针对安卓应用事件(指除去 SDK 提供的已有规则确定事件外的用户事件和其他事件)回调方法,有定义 5.

定义 5(事件回调方法之间的次序关系).

(1) 同一个事件产生的多个回调方法之间是有序的.

(2) 不同事件产生的回调方法之间是没有 HB 关系的.

(3) 事件的触发会调用相应的回调方法.然而事件的触发是不可预计的,因此事件之间是没有 HB 关系的,事件所对应的回调方法之间也没有 HB 关系.

因此,对于可疑的数据竞争候选者$\langle FA_i,FA_j\rangle$,由于它们的触发事件不同,FA_i 和 FA_j 之间是没有 HB 关系的.因此,如果没有额外的同步机制,则 FA_i 和 FA_j 可能引发数据竞争.我们使用约束求解的方法来识别 FA_i,FA_j 的上下文中是否有额外的同步措施.

约束编码广泛应用在程序中数据竞争的检测、再现和修复[29−31].基于安卓平台以及事件驱动特性和多线程模型,本文提出了以下相关约束:变量约束、条件约束、读写约束、数据竞争约束、同步约束.这些约束模拟了所有 Java 语言程序语句的特性.首先给出约束系统的基本概念.

(1) 符号变量 R_i^x,W_j^y 表示在位置 i,j 处对共享变量 x,y 进行读或者写操作.

(2) 符号变量 O_k 表示语句 k 的调度顺序.

变量约束定义了赋值语句和定义语句中对共享变量的读写约束.针对共享变量 x,位置 i 语句 $x=v$,表示为 $W_j^y=v$;语句 $x=y$ 在 i 位置,并且 y 变量定义在 j 位置,表示为 $W_j^y=y_i\wedge y_i=y_j$;语句 $z=x\oplus y$(⊕表示标准的二元运算符,例如加减乘除等),x 定义在位置 j,y 定义在位置 k,语句发生在位置 i,表示为 $z_i=x_i\oplus y_i\wedge x_i=x_j\wedge y_i=y_k$.

条件约束定义了条件语句中的约束情况.对于语句 if(z),会检查条件 z 的值,进而去判断相关的分支语句块是否被执行,表示为:$z\wedge z$ 是一个条件语句.

读写约束定义了所有对共享变量进行读写操作的语句约束情况.对于一个共享变量 x,读操作所得到的值等于之前最近一次对 x 写操作写入的值.因此,如果位置 i 读取的值等于位置 j 写入的值,则$(O_j\prec O_i)$,并且对于位置 k 对 x 的写操作,则有$(O_k\prec O_j)$或者$(O_i\prec O_k)$.表示为

$$\vee_{O_j\in writes}R_x^i=W_x^j\wedge O_j\prec O_i\wedge_{O_k\in writes\setminus(O_j)}(O_k\prec O_j\vee O_i\prec O_k).$$

数据竞争约束定义了两个特殊的语句次序约束,这两个语句共享同一个共享变量,并可能导致数据竞争的发生.对于语句 i 和 j,如果它们访问同一个共享变量,并且其中一个是写操作.同时我们想去确定这两个语句之间有没有 HB 关系.定义其数据竞争约束为 $O_{s_i}=O_{s_j}$.

同步约束包括线程内同步、锁同步、*Start*/*Join*/*Wait*/*Notify* 约束以及安卓中特殊的同步方法,具体介绍如下.

(1) 线程内同步:同一个线程中,所有的语句都是顺序执行的,表示为 $O_1\prec O_2\prec\ldots\prec O_n$.

(2) 锁同步:用 $l_{ij}^i=l_{mn}^m$ 表示语句 i,j 和语句 m,n 定义了两个临界区,两个临界区共享同一把锁,请求锁的操作发生在 i,m,释放锁的操作发生在 j,n.表示为 $l_{ij}^i=l_{mn}^m\Rightarrow O_j\prec O_m\vee O_n\prec O_i$.

(3) *Start*/*Join*/*Wait*/*Notify* 约束:*Start*/*Join*/*Wait*/*Notify* 是 Java 语言中用于同步线程常用的操作,*Start* 的操作用于开启运行一个新的线程,*Join* 操作用于通知该线程已经执行完毕;*Wait* 操作用于等待获取一个锁,*Notify* 操作用于通知已经释放了锁.因此,表示为 *Start*≺*Join*,*Wait*≺*Notify*.

(4) 安卓中特殊的同步:对安卓 HB 规则约束编码,对于两个语句 i,j,如果它们遵循 HB 规则,有 $O_i\prec O_j$.

例如,我们会检查如下线程间同步机制:*startActivity*,*startService*,*AsyncTask.execute*,*addListener* 等,并用 *methodinit*(m),*methodexit*(m)表示开始执行当前方法和已经执行完毕当前方法.

对所有的语句进行约束编码之后,将得到关于可疑数据竞争候选者的约束文件.约束文件中,首先,对约束变量进行定义;其次,根据以上规则对数据竞争候选者的上下文语句进行编码;最后,将约束文件放入 Z3solver (https://z3.codeplex.com/)中去检测是否有解,从而去识别真正的数据竞争.

