软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(5):2006−2025 [doi: 10.13328/j.cnki.jos.007203] [CSTR: 32375.14.jos.007203]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于事件标记的多粒度结合安卓测试序列约减*

郝 蕊, 冯 洋, 李玉莹, 陈振宇

(计算机软件新技术国家重点实验室 (南京大学), 江苏 南京 210093)

通信作者: 冯洋, E-mail: fengyang@nju.edu.cn

摘 要: 针对安卓自动化测试工具生成的崩溃测试序列包含过多冗余事件, 造成测试回放、缺陷理解与修复困难的现状, 很多测试序列约减工作被提出. 但目前工作仅关注应用界面状态变化而忽略了程序执行过程中内部状态变化, 此外, 目前工作仅在单一抽象粒度上对应用状态进行建模, 例如控件粒度或活动粒度, 导致约减后测试序列过长或约减效率低下. 针对以上问题, 提出基于事件标记的多粒度结合的安卓测试序列约减方法, 结合安卓生命周期管理机制、程序静态数据流分析等对触发程序崩溃的关键事件进行标记, 缩小序列约减空间, 并设计了低粒度粗筛选、高粒度细约减的策略. 最后, 收集包含程序间交互、用户输入等复杂场景的崩溃测试序列集, 在此数据集上与其他测试序列约减工作的对比评估结果也验证了所提方法的有效性.

关键词: 安卓测试; 崩溃回放; 测试序列约减

中图法分类号: TP311

中文引用格式: 郝蕊, 冯洋, 李玉莹, 陈振宇. 基于事件标记的多粒度结合安卓测试序列约减. 软件学报, 2025, 36(5): 2006–2025. http://www.jos.org.cn/1000-9825/7203.htm

英文引用格式: Hao R, Feng Y, Li YY, Chen ZY. Multi-granularity Fusion Android Test Sequence Reduction Based on Event Labeling. Ruan Jian Xue Bao/Journal of Software, 2025, 36(5): 2006–2025 (in Chinese). http://www.jos.org.cn/1000-9825/7203.htm

Multi-granularity Fusion Android Test Sequence Reduction Based on Event Labeling

HAO Rui, FENG Yang, LI Yu-Ying, CHEN Zhen-Yu

(State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210093, China)

Abstract: As too many redundant events included in crash test sequences generated by Android automated test tools may result in test replay, defect comprehension, and repairing difficulty, a great number of test sequence reduction works have been proposed. While current works only focus on the application interface changes and ignore the internal state changes during program execution. Moreover, current works only model application states at a single and abstract granularity, such as control widget granularity or activity granularity, resulting in long test sequences after reduction or inefficient reduction. This study proposes an Android test sequence reduction method combined with multi-granularity based on event labeling. By taking into account the Android lifecycle management mechanism and data flow analysis to label critical events that trigger crashes, this method can narrow the sequence reduction space and design a strategy of rough selection under low granularity and detailed reduction under high granularity. At last, a crash test sequence set containing complex scenarios such as inter-application interaction and user input is collected, and the comparison with other test sequence reduction works on this set verifies the effectiveness of the method proposed in this study.

Key words: Android test; crash replay; test sequence reduction

截至 2022 年第 3 季度, Google Play Store 拥有超过 350 万个安卓应用[1], 下载次数超过 270 亿次[2], 占据了市场份额的 80%, 与此同时, 复杂的交互环境、网络状态、多种系统兼容要求等也使得安卓应用相比传统的软件更易发生崩溃问题, 严重影响用户体验, 这也是导致应用程序用户评分低的最重要的问题之一[3−5]. 为了保障安卓应

* 部分工作为郝蕊在武汉理工大学做博士后期间完成
收稿时间: 2023-10-01; 修改时间: 2024-01-08; 采用时间: 2024-04-08; jos 在线出版时间: 2024-06-20
CNKI 网络首发时间: 2024-06-21

用质量与安全, 目前学术界及工业界发布了很多自动化测试工具, 例如 Monkey[6]、GUIRipper[7]、DynoDroid[8]、SwiftHand[9]、Sapienz[10]、Stoat[11]等, 它们通过采用随机化、模型化、系统化等不同搜索策略对应用进行功能测试, 并生成崩溃测试报告、崩溃测试序列、截图等, 方便开发者进行缺陷回放、理解与修复. 但测试工具生成的复现测试序列往往过长, 包括很多对触发程序崩溃无用的冗余事件[12−14], 使得开发者在进行程序回放、理解与修复时无法快速准确地找到触发程序崩溃的关键事件.

