Extended Atomicity Checking with Blame Assignment for Android Applications

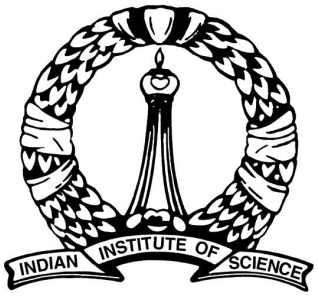
A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Master of Engineering

IN
COMPUTER SCIENCE AND ENGINEERING

by

Priyanka Mondal



Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

JUNE 2015

DEDICATED TO

My Mother

for continuous support and encouragement

Signature of the Author: .....

Priyanka Mondal
Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore

Signature of the Thesis Supervisor: .....

Aditya Kanade
Assistant Professor
Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore

Acknowledgements

I am deeply grateful to Dr.Aditya Kanade for his incomparable guidance. I have been extremely lucky to work under his supervision. He has always been a source of inspiration to me.

I would like to thank Pallavi Maiya for her guidance and support through out this project. It had been a great experience to work with her.

My thanks and appreciations also go to my fellow lab mates for all the help and suggestions. I would also like to thank Arti Bhat for all those long discussions. I thank my friends who made these two years in IISc so fun and exciting.

Finally, I would like to express gratitude to my parents for their inspiration and love that no amount of thanks can be sufficient. This project would not have been possible without their constant support and encouragement.

Vita

IISc was established in 1909 with active support from Jamsetji Tata. It is locally known as the "Tata Institute". IISc is widely regarded as India's one of the finest institutions. IISc celebrated its Golden Jubilee in 2008 with a great fanfare.

Abstract

Smartphones have become an integral part of our everyday life. Nearly everyone has at some point experienced difficulties while using smartphone applications. Most of the high severity bugs in Android apps are due to concurrency issues. However concurrency bugs are difficult to find and fix because of the non-determinism of interleavings. Android environment includes both multi-threading and asynchrony. This makes locating the concurrency bugs even harder.

Out of different types of concurrency bugs, atomicity violations are particularly hard to isolate. In addition atomicity violation has not received any attention in case of Android applications. Here we present an algorithm, which is the first work to detect atomicity violations in Android applications. We detect atomicity violations based on the *happens-before* rules we have defined for Android Concurrency model. These *happens-before* rules has been extended from Velodrome, where these rules are used for detecting atomicity violation in multi-threaded programs. However, we extend this idea for multi-threaded-event-driven platform. We have implemented two types of atomicity check, one is tree based atomicity check, and the other one is lifecycle based atomicity check. In this report we talk about lifecycle based atomicity check in detail. Our proposed algorithm finds *cycles* in the traces generated by running Android apps and classifies the reported cycles into four groups.The algorithm assigns *blame* and also filters duplicate cycles. We ran our algorithm on our benchmark apps and then on 48 real world apps including some popular apps like GMail, Wikipedia, My Tracks and got warnings in 21 apps.

Keywords

Android, Atomicity, Lifecycle

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Smartphones and the applications running on them continue to grow in popularity [4] and revenue [2]. Around 40% of smartphone owners use their smartphones before they have even got out of bed [5]. This increase is shifting client-side software development and use, away from traditional desktop programs and towards smartphone apps [7, 11]. According to the results in [22], almost everyone has at some point experienced difficulties with attempting to use smartphone apps. A Study says that around 66% of high-severity bugs in Android apps are due to concurrency issues [3]. Concurrency bugs can be classified into four catagories: atomicity violations, order violations ("An order violation occurs if a programming assumption on the order of certain events is not guaranteed during the implementation" [13, 20]), data races, and deadlocks. Of these bugs, atomicity violations are particularly hard to isolate [16].

Android is one of the fastest growing smartphone platforms. Android has 69% of the smartphone market[23]. Android environment exposes a concurrency model that combines both multi-threading and event-driven property. Each Android app runs on their own process. One app can run many threads concurrently. These threads can have task queues associated with them. In these queues tasks(a task is a collection of activities or events that users interact with when performing a certain job) are posted. A thread can post task to itself or to other threads. Tasks are dequeued from the front of the task queue except the tasks which are posted with a delay. This means that in a process multiple threads can run concurrently and at the same time these threads can post tasks to each other including themselves.

In event-driven programming, code is executed upon activation of events. An event can be defined as a type of signal to the program that something has happened. The event is generated by external user actions such as swipe, clicks on buttons, incoming sms, or by the operating system, such as a timer [11].

Use of multi-threaded-evet-driven approach makes the system faster but at the same time

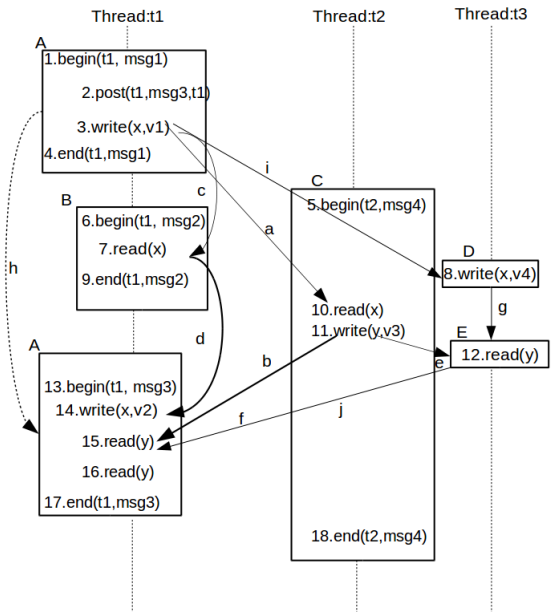


Figure 1.1: Example trace.

many subtle concurrency bugs can hide because of this complex programming model. There is non-determinism in interleaving of threads and as well as in the order of posting tasks (both the words "task" and "message" indicate events). A particularly insidious type of concurrency bug is atomicity violations [16]. Our work presents an analysis for detecting atomicity violations in Android programming environment. This analysis combines ideas from the papers Velodrome [9] and DroidRacer [15].

Our algorithm is designed based on some rules. These rules reasons about the exact dependencies between instructions in the trace of some app and reports error message if the observed trace is not serializable.

Let us illustrate our analysis with the example trace of fig.1.1. In this report we use the words node and transaction interchangeably. A task or a set of tasks is atomic when they are serializable with respect to other tasks of the same thread and the instructions of other threads [6]. We define atomicity in details in section 2. In the fig.1.1 *msg1* and *msg3* are posted to thread *t1*, and they belong to same atomic block. Whereas *msg2* and *msg4* are posted to thread *t1* and thread *t2* respectively and are part of two other atomic blocks but not part of the atomic block to which *msg1* and *msg3* belong. Atomic blocks are indicated by rectangular boxes, which we refer as transactions [9]. *msg1* and *msg3* belong to two different rectangular box as they are two different tasks though they belong to same atomic block. Instructions outside an atomic block execute in their own unary tranaction. As *msg1* and *msg3* belong to same atomic block they belong to same transaction. To analyze whether a transaction is

atomic or not we have to check whether it is serializable or not. Based on our concurrency semantics [6] we define *happens before* relation($\lesssim$) over the instructions of the trace. There can be two types of *happens before* relation. One is *may be happens before* and the other is *must be happens before* realtion. Two instructions x and y are realted by *may be happens before* relation i.e. $x \lesssim y$ if x occurs before y in the current trace but in some other trace the order may switch. If x always comes before y in all possible traces then we say x and y are related by *must be happens before* realation. Both of the *happens before* realtions are denoted by $\lesssim$. We distinguish them by understanding the context of the instructions. Edge h in fig.1.1 is a *must be happens before* edge, which comes because of the post of *msg3* from *msg1*. Edges *a,b,c,d* are *may be happens before* edges.