具体到本文示例 OI File Manager,首先,通过线程共享变量分析,可以定位到共享变量 *window*,根据共享变量相关的上下文以及其读写集合,可以得到两个事件回调方法 *onDestroy*(·)和 *onPostExecute*(·),对这两个回调方

法对语句进行约束编码.具体到 *super.onDestroy*(·)和 *dialog.dismiss*(·),可以得出 *window*(*write*)=*null* 和 *y*=*window*(*read*);同时,设定它们的调度序列 O_1,O_2.针对数据竞争约束条件,可以将此编码为 O_1=O_2,然后把两个语句相关的约束语句和数据竞争约束 O_1=O_2 输出到一个约束文件中,并放入 Z3 求解器中进行求解,去判断数据竞争约束在安卓并发语义的 Happens-Before 规则下面是否真实发生,即在所有的事件调度和线程调度解空间中是否有解.如果有解,则认定它们为数据竞争.

4 实验分析

本节主要介绍工具 RaceDetector 的实现、数据集、实验及其结果分析,并结合案例讨论与现有工具的异同.

4.1 工具RaceDetector的实现

本文使用 Java 语言实现了工具 RaceDetector.静态分析模块通过静态解析整个安卓应用 APK 文件来大幅度地提高源代码的覆盖率.这里使用 SOOT 工具将 APK 文件转换为三地址码(Jimple 格式)的简单语义文件,继而进行安卓应用线程共享变量分析,并优化了传统的逃逸分析算法(第 4.2 节).根据线程共享变量,得出该共享变量的读写语句集合,从而得出数据竞争候选者集合(第 4.3 节).在约束求解模块中,我们根据安卓应用的先后序关系规则对疑似数据竞争对应的两个代码片段进行约束编码,得到一个定义了两个代码片段发生序关系的约束文件,并通过 Z3 求解器求解,判断有数据竞争约束的两个语句是否有 HB 关系,从而找到真正的数据竞争.

本文借助了两个辅助工具——SOOT 和 Z3 求解器.

- SOOT 是一个可以分析安卓应用程序的框架平台,能够将安卓应用的 APK 文件解析为三地址码格式.SOOT 还提供了别名分析和数据流分析等常用的静态分析方法.SOOT 工具的 API 可以识别安卓应用每一个类的信息、属性信息、方法信息以及类与类之间、方法与方法之间的调用信息,从而可以进行线程共享变量分析和疑似数据竞争分析.
- Z3 求解器是一个在广泛的解空间中检测某些约束条件是否有解的工具.它接受一系列的约束条件作为输入.这些约束条件定义了相关变量之间的关系.针对数据竞争的约束文件,Z3 求解器输出 SAT,表明两个包含数据竞争约束条件的语句并没有违反安卓应用中的先后序关系规则,也即这两个语句之间没有先后序关系,因此它们是数据竞争.

4.2 数据集

本文根据以下标准挑选出了 15 个流行的 App 作为数据集.

(1) 广泛流行.这些 App 来自 Google Play Store 排行榜的前列,或者来自于其他论文中的数据集[8−10].

(2) 不同的类别,包含了媒体、工具到社交、新闻等类别.

(3) 不同的大小,小的有 8k 多行,大的有 400k 多行(这里是按照 Jimple 格式显示的代码行).

表 1 展示了这些 App 的相关信息,按照 Jimple 格式代码行的大小,从小到大进行排列.表 1 的第 2 列是代码行数.第 3 列展示了每一个 App 中的线程共享变量,斜线左边的是可共享变量的数目,右边是经过 ATSA 实际得出的、真实的共享变量数目.可以看出,线程可共享变量广泛存在于 App 中,数量从几百到几千不等.但是真实共享变量数目远远小于可共享变量数目.第 4 列~第 7 列分别是安卓应用的活动、服务、异步任务和线程的数量信息.这些信息展现了每一个应用中的线程数量:安卓应用中,每一个活动都运行在主线程;每一个服务代表了一个在后台执行的任务;AsyncTask 是安卓应用中发起异步执行任务的框架;Thread 则是传统的 Java 语言线程.最后一列是监听器 Listener 的数目,每个监听器 Listener 会监听相应的输入事件,并引发一个或者多个回调函数的执行.

由此可见,大小应用中均存在多线程运行的场景,即多线程在安卓应用中是普遍存在的.每个应用都有相应的监听者,对应着各种输入事件,并且输入事件的类型多且复杂.因此,输入事件的不可预期性、无序性以及相互之间的调度也是数据竞争问题产生的原因.

