.cn

Baoquan Cui
State Key Lab. of Computer Science
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China
baoquan@iscas.ac.cn

Hao Liu
Beijing University of Technology
Beijing, China
lansedeyu94@live.com

Jiwei Yan
Tech. Center of Softw. Eng
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China
yanjw@ios.ac.cn

Siqi Wang
State Key Lab. of Computer Science
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China
wangsiqi@emails.bjut.edu.cn

Jun Yan*
State Key Lab. of Computer Science
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China
yanjun@ios.ac.cn

Jian Zhang*
State Key Lab. of Computer Science
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China
zj@ios.ac.cn

ABSTRACT

Facing the limited resource of smartphones, asynchronous programming significantly improves the performance of Android applications. Android provides several packaged components to ease the development of asynchronous programming. Among them, the AsyncTask component is widely used by developers since it is easy to implement. However, the abuse of AsyncTask component can decrease responsiveness and even lead to crashes. By investigating the Android Developer Documentation and technical forums, we summarize five misuse patterns about AsyncTask. To detect them, we propose a flow, context, object and field-sensitive interprocedural static analysis approach. Specifically, the static analysis includes typestate analysis, reference analysis and loop analysis. Based on the AsyncTask-related information obtained during static analysis, we check the misuse according to predefined detection rules. The proposed approach is implemented into a tool called AsyncChecker.

We evaluate AsyncChecker on a self-designed benchmark suite called AsyncBench and 1,759 real-world apps. AsyncChecker finds 17,946 misused AsyncTask instances in 1,417 real-world apps (80.6%). The precision, recall and F-measure of AsyncChecker on real-world applications are 97.2%, 89.8% and 0.93, respectively. Compared with existing tools, AsyncChecker can detect more asynchronous problems. We report the misuse problems to developers via GitHub. Several developers have confirmed and fixed the problems found by AsyncChecker. The result implies that our approach is effective and developers do take the misuse of AsyncTask as a serious problem.

*Corresponding Author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409699

CCS CONCEPTS

• **Theory of computation** → **Program analysis**.

KEYWORDS

Android, AsyncTask, Asynchronous Programming, Static Analysis, Misuse Detection

ACM Reference Format:
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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409699

1 INTRODUCTION

Android has long occupied the dominant position in the mobile operating system market. As of June 2019, there are more than 2.7 million available apps in the Google Play Store [54]. In recent years, computation power and memory size of smartphones are

growing rapidly. However, compared with desktop computers, resources of smartphones are still limited, which always decreases the responsiveness of applications and even leads to bugs [15, 36].

Under limited resources, asynchronous programming is a significant technique to keep application responsive. In asynchronous programming, the main thread only accounts for the UI update while time-consuming and CPU (IO)-blocking tasks are encapsulated into background threads. In order to simplify asynchronous programming, Android provides several packaged asynchronous components such as AsyncTask and IntentService. Among them, AsyncTask is easy to implement and thus widely used by Android developers [34].

In fact, AsyncTask is merely suitable for short operations [7]. The improper use of AsyncTask can lead to many problems that can decrease the performance and even lead to the crash of applications. However, many developers still choose AsyncTask as their first choice because of its convenience. To solve this problem, Lin et al. [34] proposed a methodology to refactor AsyncTask into IntentService [26] which evades the problems caused by AsyncTask, yet IntentService cannot totally replace AsyncTask in all cases and developers still prefer to use AsyncTask. Fan et al. [15] proposed three rules for async programming and detected async errors based on these rules. In fact, these common rules are not precise enough to instruct the developers to use specific async components such as AsyncTask. From these previous works, we can find that the misuse of AsyncTask widely exists in real-world applications. Moreover, misuses of AsyncTask can lead to the crash of apps. Currently, the research community pays little attention to AsyncTask and there is a lack of detection on the misuse of AsyncTask. Therefore, it is necessary to carry out a thorough analysis of AsyncTask.

In this paper, we propose and detect five kinds of misuse patterns of AsyncTask, i.e., StrongReference, NotCancel, NotTerminate, EarlyCancel and RepeatStart. These patterns are summarized from the Android documentation [11] and technical forums such as StackOverflow [42]. These misuse patterns can lead to problems such as memory leak, result loss, energy waste and wrong semantics. Some of them can even cause the crash of applications. These patterns are related to code segments defined in both AsyncTask and Activity, which makes the detection more difficult. In accordance with the feature of AsyncTask, we propose a static analysis approach to detect the misuse of AsyncTask. Considering the characteristics of AsyncTask, we construct the AsyncTask State Transition (ATST) model to depict how the state of an AsyncTask object (an object whose type is AsyncTask or subclass of AsyncTask) transits in accordance with operations defined in AsyncTask. Based on the ATST model, we carry out typestate analysis to detect NotCancel, EarlyCancel and RepeatStart. Besides, we define possess operation to perform reference analysis so that we can identify through which fields an AsyncTask object holds strong reference to Activity. Last but not least, we also perform loop analysis to detect NotTerminate. In other words, we identify loop segments in `doInBackground()` and determine whether developers check the status of AsyncTask to jump out of the loop.

To summarize, the contributions of this paper are as follows.
1. Problem Description: This paper systematically depicts the misuse problems of AsyncTask and proposes five misuse patterns;
2. Methodology: We provide a flow, context and field-sensitive inter-procedural static analysis methodology to detect the misuse patterns of AsyncTask;
3. Tool: We have developed AsyncChecker, a static analysis tool that specializes in detecting misuses of AsyncTask;
4. Evaluation: This paper designs AsyncBench, an open micro benchmark containing 69 apps as a test suite for AsyncTask misuse detection. We have evaluated AsyncChecker on both AsyncBench and 1,759 real-world apps. The result shows that the misuses of AsyncTask widely exist (up to 80.6% of the 1,759 apps).

The tool and benchmarks can be viewed at https://github.com/pangeneral/AsyncChecker.

2 BACKGROUND

In this section, we first perform an empirical study to show how async components are used in real-world apps. Then we present a brief introduction to AsyncTask, a widely used async component in Android. At last, we show the problem caused by misuse of AsyncTask via a motivating example.

2.1 Async Components in Real-World Apps

In order to understand the usage of async components in real-world applications, we perform an empirical study. In particular, the study tries to answer the following empirical questions:

- **EQ1:** What are usage frequencies of different async components in Android applications?
- **EQ2:** What are the differences in usage of async components among different repositories?

Repository. In order to answer these questions, we built three repositories including: 1,184 apps from F-Droid [14], an open-source app market, 6,808 apps from Google Play Store [19] and 5,669 apps from Wandoujia [60], a Chinese commercial app market. For F-Droid, we collect all available apps. For Google Play Store and Wandoujia, the number of applications is so huge that it is difficult to download all of them. Therefore, we downloaded 500 apps under each category of the market. Apps that cannot be decompiled are excluded from repositories.

Methodology. We pay attention to six common async components, i.e., AsyncTask, IntentService, HandlerThread, AsyncTaskLoader, ThreadPoolExecutor and Thread. Considering the features of async components, our empirical study is based on static analysis, which is carried out on Soot framework and Jimple intermediate representation.

On one hand, some of the six kinds of async components mentioned above are abstract classes, which cannot be instantiated directly. Technically, developers can only instantiate subclasses of an async class. It is quite reasonable to carry out static analysis so that we can obtain the class inheritance relationships accurately.