Let, *msg1* and *msg3* have transaction id A (as they belong to the same atomic block) and *msg2* and *msg4* have transaction id B and C respectively. The two unary transactions of thread *t3* are transaction D and E respectively. From the given trace our analysis infers that $msg1 \lesssim msg4$ (via write-read edge on x), $msg4 \lesssim msg3$ via(write-read edge y). As *msg1* and *msg3* belong to same transaction our analysis detects a cycle, $msg1 \lesssim msg4 \lesssim msg3$, or say $A \lesssim B \lesssim A$ which is a multi-threaded cycle. Our analysis also detects the single-threaded cycle, $msg1 \lesssim msg2 \lesssim msg3$ i.e. $A \lesssim C \lesssim A$. These tells that the given trace is not serializable and that is why not atomic.

We are the first to introduce the concept of single-threaded atomicity violation and we have observed that it may even lead to crash of apps in some cases. Single threaded atomicity takes place because of the non-determinism of the posts of events from Android framework and cross thread posts.

We have also added **Blame assignment** like [9] to localize the errors. Like in our example we have got two cycles. $A \lesssim B \lesssim A$ and $A \lesssim C \lesssim A$. In both cases we can see that the paths interleave transaction A with other conflicting instructions and so there is no equivalent trace where A can execute serially or say atomically. So transaction A is blamed for this atomicity violation. For the cycle $A \lesssim B \lesssim A$, blame assignment will tell that instruction 3,7,14 are involved. For the cycle $A \lesssim C \lesssim A$, blame assignment will tell that instruction 3,10,11,15 are involved in it (refer fig.1.1).

We classify the cycles in four groups. The cycle $A \lesssim B \lesssim A$ is a **cycle of priority 1** as instructions of a single thread are involved in the cycle. The cycle $A \lesssim C \lesssim A$ is a **cycle of priority 2** as instructions of more than one thread(thread *t1* and *t2*) are involved in the cycle. This classification will be discussed in detail in section 3.

Our algorithm also **filters** duplicate cycles. As in fig.1.1 the cycle with edges **a** and **f** will be filtered by our algorithm as it is duplicate to the cycle with edges **a** and **b** i.e. the cycle

$A \lesssim C \lesssim A$.

Summary. This report talks about the following works:

- We define atomicity for multi-threaded-event-driven environment. We present the first analysis of atomicity violation for multi-threaded-event-driven programs. In this report we talk about the analysis for lifecycle based atomicity check.
- By adding *happens before* edges we get cycles. For each cycle our analysis blames one transaction and also outputs the cycle i.e. the particular instructions involved in the cycle.
- We classify(prioritize) the cycles in four groups.
- We also filter out the duplicate cycles.
- On a range of benchmarks our analysis detects almost all non-atomic (non-serializable) transactions.
- We checked 48 real world apps and got cycles in 21 apps.
- We also have validated many cycles and got some true positives.

Outline. The report is arranged as follows. In chapter 2 we have talked about android components and in 3 lifecycle based atomicity is defined. In chapter 4 we have discussed about the implementation of our proposed algorithm, blame assignment, priority based classification of cycles, optimization done in the algorithm. In chapter 5 we have presented one of our benchmark example, in which we have seeded bug and our algorithm detects it perfectly. Chapter 6 describes about evaluation and results got by running the algorithm on several apps. In chapter 7 we mentioned about future works .Chapter 8 talks about related works and section 8 talks about future work.

Chapter 2

Android Components

Components are the essential building blocks of an Android application. These components are loosely coupled by the application manifest file that describes each component of the application and how they interact. There are four components that can be used within an Android application. Those are Activity, Service, Broadcast Receiver and Content Provider. Lifecycle of a component consists of several callbacks. Callback is a piece of code passed as an argument to another piece of code to do some job. We consider all the callbacks of a component as atomic. To define the notion of lifecycle based atomicity, first we need to describe about the components of android platform.

2.1 Activity

"An activity represents a single screen with a user interface" [1]. For example, WhatsApp might have one activity that shows a list of contacts, another activity to chat with someone, where we can see one text box to type message. Each activity is independent of the others, though the activities work together for an application. A different application can start any one of these activities (if WhatsApp allows it). For example, a *Photos* application can start the activity in WhatsApp that sends photos to some contact.

The lifecycle of activity component looks like fig. 2.1. Here the firm edges indicate *must be happens before* and the dotted edges indicate *may be happens before* relations.

2.2 Service

"A service is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface" [1]. For example, a service might download a song in the background while the user is using a

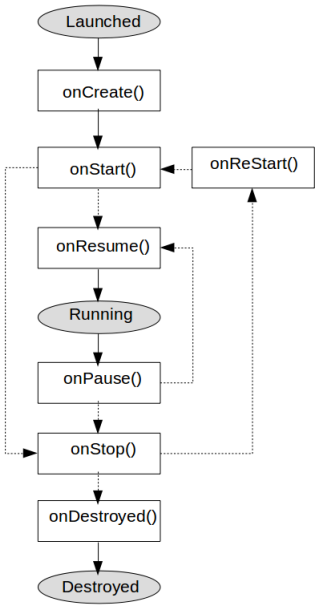


Figure 2.1: LifeCycle of Activities.

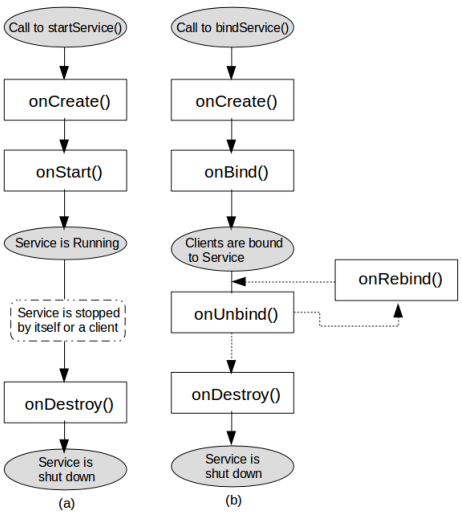


Figure 2.2: LifeCycle of (a)Unbounded Service (b)Bound Service

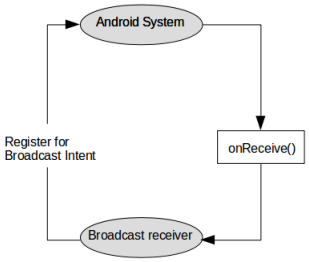


Figure 2.3: LifeCycle of Broadcast Receivers

different application, or it might fetch data over the network (e.g. current location) without blocking user to interact with some activity. Another component(e.g. Activity) can start a service (startService) and let it run or bind (bindService) to it in order to interact with it. The lifecycle of startService and bindService looks like fig.2.2(a) and fig.2.2(b) respectively.

2.3 Broadcast Receivers

"A broadcast receiver is a component that responds to system-wide broadcast announcements" [1]. Many broadcasts originate from the framework. For example, a broadcast which announces about incoming call, so that other activities call onPause() and the activity for incoming call comes in foreground, the battery is low, or a picture was captured etc.