Table 1 Information statistics of data set

表 1 数据集信息统计

应用	代码行	共享变量	活动	服务	异步任务	传统线程	监听器
SGTPuzzles	8 145	121/5	26	0	0	18	23
OI File Manager	9 185	180/2	30	1	7	6	35
Tomdroid	15 693	222/8	54	2	0	11	28
Connectbot	18 251	333/4	23	1	1	14	59
Facebook	31 757	449/6	7	4	0	61	32
Aard Dictionary	37 687	359/1	13	1	0	23	11
Remind Me	56 771	1380/12	10	4	0	55	83
Browser	66 589	1126/16	53	1	18	73	94
FedEx	182 951	2101/27	77	0	12	118	107
Netflix	202 720	2860/42	70	20	3	253	299
Music	205 896	1645/6	48	2	11	211	131
Tokopedia	279 523	5831/3	102	6	14	146	432
Flipkart	288 390	5724/77	50	10	32	257	429
Pandora	353 265	5276/131	145	12	222	472	391
Instagram	396 784	7233/19	194	17	4	464	783
平均	143 572	2 353	60	5	22	145	196

4.3 实验结果分析

本文实验均使用处理器为 Intel Core i5-4570 3.20GHz、内存为 8GB、操作系统为 64 位 Windows 7 专业版的计算机所完成.实验工具由 Java 语言来实现,使用的开发环境为 JDK 7.0,使用的 IDE 为 Eclipse Kepler.静态分析部分的功能使用了 SOOT 工具辅助,约束求解部分使用了 Z3 求解器.表 2 展示了相应的实验结果.

Table 2 Information statistics of experimental results

表 2 实验结果信息统计

应用	时间(s)	共享变量	函数对	数据竞争			有害竞争	误报数
				单	多	总数		
SGTPuzzles	11.8	5	23	2	47	49	4	0
OI File Manager	8.3	2	9	0	11	11	8	0
Tomdroid	6.3	8	25	80	26	106	3	4
Connectbot	12.6	4	38	21	0	21	2	1
Facebook	9.9	6	49	3	24	27	0	7
Aard Dictionary	10.1	1	5	1	3	4	3	2
Remind Me	41.7	12	27	9	49	58	0	3
Browser	45.8	16	30	109	28	137	7	15
FedEx	39.5	27	156	41	249	290	9	39
Netflix	57.4	42	110	302	969	1 271	14	192
Music	73.5	6	26	4	159	163	5	4
Tokopedia	168	3	18	0	46	46	0	0
Flipkart	173.6	77	172	302	478	780	37	123
Pandora	184.2	131	239	715	1224	1 939	137	256
Instagram	121.2	19	42	0	192	192	2	8
平均	64.22	24	65	106	234	340	15	44

表 2 的第 2 列展现了工具的执行时间,包括线程共享变量分析的时间、约束编码的时间和求解器求解的时间.可以看出,工具的执行时间随着应用大小近似线性增长.对于小的应用,RaceDetector 基本都在 1min 内完成任务.执行时间最短的应用是 Tomdroid,仅需要 6.3s.在表 2 的底部区域,对于大的应用来说,执行时间几乎都超过了 1min,最慢的应用是 Pandora,其执行时间超过了 3min.除此以外,我们发现执行时间也受到共享变量、数据竞争候选者数目的影响,例如,Instagram 是 15 个应用中最大的,与 Pandora 相比,其执行时间少了近 1min.这是因为它的数据竞争候选者数目(42)远小于 Pandora 的候选者数目(239).另外我们发现,约束求解执行的时间平均在 1min 左右,约束编码占据了大部分时间.这是因为我们把全局的先后序关系约束分解为针对每一个候选者的约束文件,有效降低了约束求解的执行时间.随着候选者数量的增多,约束求解执行的时间也随之增长.

表 2 的第 3 列是经过 ATSA 分析过后得到的实际情况下线程间共享变量的数目.在安卓应用中,开发人员经常会定义作用域超出自身所在线程或者自身所在类别的对象,但是实际情况下,真正能够被多线程共同访问的共享变量数目是极少的.因此,优化后的 ATSA 分析方法极大地降低了需要分析的共享变量数目,减少了程序的

空间消耗和时间消耗.

表 2 的第 4 列是可疑的数据竞争候选者(函数对)数目,对照表 2 的第 3 列可以看出,可疑的数据竞争候选者数目大于线程共享变量的数目,即每一个线程共享变量会存在多个数据竞争候选者.因此,会存在许多个线程或者 Listener 共同访问同一个共享变量.每一个数据竞争候选者对应一个函数对,符合数据竞争定义的第 1 个数据竞争条件,针对每一个函数对,我们会产生一个约束文件,并放入求解器中进行求解.

表 2 的第 5 列~第 8 列展现了具体检查出的数据竞争结果.本文将数据竞争分类成单线程数据竞争和多线程数据竞争,并识别出其中有害的数据竞争(第 8 列).根据实验结果,可以得出以下结论.

(1) 相对于安卓应用中的单线程数据竞争,多线程数据竞争的数目更多.

(2) 每个函数对代表了针对一个共享变量的约束文件,平均每个约束变量都会引发 5 个以上的数据竞争.

(3) 在数据竞争中,我们关心的有害数据竞争(空引用)仅占据很小的比例.

(4) 针对每个应用,RaceDetector 平均报告了 340 个数据竞争.针对应用 Pandora 检测到的数据竞争最多,达到 1 939.结合表 1 和表 2 可以看出,Pandora 应用使用了大量的线程并发执行代码.

表 2 的最后一列展现了误报的数据竞争的数目,平均的误报率为 13%(44/340).误报的数据竞争数目是通过实验过后人工审查分析得出的结果.我们对这些误报的数据竞争特性进行了总结,具体原因分析如下.