测试序列约减 (或测试输入约减) 是解决自动化测试工具生成测试序列过长问题的一种有效手段, 其中增量测试[15]是测试输入约减领域最早的经典算法之一, 但由于其采用随机搜索策略, 导致其生成的测试输入存在很多“非法约减”, 例如在安卓测试领域, 其生成的测试序列中的测试事件无法点击到有效控件. 后续研究工作者[16−19]结合安卓测试特点, 通过分析测试过程中应用 GUI 状态变化对测试序列构建树结构或图结构模型, 并在模型基础上进行测试序列约减, 大大提升了约减效率.

但目前的测试序列约减仍存在几个问题. 首先, 目前测试序列约减工作仅在单一粒度上对应用界面状态进行抽象, 例如 Jiang 等人[16]的工作 SimplyDroid 在活动粒度抽象, 只关注界面 Activity 名称是否变化, Sui 等人[18]提出的 ECHO 则在控件粒度抽象, 包括了页面控件及其属性的所有变化. 抽象粒度越低, 其关注的变化就会越细微, 导致序列约减时无法去除一些会造成界面变化但对触发程序崩溃无用的“次要事件”, 但抽象粒度太粗糙则会导致事件之间无法有效区分, 搜索空间过大, 约减时间过长. 此外, 目前测试序列约减工作仅关注应用 GUI 状态变化, 忽略了测试事件导致的程序内部状态变化, 比如测试事件对程序某些设置变量的改变造成程序崩溃, 这些事件虽然没有造成界面变化, 但却属于约减中需要关注的“重要事件”. 最后, 目前测试序列约减工作的评估只是在简单场景下进行, 缺少对程序间调用、设备网络状态改变、用户输入等复杂交互的支持与分析.

针对以上问题, 我们提出了基于事件标记的多粒度结合测试序列约减方法, 在控件粒度、页面布局粒度对比分析程序执行状态并构建模型, 分别进行粗筛选与细约减工作, 此外, 通过结合安卓系统生命周期管理机制、程序静态数据流分析技术, 我们对测试序列中的关键事件进行识别标记, 缩小序列搜索空间, 提升约减效率. 最后, 我们收集了来自安卓应用的 66 个崩溃测试序列, 其包含了程序切换、设备状态改变、用户输入等复杂交互场景, 评估结果验证了我们方法的有效性.

本文第 1 节介绍测试序列约减的相关方法和研究现状. 第 2 节介绍本文的动机示例. 第 3 节介绍本文的测试序列约减方法. 第 4 节介绍实验设置与结果讨论. 最后第 5 节对全文进行总结.

1 测试序列约减相关工作

增量测试是解决测试输入约减问题的一个经典算法[15], 其主要思想是不断迭代简化、验证测试输入, 直至找到触发崩溃的最小输入集合, 但由于其对所有输入一视同仁, 采用随机策略选取待验证输入, 导致其生成很多语义不可行的测试输入, 效率低下, 后续研究工作者们通过对程序输入结构进行分析, 采用层次化方式[20]、树形结构[21]对测试输入建模, 并设计相关约减算法以提升效率.

此外, 不同编程语言、不同系统的测试输入特性不尽相同, 有很多工作就是结合测试输入特性设计约减算法, Regehr 等人[22]提出了针对 C 语言编译器测试用例约减方法, 为了解决传统的增量测试约减算法产生很多过长或者无意义的测试用例 (编译无法通过) 的问题, 他们将定义了测试用例有效性验证, 并结合 C 语言特性提出了 3 种新的测试用例约减器. 基于模型的测试能够依照测试模型生成大量测试用例集, 从而提高测试覆盖率, 但也会导致生成的测试用例过长, 给故障定位带来挑战, 针对这个问题, Kanstrén 等人[23]提出了针对模型测试的用例约减, 他们不断生成出错序列的变异体, 并对变异体进行模式挖掘以辅助故障定位. 针对 JavaScript 应用程序中测试用例的回放约减, Wang 等人[24]利用测试用例中事件的变量使用信息构建事件上下文, 并要求约减后测试用例中的事件上下文与原始事件上下文保持一致, 以避免生成语法不可行的子事件序列. 在传统增量测试中, 每删除测试输入中的一个子输入集, 就可能导致需要对所有已经访问过的子输入集再次重验证, 对于大输入集, 这会造成巨大开销, 为了消除这种重新访问, Gharachorlu 等人[25]提出了测试输入减少的 3 个独立条件: 公共依赖顺序、无歧义性、

延迟移除, 在 C、C++、Rust 等语言测试用例集上的实验也验证了他们的方法效果.