Broadcast receivers do not have user interface. However they may create status bar notifications, or pop up dialog box to alert the user when a broadcast events occur. Broadcast Receiver can even initiate a service to perform some work based on some event [1].

2.4 Content Providers

A content provider component supplies data from one application to others on request. We have not considered this component in our analysis.

Chapter 3

Lifecycle based Atomicity

In fig.3.1 onCreate() and onResume() belong to callbacks of same activity and msg3 is posted from onResume() to the same thread.

In tree based atomicity analysis we will get two atomic blocks. One is an atomic block only with the onCreate() message, and the other atomic block consists of the callback onResume() and msg3. However in lifecycle based atomicity analysis, all the three messages will be considered as one single atomic block. We consider all the callbacks of a component to be part of same atomic block. So onCreate() and onResume() will belong same atomic block. We also consider all the messages posted from a callback message to the same thread as part of that atomic block. So, msg3 will belong to the same atomic block as that of onResume().

The instrumented semantics for tree-based atomicity is already described in [6].We need to change a few things to make those set of rules compatible for lifecycle based atomicity check. We have used the data-structures $\mathcal{L}, \mathcal{W}, \mathcal{R}, \mathcal{U}, \mathcal{H}, \mathcal{C}, \mathcal{P}, \mathcal{F}, \mathcal{T}, \mathcal{Q}$ in this and the following sections.

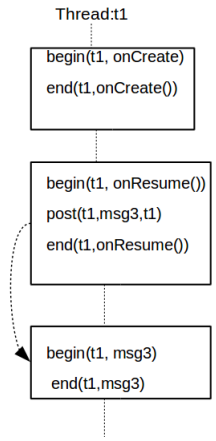


Figure 3.1: Diffrence between tree based atomicity and lifecycle based atomicity.

These are described in detail in [6].
$\mathcal{C} : Tid \rightarrow Node_{\perp}$ identifies the current transaction node corresponding to each thread
$\mathcal{L} : Tid \rightarrow Node_{\perp}$ identifies the transaction that executed last operation on the thread
$\mathcal{U} : Lock \rightarrow Node_{\perp}$ identifies the last transaction, if any, that released each lock
$\mathcal{R} : Var \times Tid \times root \rightarrow Node_{\perp}$ in case of lifecycle based atomicity, identifies the last read of each variable in each component of each thread. If the thread is without queue then it identifies the last transaction of the thread that read from each variable.
$\mathcal{W} : Var \rightarrow Node_{\perp}$ identifies the last transaction to write to each variable
$\mathcal{H} \subseteq Node \times Node$ is *happens-before* relation on transitions
$\mathcal{Q} : Tid \rightarrow q$ defines queue state of each thread where q denotes event queue associated with a thread.
$\mathcal{F} : Tid \rightarrow Node_{\perp}$ till *threadexit*, identifies the node from which thread is forked and after that identifies the node where *threadexit* was executed for the thread
$\mathcal{T} : Tid \rightarrow 2^{Trees} \cup null$ identifies the trees associated with each thread, if no tree is associated with a thread it is mapped to null
$\mathcal{P} : TaskId \rightarrow Node_{\perp}$ identifies the node from where the task was posted

Node, Var, Tid, Lock are sets of all node ids, all variables, all thread ids, all lock objects respectively and defined in [6]. $Node_{\perp}$denotes $Node \cup \{\perp\}$

The getNode(A) function in tree based atomicity check returns the node id of the task from where A was posted and $\perp$ if A is the root. In lifecycle based atomicity check we consider all the callbacks of a component as part of same atomic block. In case of lifecycle based atomicity check we use function getId(A), where A is a message which belongs to some activity. getId(A) first checks component instance of A and if that component instance is already seen it will return the node id corresponding to that component instance. If A is the first message of some component then in that case getId(A) returns $\perp_l$ and a fresh node id will be assigned to that message. This node id will be remembered corresponding to the component instance of A by the method setId(A). All the messages of a component has same component instance. So in future that node id will be returned by getId(Y) method for all messages Y of that component. The messages which are posted from any callback of some component will also get same node id corresponding to the component instance of the component.

[INS-ENTER-LIFE]

$\mathcal{C}(t) = \perp$

if d = -1 or $getId(A) = \perp_l$ then n is fresh and $setId(A) = n$

else $n = getId(A)$

$\mathcal{C}^{'} = \mathcal{C}[t = n]$

$\phi \overset{beginAt(A,t,d)}{\Longrightarrow} \phi[\mathcal{C} = \mathcal{C}']$

In the rule below *LifeId* is the set of component instances of all components seen so far. It is a dynamic set. Whenever a new component instance is seen it is added to this set.

[INS-WRITE-LIFE]

$n = \mathcal{C}(t) \quad n \neq \perp$

$\mathcal{W}' = \mathcal{W}[x := n]$

$\mathcal{H}' = \mathcal{H} \uplus (\{(\mathcal{R}(x, t', rl), n) | t' \in Tid, rl \in LifeId\} \cup \{(\mathcal{W}(x), n)\})$

$\phi \overset{wr(t,x,v)}{\Longrightarrow} \phi[\mathcal{W} = \mathcal{W}', \mathcal{H} = \mathcal{H}']$

The function *get_component_instance*(n) returns component instance corresponding to node id n

[INS-READ-LIFE]

$n = \mathcal{C}(t) \quad n \neq \perp \quad rl = get\_component\_instance(n)$

$\mathcal{R}' = \mathcal{R}[(x, t, rl) := n] \quad \mathcal{H}' = \mathcal{H} \uplus \{(\mathcal{W}(x), n)\}$

$\phi \overset{read(t,x,v)}{\Longrightarrow} \phi[\mathcal{R} = \mathcal{R}', \mathcal{H} = \mathcal{H}']$

All the other rules are same as rules defined in [6] in section 4.

All the callbacks of the Activity component i.e onCreate(), onStart(), onPause(), onResume(), onDestroy() are considered to be part of same atomic block. A message whose root is any callback message of any component is also considered to be part of same atomic block as that of the root.

All the service callbacks are considered to be part of same atomic blocks. A component can have both type of services (startService and bindService) running at the same time. The problem here is, same onCreate() and onDestroy() callback is shared by both startService and bindService. And if there are more than one bindService running they will share the same onBind() and onUnbind() callbacks. The onCreate() is executed when startService() or bindService() is issued by a client. If a service is already created and a startService or bindService

is issued then onCreate() is not executed. So the onStart() or onServiceConnected() callbacks will become part of the onCreate() which was created before. The client executes onStart()(for startService) and onServiceConnected()(for bindService) after it receives a reference to service object. If bindservice is already created and client issues bindService again then onBind() will not be executed again. So, for Service component the atomic blocks are a bit overlapped. Here for a new transaction a new transaction id is assigned and at the same time in a separate data-structure it memorizes the overlapped transaction ids.

Broadcast Receiver component has only one callback message, onReceive(). So atomic block for Broadcast Receiver component consists of only one message.

Chapter 4

Implementation

Our implementation is based on the instrumented semantics defined in section 4 of [6]. We first generate traces by running the app we want to test with the help of DroidRacer tool [15]. This tool is used for race detection in Android apps, but we have disabled the race detection part and are only using the trace generation part. We have generated at most 10 traces per app, each with at most 8 events. We implemented our algorithm in Java.