(1) 一些数据竞争发生在用户代码和安卓系统 API 调用之间,表现为多个线程共同访问同一个共享变量,访问的语句直接或间接调用系统 API.我们认为:这些数据竞争对线程共享变量进行了读或写的操作,没有同步措施.而真实情况是,系统 API 对线程共享变量已采取了相应的同步措施.由于本文未解析安卓 API 代码,因此产生误报.

(2) 单线程数据竞争往往发生在活动 Activity 和与相关的 Listener 所触发的回调方法之间,但只有当活动处于前端时,用户才可执行交互;同时,工作线程不能改变主进程 UI 组件的状态.具体展现的是在某些 Listener 所监听的事件处于活动进行前后台切换情况下,处于前端的 Activity 会转入后台,并触发调用一些回调方法进行资源的释放和销毁.在实验中,我们未覆盖到相关事件类型所导致的额外回调方法的执行,因此产生误报.

(3) 数据竞争检测阶段所使用的 Happens-Before 规则和安卓应用的并发语义是基于前人的相关工作,但是随着安卓系统的发展和更新,Happens-Before 规则和并发语义并没有全方位覆盖到.因此在实际排查中,我们发现具体实验过程中会遗漏一些 Happens-Before 规则.

4.4 案例讨论

表 1 已列出的线程类型有 5 种:活动(activity)、服务(service)、Thread、AsyncTask、Listener.我们进一步统计了不同线程类型之间的数据竞争,详见表 3.

Table 3 Information statistics of data race types

表 3 数据竞争类型信息统计

应用	Act-Lis	Act-ST	Act-Async	ST-ST	ST-Async	ST-Lis	Async-Async	Async-Lis
SGTPuzzles	2	35	0	1	0	11	0	0
OI File Manager	0	0	8	3	0	0	0	0
Tomdroid	80	0	0	1	0	26	0	0
Connectbot	21	0	0	0	0	0	0	0
Facebook	3	24	0	0	0	0	0	0
Aard Dictionary	1	2	0	0	0	1	0	0
Remind Me	9	0	0	0	0	49	0	0
Browser	109	10	0	0	2	0	0	6
FedEx	41	78	21	150	0	0	0	0
Netflix	302	26	0	802	0	141	0	0
Music	4	0	2	157	0	0	0	0
Tokopedia	0	0	0	36	0	10	0	0
Flipkart	302	10	0	67	0	266	69	66
Pandora	715	252	0	449	2	521	0	0
Instagram	0	0	0	14	0	178	0	0

在所定义的数据竞争中,数据竞争可以发生在不同的线程之间.由于安卓应用存在多种类型的线程框架,根据发生数据竞争所在的两个线程的不同类型,具体有 8 个类型:Act-Lis、Act-ST、Act-Async、ST-ST、ST-Async、ST-Lis、Async-Async、Async-Lis(Act:Activity,ST:Service 或者 Threads,Async:AsyncTask,Lis:Listener).因为 AsyncTask、Service 和 Thread 可以运行在后台,而 Listeners 一部分与 Activity 绑定,一部分与 Sensor 绑定.

表 3 表明,Activity、Service、Listener 间的数据竞争占据很大比例,数据竞争大多集中在 Act-Lis、Act-ST、ST-ST、ST-Lis 这几个类型,说明了 Listener 执行的不可预计性.我们选取数据集中的 3 个应用进行案例讨论.

- OI File Manager

在表 2 中,RaceDetector 报告了 11 个数据竞争,其中有 8 个有害数据竞争.它们发生在 Activity 和 AsyncTask 之间.当 Activity 正在执行异步任务时,会访问一个 Dialog,这时用户可能按下回退按钮中止 Activiy,同时中止相应的 Dialog.这样,正在进行异步任务的 AsyncTask 访问 Dialog 时会获取一个 null 值,从而导致有害数据竞争的发生.

- Connectbot

数据竞争发生在 Activity 和 Listener 之间.这个应用里,一个数据库用来存储数据,相应 Listener 会监听数据库状态,从而会更新数据竞争的状态.大部分数据竞争发生在 Activity 和 Listener 的回调方法之间,多数为读取数据库状态的操作,其中存在两个有害数据竞争,都涉及到了对数据库的写操作或者对数据库进行销毁的操作,具体发生在 Activity 的 *onStop*(·)方法和 Listener 的回调方法之间.一旦用户操作导致 Activity 的中止,*onStop*(·)方法会销毁数据库,此时数据库更新数据的操作会获取到一个 null 值,从而导致有害的数据竞争.

- Music

这是一个音乐播放的应用,用户可以改变 Music 音乐播放器的状态或者下载更新相关的音乐.所导致数据竞争发生的核心共享变量由一个表示音乐播放器的状态变量 *state* 所导致.该应用中,有多个线程拥有访问该状态变量的权限.多数的数据竞争为正常的更新状态变量来改变应用的状态,并不会导致有害的行为.有害的数据竞争发生在后台任务下载音乐和播放音乐之间,当发起一个后台任务下载音乐时,用户的不同操作会导致应用处于前台的不同状态.当后台任务下载完成更新数据时,如果用户销毁相关页面,将会导致数据竞争.