On the other hand, through Soot and Jimple, we can analyze apps under an exclusive intermediate representation that simplifies the disposal to some language features such as inner class and anonymous class. Besides, according to Li et al. [33], Soot and Jimple are the most adopted basic support tool and format for static analysis of Android apps respectively.

Results. Table 1 shows the information about six async components in three repositories. The column *Class* denotes the number of the corresponding async classes, the column *App* denotes the

number of apps that contain the async classes and the column *Perc* denotes the percentage of apps that contain the async classes.

From the empirical results, we can obtain some useful findings. First, AsyncTask is the most popular async component in all of the three repositories. Thread is nearly as popular as AsyncTask. Yet, Thread is not an Android specific async component. Considering the popularity of AsyncTask, it is necessary to conduct further study on it.

Second, the usage of six async components in F-Droid is much lower than GooglePlay and Wandoujia. From the results, we can see that the difference between open-source and commercial apps is obvious.

2.2 AsyncTask in Android

When an application is launched, Android activates a thread, which is called main thread or UI thread, to run the application [46]. If the main thread executes CPU-blocking or IO-blocking tasks, then the application will be blocked, which may lead to a problem known as Application Not Responding (ANR). To avoid unresponsiveness, time-consuming tasks need to be executed in background threads. In order to ease asynchronous programming, Android provides AsyncTask, an encapsulation of the concurrency framework. With the help of AsyncTask, developers can perform background operations and publish results on the UI thread without direct threads manipulation [7].

Technically, AsyncTask encapsulates the functionality of background operation and interaction with UI thread into five callback methods, i.e., `onPreExecute()`, `doInBackground()`, `onProgressUpdate()`, `onPostExecute()` and `onCancelled()`. Among these methods, only `doInBackground()` runs in the background thread. It encapsulates tasks that need to be executed in the background thread. The rest run in the UI thread and take charge of communication between background thread and UI thread.

Since AsyncTask is designed as an abstract class, developers cannot instantiate it directly. Instead, developers can only instantitate a non-abstract subclass of AsyncTask. In this paper, we use the terminology AsyncTask class to denote a non-abstract subclass of AsyncTask. We use AsyncTask object to denote the object whose type is a subclass of AsyncTask class.

Figure 1 shows the execution sequence of methods defined by AsyncTask. To start a background thread, developers need to create an instance of AsyncTask class and then invoke its `execute()` or `executeOnExecutor()` method to start a background thread. After the AsyncTask is started, UI thread first executes `onPreExecute()`, then `doInBackground()` is executed. While the background thread is running, it interacts with UI thread through `publishProgress()` and `onProgressUpdate()`. After `doInBackground()` finishes, `onPostExecute()` is executed to update UI. Note that developers can invoke `cancel()` method to cancel an AsyncTask. If `cancel()` method is invoked before `doInBackground()` finishes, then `onCancelled()` instead of `onPostExecute()` will be executed after `doInBackground()` finishes. AsyncTask provides `isCancelled()` method so that developers can check whether `cancel()` is invoked.

Figure 1: Sequence Diagram of AsyncTask

2.3 Motivating Example

In this section we show the typical problem caused by misuse of AsyncTask via a motivating example. Listing 1 defines two Activities: Main (line 1-10) and Dialog (line 11-25). We can switch from Main to Dialog via switchButton (line 3-8) defined in Main Activity. In Dialog Activity, we define a large array (line 14) and we can start an AsyncTask (line 26-39) via clicking the button called asyncButton (line 18-23). In `doInBackground()` method (line 31-38), we simulate a long-running task via thread sleeping operation.

Listing 1: Problems Caused by Misuse of AsyncTask

```
1  class Main extends Activity {
2     protected void onCreate(Bundle savedInstanceState) {
3        switchButton.setOnClickListener(new OnClickListener() {
4           public void onClick(View v) {
5              Intent newIntent = new Intent(...);
6              startActivity(newIntent);
7           }
8        });
9     } ...
10 }
11 class Dialog extends Activity {
12    private Tasker currentTask;
13    private TextView tv;
14    private int[] byteArray = new int[15000000];//large array
15    protected void onCreate(Bundle bundle) {
16       tv = (TextView) findViewById(R.id.tv);
17       asyncButton = (Button) findViewById(R.id.asyncButton);
18       asyncButton.setOnClickListener(new OnClickListener() {
19          public void onClick(View v) {
20             currentTask = new Tasker(tv); // strong reference
21             currentTask.execute("");
22          }
23       });
24    } ...
25 }
26 class Tasker extends AsyncTask<String, String, String> {
27    private TextView referenceTv;
28    public Tasker(TextView showTv) {
29       referenceTv = showTv;
30    }
31    protected String doInBackground(String... params) {
32       Thread.sleep(10000);
33       return null;
34    } ...
35 }
```

Table 1: The usage frequency of async components in three different repositories

Async Components	F-Droid			GooglePlay			Wandoujia		
	Class	App	Perc(%)	Class	App	Perc(%)	Class	App	Perc(%)
AsyncTask	2,482	421	35.6	117,295	5,660	83.1	88,010	3,779	66.7
IntentService	190	118	10.0	12,501	3,535	51.9	8,902	2,299	40.6
HandlerThread	63	60	5.1	4,096	2,420	35.5	2,906	1,487	26.2
AsyncTaskLoader	83	23	1.9	797	173	2.5	586	127	2.2
ThreadPoolExecutor	125	111	9.3	8,526	3,714	54.6	3,548	1,966	34.7
Thread	1,299	367	31.0	58,426	5,524	81.1	118,084	3,413	60.2

We can trigger an out of memory (OOM) error through following manipulations: first, click switchButton to switch from Main to Dialog; second, click asyncButton to start an AsyncTask; then, press back key to switch into Main; at last, click switchButton and the app will crash.

The crash occurs because the memory of Dialog cannot be garbage collected as AsyncTask holds strong reference to Dialog (line 20 and line 29). When the user clicks switchButton again, allocating memory for a new large array (line 14) triggers the OOM error. In other words, the misuse of AsyncTask leads to memory leak which may crash applications.

3 MISUSE PATTERNS OF ASYNCTASK

In this section, we introduce five misuse patterns of AsyncTask. These patterns are related to both functional and performance problems.

3.1 StrongReference

If an instance of AsyncTask class holds strong reference to GUI elements of Activity when the instance is started, then the memory of Activity cannot be garbage collected while the instance is running, which leads to memory leak and may cause application crash directly. Here, the GUI element denotes the instance of GUI classes, i.e., the subclasses of `View`. Technically, AsyncTask holds the reference of Activity via its fields. In the following code segment, we show an example of StrongReference pattern:

```
1  public class MainActivity extends Activity {
2     private TextView view1;
3     private AsyncTask task;
4     protected void onCreate(Bundle bundle) {
5        super.onCreate(bundle);
6        view1 = (TextView) findViewById(R.id.view1);
7        task = new Tasker(view1); // holding strong reference
8        task.execute();
9     }
10    protected void onDestroy() {
11       super.onDestroy();
12       task.cancel();
13    }
14 }
15 class Tasker extends AsyncTask {
16    private TextView view2;
17    public Tasker(TextView view1) {
18       view2 = view1;
19    }
20    ...
21 }
```

The instance of AsyncTask class `Tasker` holds strong reference of `MainActivity` via a GUI element `TextView` (line 7 and line 18) when it is started (line 8). Note that if an AsyncTask class is a non-static inner class belonging to Activity, then it holds the reference of Activity by default. According to the grammar of Java, an anonymous inner class must be a non-static class. In other words, developers should not define AsyncTask class as the anonymous inner class of Activity. In order to achieve balance between memory leak and UI update, AsyncTask can hold weak reference of Activity, which will not cause memory leakage. The detailed information about weak reference can be viewed at [61].