The algorithm for tree based atomicity and lifecycle based atomicity would be almost same with a little difference. For tree based atomicity all the tasks belong to some tree. So all the tasks are annotated with a proper depth. The concept of depth is defined in [6]. In lifecycle based atomicity checking all the callbacks of a component are part of same atomic block. Here the concept of depth is not required.

4.1 Algorithm

Alogorithm 1:Algorithm for checking Lifecycle based Atomicity

Input: Preprocessed trace.

Output: Cycles with blame assignment.

n = $\perp$, $EdgeStrore = \mathcal{E}$, $Edge\ e = null$, $String\ inst = get\_next\_instruction()$
while $inst! = null$ **do**
  if n=$\perp$ and $!inst.contains(beginAt)$ **then** $n = get\_fresh\_nodeId()$
  if $inst.equals(beginAt(A,t,d))$ and $getId(A) = \perp$ **then** $n = get\_fresh\_nodeId()$, $setId(A,n)$, $C(t) = n$
  else if $inst.equals(beginAt(A,t,d))$ and $getId(A)$!= $\perp$ **then** $n = getId(A)$, $C(t) = n$
  else if $inst.equals(endAt(A,t,d))$ **then** $C(t) = \perp$
  else if $inst.equals(begin(A,t))$ **then** $Q(t) = Q(t) \ominus A$, $L(t) = n$
  else if $inst.equals(Post(t',A,t))$ **then** $Q(t) = Q(t) \oplus A$, $P(A) = n$,
    if $t \neq t'$ **then** $T(t') = T(t') \cup CreateTree(A)$
    else $e = createTreeEdge(getCurrTask(t'), A)$, $T(t') = T(t') \cup getTree(t',n) \otimes e$
  else if $inst.equals(fork(t,tid))$ **then** $F(tid) = n$
  else if $inst.equals(threadInit(t))$ **then** $list = F(t) \cup H.get(F(t))$, $H.put(n,list)$, $e = createEdge(F(t),n)$, $EdgeStore.put(e)$
  else if $inst.equals(threadExit(t))$ **then** $F(t) = n$
  else if $inst.equals(join(t,tid))$ **then** $list = H.get(F(tid))$, $e = createEdge(F(tid),n)$, $EdgeStore.put(e)$
    if list.contains(n) **then** $ReportCycleWithBlame(n)$;
    else $list = list \cup \{n\}$
    $H.put(n,list)$
  else if $inst.equals(attachQ(t))$ **then** $Q(t) = \top$
  else if $inst.equals(acquire(t,m))$ **then**
    if $U(m).getNodeId()! = n$ **then** $list = H.get(U(m))$, $EdgeStore.put(e)$,
      if $list.contains(n)$ and $Q(t) = \top$ **then** $ReportCycleWithBlame(n)$
      else if $Q(t) = \top$ or ($Q(t)! = \top$ and $U(m).getTid()! = tid$) **then** $list = list \cup \{n\}$, $e = createEdge(U(m).getNodeId(), n)$,
        $EdgeStore.put(e)$
      $H.put(n,list)$
  else if $inst.equals(release(t,m))$ **then** $U.put(m,(n,t))$
  else if $inst.equals(read(t,x,v))$ **then** $rl = get\_component\_id(n)$, $R(x,t,rl) = n$ $list = H.get(W(x))$
    if $H.get(W(x)).contains(n)$ **then** $ReportCycleWithBlame(n)$
    else $list = list \cup \{n\}$
    $H.put(n,list)$,$e = createEdge(W(x),n)$, $EdgeStore.put(e)$
  else if $inst.equals(write(t,x,v))$ **then** $list1 = H.get(W(x))$
    if $H.get(W(x)).contains(n)$ **then** $ReportCycleWithBlame(n)$
    else $list = list \cup \{n\}$
    $H.put(n,list1)$, $e = createEdge(W(x),n)$, $EdgeStore(e)$, $e = createEdge(W(x),n)$, $EdgeStore.put(e)$
    for all $t \in Tid$ **do**
      for all $rl \in Taskid$ **do** $list = H.get(R(x,t,rl)$
        if $H.get(R(x,t,rl)).contains(n)$ **then** $ReportCycleWithBlame(n)$
        else $list = list \cup \{n\}$
      $H.put(n,list)$, $e = createEdge(R(x,t,rl),n)$, $EdgeStore.put(e)$
    $W(x) = n$
inst = get_next_instruction()

4.2 Algorithmic Details

4.2.1 Description of the functions used

First we will discuss about the functions used in the algorithm.

- *get_next_instruction*() function fetches instruction one by one from the input trace file.
- $get\_fresh\_nodeId()$ returns a fresh integer number.
- $setId(A, n)$ first checks the component instance of A and then stores node id n corresponding to that component instance.
- $CreateTree(A)$ initializes a tree with task A as root.
- $createTreeEdge(a, b)$ function adds task a as a child of task b to the tree to which task b belongs.
- $getCurrTask(t)$ returns the currently running task of thread t.
- $createEdge(a, b)$ functions takes two nodes a and b as source and destination and creates an *edge* which is stored in *EgdeStore*
- $getNodeId()$ function returns the node id stored in U corresponding to a lock object.
- $ReportCycleWithBlame(n)$ function returns the cycle along with the actual instructions involved in the cycle, also blames the transaction which is nor serializable.
- *get_component_id*(n) function returns the component instance corresponding to the node id n.

4.2.2 Other Details

The proposed algorithm detects two types of atomicity violation. One is tree based atomicity violation and another is lifecycle based atomicity violation. Before checking atomicity, the trace is preprocessed and *beginAt* and *endAt* annotations are added at proper places according to the user programmer specification. For tree-based atomicity all the trees (refer section 3.2 of [6]) are assumed to be a single atomic block and are annotated like shown in section 3.4 in [6]. For lifecycle based atomicity check we check atomicity of the Activity , Service and Broadcast Receiver components. Applications are usually associated with a Main Activity, and other components like service, Broadcast Receiver may or may not be present. For activity

component the callbacks which belong to same activity gets same component instances. Based on these instances the algorithm will know which callbacks are part of same lifecycle. For Service component we get overlapped atomic blocks as discussed in section 2, and our algorithm handles it properly.

Algorithm assigns transaction ids to the instructions in the trace. The instructions belonging to the same atomic blocks get the same transaction ids. So all the instructions belonging to same tree or same component get same transaction id.

Happens before edges are added based on the rules in section 4 of [6]. We use hash-map H to store the ancestors of a node. By ancestor of a node we mean the nodes from which a node has incoming edge(because of *happens before* rules). By edge we mean (source, destination) pair, where source and destination both stores the pair (transaction-Id, line-Number). This line number is useful for blame assignment, which will be described in detail in this section later. So while executing, the algorithm checks each line of trace and assigns a transaction id(a fresh one or an already generated one according to the rule [INS-ENTER] in [6]) to it and checks whether any incoming edge can be added or not, if yes then it adds the transaction id and the ancestor set of the transaction from where edge is added to the ancestor set of the current transaction in the hash-map H. Line number 3 in Algorithm 1 says that when algorithm is outside atomic block then only assign a fresh node id to the instruction. Line number 4 and 5 in Algorithm 1 says that if the line contains *beginAtomic* then either new node id will be assigned if the component instance is seen for the first time, else a fresh node id will be assigned and that will be remembered by the $setId()$ function.

