Detection of Java Basic Thread Misuses Based on Static Event Analysis

Baoquan Cui[1,3], Miaomiao Wang[2,3] Chi Zhang[1,3], Jiwei Yan[2†], Jun Yan[1,2,3†] and Jian Zhang[1,3]
[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China
Email: {cuibq, zhangchi, yanjun, zj}@ios.ac.cn, {wangmiaomiao20, yanjiwei}@otcaix.iscas.ac.cn

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00145

***Abstract*—The fundamental asynchronous thread (`java.lang.Thread`) in Java can be easily misused, due to the lack of deep understanding for garbage collection and thread interruption mechanism. For example, a careless implementation of asynchronous thread may cause no response to the interrupt mechanism in time, resulting in unexpected thread-related behaviors, especially resource leak/waste.**

Currently, few works aim at these misuses and related works adopt either the dynamic approach which lacks effective inputs or the static path-sensitive approach with high time consumption due to the path explosion, causing false negatives. We have found that the behavior of threads and the interaction between threads and its referencing objects can be abstracted. In this paper, we propose an event analysis approach to detect the defects in Java programs and Android apps, which focuses on the existence or the order of the events to reduce the false negatives. We extract the misuse-related events, containing the thread events and the destroy events of the object referenced by the thread. Then we analyze the events with loop identification, happens-before relationship construction and alias determination. Finally, we implement an automatic tool named `Leopard` and evaluate it on real world Java programs and Android apps. Experiments show that it is efficient when comparing with the existing approach (misuse: 723 vs 47, time: 60s vs 30min), which also outperforms the existing work in precision. The manual check indicates that `Leopard` is more efficient and effective than existing work. Besides, 66 issues reported by us have been confirmed and 21 of them have been fixed by developers.

***Index Terms*—Thread Misuse, Basic Thread, Runnable, Static Analysis, Java Program**

I. Introduction

The latest ranking shows that Java remains one of the most popular programming languages [1]. The classes, class *java.lang.Thread* and its interface, *java.lang.Runnable*, are the most basic units of the asynchronous task in Java, and they are commonly used especially in multi-threaded programming and thread pool usage, which we call the basic thread in this paper. Existing researches about thread misuse analysis and detection are based on some asynchronous components in Android with Android platform features [2–7]. And others focus on data races [8–16] , especially on the Android platform [17–24]. They adopt either the dynamic approach which lacks effective inputs or the static path-sensitive approach with high time consumption due to the path explosion. Besides, as Android versions are updated, some Android asynchronous components have been replaced by new ones or eliminated. For example, `AsyncTask` and `IntentService` mentioned in [4, 7] are deprecated in API level 30 [25, 26]. It is officially recommended by Android to use the basic concurrency components in Java to efficiently execute Android asynchronous tasks [25].

However, few works aim at the basic thread class (*java.lang.Thread*), which is prone to being misused. For example, an object reference held by a thread makes it difficult to be recycled and released in time. In particular, if a thread object has a strong reference [27] to another object which invokes a destroy method before the thread finishes, its recycling and releasing will be blocked by the thread holding its reference. It wastes memory resources, and can cause an out of memory (OOM) crash. Moreover, Java provides the interrupt mechanism to terminate a thread, but the *interrupt()* method invocation will only set the status of the thread if it is running [28]. If a thread is notified that it is interrupted via the invocation but there is no corresponding interrupted status checking to end running in time, it makes *interrupt()* method invocation unresponsive. This leads to unnecessary code execution and waste of resources.

To detect these basic thread misuses, there are some challenges. The first challenge is to analyze the happens-before relationship between the end of the thread and the call of the destroy method of its referenced object. The scheduling of multiple threads is non-deterministic, leading to different program behaviors. And identifying methods with the destruction semantics is also a challenge, which is vague in practice. It depends on many factors, documentation, comments or even just developers' habits. Finally, reducing false negatives without precision decrease is a challenge too. Coarse-grained static analysis suffers from notorious high false positives while a path-sensitive approach struggles with path explosion resulting false negatives in a limited time.

Our Approach. In this paper, we identify three typical misuse patterns of the basic thread, which are called HardToRelease (HTR), InterruptNoResponding (INR) and NotTerminatedInTime (NTT), and propose a lightweight approach to detect the thread related defects based on event analysis

†Corresponding Authors.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00145

(Section IV), avoiding the multiple exploration of a large portion of the code. Firstly, we extract the misuse-related event, containing the destroy events of the error-prone object referenced by a thread and the thread events (Section IV-A). For the former, we present a statistical approach to the identification of methods that have destruction semantics in practice (Section IV-A1) and are misused easily. The data shows that the classes with a destruction method are widely distributed in various platforms like JDK, Android SDK, and Spring Framework (Section II-B), indicating their instances are widely distributed as well. For the later, we focus on the thread event affecting its status and task, such as the initialization, start, termination, interruption and so on (Section IV-A2).

Then we analyze the misuse-related events based on the interaction between the threads (Section IV-B). Loop analysis of the event *run()* method is used to determine whether it will respond to an interrupt invocation (Section IV-B1). The happens-before relationship is used to determine the order between the thread ending and the invocation of the destroy method of an object referenced by the thread (Section IV-B2). We specifically handle the happens-before relationship brought by the *Thread.start()/join()*, representing the start of an asynchronous task and the start of thread synchronization. Furthermore, many tasks in the thread are implemented by the *java.lang.Runnable* interface, so that we build a special alias relationship between the thread and the task to detect misuses more thoroughly (Section IV-B3). With the analysis result, we design algorithms based on the three misuse patterns (Section IV-C). Finally, we build a tool named `Leopard` to detect the misuses automatically according to the algorithms. We evaluate `Leopard` and the experimental data shows `Leopard` is powerful to detect these misuses (Section V).

In summary, our contributions are as follows:

- We identify three misuse patterns, *i.e.*, HTR, INR and NTT, to characterize the misuse of the basic thread in Java.
- We propose a lightweight flow-sensitive approach based on event analysis to detect the thread related misuses with the patterns. It focuses on the existence or the order of the thread events and the destroy events of objects it refers to. We also propose clear rules to identify the object which may be hard to be recycled and released by GC when referenced by a thread.
- We design succinct and efficient algorithms based on the analysis and implement them with a lightweight tool called `Leopard`. We evaluate it on 9 large Java programs from Github and 147 real-world apps from F-Droid. Experimental results demonstrate it can efficiently detect thread-related misuses. We report issues and 66 of them are confirmed by developers where 21 are fixed.

II. Background

In this section, we will present the thread component, the garbage collector (GC) in Java with a statistical analysis to study the classes containing "destroy" method, and the thread interrupt mechanism. At last, a motivating example shows the problems caused by misuses of the basic thread.

TABLE I
Destructible Class Distribution

Classification	#Destr. Class	Rule (in Package)
JDK	1,564	java.*, javax.*,"sun.*","com.sun.*", "jdk.*", "org.omg.*", "org.w3c.*", "org.xml.*", "org.jcp.*", "org.ietf.*"
Android SDK	2,027	android.*, androidx.*
SpringFramework	1,117	org.springframework.*
Third-party Impl.	16,232	the remaining
Total	20,940	-

A. Async Component in Java and Android

The asynchronous component refers to the encapsulation or implementation of the original thread to quickly implement asynchronous tasks. JDK provides the most basic thread components: thread implementation (*java.lang.Thread*) and reserved interface (*java.lang.Runnable*). There are only two ways to create a new thread of execution. One is to declare a class to be a subclass of *java.lang.Thread* and the other way is to declare a class that implements the interface *java.lang.Runnable* while pass its instance to a thread. Some of the asynchronous components provided by Android are the target research objects in previous works [2–4, 6], such as `AsyncTask`, `AsyncTaskloader`, `IntentService`. But they are deprecated by Android officially for performance reasons [25, 26, 29]. It is officially recommended to use the standard concurrent components to efficiently execute Android asynchronous tasks [25]. This is also one of the motivations why we analyze and detect misuses of the basic thread.