3.2 NotCancel

The invocation of the `onDestroy()` method leads to the destruction of Activity's GUI. If `cancel()` method is not invoked before the destruction of Activity, then `onPostExecute()` will be executed. If `onPosteExecute()` method contains UI update operation and the GUI does not exist any more, the invocation of `onPostExecute()` will be meaningless and even crash the applications. The following code segment shows an example of NotCancel which could lead to crash if dialog is dismissed (line 15) while the Activity has been destroyed.

```
1  public class MainActivity extends Activity {
2     private ProgressDialog searchDialog;
3     private AsyncTask task;
4     protected void onCreate(Bundle bundle) {
5        ...
6        task = new Tasker(searchDialog);
7        task.execute(); // not cancel
8     }
9     private static class Tasker extends AsyncTask {
10       private WeakReference<ProgressDialog> dialog;
11       public Tasker(ProgressDialog theDialog) {
12          dialog = theDialog;
13       }
14       protected void onPostExecute() {
15          dialog.dismiss(); // potential crash
16       }
17       ...
18    }
19 }
```

To avoid the potential error caused by improper invocation of `onPostExecute()`, developers should invoke `cancel()` method of AsyncTask before the Activity is destroyed such that `onCancelled()` instead of `onPostExecute()` is invoked after `doInBackground()` finishes. To do so, we recommend developers assign the instance of AsyncTask classes to the field of Activity so that they can operate AsyncTask instances in any callback method of Activity including `onDestroy()`. Otherwise, we cannot operate AsyncTask objects which are only assigned to local variables across the different lifecycle methods of Activity.

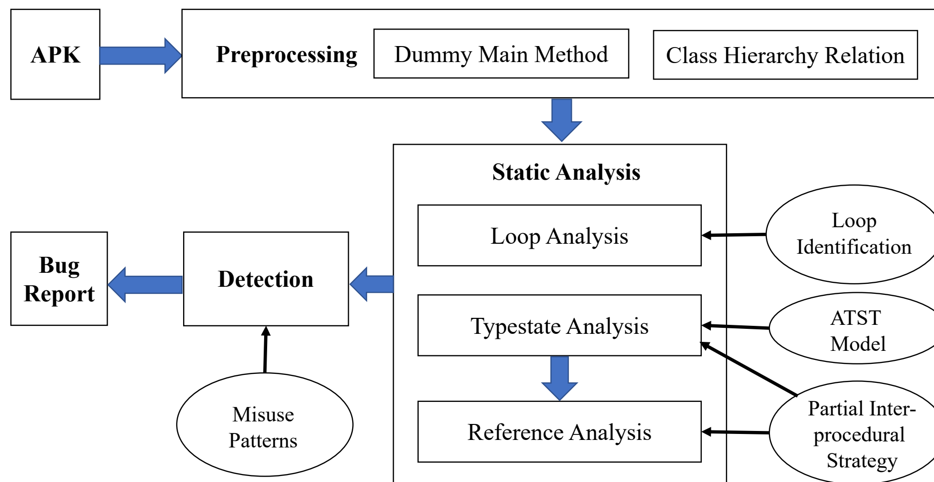

Figure 2: Overview of Misuse Detection for AsyncTask

3.3 NotTerminate

The invocation of `cancel()` method can avoid improper interaction between background thread (AsyncTask) and main thread (GUI of Activity), yet it cannot terminate `doInBackground()`. In fact, it only changes the status of AsyncTask object. Apparently, it is a waste of energy to continue running `doInBackground()` after `cancel()` is invoked. In order to save energy, developers should terminate the background thread represented by AsyncTask as soon as possible if it is cancelled. More specifically, developers should check the return value of `isCancelled()` periodically (within a loop) in `doInBackground()` and terminate AsyncTask if `isCancelled()` returns true [7].

```
1  class Tasker extends AsyncTask {
2     protected String doInBackground(String[] arg) {
3        while ( ... ) {
4           // loop should terminate if cancel() is invoked
5        }
6        return "message";
7     }
8  }
```

3.4 EarlyCancel

Developers can cancel a running AsyncTask by invoking `cancel()` method. However, if an AsyncTask instance is cancelled before it is started, then `onPostExecute()` will never be executed, which is incorrect semantically. In other words, `cancel()` method should be invoked after AsyncTask is started. We show an example of EarlyCancel in the following code segment:

```
1  protected void onCreate(Bundle bundle) {
2     ...
3     task = new Tasker();
4     task.cancel(true); // task is cancelled before it is
         started
5     task.execute();
6     ...
7  }
```

3.5 RepeatStart

According to Android Developer Documentation [7], an instance of AsyncTask can only invoke `execute()` or `executeOnExecutor()` once. If a second execution is attempted then an exception will be thrown, which means developers should create a new instance of AsyncTask each time they want to start a background thread.

```
1  class MainActivity extends Activity {
2     private Tasker task;
3     protected void onCreate(Bundle bundle) {
4        task = new Tasker();
5        ...
6     }
7     protected void onStart() {
8        task.execute(); // task will execute twice if user
            navigates back
9        ...
10    }
11 }
```

The above code segment contains a potential RepeatStart error. According to the lifecycle of Activity, `onStart()` will be invoked again when users navigate back to the Activity, which means the instance of AsyncTask pointed to by the variable `task` will be executed twice.

4 STATIC MISUSE DETECTION

Figure 2 shows an overview of our approach to statically analyze AsyncTask-related misuse patterns. There are three steps in our approach:

- **Preprocessing.** The input is an apk file. During preprocessing, we first obtain its class hierarchy relation through APIs provided by Soot [30]. Then we unify lifecycle callback methods and user-defined callback methods into a dummy main method for each Activity just as FlowDroid [6] does.
- **Static Analysis.** After preprocessing, we perform flow, context, object and field-sensitive inter-procedural static analysis to collect typestate, reference and loop information about AsyncTask.

- **Detection.** At last, we detect whether usage of AsyncTask is correct via predefined detection rules based on the collected information during static analysis. Any code segments containing AsyncTask misuse will be recorded in bug reports.

Partial Inter-procedural Strategy. The typestate analysis and reference analysis take the dummy main method of Activity as the entry method. Apparently, most of the operations in an app are AsyncTask-irrelevant. In order to improve efficiency, we need to eliminate irrelevant operations and concentrate on AsyncTask-related ones. In view of this, we only perform inter-procedural analysis for methods that contain AsyncTask-related statements, i.e., statements that contain AsyncTask variables. Due to the feature of polymorphism, it is difficult to determine which method is invoked at the non-static call site. In order to keep precision, we first generate a list of possible invoked methods based on class hierarchy relation. Then we traverse the statements of these possible invoked methods one by one. If any of the methods in the list contains AsyncTask-related statements, then we take the call site as AsyncTask-related one. In other words, we perform inter-procedural analysis at any call site that may be AsyncTask-related. For call sites that must not be AsyncTask-related, we take the invoked method as a library method and do not unfold it.