In case of tree based atomicity the function $getNode()$ (described in [6]) will be used instead of $getId()$ in line number 5 in Algorithm 1 there will be no such function named $setId()$ at all in the algorithm for tree based atomicity check. The function $get\_root\_id(node)$ (returns root task [6]) will be used instead of function $get\_component\_id()$ in line number 26 in Algorithm 1.

node-Id is saved corresponding to all the last writes($\mathcal{W}$) of all the variable, all the last read($\mathcal{R}$) of all variables of all trees(in tree-based atomicity checking) or components(in life-cycle based atomicity checking), and all the last unlocks($\mathcal{U}$) of all lock-objects. So when a read, write or lock is encountered an edge is added from the previous write(according to the rule [INS-READ-LIFE], line number 29 in Algorithm 1), read and write (according to the rule [INS-WRITE-LIFE],line number 33 and 38 in Algorithm 1), and unlock (according to the rule [INS-ACQUIRE] in [6], line number 22,23 in Algorithm 1) respectively, and the $\mathcal{W}, \mathcal{R}, \mathcal{U}$ data-structures are updated appropriately (by rules [INS-WRITE-LIFE],[INS-READ-LIFE] and [INS-RELEASE] [6],).

When there is $fork(t, tid)$ instruction in thread t node id is saved in $\mathcal{F}$ data-structure

(rule [INS-FORK] in [6], line number 11 in Algorithm 1) as an edge from this node to node for $threadinit(tid)$ will be added in future(rule [INS-THDINIT] in [6], line number 12 in Algorithm 1).

For $threadexit(tid)$ again the node id is saved in $\mathcal{F}$ data-structure (rule [INS-THREADEXIT] in [6], line number 13 in Algorithm 1), as an edge from this node to node $join(t, tid)$ will be added in future (rule [INS-JOIN] in [6], line number 14 in Algorithm 1).

When a $post(t', A, t)$ is encountered, if $t \neq t'$ then A becomes a root node of a new tree (rule [INS-POST] in [6], line number 8 in Algorithm 1)). If $t = t'$ then current node id is saved in $\mathcal{P}$.

When the algorithm adds a nodeA as ancestor of nodeB it adds the whole ancestor set of nodeA including nodeA to the ancestor set of nodeB. Now if the algorithm finds that nodeB is already present in the ancestor set of nodeA that means cycle has been detected and it reports the cycle and adds the ancestor set of nodeA to the ancestor set of nodeB excluding nodeB. It adds the ancestor set of nodeA to the ancestor set of nodeB in spite of getting cycle to not to miss any cycle of larger depth later.

Our algorithm can deal with multiple atomic block at the same time. In tree based atomicity check all the trees of all the threads are atomic and atomicity check is done for all the atomic block in one run. In lifecycle based atomicity check also atomicity of all the lifecycle components of all the threads are checked in one run.

The algorithm is trace specific. So, on same app in different traces it can report different number of cycles, and it reports both single and multi-variable cycles.

4.3 Blame Assignment.

Blame assignment helps to localize the error in the given cycle. Cycle is detected based on the transaction ids assigned to the instructions by the algorithm. But there can be thousands of instructions having the same transaction id. So if we just report the cycle it wont be enough for us to understand the behaviour of the cycle i.e for which instructions the cycle is formed. Blame assignment locates the exact instructions in the the trace which are involved in a cycle. We use another data-structure to store all the edges added by the algorithm. We call it Edge-Store (In H we only store the ancestor node ids corresponding to a particular node, no line number is stored there, H is used to check cycles but Edge-store is used for finding the path of cycles and assigning blame). The edges are added as (source,destination) pair in the Edge-Store. When one cycle is detected a recursive method is used to generate the path of the cycle. The path it returns is the instructions in the trace file involved in the cycle, where the node id of the source instruction and the destination instruction are always same. From this we can easily figure out the actual instructions involved in the source code.

4.4 Priority based classification

Our algorithm not only detects cycle but also classifies the cycles into four categories. They are classified based on the the type of edges involved in the cycle. There can be two types of *happens before* edges. One is *must be happens before* edge. The edges like fork-threadinit, threadexit-join, post-call, enable-trigger and the order edges(the $\mathcal{L}, \mathcal{F}, \mathcal{P}$ data structures saves the sources of these edges, [6]) falls in this category. These edges can be considered as causal edges. Other one is *may be happens before* edge. The edges like read-write, write-read, write-write, unlock-lock falls in this category. We refer an edge between instructions of same thread as single-threaded edge and a edge between instructions of different thread as multi-threaded edge. Our classification of cycles are as follows.

4.4.1 Cycles of Priority 1

These cycles are generated due to the edges between instructions of same thread but different atomic blocks i.e single-threaded edge. No inter-thread edge or *must be happens before* edge is involved in these cycles. Till now there is no existing tool that detects single-threaded atomicity violation in Android apps. This is the reason we give higher priority to these cycles.

4.4.2 Cycles of Priority 2

These cycles involve multi-threaded edges, but no *must be happens before* edge. The cycle can contain single-threaded edge also but at least one multi-threaded edge has to be involved in the cycle. These we can refer as multi-threaded atomicity violation.

4.4.3 Cycles of Priority 3

The cycles with at least one *must be happens before* edge and at least one *may be happens before* edge fall under this category. The *must be happens before* edges can occur because of $\mathcal{L}$. These cycles can also be considered as multi-threaded cycles.

4.4.4 Cycles of Priority 4

All the edges of these cycles are *must be happens before* edges. These cycles are the cycles we can not reorder or avoid. These cycles are not considered as violation, and that's why are given the least priority.

We observed that 75% of the reported cycles are of Priority 1, 4% are of Priority 2 and 21% are of Priority 3. No cycles of Priority 4 were reported. The number of edges in a cycle varies from 2 to 53(as far as we validated).

4.5 Filtering of cycles

We have extended our algorithm to filter the cycles. To do this we had to store some more information in the Edge-store. The Edge-store now stores task-id, node-id, line-number, and object-id, field-id and instruction-type(e.g. read , write lock , post etc) of the source and destination of each edge. Instructions like post, call etc does-not have object-id and field-id. In those cases we put null. Cycles are filtered based on the task-id, object-id, field-id and access-type. This means if there are more than one cycle whose all the edges are identical with respect to task-id, object-id, field-id and access-type, then only first cycle will be reported. We have observed that due to filtering, the number of reported cycles reduced from 0% to 80%.

4.6 Optimization

Two features we have added to the algorithm to optimize it.

- The first optimization is done for lifecycle atomicity check. A task which is outside atomic block, if does not have any incoming edge then that task can never take part in formation of any cycle. So that task is removed from the hash-map H as soon as it finishes. And if there is any out-going edge from that task in the Edge-Store then those are also removed. This saves some space. $Range_C$ is the set of transaction ids which are active at present. If a transaction is not active at present but it was active before and can again be active in future as that transaction id belong to some atomic block, then that transaction id will also belong to $Range_C$. $Range_H$ is the set of transaction ids incoming edges can be added in future. Thus if a transaction n which is not part of any atomic block has already finished i.e $n \notin Range_C$ [9] then no other incoming edges can be added to it in future(i.e. $n \notin Range_H$) and that node can never appear in cycle. So that node can be removed from *happens before*graph H. This optimization is also done in [9]. We formalize this process by the following rule, which is used when a task finishes i.e the return statement of that task is executed and after applying the rule [INS-EXIT].