B. GC and Thread Preemption

Java uses the Garbage Collector (GC) to automatically recycle memory for objects. GC adopts the root search algorithm to determine whether an object is "garbage", that is, to determine whether an object is alive or not. The active thread is one of the important components in the GC roots where the search starts. It means that if a thread is alive, objects referenced strongly by it directly or indirectly will not be marked for garbage collection.

Java does not have an explicit destructor as in C++. Objects may have a *destroy()*, *close()* method to release occupied resources actively or via *onDestroy()* callback passively. It is not the actual destruction of the object, but the preparation before the object is recycled. After the destroy method of an object is called, the resources occupied by it will be released, and the object itself should be recycled by GC in time and no longer be accessed again theoretically. We perform a statistical analysis, trying to answer the following empirical question (EQ):

- **EQ:** How many classes are there with the destroy semantic method in Java programs and Android applications?

EQ Result. We use the experimental dataset in Section V, 9 large Java programs and 147 apps, for statistics, including JDK and Android SDK. Here, a method with a name containing "destroy", "close" will be roughly regarded as the destroy semantics one and its declared class will be counted as the destructible class. Table I shows the result. We have identified

a total of 20,940 destructible classes without duplicates in about 400,000 classes. It can be seen that they are not only widely distributed on various platforms and frameworks, but also quite common in upper-layer applications and programs.

The thread scheduling is preemptive in Java. It means that the scheduling of threads is the responsibility of the operating system rather than the program itself. That is to say, Java does not provide a solution for manual management or termination of the thread in time directly. What all the developers can do is to call the *interrupt()/stop()* method. If the thread is running, then the *interrupt()* invocation will do nothing except setting an interrupted status [28]. At this time, if the thread has not checked the interrupted status during executing with the asynchronous loop (usually seen as time consuming), the invocation will lose responding and the thread will not terminate as soon as expected.

C. Motivating Example

We will show the typical problems caused by the misuse of the basic thread via a motivating example. Listing 1 defines two classes: a destructible object class named `DestructibleObject` with the *destroy()* method and a subType of thread named `WorkerThread` (lines 7 and 18). The *main()* method declares a `DestructibleObject` object *obj*, and calls its *init()*, *startTask()*, and *destroy()* methods sequentially (lines 2-5). The class, `DestructibleObject`, initializes a thread object as its field (line 8) and starts it to perform asynchronous tasks (line 15). The thread object receives a `DestructibleObject` argument and assigns it to its own field named *dObj* (line 21). After the *startTask()* method invocation, the thread *workerThread* will be executed and its *run()* method will be invoked if it is selected by the thread scheduler. At the end of the *run()* method, the data has been assembled, and the object will be updated (lines 24-25).

Listing 1. A Motivating Example

```
1  public static void main(String[] args){
2     DestructibleObject obj = new DestructibleObject();
3     obj.init();
4     obj.startTask();
5     obj.destroy();
6  }
7  class DestructibleObject{
8     Thread workerThread = null;
9     public void init(){
10       workerThread = new WorkerThread(this); }
11    @PreDestroy
12    public void destroy(){
13       if( workerThread != null ){
14       workerThread.interrupt();} }
15    public void startTask(){ workerThread.start(); }
16    public void update(Object data){ ... }
17 }
18 class WorkerThread extends Thread{
19    DestructibleObject dObj = null;
20    public WorkerThread(DestructibleObject obj){
21       dObj = obj; } // dObj will be hard to be
              released after its destroy method
              invocation if the thread is alive
22    public void run(){
23       while (...){ ...}//No Response to the
              interruption call in line 14
24       data = ...;
25       dObj.update(data); ...}
26 }
```

Notably, the execution of the code in the *run()* method is asynchronous. So there is an indeterminate happens-before relationship between the *destroy()* method invocation (line 5) and the end of *run()* method (line 25). Since the *workerThread* object holds a strong reference to the `DestructibleObject` object, even if the *destroy()* method is called first, the `DestructibleObject` object cannot be recycled in time by the GC as mentioned before. This causes a potential memory leak defect, especially if the destructible object is a large object, such as a handle of a database, a reference of an Android Activity, or an object holding a native library resource. Besides, although the `DestructibleObject` *obj* tries to finish the execution of the *workerThread* (lines 13-14), there exists no interrupted status check in the loop (seen as time-consuming) of the *run()* method to respond to the intention of terminating the execution as soon as possible (line 23). This also leads to a potential resource wasting.

III. Misuse Pattern

We identify three misuse patterns on the basic thread in Java based on the Java document and the projects in practice. They can cause both functional and performance problems, described as follows.

- **HardToRelease (HTR)**. As mentioned before, if an instance of the basic thread holds a strong reference [27] of a destructible object (Listing 1 line 22), the object will never be actually released by the GC until the thread finishes. In this paper, we call the prone mis-referenced object, who has a destruction semantic method, a destructible object. That is, the strong reference held by an active thread is preventing the object from being recycled, which makes it difficult for the object to be released. This can lead to unnecessary memory usage and even memory leaks.
- **InterruptNoResponding (INR)**. When a thread is running, there is no perfect way to end it immediately. Its *interrupt()* method can act as a bridge of the interaction between the main thread and the worker thread. If the thread is in the `Running` state, then its interrupt status will be cleared and it will receive an `InterruptedException` when the *interrupt()* method is called. Otherwise, it is just to set the interrupted flag [28]. At this time, if the loop (considered time consuming) in its *run()* method does not check the interrupted status, then the invocation loses any responding as expected (Listing 1 line 24). We call it InterruptNoResponding (INR). This problem will lead to execute useless program instructions, resulting in unsatisfied program intent and the waste of resources.
- **NotTerminatedInTime (NTT)**. If a destructible object is referenced by an active thread, the thread should be interrupted before the destructible object destroy, i.e., its destroy method invocation, as it may use the object before its termination. we call the misuse NotTerminatedInTime (NTT). In detail, NTT occurs when an active thread holds a reference (whether it is a strong reference, a weak reference [30], or the other reference) of a destructible object, and the invocation order of the object's destroy method and the termination of the thread

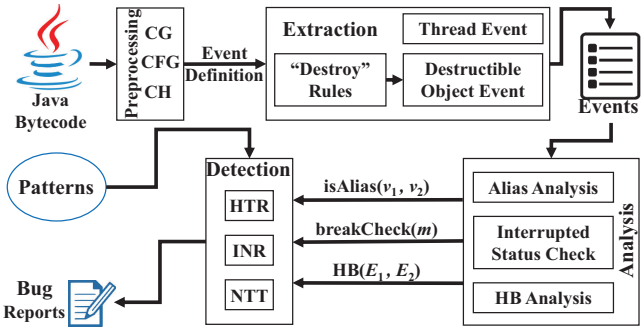

Fig. 1. Overview of Our Approach

are indeterminate. A better usage in practice is to send an interrupted message to end the thread as soon as possible before the object is destroyed, because at this time accessing the object is unnecessary or illegal.