4.1 Typestate Analysis

A typestate denotes the state an object can occupy during execution [17]. As mentioned in section 2, the operations on an AsyncTask object such as `execute()` and `cancel()` can change its state. Therefore, we define the AsyncTask State Transition (ATST) model for AsyncTask objects, based on which those typestate related misuse patterns of AsyncTask can be easily detected.

*Definition 4.1 (**ATST Model**).* The ATST model is a 4-tuple $\mathcal{M} = \langle Q, \Sigma, \delta, s_0 \rangle$, where

- Q={*Initial*, *Running*, *Canceled*, *Wrong*} is the set of states of AsyncTask object. Among the four states, *Wrong* is abnormal. If an AsyncTask object is in *Wrong* state, then an error occurs.
- Σ={*S*, *C*} is the set of operations, where *S* denotes starting an AsyncTask via invoking `execute()` or `executeOnExecutor()` and *C* represents canceling an AsyncTask via invoking `cancel()`.
- $\delta : Q \times \Sigma \rightarrow Q$ is the state-transition function, which is shown in figure 3.
- $s_0 = Initial \in Q$ is the initial state of AsyncTask object.

During typestate analysis, we keep the state of each AsyncTask object. Whenever a start operation or cancel operation is detected, we first determine which AsyncTask object is under manipulation. The object sensitivity is achieved through a store-based heap model. Specifically, we maintain a map from reference variables to AsyncTask allocation sites when an AsyncTask object is instantiated via NewExpr. Then, we transfer the typestate of the AsyncTask object according to the ATST model. Through typestate analysis, we can directly check some state-related misuse patterns. Besides, it is also the foundation for reference analysis.

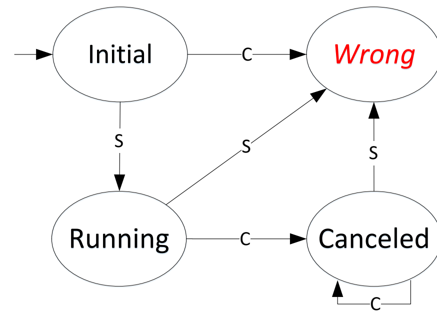

Figure 3: AsyncTask State Transition Function

4.2 Reference Analysis

In order to check whether an AsyncTask object holds strong references to the GUI elements of Activity, we perform reference analysis to record the mapping relation from the fields of AsyncTask to the GUI objects they refer. The map is denoted by *refer_map(ao)* where *ao* is an AsyncTask object. For each instantiated AsyncTask object, we maintain such a map independently. In fact, the map is built through Possess operation:

*Definition 4.2 (**Possess**).* An AssignStmt $s = \langle op_l, op_r \rangle$ is a Possess operation if:

- the left operand op_l is a field of AsyncTask object; and
- the right operand op_r is or points to an object whose type is the subclass of View or Activity.

View is the basic building block for user interface components [59] and we take the objects whose type is the subclass of View as the GUI elements of Activity. During reference analysis, we traverse along the CFG of the dummy main method and update the mapping relations when a Possess operation is detected. When an AsyncTask object is started, we check its reference map to judge whether StrongReference occurs. Listing 2 shows an example about reference analysis.

Listing 2: An example of Reference Analysis

```
1  class MainActivity extends Activity {
2     protected void onCreate(Bundle bundle) {
3        TextView view1 = (TextView) findViewById(R.id.view1);
4        TextView view2 = (TextView) findViewById(R.id.view2);
5        String mark = "tasker";
6        Tasker task = new Tasker(view1, mark);
7        task.setView2(tv2);
8        task.execute("begin"); // AsyncTask is started
9     }
10 }
11 class Tasker extends AsyncTask<String,String,String> {
12    private TextView v1;
13    private WeakReference<TextView> v2;
14    private String s1;
15    public Tasker(TextView tv1, String mark) {
16       this.v1 = tv1; // strong reference to GUI object
17       this.s1 = mark; // strong reference to non-GUI object
18    }
19    public void setView2(TextView tv2) {
20       this.v2 = new WeakReference<TextView>(tv2); // weak
             reference to GUI object
21    }
22 }
```

We define an AsyncTask class called Task which has three fields whose types are TextView, WeakReference and String respectively. In the *onCreate()* method of Activity, we create an AsyncTask object (line 6) and start it to begin a background thread (line 8). Apparently, there exists a StrongReference misuse since the AsyncTask object holds strong reference to GUI elements of Activity via its field v1. Note that the fields v2 and s1 do not lead to StrongReference misuse since they do not satisfy the condition of StrongReference. v2 holds weak reference to GUI object (view2) and s1 holds strong reference to non-GUI object (mark). Therefore, the reference map for the AsyncTask object is *ref_map(task)* = {v1=view1}.

4.3 Loop Analysis

In order to check NotTerminate misuse, we need to identify loop structure and the invocation of `isCancelled()` in `doInBackground()`. The original `doInBackground()` defined in AsyncTask is an abstract method. What we need to analyze are redefined `doInBackground()` methods in AsyncTask classes. Based on the class hierarchy relation of the input apk, we can easily find all AsyncTask classes and `doInBackground()` methods. Then we implement algorithms introduced in [63] to identify loop structures in approximately linear time. For each identified loop, we detect whether `isCancelled()` is invoked to jump out of the loop. If there exists any loop that does not invoke `isCancelled()`, then the NotTerminate error occurs. For each `doInBackground()` method, we use a set to save loops that do not contain terminate operation. When an AsyncTask object is started, we check the set to judge whether NotTerminate occurs.

Different from typestate analysis and reference analysis, we perform loop analysis in bottom-up manner via topological sort. We calculate the loop sets for callees and merge them to callers until we reach `doInBackground()`, the top level caller. To do so, we first build acyclic call graph in which the root node represents the `doInBackground()` method.

List 3 shows an example of loop analysis. There are two loops within the `doInBackground()` method. In the first loop, there is a terminate operation (line 4), which is correct. The second loop lacks termination operation (line 10-12) and thus leads to NotTerminate misuse. We first calculate the loop set for `theLoop()` which is *loop_set(theLoop)*: {secondLoop}. Then we merge it to `doInBackground()`. The loop set for the entry method is *loop_set(entry)*: {secondLoop}. In practice, a loop is denoted by the combination of header statement and all statements in the loop.

Listing 3: An Example of Loop Analysis

```
1  class Task extends AsyncTask<String,String,String> {
2     protected String doInBackground(String[] args) {
3        for (int i = 0; i < 100; i++) { // First loop
4           if (isCancelled()) break; // Terminate operation
5           ...
6        }
7        theLoop();
8     }
9     private void theLoop() {
10       for (int j = 0; j < 100; j++) { // Second loop
11          ... // Lack Terminate operation
12       }
13    }
14    ...
15 }
```

4.4 Detection Rules

Table 2 shows the detection rules for misuse patterns of AsyncTask. In the table, *ao* is the AsyncTask object under analysis. *s* denotes current state of *ao*. The operation defined in ATST model is represented by *o*. *refer_map(ao)* denotes the mapping relation from AsyncTask fields to *ao*, which is constructed during reference analysis. *loop_set(ao)* denotes the set of invalid loops for `doInBackground()` defined in the AsyncTask class which is the type of *ao*. Similar to *refer_map*, *loop_set* is constructed during loop analysis.