[INS-EXIT-EXTENDED]

$n \notin Range_C$, $n \notin Range_H$ and $isAtomic(A) = 0$

$\mathcal{L}' = \mathcal{L} \backslash \{n\}$ $\mathcal{R}' = \mathcal{R} \backslash \{n\}$

$\mathcal{W}' = \mathcal{W} \backslash \{n\}$ $\mathcal{U}' = \mathcal{U} \backslash \{n\}$

$\mathcal{H}' = \mathcal{H} \backslash \{n\}$

$\phi \overset{endAt(A,t,d)}{\Longrightarrow} \phi[\mathcal{L} = \mathcal{L}', \mathcal{U} = \mathcal{U}', \mathcal{R} = \mathcal{R}', \mathcal{W} = \mathcal{W}', H = H']$

Here the updates of $\mathcal{W}, \mathcal{R}, \mathcal{U}, \mathcal{L}, \mathcal{H}$ are done as follows(Shown for $\mathcal{W}$ only).
$\mathcal{W}\backslash\{n\} = \lambda x.$ if $\mathcal{W}(x) = n$ then $\perp$ else $\mathcal{W}(x)$
$\mathcal{H}\backslash\{n\} = \{(n_1, n_2) \in \mathcal{H} | n_1 \neq n, n_2 \neq n\}$

Here $isAtomic(A)$ returns 1 if A is task and it belongs to atomic block, else if A is a task and does not belong to atomic block then returns 0. If A is not a task then isAtomic() returns -1.

- The second optimization again done for lifecycle atomicity check. We have explained that the instructions outside atomic block are considered as unary transaction. Each one of them will be assigned a fresh node id. This will be done for even all the instructions of a task outside atomic block. But we know that a single task in a thread should always be atomic with respect to that thread, as one thread can run only one task at a time. So we have modified the algorithm in such a way that all the instructions in a task even outside atomic block gets same node id. Because of this we have to store lesser number of nodes in the hash-map H. For this we have have changed the [INS-ENTER] and [INS-OUTSIDE] rules a little.

[INS-ENTER-EXTENDED]

$\mathcal{C}(t) = \perp$

if (d = -2 and $isAtomic(A) = 0$) or d = -1 or $getNode(A) = \perp$ n is fresh

else $n = getNode(A)$

$\mathcal{C}' = \mathcal{C}[t = n]$

$\phi \overset{beginAt(A,t,d)}{\Longrightarrow} \phi[\mathcal{C} = \mathcal{C}']$

Here the task A does not belong to any atomic block so $isAtomic(A)$ is 0.

[INS-OUTSIDE-EXTENDED]

$\mathcal{C}(t) = \perp$ l is a fresh label

$a \in \{acq(t, m), rel(t, m), read(t, x, v), write(t, x, v),$

$post(t, A, t'), attachQ(t), fork(t, t'), join(t, t'),$

$threadinit(t), threadexit(t), begin(A, t), end(A, t)\}$

$\phi \overset{beginAt(B,t,d)}{\Rightarrow} \phi_1 \quad \phi_1 \overset{a}{\Rightarrow} \phi_2 \quad \phi_2 \overset{endAt(B,t,d)}{\Rightarrow} \phi'$

$\phi \overset{a}{\Rightarrow} \phi'$

Here B is A and d is -2 for begin(A,t),end(A,t) statements and B is "dummy" and d is -1 for other statements.

After doing these optimization our algorithm was able to check applications with very long traces (e.g. Tomdroid), which was not possible before.

Chapter 5

Example

In this section we describe about one of our benchmark app, in which we seeded one single threaded bug, which we supposed to be detected by our algorithm as a life cycle based cycle, and our algorithm did it precisely along with blame assignment.

In fig 5.1 we have shown the execution trace of the scenario we are going to talk about. In this app, the onCreate() of the main activity issues a broadcastIntent and starts a bind service.That is why there is enable instructions of onBind() and onReceive() inside onCreate(). In onCreate() of the main activity variable *val* is initialized with some value. In onBind() callback of bindService the variable *val* is read and is set to some particular value, which is expected to be read in onServiceConnected() callback of the service, but in the execution we see that onReceive() of the broadcast issued from onCreate() comes in between onBind() and onServiceConnected() and changes the value of *val*, and that's why in onServiceConnected() we dont get the expected value of *val*. We ran the app several times and saw sometimes onReceive() executed after onServiceConnected(), and we did not get any cycles in those interleavings. So both interleavings are possible. This is a cycle which can only be detected by lifecycle based atomicity check, as in lifecycle based analysis the onBind() and onServiceConnected() callbacks are part of same atomic block, as they belong to same service component. onBind() and onServiceConnected() are not related by parent child relationship, so it is not possible to detect this cycle in tree based atomicity analysis.

As the messages onBind() and onServiceConnected() are part same service component they gets same transaction id B. onCreate() and onReceive() messages get transaction id A and C respectively. We see that the variable *val* is first initialized in onCreate(), then it is read in onBind() where it's value is changed. In message onReceive() value of *val* is again changed, and this value is read in message onServiceConnected(). There we get two cycles. One is because of the edges a(write-write edge on *val*) and b(write-read edge on *val*)(refer fig. 5.1).

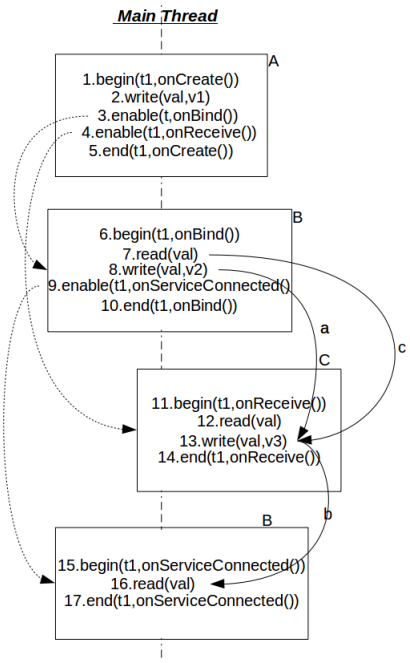


Figure 5.1: Execution trace corresponding to code snippet of fig.5.3, 5.4 and 5.5.

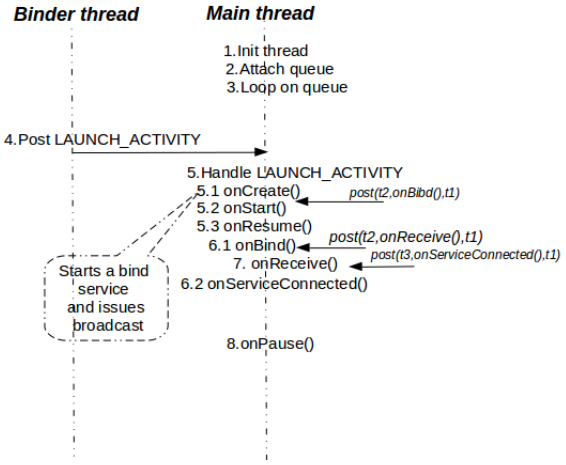


Figure 5.2: Execution scenario corresponding to code snippet of fig.5.3, 5.4 and 5.5.

The other cycle is because of the edges c(read-write edge on *val*)) and b(write-read edge on *val*).