IV. Approach

Our insight is that the thread-related code occupies only a small part of the program and these misuses can be reduced to the event existence or the order determination between events. Therefore, the approach based on event analysis can avoid the analysis of all paths and all variables, thereby improving efficiency. Figure 1 shows the overview of our approach. We take the Java bytecode as input to get the basic program properties such as the class hierarchy (CH) relationship, the control flow graph (CFG), and the call graph (CG) of the program. Then we extract the misuse related events. Combining previous work [31, 32] and practical experience, we propose explicit rules for identifying these destructive objects. After the extraction of the events, we analyze them with loop analysis, happens-before relationship construction and alias relationship between variables for the misuse detection. Finally, misuses will be detected via the algorithm based on the patterns and the analysis result, and recorded into the report.

For convenience of presentation, we introduce ***Event*** to represent method invocation and specify it with thread related invocations driven by the misuses detection demand in the following section.

Definition IV-1 (Event). *An event is a 3-tuple*

$$\boldsymbol{E} = \langle caller, method, arguments \rangle$$

presenting a method invocation consisting of the caller, the method and the argument list, where the caller is the instance the underlying method is invoked from and the argument list is the arguments used for the method call.

For an ***Event***, its caller and method always exist. In general, the number of arguments in an invocation may be any non-negative number. But we assume that the number is zero or one, as we focus on `Thread`, `Runnable` and destructible objects. Particularly, we take the assignment statement to the field variable as a set method invocation, that is, $\boldsymbol{E} = \langle caller, \text{“}setName\text{”}, \text{“}Bob\text{”} \rangle$ represents $caller.name = \text{“}Bob\text{”}$.

A. Event Extraction

The misuses involve the destroy event and the thread event which need to be extracted at first.

1) Destroy Event Extraction: A destroy event ($\boldsymbol{E}_{Destroy}$) is the destroy method invocation of a destructible object. The bean object under the `Spring` framework has the *destroy()* method. The `Activity` and `Fragment` in Android has the *onDestroy()* callback triggered by the framework.

Destroy Event Identification. In fact, JDK provides an annotation, `@PreDestroy` in the package *javax.annotation*, with the same meaning since JDK 1.6. This annotation can modify a method to mark the method a pre-destroy one (as marked in Listing 1, line 12). Its meta annotation, `@Retention`, has the value of `@RUNTIME`, which means the annotation could be obtained even at runtime. It is widely used, especially in Spring projects. Searching for “PreDestroy” in GitHub, the results contain about 303K code, 11K commits, 2K issues. It is said that the method annotated with `@PreDestroy` is typically used to release resources that the instance holds in Java Specification Requests 250 (JSR 250) [32]. So it is a generalizable method to help us identify the destructible object from the bytecode.

Based on the above knowledge and the observation in Section II-B, we identify the destructible object as well as the destructible event. At first, we treat an object of a class as the potential candidate destructible object if it satisfies any one of the following rules: **(1)** it has a method marked with the annotation `@PreDestroy`; **(2)** it has a method with a name containing “destroy” ignoring case; **(3)** it has a method named “close”. We then refine the rules via code analysis to reduce false positives.

Refinement. The destroy event identified by the rules above works on most cases, representing the destruction semantics really. But there are false positives caused by the special implementation without any code that destroys or releases resources which should be excluded. In Java, a common way to release a reference to a filed is to assign it the value `NULL`. Another semantic representation of resource release is that its field destroys itself in the destroy method, i.e., the field is also a destructible object. Thus, we can exclude false positives due to empty destroy methods and purified methods whose bodies do not contain any field operations. Such methods are usually reserved for destroy method semantics due to some programming habit or specification, but currently do not perform any tasks in their implementations. Finally, the release of native resources in Java is usually through JNI, so we think that calling JNI in the destroy semantic method is a signal to release resources. Through the above, we can distinguish `ByteArrayInputStream` and `FileInputStream`. Both of them have a *close()* method but the former does nothing while the later releases a native file handle.

In summary, we identify a destructible object if it has a candidate destroy method whose body contains: (1) an assignment statement for its field with the value `NULL`, (2) an invocation statement of its field destroy method, or (3) a JNI invocation statement.

2) Thread Event Extraction: For these misuses, we specify thread events to facilitate detection detailed as follows:

- $\boldsymbol{E}_{Init}$ represents the constructor invocation of a thread, where a `Runnable` argument r may be passed; it means the caller has been initialized.
- $\boldsymbol{E}_{Start}$ means to start a thread, after which the thread *run()* method is ready to be scheduled.
- $\boldsymbol{E}_{Terminate}$ is a dummy method that indicates the thread's *run()* method has finished executing.
- $\boldsymbol{E}_{Interrupt}$ is the *interrupt()/stop()* (which is unsafe) method invocation of a thread object to interrupt the thread.
- $\boldsymbol{E}_{InterruptCheck}$ is the *isInterrupted()* method invocation of a thread object to get the interrupted status of the thread.
- $\boldsymbol{E}_{Join}$ is the *join()* method invocation to wait for the thread to finish executing.
- $\boldsymbol{E}_{SetField}$ is the set field method invocation of a thread object, which may be a shorthand for the field assignment statement as mentioned above.

With the event illustration, we can locate them in the program for further analysis.

B. Event Analysis

We analyze the events for the misuses: (1) the loop analysis can be used to check if there exists interrupted status check in the asynchronous task *run()* method; (2) happens-before construction helps to link the events inter- or intra- thread; and (3) alias analysis is used to identify the real task which may extend the `Thread` or implement the `Runnable` interface as well, and the thread where the event happens.

1) Interrupted Status Check: To detect the misuse, we need to identify if there is an $\boldsymbol{E}_{InterruptCheck}$ to break for each loop in its *run()* method. According to the implementation of the thread from JDK, the *run()* method may belong to the thread object or its field *target* if the field is not `NULL`. Its field *target* is a `Runnable` type and passed in from its constructor invocation. We take both cases into account. We implement an efficient loop recognition algorithm from [33]. For each loop belonging to the *run()* method, it will be checked whether there is *isInterrupted()* check in it after loop identification. We denote the result as `breakCheck`(m), where m is the *run()* method. In fact, we will first analyze and save the results of the *run()* methods in all of `Runnable` and `Thread` classes. All instances of each class have the same result, so that we do not need to perform redundant analysis, and directly take the cache result if necessary.

2) Happens-before Relationship Construction: To detect these misuses, one of the subtasks is to determine the order in which certain method calls occur, such as the end of a thread and the destruction method of its referencing object, as shown in Figure 2. The happens-before relationship is a partial order relation, which can be formally defined such that:

- If event E_1 and E_2 occur in the same thread, $E_1 \rightarrow E_2$ if the occurrence of event E_1 precedes the occurrence of event E_2;
- If event E_1 is the sender of a message and event E_2 is the receiver of the message, $E_1 \rightarrow E_2$;

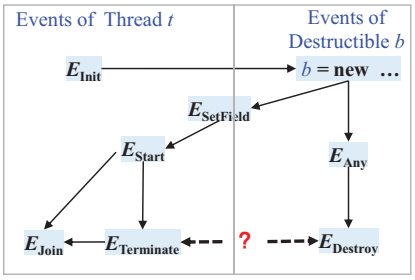

Fig. 2. Happens-before Relationship between Events. $\boldsymbol{E}_{Any}$ is any event between the initialization and destruction of the destructible object b.

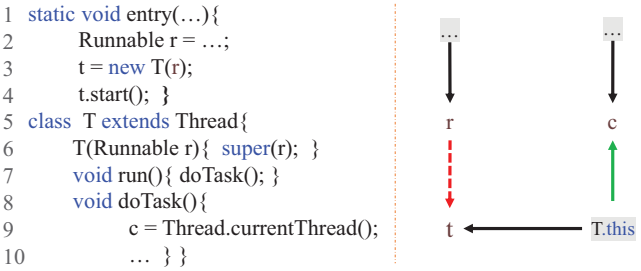

Fig. 3. Alias Analysis

where $\rightarrow$ means happens-before. It can be represented also as a function HB whose range has three elements: $HB(E_1, E_2) \in \{true, false, unknown\}$. The unknown value means that if two events happen in different isolated threads, then the two threads are said to be concurrent, that is neither $HB(E_1, E_2)$ nor $HB(E_2, E_1)$ is true. Like all strict partial orders, the happens-before relation is transitive which means if $E_1 \rightarrow E_2$ and $E_2 \rightarrow E_3$, then $E_1 \rightarrow E_3$.