与以上这些工作不同, 安卓界面测试的测试用例基本没有编程语言规则约束, 但由于其用例执行耗费时间长, 因此, 安卓测试用例执行对于其用例回放、错误理解与修复都至关重要. 近些年有一些针对安卓测试用例的约减工作被提出, Jiang 等人[16]认为安卓测试序列是由很多小会话组成, 每个小会话又包含几个独立操作事件, 这些事件的会话构成了测试用例约减的自然边界, 因为它们通常可以以高概率一起被约减, 因此他们利用分层结构对这些会话进行建模, 并在此基础上完成测试用例约减. Sui 等人[18]则从当前执行过程中获取当前屏幕界面 XML 信息, 并分析不同界面状态之间的差异来构建程序运行状态图, 并寻找图中最短路径算法作为约减序列. 此外, 为了有效记录大量事件执行的 GUI 状态, 他们还提出了基于滑动窗口的自适应模型来有选择性地注入界面状态检测. Yan 等人[19]通过对 Monkey 生成的测试序列进行手动分析, 定义了无操作、单一和组合无效果事件 3 种类型的无效事件, 并为其设计了 9 种约减规则, 此外, 他们还提出了一种静态界面状态层次树引导的测试序列约减方法. Choi 等人[26]提出了一种启发式的对安卓测试套件 (test suite) 进行约减的方法, 他们定义了 3 种可被直接约减的测试用例冗余模式: 对程序代码及界面覆盖率均无影响的测试用例、测试用例内对覆盖率无影响的环路、测试用例间共享的子测试序列.

此外, 安卓测试用例复现会面临不确定性问题, 例如动态布局、弹出窗口、网络连接状态改变等, 这些将会导致用例复现失败, Clapp 等人[17]提出了一种生成小测试事件集来大概率到达预期活动页面的技术, 从而可并非处理增量测试中对子序列集的挑选及验证, 以提升约减效率. Jiang 等人[27]则结合确定性回放工具来处理测试序列约减中可能会出现的不确定性情况, 并结合事件分组的方式缩短约减时间.

目前对安卓测试用例约减的工作仅从界面状态层次对事件进行分析, 并没有考虑操作事件对程序内部状态的作用, 此外, 目前工作均仅从单一粒度对界面状态建模, 使得方法的通用性不强, 生成的测试序列存在过长或无法约减的问题, 我们的工作则从控件粒度、页面布局粒度上分别对测试序列进行约减, 并且结合了静态数据流分析对操作事件对程序内部状态的影响进行了标记.

2 动机示例

LibreNews 是一款突发新闻通知的安卓开源软件, 它允许用户自定义新闻消息来源, 并且可进行通知频率、自动更新与否等方面的个性化设置. 图 1 给出了来自 LibreNews 应用的两个触发程序崩溃的测试序列. 序列 (a) 共包含 48 个事件, 其最短路径包括 3 个事件: 打开应用后点击“GO TO LIBRENEWS”按钮进入应用主页 (e_{60}), 然后按下后退按钮 (e_{81}), 程序页面并未发生变化, 然后点击“Automatically refresh”按钮 (e_{82}), 程序崩溃. 序列 (b) 共包含 124 个事件, 其最短路径包括 5 个事件: 打开应用后 (e_6) 点击“Server”, 将其地址修改为不合法地址“123456” (e_{113} 到 e_{115}), 然后点击“REFRESH”按钮 (e_{124}), 程序崩溃.

在对应用代码分析后, 我们发现序列 (a) 是因为事件 e_{47} 点击返回操作, 应用本应该回到欢迎界面 s_0 , 但却错误地进入了之前未被销毁的主页 s_6 , 这是典型的生命周期回调函数处理不当造成的崩溃错误. 而序列 (b) 则是因为事件 e_{114} 将获取新闻来源的服务器地址设置成了非法格式“123456”导致程序出错.

若以传统的测试序列仅考虑 GUI 界面变化的约减方法对序列 (a)、(b) 显然不合适, 例如序列 (a) 中导致程序崩溃的事件 e_{47} 在界面上没有导致任何变化. 此外, 通过对动机示例序列的分析, 可知合适的抽象粒度的选择对于提升序列约减效率也很重要, 例如仅在 Activity 层次上对程序状态进行抽象, 则状态 s_8 、 s_9 会被合并, 但显然这两个状态之间的变化对于崩溃能否被触发非常关键.