Table 2: Detection Rules of Misuse Patterns

Misuse Pattern	Detection Rule
StrongReference	s=*Initial* $\wedge$ o=*Start* $\wedge$ *refer_map(ao)* $\neq \emptyset$
NotTerminate	s=*Initial* $\wedge$ o=*Start* $\wedge$ *loop_set(ao)* $\neq \emptyset$
EarlyCancel	s=*Intitial* $\wedge$ o = *Cancel*
RepeatStart	$s \in$ {*Canceled, Running*} $\wedge$ o = *Start*
NotCancel	s=*Running* $\wedge$ $o = \emptyset$

The AsyncTask object *ao* is in the *Initial* state when it is instantiated. The state of AsyncTask object under analysis transits in accordance with the operation defined in the ATST model. If current state is *Initial* and the operation is *Start*, then a background thread is started. We need to check whether *StrongReference* and *NotTerminate* occur. First, if *refer_map(ao)* is not empty, then *StrongReference* occurs. Second, if *loop_set(ao)* is not empty, then *NotTerminate* is detected. Third, if *ao* is in *Initial* state and *o* is *Cancel*, then *EarlyCancel* is recorded. Fourth, RepeatStart occurs if *ao* is in *Running* or *Cancenled* state and *o* is *Start*. Finally, if *s* is in *Running* state after Activity is destroyed, then *NotCancel* occurs.

5 IMPLEMENTATION

We have implemented the approach of misuse detection on AsyncTask into a tool named AsyncChecker. It is written in Java based on Androlic [44], a flow, context, object, field-sensitive static analysis framework. As figure 2 shows, AsyncChecker generates the class hierarchy relation and dummy main method for subsequent analysis. We can analyze the semantics of statements via the core engine of Androlic. In accordance with the misuse detection of AsyncTask, we implement some APIs of Androlic.

At first, we implement partial inter-procedural strategy via the interface `IMethodInterProceduralJudge` to check whether a call site is AsyncTask-related. Then, we define `AsyncTaskRefObject` and `AsyncTypeState` to represent AsyncTask object and its typestate respectively. The class `AsyncTaskRefObject` is a subclass of `NewRefHeapObject`, which denotes the object instantiated via explicit allocation site in Androlic. Through self-defined AsyncTask object, we can easily perform reference analysis to maintain the *ref_map*. In order to perform typestate analysis, we need to identify operations defined in the ATST model. We implement the interface `ILibraryInvocationProcessor` to process the library method invocation of AsyncTask, i.e., identify AsyncTask-related operations. Besides, we implement `ISymbolicEngineInstrumenter` to check NotCancel when a path ends. During loop analysis, we define a summary to save the *loop_set* for methods invoked in `doInBackground()`. At last, we apply the detection rules on collected AsyncTask information and record any misuse in the detection reports.

6 EVALUATION

In this section, we firstly apply AsyncChecker on both self-designed benchmark suites and real-world applications. Then we compare AsyncChecker with existing tools. The evaluation tries to answer the following four questions:

- **RQ1:** How effective is AsyncChecker on self-designed benchmark?
- **RQ2:** Can AsyncChecker find the misuse of AsyncTask in real-world applications?
- **RQ3:** How effective is AsyncChecker against existing tools?
- **RQ4:** Do developers take the misuse of AsyncTask as a serious problem?

6.1 RQ1: AsyncChecker on Benchmark

AsyncBench. There are many benchmark suites for Android static analysis such as DroidBench [12] and ICC-Bench [25]. However, there is no benchmark suite for AsyncTask-related misuse. For a better study of this problem, we have designed AsyncBench, an AsyncTask-specific benchmark suite. Currently, AsyncBench contains 69 manually written Android apps. These apps are divided into six groups in which five groups represent the five types of misuse patterns of AsyncTask and one corresponds to the combination of different types of misuse patterns. We have taken various situations of misuse of AsyncTask into consideration and developed corresponding apps. For instance, the AsyncTask operations across different lifecycle callback methods of Activity and user-defined callback methods such as `onClick()` can greatly affect the detection results. We designed several cases to cover these callback methods. For loop analysis, we consider different loop structures in Java and design corresponding test suites. Note that we do not drop out of occasions that currently cannot be processed by AsyncChecker in order to obtain an objective result and make a future extension on AsyncChecker. We hope AsyncBench can facilitate researchers who are interested in misuse detection of async components.

We conducted experiments on AsyncBench in windows 7 with 8GB RAM. Table 3 shows the experimental result. As we can see, there are several false negatives on AsyncBench. The false negative is caused by two reasons, i.e., strong reference via containers such as Map and List, and AsyncTask operation across different callback methods. Since the benchmark is constructed manually, the experimental result only provides basic information for the detection of AsyncTask misuse. The experimental result on AsyncBench is taken as a baseline. We take various scenarios about the usage of AsyncTask into consideration. As an open source benchmark, AsyncBench can facilitate researchers who are interested in misuse detection of async components.

Table 3: Experimental Results on AsyncBench

Benchmark Group	TP	TN	FP	FN
StrongReference	2	3	0	2
NotCancel	8	4	0	0
NotTerminate	8	7	0	0
EarlyCancel	6	6	0	1
RepeatStart	8	0	0	2
Combination	13	3	0	4
Sum	45	23	0	9

6.2 RQ2: AsyncChecker on Real-World Apps

In order to evaluate AsyncChecker in wild, we collected real-world apps from F-Droid, Google Play Store and Wandoujia App Store. As mentioned in Section 3, it is difficult for a local AsyncTask variable to avoid NotCancel as it cannot be manipulated across lifecycle callback methods in Activity. That is to say, an AsyncTask variable should be defined as a field rather than a local variable. Therefore, we only kept apps that have at least one AsyncTask field defined in Activity. After filtering, we obtained 1,759 apps, among which 71 from F-Droid, 1,109 from Google Play and 579 from Wandoujia. The experiments on 1,759 real-world apps have been performed in Docker Ubuntu 16.04.3 with 26 Intel Xeon E5-2680 processors (2.40GHz) and 50 GB RAM.

Results. The result about misuse of AsyncTask on three real-world repositories are recorded in Table 4 and 5. Table 4 shows the number of misused AsyncTask instances and the number of apps that contains misused AsyncTask instances. The column *Correct* and *Misused* under the column *Inst* denote how many analyzed AsyncTask classes are correctly used and misused respectively. Similarly, the last two columns show the number of apps that use AsyncTask correctly or incorrectly. As we can see, 6,190 AsyncTask instances are correctly used and 17,946 are misused. For the apps under analysis, up to 1,417 (80.6%) apps contain misused AsyncTask instance. In a word, the misuse of AsyncTask is a widely existing problem among real-world applications.

Table 4: Number of Misuse in AsyncTask Instance and Real-world Applications

Respository	Inst		App	
	Correct	Misused	Correct	Misused
F-Droid	50	416	13	58
GooglePlay	4,033	13,098	178	931
Wandoujia	2,107	4,432	151	428
Sum	6,190	17,946	342	1,417

Table 5 shows the distribution of different kinds of misuse patterns of AsyncTask. The column *Inst* denotes the number of AsyncTask instances which have certain kinds of misuse pattern. The column *App* denotes the number of applications that contain certain kinds of misuse pattern. From the result we can conclude that StrongReference, NotCancel and NotTerminate are much more common than the rest two misuse patterns. The reason is straightforward. On one hand, different from other misuse patterns, RepeatStart can definitely trigger the crash of applications. Developers can easily find the problem during testing and fix it before publishing applications. On the other hand, the wide existence of NotCancel proves that many developers do not know the AsyncTask can be cancelled, let alone EarlyCancel. Besides, EarlyCancel does not confront the normal semantics of AsyncTask. The experimental results show that most developers avoid making such mistake.