通过分析上述应用中数据竞争产生的具体情况可知,多线程并发执行任务是安卓应用中常见的场景,如果没有完善的同步措施,数据竞争很可能发生并导致不同程度的危害.

4.5 工具对比

本节将对相关的安卓应用数据竞争检测工具进行比较和分析,包括 CAFA、DroidRacer 和 EventRacer.

(1) 分析方法和覆盖率

DroidRacer、CAFA 和 EventRacer 使用动态分析方法进行建模,对原 App 插桩,并动态执行 App 获取相关的执行轨迹.其中,CAFA 主要检测空指针异常所导致的数据竞争;DroidRacer 主要关注发生在用户代码中的事件交互所导致的数据竞争;EventRacer 不仅识别了用户代码中的数据竞争,同时也对安卓框架 SDK 进行了分析.但是动态分析覆盖率较低,尤其在安卓应用中,一方面,一次动态执行很难触发所有事件去覆盖回调方法;另一方面,一次事件执行的轨迹是固定的,但多线程的事件和线程调度是不确定的,如果要触发更多的执行轨迹,将会显著增加性能消耗.

考虑到动态分析方法在覆盖率方面的缺陷,RaceDetector 使用了静态分析方法,能够保证具有较高的覆盖率,另外还使用了约束求解方法去动态生成所有可能的事件和线程调度.

(2) 执行轨迹的产生和性能

CAFA、DroidRacer 和 EventRacer 都是通过插桩并执行 App 来获取执行轨迹.这里,我们详细分析 EventRacer 工具.EventRacer 通过 AndroidMonkey 产生随机的输入事件触发 App 的执行,相关的命令为 adb shell monkey –s 42 –throttle 60 –v 1000.这里产生了 1 000 个输入事件,运行时间在 1min 左右.随着输入事件的增加,执行时间也会相应增加.由于这些事件是随机产生的,而安卓应用中界面的跳转往往需要触发特定的事件,因此随机事件存在大量的冗余,通常只是在固定的几个界面中执行重复的动作,而不能执行特定事件,进而覆盖到其

他界面.

相反,RaceDetector 抽取所有组件、线程和事件回调方法信息,并检查所有的事件回调方法间是否会满足数据竞争定义的第 1 个条件;另外,还通过求解器判断是否满足数据竞争定义的第 2 个条件,保证覆盖率和性能.

(3) 模型

CAFA、DroidRacer 和 EventRacer 都使用发生序 HB 模型,它们构建了全局 HB 图.其中,EventRacer 扩展了 CAFA 和 DroidRacer 的 HB 图,并优化了图查找识别算法.它们所构造的 HB 图是基于动态执行产生执行轨迹.

与此相反,考虑到构造一个全局的 HB 模型会对性能产生很大的影响,RaceDetector 对每一个可疑的数据竞争候选者上下文构建局的 HB 图,即把全局的 HB 图划分为多个小的局部 HB 图,优化了算法和执行时间.

(4) 实验结果

我们使用 EventRacer 和 RaceDetector 对本文的数据集进行了对比实验.因为相关安卓平台和版本的差异,数据集中的部分应用无法直接进行对比,最终 6 个应用的比较结果展现在图 4 中.所有的实验都是在 EventRacer 默认的设置下进行的,EventRacer 默认设置产生 300 个输入事件去获取执行轨迹.

表 4 的第 1 列显示了 6 个应用名称.第 2 列显示了 EventRacer 和 RaceDetector 执行时间.EventRacer 的执行时间包括以下几个部分:启动安卓模拟器、执行 App、检索执行轨迹文件、分析执行轨迹文件.EventRacer 的平均执行时间约为 50s.这是由于其默认生成的 300 个输入事件数量较少,事实上,某些 App 的 Listeners 就已经远远超过了 300 个.如果 EventRacer 产生更多随机事件去覆盖 Listeners,相应的执行时间也会大为增加.RaceDetector 分析小的安卓应用只需要极少的时间,最少的 OI File Manager 只需要 8s.RaceDetector 平均需要 49.2s 的执行时间,但考虑到 RaceDetector 覆盖了全部的 Activity、Thread 和 Listeners,这个时间还是可以接受的.

Table 4 Data race detection reports by EventRacer and RaceDetector

表 4 EventRacer 和 RaceDetector 检测的数据竞争报告

应用	时间 (EventRacer/ RaceDetector)	EventRacer 工具				数据竞争 (EventRacer/ RaceDetector)	有害的 (EventRacer/ RaceDetector)
		同步	用户代码	用户调用框架	无害		
OI File Manger	50(22+15+5+8)/8.3	0	0	0	785	785/11	0/8
Tomdroid	50(22+15+5+8)/6.3	0	0	216	1 389	1 605/106	0/3
Connectbot	49(21+15+5+8)/12.6	1	0	36	1 400	1 437/21	1/2
Aard Dictionary	50(24+14+6+6)/10.1	1	1	160	440	602/4	2/3
Music	65(22+18+10+15)/73.5	8	0	189	3 553	3 750/163	8/5
Pandora	53(22+18+5+8)/184.2	0	0	0	1 758	1 758/1939	0/137
平均	53/49.2	1.7	0.17	100	1 554	1 656/374	2/26

表 4 第 3 列~第 6 列是详细的数据竞争信息.EventRacer 对数据竞争划分了 4 个不同的类型.