According to thread event semantics, events of the same thread declared by us above may have implicit happens-before relationship. That is, $\boldsymbol{E}_{Init} \rightarrow \boldsymbol{E}_{Start}$, and $\forall \boldsymbol{E} \rightarrow \boldsymbol{E}_{Terminate}$. In particular, in the scenario where two threads interact with messages through events, some events between the two threads will generate a definite happens-before relationship. That is the event $\boldsymbol{E}^{t_1}_{Start_{t_2}}$ which starts the thread t_2 in the thread t_1 happens before the first event in the *run()* method of the thread t_2, and any event in t_2 happens before its $join()$ event. They can be represented briefly as shown in Figure 2. Based on the program call graph, control flow graph and the above happens-before relationship as the critical connection, we can infer the sequence of events more completely. Next, we will determine which events belong to the same object.

3) Event-Caller Analysis: To determine which events belong to the same caller, we need alias analysis. Our alias analysis is based on the existing analysis result of FlowDroid. In particular, in alias analysis, we build a value flow edge with the caller and argument $(t := r)$ in event $\boldsymbol{E}_{Init}$ as shown in Figure 3 (red dotted arrow). It means that the `Runnable` variable r is assigned as an argument to the field *target* of the thread object t. If the field *target* is not *NULL*, then the *run()* method in the thread actually executes the *run()*

Algorithm 1: MisuseDetection

```
   Input: bytecodeProgram
1  events = eventAnalysis(bytecodeProgram);
2  allClasses = getAllClassesInApk(bytecodeProgram);
3  unCheckedClasses = ∅;
4  for each c ∈ allClasses do
5      if c inherits from Runnable then
6          if BreakCheck(c.run()) then
7              unCheckedClasses.add(c);
8  for each s ∈ events.startEvents do
9      for each f ∈ events.setFieldEvents do
10         if isAlias(s.caller, f.caller) ∧
             isDestructibleClass(f.arg.getClass()) then
11             recordHTRMisuse(s); // record HTR
12     for each c ∈ s.caller.pointClasses do
13         if unCheckedClasses.contains(c) then
14             for each i ∈ events.interruptEvents do
15                 if isAlias(s.caller, i.caller) then
16                     recordINRMisuse(s); // record INR
17     for each d ∈ events.destroyEvents do
18         if s.caller references d.caller then
19             for each i ∈ events.interruptEvents do
20                 if isAlias(s.caller, i.caller) then
21                     if ¬HB(i, d) then
22                         recordNTTMisuse(s); // record NTT
23             if ¬ events.destroyEvents.isEmpty() ∧
                 interruptEvents.isEmpty() then
24                 recordNTTMisuse(s); // record NTT
```

method of the `Runnable` object variable. At this point, the *run()* invocation, $\boldsymbol{E}_{SetField}$, $\boldsymbol{E}_{Interrupt}$, and $\boldsymbol{E}_{Terminate}$, are all consistent with the Runnable variable. Moreover, we need to consider the static assignment of the thread, *i.e.*, *c = Thread.currentThread()*, which is a native implementation and returns the current thread object. We use the backward analysis recursively to find the *run()* method which calls this method and is the first method called after the thread start. That is, we get the thread class containing the *run()* method. Then we obtain the thread object according to point-to analysis (green solid arrow). If there is no explicit predecessor *run()* method, we point it to the main thread. We use a function to express whether n variables are aliases to each other, denoted as $\texttt{isAlias}(v_1, ..., v_i, ..., v_n)$, where $i, n \in N^+$ and $2 \leq n$.

C. Detection Algorithm

We detect misuses according to Algorithm 1, based on the event analysis before. It takes the Java bytecode program (Android apk can be regarded as a special one) as input and attains all the events we focus on, such as $\boldsymbol{E}_{Start}$, $\boldsymbol{E}_{Interrupt}$, $\boldsymbol{E}_{Destroy}$ and so on (line 1). As all basic threads are subclasses of `Runnable` (threads also implement this interface), we analyze their *run()* methods at first, and record classes that do not perform interrupt check to avoid redundant analysis for INR detection (line 2-7). Thread misuse detection is aimed at objects that have been started (line 8). In other words, those threads that have not been in the Running state will not be misused. If a thread has an $E_{setField}$ event and is assigned a destructible object to its field, a HTR will be recorded (line 9-11). Besides, for each thread with an interrupt event, if any class of the object it points to has no interrupt check in the *run()* method (that is, the class is in the cached unchecked classes), then an INR occurs (line 12-16). Moreover, for each thread holding a destructible object, whether it is strong reference or not, and if the thread's interrupt method is not called before the destructible object is destroyed, a NTT is logged (line 17-22). In particular, the NTT is logged if there is no interrupt method call at this time too (line 23-24). Finally, after the detection, we can get the report as a result.

V. Evaluation

We have developed a tool named `Leopard` to detect misuses of the thread with the event analysis approach mentioned before, which is open source and public accessible[1]. It is based on `FlowDroid` [34] and the intermediate representation `Jimple`. We firstly identify the classes of destructible objects referenced by threads. For Android apps, `Leopard` takes dummy main methods as the main entry methods. But for Java programs, all public methods are considered as entrances as they may be called as third-party libraries. Misuses that are not reachable from these methods will be ignored. Then `Leopard` collects and analyzes all thread related events. Finally, it performs a detection based on the algorithms.

We will evaluate `Leopard` with the following research questions:

- **RQ1:** Can `Leopard` find the thread related misuses in Java programs and Android apps?
- **RQ2:** How efficient is `Leopard` against the existing approach?
- **RQ3:** Do developers take the misuse as a serious problem?

A. Experiment Setup

All the following experiments run in a docker container, where the operating system is Ubuntu with 46 cores (Intel (R) Xeon (R) E5-2680) and 50G RAM. And the experiments are running under the environment of JDK-1.8.

DataSet (9 large Java Programs and 147 Android Apps). These open source Java programs are collected through the search website, *grep.app*, which can search for common git code hosting platforms, such as GitHub, and the projects with more stars or forks will be returned first. We use "PreDestroy" as the keyword and restrict the result to the Java programming language. Thus 3,150 results are obtained, and we select the top 10 repositories in the list. But one program is excluded which is a teaching project with fragmented code and asynchronous tasks. Their basic information is shown in Table II, and it can be seen that they are very popular. Among them, the Apache Tomcat [35] is an open source implementation of the Java Servlet, JavaServer Pages, Java

[1]https://github.com/cuixiaoyiyi/Leopard

TABLE II
MISUSES DETECTED BY LEOPARD IN JAVA PROGRAMS

Project	Fork[1]	Star	LOC	HTR	INR	NTT	Time (s)
cxf[44]	1.3K	758	1,066K	22	58	7	6,184
hivemq[45]	211	738	737K	1	4	1	251
jetty[46]	1.8K	3.3K	181K	7	12	3	53
junit5[36]	1.2K	5.1K	31K	1	2	0	12
micronaut[47]	922	5.5K	531K	12	19	5	953
quarkus[48]	1.8K	9.7K	434K	17	34	3	685
spring[49]	33.3K	47.1K	1,507K	5	20	2	321
tomcat[35]	4.1K	6K	193K	6	18	1	59
wildfly[50]	2.1K	2.7K	1,443K	8	9	8	1,035

[1] Here, *Fork* is a term in Github platform, which means to make a copy of the repository into another account; not a fork operation in multithreading.