The dotted edges are the must-be happens before edges and the firm edges are the may-be happens before edges added by our algorithm. Two more edges are there from the write on *val* of onCreate() to read and write on *val* on onBind() but these are shown in the trace as they never take part in any cycle.

The fig. 5.2 shows the execution scenario corresponding to the execution trace shown in fig. 5.1. Here it is shown that the post of onReceive(), onBind(), and onServiceConnected() are post from native threads to main thread, so their order can never be deterministic.

Fig. 5.3, 5.4 5.5 shows part of the source code of the benchmark app we have been discussing in this section. We see in fig. 5.3 onCreate() of MainActivity issues broadcastIntent at line 14 and binds service at line number 17. Value of variable *val* is initialized at line number 10 by calling myFunc() function, which is defined at line 57. onServiceConnected() method in line number 41 of fig. 5.3 reads value of *val* at line number 48.

LocalService class in fig. 5.4 extends Service class and overrides the onBind() method. In onBind() method, it reads the value of *val* at line number 10 and changes it's value in line number 12.

MyReceiver class in fig. 5.5 extends BroadcastReceiver and overrides the method onReceive(). It reads the value of *val* at line 12 , and writes on it at line 14.

Assigning Blame. One of the key feature of our algorithm is that it not only detects a cycle but it also precisely reports all the instructions which caused the cycle. In this case instructions involved in first cycle are line number 12 in fig. 5.4, line number 14 in fig. 5.5 and line number 48 in fig. 5.3 (write-write-read on *val*). The instructions involved in the other cycle are line number 10 in fig. 5.4, line number 14 in fig. 5.5 and line number 48 in fig. 5.3 (read-write-read on *val*). Our Algorithm detects the corresponding lines in the trace properly.

```
1  public class MainActivity extends Activity {
2    private LocalService s;
3    public static int val;
4
5    @Override
6    public void onCreate( Bundle savedInstanceState)
7    {
8      super.onCreate(savedInstanceState);
9      setContentView(R.layout.activity_main);
10     myFunc();
11     Intent intent = new Intent();
12     intent.setAction("MyBroadcast");
13     intent.putExtra("value", value);
14     sendBroadcast(intent);
15
16     Intent intent2= new Intent(this, LocalService.class);
17     bindService(intent2, mConnection,
18     Context.BIND_AUTO_CREATE);
19     Toast.makeText(MainActivity.this, "Value Initialized to:
20 "+val, Toast.LENGTH_SHORT).show();
21   }
22   @Override
23   protected void onResume()
24   {
25         super.onResume();
26   }
27
28   @Override
29   protected void onPause()
30   {
31           myFunc();
32           Toast.makeText(MainActivity.this,
33           "Value changed back to: "
34           +val, Toast.LENGTH_SHORT).show();
35           super.onPause();
36           unbindService(mConnection);
37   }
38   private ServiceConnection mConnection =
39   new ServiceConnection()
40   {
41     public void onServiceConnected(ComponentName
42     className, IBinder binder)
43     {
44       LocalService.MyBinder
45       b = (LocalService.MyBinder) binder;
46       s = b.getService();
47       Toast.makeText(MainActivity.this, "Connected:"
48       +val,
49       Toast.LENGTH_SHORT).show();
50     }
51     public void onServiceDisconnected
52     (ComponentName className)
53     {
54       s = null;
55     }
56   };
57   public void myFunc()
58   {
59           val =1000;
60   }
61 }
```

Figure 5.3: Sample Android application code snippet I (MainActivity)

```
1  public class LocalService extends Service
2  {
3    private final IBinder mBinder = new MyBinder();
4
5    @Override
6    public IBinder onBind(Intent arg0)
7    {
8                Toast.makeText(LocalService.this,
9                "im_in_onBind_"+
10               MainActivity.val, Toast.LENGTH_SHORT)
11               .show();
12               MainActivity.val = 9900;
13               return mBinder;
14   }
15
16   public class MyBinder extends Binder
17   {
18     LocalService getService()
19     {
20       return LocalService.this;
21     }
22   }
23 }
```

Figure 5.4: Sample Android application code snippet II

```
1  public class MyReceiver extends BroadcastReceiver
2  {
3    @Override
4    public void onReceive(Context context, Intent intent)
5    {
6      CharSequence text = "we_are_in_onReceive:";
7      Bundle extras = intent.getExtras();
8      if (extras != null) {
9       if(extras.containsKey("value"))
10      {
11           Toast.makeText(context, text+"_"
12           +MainActivity.val, Toast.LENGTH_SHORT)
13           .show();
14        MainActivity.val = 3456;
15      }
16    }
17 }
```

Figure 5.5: Sample Android application code snippet III

Chapter 6

Evaluation

We ran our algorithm on 38 open source apps, 10 proprietary apps, and 6 of our benchmark apps(refer table 6.1). Our algorithm maximum takes 5 to 6 seconds to run 5 to 10 traces of an app and it runs both types of atomicity check at the same time. Among 38 open source apps we got cycles in 17 apps (in lifecycle based atomicity check). Among 10 proprietary apps we got violations in 4 apps (in lifecycle based atomicity check). We validated many cycles and got some true positives. In table 6.2 we have shown how many warnings our algorithm generated for tree based (column 2) and lifecycle based atomicity (column 2) check and for how many of them blame assignment(inside braces) correctly identified the instructions involved in the cycles, for open source apps. Similarly in table 6.3 we have shown how many cycles were got from proprietary apps and how many of them were blamed correctly (inside braces).

We have ran the algorithm on 6 benchmarks (refer table 6.4). In 5 cases we seeded bugs and our algorithm was able to detect all the bugs in all the cases. In Seal1 there is no bug seeded and our algorithm reports no cycle for it. Seal2 has been discussed in section 4.

Seal3 is an app where we seeded one multi-threaded and one single-threaded atomicity violation. Our algorithm reports two cycles for both of the cases. In Seal4 bug seeded only

App-Type	Total Apps Checked	Apps with warnings in tree based(#warnings)	Apps with warnings in lifecycle based(#warnings)
Benchmark	6	4(4)	5(14)
Open Source	38	16(284)	17(642)
Proprietary	10	2(11)	4(16)
Total	48	22(299)	26(672)

Table 6.1: Observations

Open Source Android App	Warnings for tree based(Blame)	Warnings for lifecycle based(Blame)
Apollo	6 (6)	42 (42)
Character Recognition	4 (4)	9 (9)
Barcode Scanner	4 (4)	34 (34)
Turtle Player	3 (3)	4 (4)
Tomdroid	6 (6)	1 (1)
FBReader	31 (31)	6 (6)
Music Player	5 (5)	166 (166)
Messenger	3 (3)	10 (10)
NewsBlur	6 (6)	44 (44)
Andless	23 (23)	5 (5)
Camera Timer	7 (7)	3 (3)
Browser	28 (28)	14 (14)
Wikipedia	7 (7)	7 (7)
Search	28 (28)	80 (80)
Jamendo	6 (6)	2 (2)
JustPlayer	132 (132)	195 (195)
SicMuPlayer	20 (20)	20 (20)

Table 6.2: Warnings generated for open source apps by our algorithm for tree and lifecycle based atomicity check

Proprietary Android App	Warnings for tree based(Blame)	Warnings for lifecycle based (Blame)
Google Fit	10(10)	10(10)
GMail	1(1)	3(3)
My tracks	0(0)	2(2)
Google Keep	0(0)	1(1)

Table 6.3: Warnings generated for proprietary apps by our algorithm for tree and lifecycle based atomicity check

Benchmark Apps	Warnings in tree based(Blame)	Warnings in lifecycle based(Blame)	True Positives
Seal1	0(0)	0(0)	0
Seal2	0(0)	7(7)	7
Seal3	2(2)	2(2)	2
Seal4	0(0)	3(3)	3
Seal5	1(1)	1(1)	1
Seal6	1(1)	1(1)	1

Table 6.4: Warnings generated for Benchmark apps by our algorithm for tree and lifecycle based atomicity check

for lifecycle atomic check. Our algorithm detects those accurately. In Seal5 we use adhoc synchronization and our algorithm reports cycle on synchronization variable. Seal6 is an app consisting of only one thread. Here we have shown that single-threaded atomicity violation can not be avoided using locks.