EventRacer 和 RaceDetector 检测到的数据竞争总数在第 7 列.我们可以看出,RaceDetector 检测出的数量远远小于 EventRacer.这是由于 RaceDetector 只检测了用户代码,EventRacer 检测了用户代码和安卓相应版本的 SDK 代码.然而,安卓 SDK 对线程共享变量已经做了很好的同步,没进行同步的是良性的数据竞争,对 App 的行为并没有有害的影响.

表的最后一列显示了有害的数据竞争,可以看出,EventRacer 检测出的数据竞争中,有害的数据竞争极少.相反,RaceDetector 检测出的有害数据竞争则占据一定的比率.具体对于 OI File Manager 和 Tomdroid,EventRacer 分别报告了 785 个和 1 605 个数据竞争,没有一个是有害的数据竞争.相反,RaceDetector 报告了 11 个和 106 个数据竞争,其中,对于 OI File Manager 有 8 个有害的数据竞争,对于 Tomdroid 有 3 个有害的数据竞争.

针对 Pandora,可以发现,RaceDetector 报告的数据竞争超过了 EventRacer 报告的数据竞争.这是因为对于小的应用,EventRacer 通过分析执行轨迹和安卓 SDK 能够分析出较多的数据竞争.我们静态检索了所有的用户代码,但是应用小,报告的数据竞争也少.对于 Pandora,它是一个大的安卓应用.仅仅检索用户代码,RaceDetector 报告的数据竞争就超过了 EventRacer.因此对于越大的应用,EventRacer 将会遗漏更多的有害数据竞争.本次对比实验中,与 RaceDetector 相比,EventRacer 平均遗漏了近 20 个有害的数据竞争.

5 相关工作

- 数据竞争检测方法

早期的数据竞争检测方法是基于锁的[16,32,33],最有代表性的工具为 Eraser[33].但是基于锁的数据竞争检测方法属于保守策略,有很严重的误报问题(false positive).另一种数据竞争检测方法是基于 HB 关系[18,27,28].不同的工具通过静态或者动态分析构造 HB 图.基于静态分析的数据竞争检测方法[1,11,12,14,15,20,34]有很好的覆盖率,因为它们能够解析所有的源码和执行路径;但存在误报问题,因为无法确定动态加载的代码和实际执行的路径选择.与此同时,覆盖率的提高意味着性能的下降,解析全部的源代码在大规模应用中对性能有严重影响.如何平衡、取舍,也是静态分析方法所面临的挑战之一.基于动态分析的数据竞争检测方法适合用于大的数据集,并且能够产生比较精确的结果,但会受到低覆盖率的影响,因为通过执行程序所获取的上下文信息只能覆盖很少的一部分代码,会面临漏报问题(false negative).综合了静态分析和动态分析的优点,基于预测性分析的数据竞争检测方法[29−31,35,36]从动态分析出发获取执行轨迹,进而分析执行轨迹所依赖的限制条件,在遵循限制条件的前提下,采取策略改变语句和线程的调度,生成新的执行轨迹,从而提高覆盖率,既缓解了静态分析的误报问题,又缓解了动态分析的低覆盖率问题.还有基于 causally-precedes(CP)的方法[37],能够避免误报,但还是会有漏报.文献[38]进一步将抽象的控制流信息引入到执行模型中,弥补了 HB 和 CP 方法的不足,增加了解空间.本文扩展了预测性分析中的约束求解方法,并结合静态分析、共享变量逃逸分析和优化的 HB 图,实现了 RaceDetector.

- 安卓应用数据竞争检测

针对安卓应用中数据竞争,文献[10]首次形式化出了安卓应用中的并发语义,总结出了安卓应用中的 HB 关系,并针对多线程代码片段以及安卓中特有的单线程代码片段,构建了安卓应用的 HB 图.基于动态插桩技术和 HB 模型,文中实现了相关工具 DroidRacer.文献[9]基于安卓应用中的事件驱动模型系统,实现了工具 CAFA,动态地获取安卓应用的执行轨迹并分析检测数据竞争.EventRacer[8]扩展了 DroidRacer 的 HB 模型,并优化了检索算法.但是这些工具都是基于动态分析方法,存在固有的低覆盖率问题,尤其针对安卓应用,除了线程发生序关系,还面临定位事件发生序和事件发生时机的挑战.因此,现有的安卓并发语义和发生序关系在真实面临复杂的事件操作和线程调度往往也存在严重的误报问题.除此之外,这些方法构造了全局的 HB 图,很难处理大的应用程序.即在安卓应用中检测数据竞争,采用静态分析所面临的问题是如何平衡好覆盖率和性能,并需要对性能进行优化,为此,我们进行了共享变量分析以缩小范围;应用动态分析将面临低覆盖率的问题,因为安卓应用是基于事件驱动的,相对于传统的多线程程序,安卓应用的一次动态执行只能覆盖很小比例的事件,为此,我们采用约束求解的方法来提升覆盖率.