Validation. In order to evaluate the false positive and false negative rate of AsyncChecker, we need to manually check the source code of app under analysis. Since it is difficult to obtain the source code of commercial applications, we selected 22 apps from F-Droid which have been modified at least once since June 2019. We downloaded the source code of newest versions from github and successfully packaged them into apk files. Then we

Table 5: Distribution of Misuse Patterns of AsyncTask on Real-world Applications

Misuse Pattern	F-Droid		GooglePlay		Wandoujia		Sum	
	Inst	App	Inst	App	Inst	App	Inst	App
StrongReference	261	50	4,901	742	1,613	300	6,775	1,092
NotCancel	354	55	9,108	842	3,540	394	13,302	1,291
NotTerminate	304	58	10,916	881	3,496	391	14,716	1,330
EarlyCancel	2	2	23	16	11	8	36	26
RepeatStart	3	3	24	6	2	2	36	11

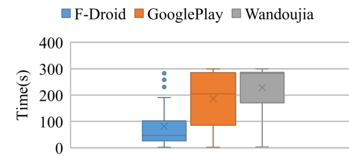

Figure 4: Running Time of AsyncChecker

analyzed these apk files with AsyncChecker. At last, we manually read the source code containing AsyncTask and bug reports of AsyncChecker to determine whether false positive or false negative occurs. The manual package and inspection took three postgraduate students about two weeks.

AsyncChecker reported asynchronous problems in 17 apps. Table 6 shows the results of manual inspection. As we can see, the overall precision, recall and F-measure of AsyncChecker on real-world applications are 97.2%, 89.8% and 0.93 respectively. The result proves that AsyncChecker can detect the misuse problem of AsyncTask precisely.

Table 6: Manual Inspection on Real-world Applications

App Name	StrongReference			NotCancel			NotTerminate		
	TP	FP	FN	TP	FP	FN	TP	FP	FN
AFWall [1]	2	0	0	2	0	0	2	0	0
AnkiDroid [2]	0	0	0	3	0	0	3	0	0
A Photo Manager [3]	2	0	0	0	0	2	2	0	0
Easy xkcd [13]	0	0	0	1	0	1	1	0	0
Kiss [28]	0	0	0	1	0	1	2	0	0
Minetest [38]	1	0	0	0	0	0	1	0	0
Mythmote [39]	1	0	0	1	1	0	1	0	0
NextCloud [40]	2	0	0	2	0	0	2	0	0
OpenBikeSharing [41]	1	0	0	1	0	0	1	0	0
Password Store [55]	2	0	0	2	0	0	2	0	0
rootless-logcat [49]	1	0	0	0	0	0	0	0	0
SatStat [50]	0	0	0	0	0	0	1	0	0
Seafile [51]	12	0	2	10	2	5	13	0	0
Simple Gallery [53]	0	0	0	0	0	1	1	0	0
syncthing-android [57]	1	0	0	2	0	0	2	0	0
Web Opac App [4]	3	0	0	2	0	0	2	0	0
Wikimedia Commons [9]	3	0	0	7	0	0	5	0	0
Sum	31	0	2	34	3	10	41	0	0

Efficiency. Figure 4 shows the analysis time of AsyncChecker on three repositories. The average running time of AsyncDetector on three repositories are 82 (F-Droid), 198 (GooglePlay) and 228 (Wandoujia) seconds respectively. Obviously, AsyncChecker can detect the misuse of asynchronous components within a relatively short period.

6.3 RQ3: Comparison with Existing Tools

In this section, we try to compare AsyncChecker with existing tools APEChecker [15] and DiagDroid [27]. APEChecker is not open source and the website of DiagDroid cannot be accessed. In order to acquire available tools, we contacted the authors of APEChecker and DiagDroid respectively. Finally, we received reply from the author of DiagDroid and they offered us the newest version of DiagDroid.

APEChecker combines static analysis and dynamic UI exploration. The author of APEChecker proposed three rules to detect asynchronous problems. On one hand, the first two rules denoted that there should not be any GUI update/creation operations in background thread, which is correct. Crash will occur if these two rules are violated. On the other hand, the last rule (Async thread should avoid accessing GUIs or performing transactions inside async callbacks) is not precise. [15] mentioned that when onPostExcute() is invoked, we don't know the state of GUIs. However, via canceling AsyncTask before GUI is destroyed we can make sure the GUI has not been destroyed when onPostExecute() is invoked. Otherwise, onCancelled() instead of onPostExecute() will be invoked. Therefore, for AsyncTask, we can operate GUIs in async callbacks (only in onPostExecute()). Apparently, those UI action patterns defined in [15] are related to functional issues which can directly lead to crash. StrongReference is related to potential memory leak, which is both performance and function related. Besides, they only applied APEChecker on 40 applications. Compared with APEChecker, AsyncChecker is based on more precise rules and it has been applied on thousands of real-world applications.

DiagDroid is based on instrumentation and random testing. To compare the effectiveness of AsyncChecker and DiagDroid on real-world applications, we applied DiagDroid on the 22 apps packaged by ourselves. The experiments are performed on a mobile phone (Nexus 5X) in the version of Android 6.0. We ran each app for 2.5 hours.

Figure 5 compared AsyncChecker and DiagDroid on the 22 apps packaged by ourselves. As we can see, DiagDroid reported 11 issues in 8 applications. We cannot assure the category of seven cases (Unkown in the figure) because the report of these cases are incomplete and we can't find categories that match those cases. The rest four cases were all categorized into Not Canceling Obsolete Tasks [27]. DiagDroid focuses on performance problems of applications and the experimental result shows that it can only find limited problems. Besides, we cannot directly judge the issue category from the report of DiagDroid and we need to read the source code, which decreases the applicability of DiagDroid. Compared with DiagDroid, AsyncChecker can detect the potential problems of applications without running the app under analysis and we can directly check the issue type from the bug report of AsyncChecker. Based on the results shown in table 6 and figure 5, we can conclude that AsyncChecker can find more misuse problems of asynchronous components. The detailed information about the validation results of AsyncChecker and DiagDroid can be found at https://github.com/pangeneral/AsyncChecker.

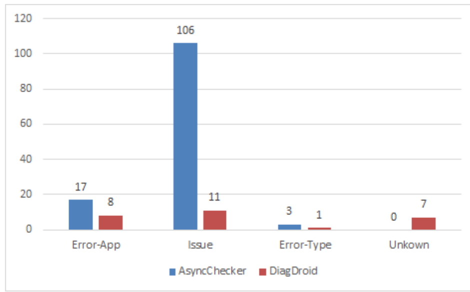

Figure 5: Comparison of AsyncChecker and DiagDroid

6.4 RQ4: Feedback from Developers

As we can see, the misuse of AsyncTask exists in many real-world applications. Do developers take these warnings as serious problems? In order to answer such questions, we reported issues of the 17 apps shown in table 6 to developers of F-Droid applications via GitHub. Finally, we got feedback from 10 apps. All of these applications are popular apps which have at least 50 stars on GitHub and 10,000 downloads in app store.