TABLE III
MISUSES DETECTED BY LEOPARD AND ASYNCCHECKER WITH DIFFERENT TIMEOUT CONFIGURATIONS IN APPS. $M(N)$ DENOTES M MISUSES DETECTED IN N APKS.

Tool		HTR	INR	NTT	Total
AsyncChecker	5min	18 (14)	9 (5)	15 (12)	42 (15)
	30min	19 (14)	12 (5)	16 (12)	47 (15)
Leopard		**333 (95)**	**94 (34)**	**296 (89)**	**723 (103)**

Expression Language and Java WebSocket technologies. It is often used when deploying web/server programs. Junit5 [36] is a Java unit test case execution framework. We also selected apps from the open source platform F-Droid [37]. It contains 1,249 apps totally. An app, where at least one method in `Activity` contains the string "java.lang.Thread" (*i.e.*, `stmt.toString.contains("java.lang.Thread")`) in its Jimple statements (definition statement, invocation statement, assignment statement, etc.), has been put in the experimental dataset. Such filtering is to allow the comparison tool to detect more problems in a short time, facilitating comparison. In fact, there are no restrictions on the use of `Leopard`. Finally, 147 APKs are filtered out. The data set is also accessible via the aforementioned anonymous link.

Comparison Tool Selection. According to the existing research [38], the tool, `LeakDroid` [39], has supported the leak detection caused by the thread while others have not, such as `Relda` [40], `Android Lint` [41] and `FindBugs` [42], `Code Inspection`. But `LeakDroid` has been inaccessible from the website (page not found error) and, we have emailed its author and received no response. Moreover, we have tried to use fuzzing tools, such as `Monkey` [43], to trigger the exceptions caused by these misuses, and they work on the self design app with enough input events and time consumption but struggle on the real world app with more complex interaction. Finally, we select `AsyncChecker` [6] as the comparison tool, which supports misuse detection of the asynchronous component `AsyncTask` with the path-sensitive approach, and we adapt it for misuse detection of threads.

B. *Effectiveness of* `Leopard`

We conduct experiments on the dataset to evaluate the error detection ability of `Leopard`. Table II shows the misuses detected in Java programs (total: 285) and Table III shows the result in Android Apps (total: 723). We have found that the proportions of the misuses of INR (176/285=61.2%) in Java programs and Android apps (94/723=13%) are totally different. Through the statistics of the number of events, we have found the reason is that there are more thread interruption calls in the Java program compared to the Android app, and the Java programs prefer to reuse the thread object allocation statement with different `Runnable` tasks. This shows that Java program developers pay more attention to thread maintenance. For the misuses of HTR, they appear much more often in Android apps (333/723=46.1%) than Java programs (79/285=27.7%). It is probably because Android threads easily reference typical destructible objects in Android, such as `Activity` and `View`. And for the misuses of NTT, it is similar with HTR because the interruption event and the destroy event always need to be invoked in upper level application as the Java program may be used as a third-party library.

Case Study. We will use a case study to illustrate the effectiveness of our approach. Listing 2 shows code segments from a real app with the package name *namlit.siteswapgenerator*. There is a method *onOptionsItemSelected (...)* to respond to the menu event from the class `MainActivity` (line 2). It responds differently depending on the different items in the menu (line 3-7). One of the responses is to save parameters via a method *saveGenerationParameters ()* invocation (line 6) declared below (line 8). This method creates a dialog instance whose class is `SaveGenerationParametersDialog` interacting with the user finally (line 12). After a series of interactions and method invocations, the *doSthInDatabase ()* method of the class `SaveGenerationParametersDialog` will be called finally. An anonymous thread is created to perform database-related asynchronous tasks in it (line 14-16). This task is carried out by an anonymous inner class that implements the `Runnable` interface. A non-static anonymous inner subclass of the `Runnable` will hold an instance reference to the class `SaveGenerationParametersDialog`. The latter one inherits from `View` which is a destructible object. Hence a HTR misuse occurs. As we can see, there are many `else if` blocks in it and the path-sensitive approach can cause a false negative due to the path explosion. However, according to our approach, only two events, the initialization of the thread (line 14) and the assignment of its `DialogFragment` field (implicit), need to be concerned, and a large number of unrelated code and paths do not need to be explored more than once. In fact, 75.6% of classes involved in the HTR misuses are non-static anonymous inner sub-classes of `Thread` and `Runnable`. They contain the implicit object reference to the outer class that is transparent for developers in the source code, which can be avoided by a better practice.

The path-sensitive approach will explore a large number of branches: on the one hand, the exploration is complex as many branches will cause the path explosion; on the other hand, there is a thread related operation so that the method *onOptionsItemSelected (...)* will not be skipped, at this time the

Listing 2. A Misuse Detected by Leopard under Path Explosion

```
1  class MainActivity extends AppCompatActivity{
2   public boolean onOptionsItemSelected (MenuItem
        paramMenuItem) {
3   int i = paramMenuItem.getItemId ();
4   if (i == 2131296281) {...}
5   else if (i == 2131296290) {
6    saveGenerationParameters (); // save
7   } ...}//7 more else if blocks are omitted here
8   public void saveGenerationParameters () {
9   ...// omitted
10   (new SaveGenerationParametersDialog ()).show (...)
        ;}
11 }
12 class SaveGenerationParametersDialog extends
       DialogFragment {
13  public void doSthInDatabase () {
14   (new Thread ( ()->{
15    ... //time-consuming task omitted
16  }).start (); ... }//omitted
```

pruning strategy will not work. Therefore, the path-sensitive solution will yield a false negative for this misuse detection under limited time or number of paths. For our approach, the thread related point in the method *onOptionsItemSelected (...)* is only the method *saveGenerationParameters ()* invocation (line 7). All other thread-independent invocations and statements have been ignored after being explored once. Eventually we have found the HTR misuse, which is a potential resource leak, reducing false negatives.

Answer to RQ1. `Leopard` can be used not only for Android apps (723 misuses), but also for large Java programs (285 misuses) effectively. Java program developers pay more attention to the thread maintenance. The anonymous inner subclasses of `Thread` and `Runnable` in destructible classes play an important role among the HTR misuses which can be improved with better programming practice.

C. Comparison on Apps

In this section, we will compare `Leopard` to the existing approach in [6] which is path sensitive, via (1) the misuse number and detection time, (2) the intermediate information (event covered during analysis) and (3) manual check for precision and recall.

Misuses and Time. We configure `AsyncChecker` with two different timeout configurations: 5 minutes and 30 minutes. The result is shown in Table III where 18 (14) represents 18 misuses detected in 14 APKs. We can see that `AsyncChecker` has detected 42 misuses in 15 apps under 5 minutes timeout and 47 misuses in 15 apps under 30 minutes timeout, while `Leopard` has detected 723 ones in 103 apps. The increase in time does not allow `AsyncChecker` to detect misuse on more apps and there is one app whose time consumption does not reach the timeout threshold under both configurations. Although there is an increase of misuses detected in the 15 apps, the increment with much more time consumption (600%), is a small proportion (0.7%) compared to the result of `Leopard`. In addition, `Leopard` has significant advantages in terms of time consumption. For the apps, the average time it takes is about 60s while the maximum time is 405s. And among the detection result

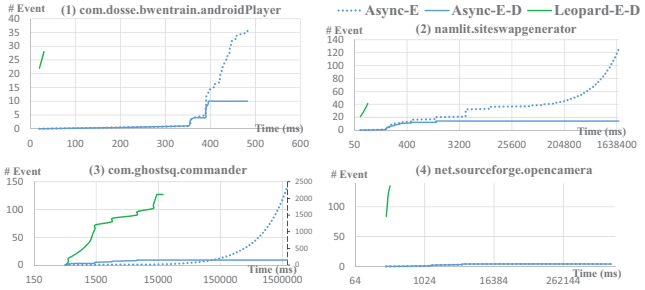

Fig. 4. Event Coverage. Async-E means the number of events covered during detection of `AsyncChecker`. Async-E-D and Leopard-E-D mean the number of events covered during detection of `AsyncChecker` and `Leopard` after deduplication, respectively. The second vertical axis on the right in the sub-figure (3) is for Async-E only.

of `AsyncChecker`, only 12 (8.2%) apps do not reach 30 minutes timeout threshold.