6 总 结

本文主要研究了安卓应用中数据竞争这类并发缺陷,针对现有工具的不足,本文提出了改进的方法.

我们首先使用 SOOT 工具解析安卓应用的 APK,并记录共享变量信息和线程、安卓组件等必要信息.针对线程共享变量信息,我们集成了安卓应用中的共享变量,并优化了传统的共享变量分析方法:逃逸分析.提出了安卓线程共线变量分析方法 ATSA,可以使得实际的线程共享变量数目远远小于可能的线程共享变量数目,极大地缩小了 RaceDetector 的分析空间并提高了执行性能.继而,我们形式化定义了可疑的数据竞争候选者集合,并针对每个可疑的数据竞争以及其上下文进行约束编码.在约束编码阶段,我们集成了安卓应用中的 Happens-Before 规则,并提出了局部的 Happens-Before 图.约束编码之后,我们将产生的约束文件放入 Z3 求解器中进行求解,Z3 求解器覆盖了所有的事件调度和线程调度,极大地提高了 RaceDetector 的覆盖率.

相对于现有工具,本文所实现的 RaceDetector 能够有效检测数据竞争,同时,我们的工作还存在一些不足之处:RaceDetector 通过 SOOT 进行相关分析,相关的性能受到 SOOT 的限制,SOOT 不能解析动态加载的代码,并且针对部分 App 会出现分析失败的结果;同时,RaceDetector 基于现有的工具所提出的并发语义和 Happens-Before 进行 HB 关系分析,相关的性能和准确性也受此影响;另外,针对有害的数据竞争,我们并没有准确定义它和良性数据竞争的边界,我们只是从空引用这一个角度对有害数据竞争进行了定义和分析.因此,还需要更深入

的研究和探索.

由于精力有限,本文未关注 Android SDK 框架层代码中的数据竞争.原因在于,一是框架层已经具备了很好的同步措施,技术相对成熟,并发缺陷较少,而应用层主要由用户的代码构成,代码快速迭代很容易产生并发缺陷;二是 SDK 框架层代码中即使有并发缺陷,也不易修改(因为 SDK 框架层过于复杂),但用户层的用户代码可以修改并完成之后的修复工作.后续我们将继续进行安卓应用数据竞争重现和修复的研究.

References:

[1] Kahlon V, Sinha N, Kruus E, *et al*. Static data race detection for concurrent programs with asynchronous calls. In: Proc. of the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. ACM, 2009. 13−22.

[2] Takala T, Katara M, Harty J. Experiences of system-level model based GUI testing of an Android application. In: Proc. of the 2011 4th IEEE Int'l Conf. on Software Testing, Verification and Validation. IEEE, 2011. 377−386.

[3] Yan D, Yang S, Rountev A. Systematic testing for resource leaksin Android applications. In: Proc. of the 23th Int'l Symp. on Software Reliability Engineering. IEEE, 2013. 411−420.

[4] Yang S, Yan D, Rountev A. Testing for poor responsiveness inAndroid applications. In: Proc. of the 1st Int'l Workshop on the Engineering of Mobile-enabled Systems. IEEE, 2013. 1−6.

[5] Grace M, Zhou YJ, Zhang, Q, *et al*. RiskRanker: Scalable and accurate zero-day android malware detection. In: Proc. of the 10th Int'l Conf. on Mobile Systems, Applications, and Services. ACM, 2012. 281−294.

[6] Holland B, Deering T, Kothari S, *et al*. Security toolbox for detecting novel and sophisticated Android malware. In: Proc. of the 37th Int'l Conf. on Software Engineering. IEEE Press, 2015. 733−736.

[7] Zhou YJ, Jiang XX. Dissecting Android malware: Characterization and evolution. In: Proc. of the 2012 IEEE Symp. on Security and Privacy. IEEE, 2012. 95−109.

[8] Bielik P, Raychev V, Vechev M. Scalable race detection for Android applications. In: Proc. of the ACM SIGPLAN Int'l Conf. on Object Oriented Programming Systems Languages & Applications. 2015. 332−348.

[9] Hsiao CH, Pereira C, Yu J, *et al*. Race detection for event-driven mobile applications. In: Proc. of the 35th Annual ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2014. 326−336.

[10] Maiya P, Kanade A, Majumdar R. Race detection for Android applications. ACM SIGPLAN Notices, 2014,49(6):316−325.

[11] Kahlon V, Yang Y, Sankaranarayanan S, *et al*. Fast and accurate static data-race detection for concurrent programs. In: Proc. of the 19th Int'l Conf. on Computer Aided Verification. Springer-Verlag, 2007. 226−239.

[12] Da Luo Z, Hillis L, Das R, *et al*. Effective static analysis to find concurrency bugs in Java. In: Proc. of the 2010 10th IEEE Working Conf. on Source Code Analysis and Manipulation. IEEE, 2010. 135−144.

[13] Naik M, Aiken A. Conditional must not aliasing for static race detection. In: Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. 2007. 327−338.