The detailed information is shown in Table 7. The developers of `Kiss` and `Password Store` have fixed the misuse problems of AsyncTask in the newest version before we came up with an issue. We listed the ID for the fixing commit. The developers of `AnkiDroid`, `Easy xkcd`, `Minetest`, `rootless-logcat`, `Web Opac App` and `Wikimedia Commons` confirmed that our reports are correct and there exists AsyncTask misuse in their applications. We listed the ID of submitted issues. The developer of `A Photo Manager` and `syncthing-android` confirmed and fixed the reported misuse problems. We listed the ID of issue and fixing commit respectively.

Table 7: Feedback from Developers of Real-world Applications on GitHub

App Name	Stars	Issue ID	Commit ID	Result
AnkiDroid	2,023	5404	/	confirmed
A Photo Manager	111	145	7fb4c03	fixed
Easy xkcd	101	164	/	confirmed
Kiss	1,281	/	fa5ab40	fixed
Minetest	4,365	8787	/	confirmed
Password Store	1,271	/	56e53d3	fixed
rootless-logcat	59	15	/	confirmed
syncthing-android	1,157	1142	b93da52	fixed
Web Opac App	108	566	/	confirmed
Wikimedia Commons	519	2791	/	confirmed

Most of the misuse problems that have been fixed by developers are caused by StrongReference which can lead to memory leakage. Some developers neglect other problems such as NotCancel, which may also cause the crash of applications. Based on the feedback of developers, we can conclude that the misuse of AsyncTask is a significant problem. Many developers have realized that such problems should be fixed to evade potential bugs.

6.5 Discussion

Threat to Validity. AsyncChecker is built on top of Androlic and Soot. Both of them still have some limitations in practice which may affect the experimental results of AsyncChecker. Currently, if AsyncTask holds strong reference to Activity via containers such as ArrayList and HashMap, there will be false negative of StrongReference. As we know, Java provides generic as syntactic sugar. In Jimple intermediate, generic is discarded and it is difficult to determine the type of objects added into collections precisely. Besides, Androlic may fail to identify some user-defined callback methods during dummy main method construction, which can lead to false negative and false positive if these callback methods contain AsyncTask-related operations. In the dummy main method, the execution order of different user-defined callback methods is fixed, which may also affect the experimental results.

Generalizability. Except NotTerminate, the detection of other misuse patterns are all based on typestate analysis, which is the cornerstone of our approach. Apparently, it is a common idea to monitor the state of an object so that we can detect its error when the state is abnormal. Typestate analysis can also be leveraged to detect the potential error of other async components such as IntentService. In other words, We can implement similar approach to detect new types of bugs and new async component after we summarize state-based bug patterns.

7 RELATED WORK

We introduce related work in two aspects. First, problems caused by asynchronous programming in Android and how researchers solve them. Second, the static analysis approach used in Android.

7.1 Android Asynchronous Programming

Currently, asynchronous programming is widely used in many areas. Android is a typical operating system based on event-driven model and single-threaded model.

The asynchronism of Android may lead to many problems such as data race. Technically, given two operations, if at least one of them is write operation and they do not happen-before [31] each other, then they may lead to data race. Maiya et al. [37] and Hsiao et al. [21] tried to find possible data race in apps based on a happens-before graph. Based on the previous two papers, Bielik et al. improved the efficiency of happens-before construction algorithm and developed EventRacer [8]. In order to reduce false positive in race detection, Hu et al. developed ERVA [22] to identify and reproduce true data race.

Due to the existence of data race, asynchronous programming may lead to unexpected bugs. Fan et al. [15] developed APEChecker to detect asynchronous programming errors (APE). Besides, AsyncDroid [43] and RacerDroid [58] are typical dynamic analysis tools which can reproduce bugs caused by data race. Liu et al. [36] concluded performance bugs in Android which includes lengthy operations in main threads, wasted computation for invisible GUI and frequently invoked heavy-weight callbacks.

Those works focused on the problems caused by asynchronous programming. Little attention has been paid to how to use async components correctly. In fact, Android provides several asynchronous components for developers. Among these components, AsyncTask is easy to use and thus very popular. Lin et al. [35] proposed a method to extract long-running operations and refactor them into AsyncTask. In [34], they presented a method to further refactor

AsyncTask into *IntentService*, a more reliable asynchronous component provided by Android. However, they do not solve the problem of how to use AsyncTask correctly. In this paper, we pay attention to how to use AsyncTask correctly and try to detect the misuse in applications.

7.2 Android Static Analysis

Static analysis is widely used in the research area of Android apps. Some static analysis frameworks such as Soot [30] and WALA [16] are widely adopted by the academia. Based on these frameworks, researchers developed some static analysis tools such as FlowDroid [6], StubDroid [5], IccTA [32] and AsDroid [24]. Technically, static analysis tools can be leveraged to perform taint analysis, symbolic execution, code instrumentation and other related tasks.

In recent years, researchers are concerned about Android specificities such as Android lifecycle, ICC (inter-component communication) and IAC (inter-app communication) during static analysis. Wu et al. [64] conducted a survey about defects in Android applications. According to [33], DidFail [29], DroidSafe [20] and FUSE [48] are the most versatile tools which take six aspects of Android specificities into consideration. Besides, Garcia et al. [18] developed LetterBomb which relied on a combined path-sensitive symbolic execution-based static analysis to generate exploits of vulnerabilities for Android apps. Huang et al. [23] came up with callback compatibility issues in Android apps and proposed a static analysis tool called Cider to detect the problem. Shan et al. [52] proposed a suite of static analysis to detect self-hiding behaviors in Android apps. Yan et al. [65] researched EAs (exported activity) and identified misexposed activities in Android apps.

As there are so many static analysis tools in the research community, some researchers paid attention to the efficiency and effectiveness of those tools. Qiu et al. [47] compared three popular tools FlowDroid, AmanDroid [62], and DroidSafe. Pauck et al. [45] proposed ReproDroid to compare Android static taint analysis tools including Amandroid, DIALDroid [10], DidFail, DroidSafe, FlowDroid and IccTA.

In this paper, we propose a static analysis approach which takes the characteristics of AsyncTask into consideration. More specifically, the state of AsyncTask changes in accordance with operations, which can be processed with typestate analysis [56]. Based on Androlic [44], the analysis approach can be easily extended for similar problems.

8 CONCLUSION

In this paper, we summarize five kinds of misuse patterns of AsyncTask, the most widely used asynchronous components in Android. In order to find these problems hidden in apps, we propose static analysis approach to collect the AsyncTask-related information and detect the misuse via predefined rules. Based on the analysis approach, we have implemented a tool called AsyncChecker. In order to evaluate AsyncChecker, we design and develop AsyncTask-specific benchmark suite AsyncBench. Besides, we analyze 1,759 real-world apps from Google Play, Wandoujia and F-Droid. AsyncChecker successfully detects AsyncTask misuse from 1,417 apps. We compare AsyncChecker with an existing tool DiagDroid and the result proves that AsyncChecker can find more asynchronous problems. We also send the misuse report to developers via GitHub. Ten developers have confirmed and fixed the problems.

Currently, AsyncChecker concentrates on AsyncTask, one of the most popular async components. In fact, there might be similar problems in other async components. In the future, we plan to conduct further studies on async components and try to detect more async-related problems.