We were able to find the the actual instruction involved due to the blame assignment implementation. Table 6.4 tells how many warnings we get for tree based(column 2) and lifecycle based (column 3) analysis along with how many cycles were properly blamed (inside braces), and how many of them were true positive(cloumn 4).

All the tables present the number of cycles after filtering. For Example in the open source app Apollo we get 17 cycles before filtering and 6 cycles after filtering in tree based atomicity check. In the same app we get 67 cycles before and 42 cycles after filtering in lifecycle based atomicity check.

6.1 Validation of cycles

Out of 48 apps we checked, the algorithm reports cycles in 21 apps in lifecycle based atomicity check. We have validated some of the the cycles. By validation we mean we have tried to reorder the the instructions involved in cycles. We were able to reorder them in 50% of cases. While validating cycles we noticed that Android Music Player crashed because of one cycle . Other than this we got some true positives where the value of the variables read becomes different after re-ordering.

Chapter 7

Conclusions and Future Work

Our algorithm is the first one to detect atomicity violation in Android apps. It does dynamic analysis. Android framework has an expressive environment. We consider all the callbacks of a component to be atomic. But we have not thought about the content provider component of Android. We wish to think about it later.

If an activity enables one service then we have observed that it would be better if we consider the service as a part of the activity's atomic block. And as onDestroy(), and onPause() are the callbacks which always happens because of external events, so these should not be considered as a part of the activity's atomic block. This is another approach of atomicity, which we want to implement in future. We hope this approach will help us to find some cycles which may lead to bad behaviour. We will also be validating all the cycle we got from our atomicity checker.

Chapter 8

Related Work

Atomicity violations are an important class of concurrency bugs and various tools have been developed so far to detect these types of bugs. Atomicity violations have been classified into single-variable (involving single shared variable) and multi-variable atomicity violations (involving more than one shared variables) [17]. Lot of research is happening to find these types of bugs in multithreaded programs. AVIO [14] is a tool to find single variable atomicity violations. For executions that fail, it reports patterns that are not present in benign patterns.

Atomizer [8] detects single as well as multi-variable atomicity violations and is based on Lipton's theory of reduction [12]. It uses Eraser's Lockset algorithm [19] to check whether a given program is serializable or not. It initializes lockset of each thread with all possible locks and whenever a thread tries to acquire a lock, it updates its lockset to be the intersection of current lockset and new acquired lock. If lockset becomes empty it generates a warning.

Cooperative Crugs Isolation [21] detects concurrency bugs by tracking if the shared memory location is successively accessed by different threads. Another tool known as Maple [24] exposes rare buggy interleavings by remembering the tested interleavings. While executing for the test input, Maple actively controls the thread schedule to expose predicted untested interleavings.

CTrigger is another work on detecting atomicity violation. It studies the interleaving characteristics of the common practice in concurrent program testing (i.e., running a program over and over) to understand why atomicity violation bugs are hard to expose. Second, it proposes CTrigger to effectively and efficiently expose atomicity violation bugs in large programs. CTrigger focuses on a special type of interleavings (i.e., unserializable interleavings) that are inherently correlated to atomicity violation bugs, and uses trace analysis to systematically identify (likely) feasible unserializable interleavings with low occurrence-probability. CTrigger then uses minimum execution perturbation to exercise low-probability interleavings and expose difficult-to-catch atomicity violation [18].

Various pattern based techniques have also been used to find atomicity violations in multi-threaded programs. UNICORN [17] groups memory accesses into pairs of problematic patterns and assigns them suspiciousness score. Based on this score, ranks are assigned to these patterns, thus making the job of finding actual bugs easy for the developer.

As far as Android concurrency model is concerned, there has been no research on detecting atomicity violations, as per our knowledge. However, other concurrency bugs like data races, have been studied for this model. DroidRacer [15] is a tool to detect data races in Android Applications. If generates a happens-before graph and finds accesses to same memory location with no happens-before ordering between them. CAFA [10] is another tool to detect reces in android apps.

Bibliography

[1] Application fundamentals — android developers. http://developer.android.com/guide/components/fundamentals.html.

[2] Articles & publications,kpmg,us. .

[3] A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios - researchgate. .

[4] Interim results for the 2014 survey — software industry survey. http://www.softwareindustrysurvey.org/.

[5] Smartphone usage experience report. http://www.ericsson.com/res/docs/2013/consumerlab/smartphone-usage-experience-report.pdf.

[6] Arti Bhat. Atomicity semantics and violation detection for android applications. ME Report, Indian Institute of Science, June 2015.

[7] Sebastian Deterding, Staffan L. Björk, Lennart E. Nacke, Dan Dixon, and Elizabeth Lawley. Designing gamification: Creating gameful and playful experiences. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '13, pages 3263–3266, New York, NY, USA, 2013. ACM.

[8] Cormac Flanagan and Stephen N Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 256–267, New York, NY, USA, 2004. ACM.

[9] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 293–303, New York, NY, USA, 2008. ACM.

[10] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, New York, NY, USA, 2014. ACM.

[11] Amy K. Karlson, Shamsi T. Iqbal, Brian Meyers, Gonzalo Ramos, Kathy Lee, and John C. Tang. Mobile taskflow in context: A screenshot study of smartphone usage. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2009–2018, New York, NY, USA, 2010. ACM.

[12] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975.

[13] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[14] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 37–48, New York, NY, USA, 2006. ACM.

[15] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 316–325, New York, NY, USA, 2014. ACM.

[16] Abdullah Muzahid, Norimasa Otsuki, and Josep Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 287–297, Washington, DC, USA, 2010. IEEE Computer Society.

[17] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. A unified approach for localizing non-deadlock concurrency bugs. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *ICST*, pages 51–60. IEEE, 2012.

[18] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architec-*

tural Support for Programming Languages and Operating Systems, ASPLOS XIV, pages 25–36, New York, NY, USA, 2009. ACM.

[19] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[20] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. Do i use the wrong definition?: Defuse: Definition-use invariants for detecting concurrency and sequential bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 160–174, New York, NY, USA, 2010. ACM.

[21] Aditya Thakur, Rathijit Sen, Ben Liblit, and Shan Lu. Cooperative crug isolation. In *Proceedings of the Seventh International Workshop on Dynamic Analysis*, WODA '09, pages 35–41, New York, NY, USA, 2009. ACM.

[22] unKnown. A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios - researchgate. .

[23] Jules White. Event driven systems and android. .

[24] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 485–502, New York, NY, USA, 2012. ACM.