Time Complexity Analysis. The reason why our approach is so efficient is that our approach has an acceptable polynomial time complexity. We first analyze the time complexity of each step. We use *C*, *M*, *S* to denote the number of classes, methods, and statements in a program respectively. Typically, *C* is smaller than *M*, and both of them are much smaller than *S*, which means that the power complexity of *C* and *M* is also less than *S*. **(1)** Event extraction is determined by a few method invocations, which is linearly related with the number *S* of statements. **(2)** For each *run()* method, the algorithmic complexity of loop recognition is O(*N*+*K***E*), where *N* is the node number, *E* is the edge number in its CFG and K is unstructuredness coefficient which is usually small [33]. The complexity of this step is approximately O(*S*) as they are much smaller than *S*. **(3)** Standard alias analysis has cubic time complexity. **(4)** Happens-before determination. Compared to the previous steps, it is more complex. In the worst case, each determination needs to traverse the super control flow graph of the entire program. At this point, the complexity is O(*S*+E_{super}), where E_{super} is the number of edges of the hypergraph, which will not exceed S^2. So the complexity of this step is O(S^2). In summary, our approach has a time complexity about O(S^4) - O(S^5) on average, and O(S^7) in worst case, which is far superior to exponential overhead.

Coverage of Events. To verify whether the event based approach in this paper matches the demand of the misuse detection, we record events covered during the detection of the tools. The events contain E_{Init}, E_{Start}, $E_{Interrupt}$ and implicit *run()* method invocation of a thread. Once the invocation statements of these methods or the methods themselves are analyzed, the time and event will be recorded. In the end, we select 4 apps to display that `AsyncChecker` can detect the most misuses, containing *com.dosse.bwentrain.androidPlayer*, *namlit.siteswapgenerator* , *com.ghostsq.commander* and *net.soucefor-ge.opencamera* . Figure 4 shows the relation of the number of covered events and detection time for both `Leopard` and `AsyncChecker`. Async-E means the number of events covered during detection of `AsyncChecker`. Async-E-D and Leopard-E-D mean

TABLE IV
DETECTION RESULT COMPARISON BETWEEN ASYNCCHECKER AND LEOPARD ON 15 ANDROID APPS

App Package Name	AsyncChecker									Leopard								
	HTR			INR			NTT			HTR			INR			NTT		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
ch.hgdev.toposuite	1	0	3	0	0	0	1	0	3	4	0	0	0	0	0	4	0	0
com.dosse.bwentrain.androidPlayer	1	0	0	5	0	0	1	0	1	1	0	0	3	1	2	1	0	1
com.ghostsq.commander	1	0	1	1	0	2	1	0	1	2	0	0	3	0	0	2	0	0
com.github.axet.audiorecorder	0	0	0	1	0	2	0	0	0	0	0	0	3	0	0	0	0	0
com.jovial.jrpn	1	0	1	1	0	0	0	0	2	2	0	0	1	0	0	2	0	0
com.spisoft.quicknote	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
de.reimardoeffinger.quickdic	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0
de.rochefort.childmonitor	2	0	0	0	0	0	1	0	1	2	0	0	0	0	0	2	0	0
godau.fynn.usagedirect	1	0	1	0	0	0	1	0	1	2	0	0	0	0	0	2	0	0
godau.fynn.usagedirect.system	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0
menion.android.whereyougo	1	0	1	0	0	1	1	0	1	2	0	0	1	0	0	2	0	0
namlit.siteswapgenerator	3	0	6	0	0	0	3	0	6	9	0	0	0	0	0	9	0	0
net.justdave.nwsweatheralertswidget	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0
net.sourceforge.opencamera	2	0	0	1	0	0	2	0	0	2	0	0	1	0	0	2	0	0
xyz.myachin.downloader	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0
Sum	18	0	13	9	0	6	15	0	17	**31**	0	0	**13**	1	2	**31**	0	1

the number of events covered during detection of AsyncChecker and Leopard after deduplication with their hash codes, respectively. The second vertical axis on the right in the sub-figure (3) is for Async-E only. With the configuration of 30 minutes timeout for each app, AsyncChecker covers 37 events while Leopard covers 232 events in much shorter time. Under the best case, AsyncChecker can cover 15.9% (37/232) of events. And Leopard can cover 86.2% (232/(37+232)) even in the worst case. Sub-figure (3) in Figure 4 reveals the reason why the recall of AsyncChecker rate is low. Although it covers 2,357 accumulated events, it covers only 9 unique ones. That is to say, it is trapped by paths that include thread events without any misuse, which can be avoided by Leopard.

Precision and False Positives. To evaluate how precise Leopard is, we have counted the false positives and false negatives of the two approaches manually in 15 apps where AsyncChecker can detect misuses. A total of 78 misuses are involved, excluding the misuses unreadable, such as the ones caused by code obfuscation. We take all the results detected by the two tools as the complete set to calculate the false negatives. The result is shown in Table IV. The precision, recall and F_1 value of AsyncChecker are 100%, 53.8%, 0.7, respectively, while those of Leopard are 98.7%, 96.1%, 0.974, respectively. The recall of AsyncChecker is inferior to the value it claims, because it is difficult to detect false negatives for the work compared to [7], distorting the recall. One among the three false negatives is because Leopard does not take the class's initialization method ("⟨*clinit*⟩ *()*") into account, and the other two are because it does not do a full point-to analysis causing the edge missing in the call graph. The result shows that our approach is able to greatly reduce false negatives.

Answer to RQ2. Leopard can detect the misuses efficiently via covering more misuse related events within much shorter time based on event analysis, reducing lots of false negatives while maintaining a high precision.

D. Feedback from Developers

We have reported the issues to the developers Leopard has detected. For Android apps, we exclude the apps which have been archived, have no issue track, require a complete pull request, whose source code cannot be public accessible or have not been updated for more than two years. Finally, we have reported issues for 39 applications and received 22 feedback so far. Two of the feedback claim that their thread object lives for a short period of time and would not care about them, and one issue has been closed as not in their repair plan. The remaining 19 feedback have confirmed the issues involving 66 misuses (HTR: 59, INR: 6, NTT: 1). Among them, 21 misuses have been fixed already. The confirmed issues are detailed in Table V. As we can see, most of them are popular and used by many people, as indicated by real downloads on Google Play, or has many forks and stars in Github. For example, *Ghost Commander* [51] has been downloaded over a million times from Google Play, which is a dual-panel file manager and *FreeRDP* has been forked more than 23,500 times with over 8,600 stars in Github, which is an open source and free implementation of the Remote Desktop Protocol (RDP). Developers pay more attention to HTR than the other two misuse patterns, probably because it is more likely to occur due to the programming habit or the memory constraint on mobile devices directly. A typical misuse of HTR is the abuse of Activity as Context or a reference to a destructible object is unintentionally generated in a non-static inner basic thread. Another common misuse is to reference a complex object where referencing only its field is enough. We recommend developers to use an ApplicationContext with a longer lifetime instead of an Activity if there are no side effects on the program, reduce usage of non-static