ACKNOWLEDGEMENTS

This work is supported by the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (Grant No. QYZDJ-SSW-JSC036), and the National Natural Science Foundation of China (Grant No. 61672505).

REFERENCES

[1] AFWall. 2019. https://github.com/ukanth/afwall.
[2] Anki-Android. 2019. https://github.com/ankidroid/Anki-Android.
[3] APhotoManager. 2019. https://github.com/k3b/APhotoManager.
[4] Web Opac App. 2019. https://github.com/opacapp/opacclient.
[5] Steven Arzt and Eric Bodden. 2016. StubDroid: automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering, ICSE'16, Austin, TX, USA, May 14-22*. 725–735. https://doi.org/10.1145/2884781.2884816
[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM-SIGPLAN Symposium on Programming Language Design and Implementation, PLDI'14, Edinburgh, United Kingdom, June 09 - 11*. 259–269. https://doi.org/10.1145/2594291.2594299
[7] AsyncTask. 2019. https://developer.android.com/reference/android/os/AsyncTask.
[8] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2015. Scalable race detection for Android applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'15, Pittsburgh, PA, USA, October 25-30*. 332–348. https://doi.org/10.1145/2814270.2814303
[9] Wikimedia Commons. 2019. https://github.com/commons-app/apps-android-commons.
[10] DialDroid. 2019. https://github.com/dialdroid-android/DIALDroid.
[11] Android documentation. 2019. https://developer.android.google.cn/docs.
[12] DroidBench. 2019. https://github.com/secure-software-engineering/DroidBench.
[13] Easyxkcd. 2019. https://github.com/T-Rex96/Easy_xkcd.
[14] F-Droid. 2019. https://f-droid.org.
[15] 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
[16] Stephen Fink and Julian Dolby. 2012. WALA–The TJ Watson Libraries for Analysis.
[17] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective typestate verification in the presence of aliasing. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*. 133–144. https://doi.org/10.1145/1146238.1146254
[18] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. 2017. Automatic generation of inter-component communication exploits for Android applications. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE'17, Paderborn, Germany, September 4-8*. 661–671. https://doi.org/10.1145/3106237.3106286
[19] GooglePlay. 2019. https://play.google.com/store.
[20] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium, NDSS'15, San Diego, California, USA, February 8-11*. https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe
[21] Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. 2014. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM-SIGPLAN Symposium on Programming Language Design and Implementation, PLDI'14, Edinburgh, United Kingdom, June 09 - 11*. 326–336. https://doi.org/10.1145/2594291.2594330

[22] Yongjian Hu, Iulian Neamtiu, and Arash Alavi. 2016. Automatically verifying and reproducing event-based races in Android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA'16, Saarbrücken, Germany, July 18-20*. 377–388. https://doi.org/10.1145/2931037.2931069

[23] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for Android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE'18, Montpellier, France, September 3-7*. 532–542. https://doi.org/10.1145/3238147.3238181

[24] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering, ICSE'14, Hyderabad, India, May 31 - June 07*. 1036–1046. https://doi.org/10.1145/2568225.2568301

[25] Icc-Bench. 2019. https://github.com/fgwei/ICC-Bench.

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

[27] 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. https://doi.org/10.1145/2950290.2950316

[28] Kiss. 2019. https://github.com/Neamar/KISS.

[29] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis, SOAP'14, Edinburgh, UK, Co-located with PLDI'14, June 12*. 5:1–5:6. https://doi.org/10.1145/2614628.2614633

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

[31] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[32] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering, ICSE'15, Florence, Italy, May 16-24*. 280–291. https://doi.org/10.1109/ICSE.2015.48

[33] 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

[34] 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

[35] Yu Lin, Cosmin Radoi, and Danny Dig. 2014. Retrofitting concurrency for Android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE'14, Hong Kong, China, November 16-22*. 341–352. https://doi.org/10.1145/2635868.2635903

[36] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering, ICSE'14, Hyderabad, India, May 31 - June 07*. 1013–1024. https://doi.org/10.1145/2568225.2568229

[37] 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

[38] Minetest. 2019. https://github.com/minetest/minetest.

[39] Mythmote. 2019. https://github.com/pot8oe/mythmote.

[40] Nextcloud. 2019. https://github.com/nerzhul/ncsms-android.

[41] OpenBikeSharing. 2019. https://github.com/bparmentier/OpenBikeSharing.

[42] Stack Overflow. 2019. https://stackoverflow.com.

[43] Burcu Kulahcioglu Ozkan, Michael Emmi, and Serdar Tasiran. 2015. Systematic Asynchrony Bug Exploration for Android Apps. In *Proceedings of the 27th International Conference on Computer Aided Verification, CAV'15, San Francisco, CA, USA, July 18-24*. 455–461. https://doi.org/10.1007/978-3-319-21690-4_28

[44] 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. https://doi.org/10.1145/3293882.3339001

[45] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep their Promises?. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE'18, Lake Buena Vista, FL, USA, November 04-09*. 331–341. https://doi.org/10.1145/3236024.3236029

[46] Processes and Threads. 2019. https://developer.android.com/guide/components/processes-and-threads.

[47] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'18, Amsterdam, The Netherlands, July 16-21*. 176–186. https://doi.org/10.1145/3213846.3213873

[48] Tristan Ravitch, E. Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. 2014. Multi-App Security Analysis with FUSE: Statically Detecting Android App Collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC'14, New Orleans, LA, USA, December 9*. 4:1–4:10. https://doi.org/10.1145/2689702.2689705

[49] rootless logcat. 2019. https://github.com/tananaev/rootless-logcat.

[50] SatStat. 2019. https://github.com/mvglasow/satstat.

[51] Seafile. 2019. https://github.com/haiwen/seadroid.

[52] Zhiyong Shan, Iulian Neamtiu, and Raina Samuel. 2018. Self-hiding behavior in Android apps: detection and characterization. In *Proceedings of the 40th International Conference on Software Engineering, ICSE'18, Gothenburg, Sweden, May 27 - June 03*. 728–739. https://doi.org/10.1145/3180155.3180214

[53] SimpleGallery. 2019. https://github.com/SimpleMobileTools/Simple-Gallery.

[54] Statista. 2019. Google Play. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[55] Password Store. 2019. https://github.com/zeapo/Android-Password-Store.

[56] Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[57] syncthing android. 2019. https://github.com/syncthing/syncthing-android.

[58] 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

[59] View. 2019. https://developer.android.google.cn/reference/kotlin/android/view/View.

[60] Wandoujia. 2019. https://www.wandoujia.com.

[61] WeakReference. 2019. https://developer.android.com/reference/java/lang/ref/WeakReference.

[62] 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

[63] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. 2007. A New Algorithm for Identifying Loops in Decompilation. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*. 170–183. https://doi.org/10.1007/978-3-540-74061-2_11

[64] Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. 2019. Analyses for specific defects in android applications: a survey. *Frontiers Comput. Sci.* 13, 6 (2019), 1210–1227. https://doi.org/10.1007/s11704-018-7008-1

[65] Jiwei Yan, Xi Deng, Ping Wang, Tianyong Wu, Jun Yan, and Jian Zhang. 2018. Characterizing and identifying misexposed activities in Android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE'18, Montpellier, France, September 3-7*. 691–701. https://doi.org/10.1145/3238147.3238164