[14] Naik M, Aiken A, Whaley J. Effective static race detection for Java. In: Proc. of the ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation. ACM, 2006. 308−319.

[15] Naik M, Park C, Sen K, *et al*. Effective static deadlock detection. In: Proc. of the 31st Int'l Conf. on Software Engineering. IEEE Computer Society, 2009. 386−396.

[16] O'Callahan R, Choi JD. Hybrid dynamic data race detection. In: Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. 2003. 167−178.

[17] Effinger-Dean L, Lucia B, Ceze L, *et al*. IFRit: Interference-free regions for dynamic data-race detection. In: Proc. of the ACM SIGPLAN Int'l Conf. on Object Oriented Programming Systems Languages & Applications. 2012. 467−484.

[18] Flanagan C, Freund SN. FastTrack: Efficient and precise dynamic race detection. In: Proc. of the 30th Annual ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2009. 121−133.

[19] Jensen CS, Prasad MR, Møller A. Automated testing withtargeted event sequence generation. In: Proc. of the 2013 Int'l Symp. on Software Testing and Analysis. ACM, 2013. 67−77.

[20] Petrov B, Vechev MT, Sridharan M, *et al*. Race detection for Web applications. In: Proc. of the 33th Annual ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2012. 251−262.

[21] Raychev V, Vechev MT, Sridharan M. Effective race detection for event-driven programs. In: Proc. of the 2013 ACM SIGPLAN Int'l Conf. on Object Oriented Programming Systems Languages & Applications. 2013. 151−166.

[22] Zheng YH, Bao T, Zhang XY. Statically locating Web application bugs caused by asynchronous calls. In: Proc. of the 20th Int'l World Wide Web Conf. ACM, 2011. 805−814.

[23] Zheng YH, Zhang XY. Static detection of resource contention problems in server-side scripts. In: Proc. of the 34th Int'l Conf. on Software Engineering. IEEE, 2012. 584−594.

[24] Zheng YH, Zhang XY, Garnesh V. Z3-str: A Z3-based string solver for Web application analysis. In: Proc. of the 9th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. ACM, 2013. 114−124.

[25] Zheng YH, Zhang XY. Path sensitive static analysis of Web applications for remote code execution vulnerability detection. In: Proc. of the 35th Int'l Conf. on Software Engineering. IEEE, 2013. 652−661.

[26] Huang J. Scalable thread sharing analysis. In: Proc. of the Int'l Conf. on Software Engineering. ACM, 2016. 1097−1108.

[27] Bond MD, Coons KE, McKinley KS. Pacer: Proportional detection of data races. In: Proc. of the 31st ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2010. 255−268.

[28] Flanagan C, Freund S. Detecting race conditions in large programs. In: Proc. of the Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001). ACM, 2001. 90−96.

[29] Huang J, Zhou JG, Zhang C. Scaling predictive analysis of concurrent programs by removing trace redundancy. ACM Trans. on Software Engineering and Methodology (TOSEM), 2013,22(1):1−20.

[30] Khoshnood S, Kusano M, Wang C. ConcBugAssist: Constraint solving for diagnosis and repair of concurrency bugs. In: Proc. of the 2015 Int'l Symp. on Software Testing and Analysis. ACM, 2015. 165−176.

[31] Machado N, Lucia B, Rodrigues L. Concurrency debugging with differential schedule projections. In: Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2015. 586−595.

[32] Elmas T, Qadeer S, Tasiran S. Goldilocks: A race and transaction-aware Java runtime. In: Proc. of the 28th Annual ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2007. 245−255.

[33] Savage S, Burrows M, Nelson G, *et al*. Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. on Computer Systems, 1997,15(4):391−411.

[34] Voung JW, Jhala R, Lerner S. RELAY: Static race detection on millions of lines of code. In: Proc. of the 15th ACM SIGSOFT Symp. on the Foundations of Software Engineering. ACM, 2007. 205−214.

[35] Huang J, Zhang C. Persuasive prediction of concurrency access anomalies. In: Proc. of the ISSTA. ACM, 2011. 144−154.

[36] Sinha A, Malik S, Wang C, *et al*. Predictive analysis for detecting serializability violations through trace segmentation. In: Proc. of the MEMOCODE. IEEE, 2011. 99−108.

[37] Smaragdakis Y, Evans J, Sadowski C, Yi J, Flanagan C. Sound predictive race detection in polynomial time. In: Proc. of the 39th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2012). 2012. 387−400.

[38] Huang J, Meredith PO, Rosu G. Maximal sound predictive race detection with control flow abstraction. In: Proc. of the 35th Annual ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2014). ACM, 2014. 337−348.

孙全(1993－),男,安徽淮南人,硕士,主要研究领域为安卓数据竞争检测.

夏昕濛(1993－),男,博士生,主要研究领域为程序分析.

许蕾(1978－),女,博士,副教授,CCF 专业会员,主要研究领域为 Web 程序设计语言分析,Web 应用恶意代码识别分析.

张卫丰(1974－),男,博士,教授,CCF 高级会员,主要研究领域为代码仓库,持续集成,程序分析.