TABLE V
CONFIRMED ISSUES IN REAL-WORLD APPS. (* FIXED)

App	Fork	Star	#Download	#Misuse	Issue ID
VocableTrainer [52]	10	27	500K+	1	93
toposuite[53]	2	12	5K+	4	3
APK-Explorer-Editor[54]	53	278	7.8K+	**1***	29
LRC-Editor[55]	9	43	100K+	3	35
Nextcloud[56]	1.5K	3.2K	100K+	7	10691
TRIfA[57]	52	220	5K+	14	350
AppManager[58]	174	2.3K	80K	1	854
Siteswap Generator[59]	3	13	1K+	9	55
TC Slim[60]	66	1.1K	10K+	2	36
blabber.im[61]	16	41	-	**6***	674
OSMDashboard[62]	8	52	500+	**1***	169
Ghost Commander[51]	-	-	1,000K+	**1***	93
Offline Puzzle Solver[63]	-	1	-	**1***	1
FitoTrack[64]	48	161	5K+	3	400
Conversations[65]	1.3K	4.2K	100K+	**2***	4366
monocles chat[66]	7	14	-	**6***	44
ccgt[67]	4	11	-	1	7
Notes[68]	121	769	10K+	**1***	1574
FreeRDP[69]	23.5K	8.6K	10K+	**2***	8158
Total	-	-	-	66 (**21***)	-

inner class of the basic thread and follow the principle of least reference.

For Java programs, we also have reported these issues to developers. Some issues have not received feedback because large Java projects have too many issues, and it may take a long time to deal with them for developers, such as cxf. Some issues are caused by other third-party libraries so we had a hard time finding their repositories. There is even a third-party library archived which is read only and its enhanced version is not based on its source code. As we take the public method as the entry method of the detection, the misuses confuse developers so that there is a feedback which hopes that we can provide a complete upper level application scenario or case. Compared to Android, the feedback on these projects is not as consistent as expected. The main reason is that these projects lack an obvious program entry method, while on Android, we can simulate its entry method better.

Answer to RQ3. Developers take these misuses as serious defects. Totally, 66 misuses have been confirmed by developers via our issue reports and 21 of them have been fixed already. Developers should try to follow the minimum reference principle to reduce unnecessary coupling or use the object with a longer life cycle as much as possible.

E. Threat to Validity

There may be other methods with destructible semantics which we do not take into account, resulting in false negatives. Correspondingly, the destroy methods we identified may not always be equivalent to the destroy semantics, causing false positives. In addition, we have found that Java programs may use the interrupt response via the exception *try-catch* rather than the *isInterrupted()* status check, which may cause false positives, although the latter responds much more quickly than the former. Furthermore, although Soot is powerful, it may cause false positives due to the suspicious edge in the CG, especially for Java programs without a definite entrance. Besides, we ignore the strict path reachability of misuses during the detection, relying on `FlowDroid`, which may result in potential false positives too. Moreover, the experimental environment may affect the accurate reproduction of the data.

VI. RELATED WORK

We introduce related works in two aspects: one is about the detection of the resource leak and the other is about asynchronous analysis.

Resource Leak. Static analysis is used to detect resource leaks [70]. A special research focuses on the memory leak such as Java collections [71]. There is a work trying to detect the problem of not closing resource object that inherits from `AutoClosable`, using accumulation analysis [31]. And `LeakDroid` and a memory leak detecting approach are based on test cases [39, 72], trying to find leak via test suites. A work constructs a benchmark with the bug oracle [38]. As a device platform with limited resources, the resource problem on Android has been studied by some researchers. The detection of Android resource leakage was proposed to mainly aim at the failure to close hardware-related object resources in time, such as cameras and sensors [40, 73–75]. A detailed empirical study conducted on 491 issues from 15 large open-source Java projects proposed taxonomies for the leak types, for the caused defects, and for the repair actions [76]. `COBWEB` and `ACETON` were present for energy testing in Android, especially considering the lifecycle and hardware state context [77, 78]. Compared to them, our work mainly focuses on the misuse of the basic thread, causing resource leak and waste such as the unnecessary execution of the CPU which is hard to be quantified and the block of GC.

Asynchronous Behaviour Analysis. Many of the thread related works have focused on data races, trying to detect the read or write to the same object that does not have an unambiguous happens-before relationship in different threads in the Java and Android platforms [8–22]. In addition, there are some works on Android apps according to the characteristics of the platform. `AsyncChecker` [6], inspiring this work, proposed five patterns to detect misuses about `AsyncTask`. We refine three of them as the patterns in this paper and ignore the other two patterns as `RepeatStart` rarely appears, and `EarlyCancel` will not cause any defects. In addition, the main difference between the three patters in this paper and [6] is that we define the destructible object in detail (instead of being limited to `Activity/View`) and generalize them to characterize the misuse of the basic thread, exposing more defects to be compatible with Android applications and Java programs. `APEChecker` combined static analysis and dynamic GUI exploration to find errors [2], Lin et al. developed a tool to refactor an `AsyncTask` into an `IntentService` due to its performance problems [3] and so on. Compared to them, we focus on the problem caused by the asynchronous threads rather than the data race. In addition, Android platform has gradually eliminated some asynchronous components provided by itself, such as `AsyncTask`, `AsyncTaskLoader`, `IntentService`, etc [25, 26, 29]. Compared to them, Java's

basic thread is a more fundamental asynchronous component used in not only Android apps but also Java programs, which is one of the motivations for our research.

VII. Conclusion

The basic asynchronous thread in Java is very useful, yet error-prone. In this paper, we propose a lightweight approach based on the event analysis to detect the misuses causing the thread related resource leak and waste. We characterize those destructible objects that are prone to mis-referencing and summarize clear rules to identify them. Based on the extraction and analysis of the thread event and the destroy event, we design efficient algorithms and develop a tool named `Leopard` to detect the defects. Experiments on real world programs and the feedback from developers show that our solution is effective and also efficient for the misuse detection. In the future, we will refine the hazard level caused by the mis-reference to the destructible object and extend this work to the improvement of performance for the thread pool in Java.

Acknowledgement

Thanks to Dr. Linjie Pan for the initial discussion for this paper, to Ms. Yajun Zhu and Prof. Yan Cai for comments on earlier drafts of this paper, and to the anonymous reviewers for their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of China (NSFC) under grants No.62132020 and No.62102405.

References

[1] TIOBE Index for August 2023. [Online]. Available: https://www.tiobe.com/tiobe-index/

[2] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, "Efficiently manifesting asynchronous programming errors in Android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 486–497.

[3] Y. Lin, S. Okur, and D. Dig, "Study and refactoring of Android asynchronous programming (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 224–235.

[4] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for Android applications through refactoring," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 341–352.

[5] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 1013–1024.

[6] L. Pan, B. Cui, H. Liu, J. Yan, S. Wang, J. Yan, and J. Zhang, "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*, 2020, pp. 952–963.

[7] Y. Kang, Y. Zhou, H. Xu, and M. R. Lyu, "DiagDroid: Android performance diagnosis via anatomizing asynchronous executions," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 410–421.

[8] S. Schulz, E. Herrendorf, and C. Bockisch, "Thread-Sensitive Data Race Detection for Java," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2021, pp. 32–42.

[9] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey, "RacerD: compositional static race detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–28, 2018.

[10] Y. Li, B. Liu, and J. Huang, "Sword: A scalable whole program race detector for java," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 75–78.

[11] B. Liu and J. Huang, "D4: fast concurrency debugging with parallel differential analysis," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 359–373, 2018.

[12] B. Swain, B. Liu, P. Liu, Y. Li, A. Crump, R. Khera, and J. Huang, "OpenRace: An Open Source Framework for Statically Detecting Data Races," in *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 2021, pp. 25–32.

[13] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey, "RacerD: compositional static race detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–28, 2018.

[14] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 162–180.

[15] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 308–319.

[16] C. Radoi and D. Dig, "Effective techniques for static race detection in java parallel loops," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, pp. 1–30, 2015.

[17] D. Wu, J. Liu, Y. Sui, S. Chen, and J. Xue, "Precise static happens-before analysis for detecting UAF order violations in android," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 276–287.

[18] P. Bielik, V. Raychev, and M. Vechev, "Scalable race detection for Android applications," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 332–348, 2015.

[19] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race detection for event-driven mobile applications," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 326–336, 2014.

[20] Y. Hu, I. Neamtiu, and A. Alavi, "Automatically verifying and reproducing event-based races in Android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 377–388.

[21] N. Salehnamadi, A. Alshayban, I. Ahmed, and S. Malek, "ER catcher: a static analysis framework for accurate and scalable event-race detection in Android," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 324–335.

[22] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for Android applications," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 316–325, 2014.

[23] B. K. Ozkan, M. Emmi, and S. Tasiran, "Systematic asynchrony bug exploration for Android apps," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 455–461.

[24] H. Tang, G. Wu, J. Wei, and H. Zhong, "Generating test cases to expose concurrency bugs in Android applications," in *Proceedings of the 31st IEEE/ACM international Conference on Automated software engineering*, 2016, pp. 648–653.

[25] AsyncTask. [Online]. Available: https://developer.android.google.cn/reference/android/os/AsyncTask

[26] IntentService. [Online]. Available: https://developer.android.google.cn/reference/android/app/IntentService

[27] Types of References in Java. [Online]. Available: https://www.geeksforgeeks.org/types-references-java/

[28] Thread. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

[29] AsyncTaskLoader. [Online]. Available: https://developer.android.google.cn/reference/android/content/AsyncTaskLoader

[30] WeakReference. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/ref/WeakReference.html

[31] M. Kellogg, N. Shadab, M. Sridharan, and M. D. Ernst, "Lightweight and modular resource leak verification," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 181–192.

[32] JSR250. [Online]. Available: https://jcp.org/aboutJava/communityprocess/mrel/jsr250/index3.html

[33] T. Wei, J. Mao, W. Zou, and Y. Chen, "A new algorithm for identifying loops in decompilation," in *International Static Analysis Symposium*. Springer, 2007, pp. 170–183.

[34] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context,

flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.
[35] Apache Tomcat. [Online]. Available: https://github.com/apache/tomcat
[36] Junit5. [Online]. Available: https://github.com/junit-team
[37] F-Droid. [Online]. Available: https://f-droid.org/en/
[38] Y. Liu, J. Wang, L. Wei, C. Xu, S.-C. Cheung, T. Wu, J. Yan, and J. Zhang, "DroidLeaks: a comprehensive database of resource leaks in Android apps," *Empirical Software Engineering*, vol. 24, pp. 3435–3483, 2019.
[39] D. Yan, S. Yang, and A. Rountev, "Systematic testing for resource leaks in Android applications," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 411–420.
[40] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang, "Light-weight, inter-procedural and callback-aware resource leak detection for Android apps," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1054–1076, 2016.
[41] Android Lint: A Code Scanning Tool for Android Apps. [Online]. Available: https://developer.android.com/studio/write/lint.html
[42] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.
[43] Monkey. [Online]. Available: https://developer.android.google.cn/studio/test/monkey
[44] Apache CXF. [Online]. Available: https://github.com/apache/cxf
[45] Hivemq. [Online]. Available: https://github.com/hivemq/hivemq-community-edition
[46] Eclipse Jetty. [Online]. Available: https://github.com/eclipse/jetty.project
[47] Micronaut-core. [Online]. Available: https://github.com/micronaut-projects/micronaut-core
[48] Quarkus. [Online]. Available: https://github.com/quarkusio/quarkus
[49] Spring Framework. [Online]. Available: https://github.com/spring-projects/spring-framework
[50] Wildfly. [Online]. Available: https://github.com/wildfly/wildfly
[51] Ghost Commander. [Online]. Available: https://sourceforge.net/projects/ghostcommander/
[52] VocableTrainer. [Online]. Available: https://github.com/0xpr03/VocableTrainer-Android
[53] toposuite. [Online]. Available: https://github.com/hgdev-ch/toposuite-android
[54] APK-Explorer-Editor. [Online]. Available: https://github.com/apk-editor/APK-Explorer-Editor
[55] LRC-Editor. [Online]. Available: https://github.com/Spikatrix/LRC-Editor
[56] Nextcloud. [Online]. Available: https://github.com/nextcloud/android
[57] TRIfA. [Online]. Available: https://github.com/zoff99/ToxAndroidRefImpl
[58] AppManager. [Online]. Available: https://github.com/MuntashirAkon/AppManager
[59] Siteswap Generator. [Online]. Available: https://github.com/namlit/siteswap_generator
[60] TC Slim. [Online]. Available: https://github.com/TrackerControl/tracker-control-android
[61] blabber.im. [Online]. Available: https://codeberg.org/kriztan/blabber.im
[62] OSMDashboard. [Online]. Available: https://github.com/OpenTracksApp/OSMDashboard
[63] Offline Puzzle Solver. [Online]. Available: https://gitlab.com/20kdc/offline-puzzle-solver
[64] FitoTrack. [Online]. Available: https://codeberg.org/jannis/FitoTrack
[65] Conversations. [Online]. Available: https://github.com/iNPUTmice/Conversations
[66] monocles chat. [Online]. Available: https://codeberg.org/Arne/monocles_chat
[67] ccgt. [Online]. Available: https://github.com/pterodactylus42/ccgt
[68] NextCloud Notes. [Online]. Available: https://github.com/nextcloud/notes-android
[69] FreeRDP. [Online]. Available: https://github.com/FreeRDP/FreeRDP
[70] D. Yan, G. Xu, S. Yang, and A. Rountev, "Leakchecker: Practical static memory leak detection for managed languages," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014, pp. 87–97.
[71] G. Xu and A. Rountev, "Precise memory leak detection for Java software using container profiling," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 3, pp. 1–28, 2013.
[72] M. Ghanavati and A. Andrzejak, "Automated memory leak diagnosis by regression testing," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 191–200.
[73] J. Liu, T. Wu, J. Yan, and J. Zhang, "Fixing resource leaks in Android apps with light-weight static analysis and low-overhead instrumentation," in *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 2016, pp. 342–352.
[74] Y.-M. Tseng, J.-L. Chen, and S.-S. Huang, "A lightweight leakage-resilient identity-based mutual authentication and key exchange protocol for resource-limited devices," *Computer Networks*, vol. 196, p. 108246, 2021.
[75] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury, "Energypatch: Repairing resource leaks to improve energy-efficiency of Android apps," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 470–490, 2017.
[76] M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak, "Memory and resource leak defects and their repairs in Java projects," *Empirical Software Engineering*, vol. 25, pp. 678–718, 2020.
[77] R. Jabbarvand, J.-W. Lin, and S. Malek, "Search-based energy testing of android," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1119–1130.
[78] R. Jabbarvand, F. Mehralian, and S. Malek, "Automated construction of energy test oracles for Android," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 927–938.