因此, 如何能够准确、快速地找出对触发程序崩溃起作用的重要事件, 对于提升测试序列约减效率非常重要, 这也启发我们结合安卓系统的生命周期管理机制对可能会导致返回栈顺序出错的事件进行识别, 并结合程序静态分析找出与出错函数有数据依赖关系的事件, 最后, 我们采用了多粒度结合的方式, 分别在控件及页面布局粒度上对程序执行状态进行抽象.

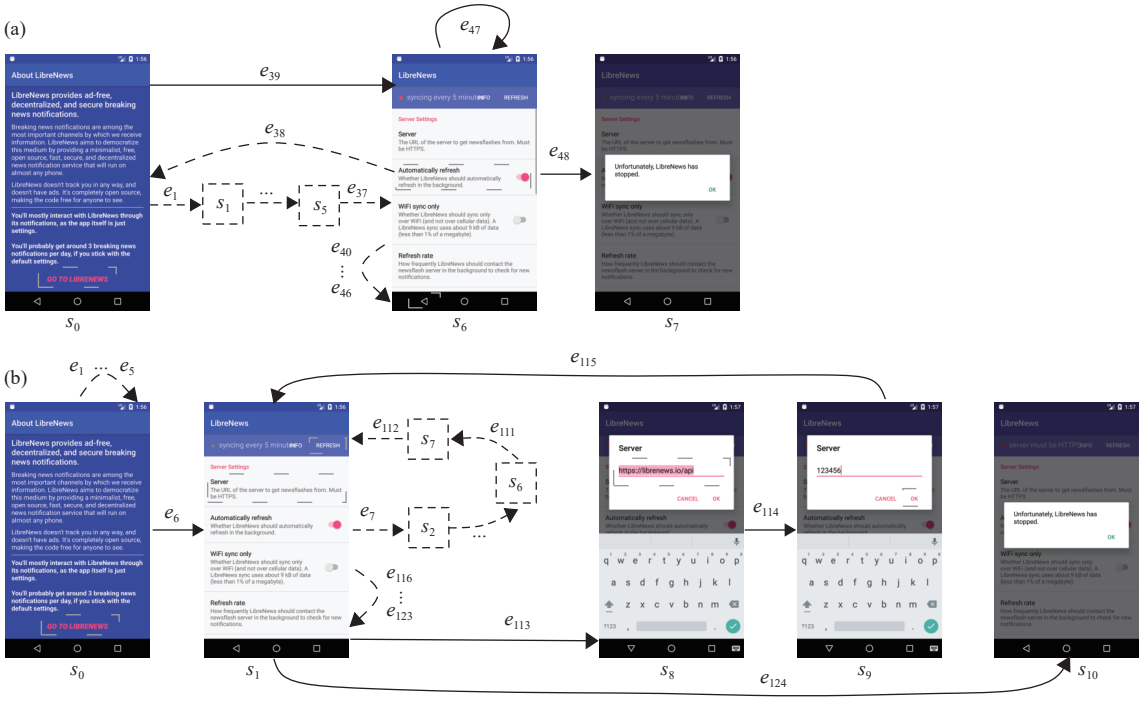

图 1 应用 LibreNews 崩溃测试序列

3 方法设计

本文提出了一种基于事件标记的测试序列多粒度约减方法 TREC, 我们将测试序列约减看作满足触发程序崩溃这个约束条件的最短路径搜索问题, 通过对事件优先级进行标记、在不同粒度上分层次约减等手段不断缩小搜索空间, 从而提升搜索效率. 方法框架如图 2 所示, 我们首先对程序状态进行细粒度上的抽象, 区分程序控件的细微变化, 这有利于我们利用相对少的时间对测试序列进行粗略约减, 找到一条不那么短的但是可以触发程序崩溃的测试序列, 然后我们将程序状态的抽象提升到仅仅考虑页面布局结构, 并对测试序列进一步约减. 在每个粒度的约减流程中, 我们首先去除了不会对测试界面及程序逻辑产生任何效果的无效事件, 然后构建程序执行过程中的状态跳转图, 跳转图包含了一个起始节点 (程序启动状态) 与结束节点 (程序崩溃状态), 我们的问题也变成了在状态跳转图中搜索连接起始节点和结束节点、并且可正确触发程序崩溃的最短路径.

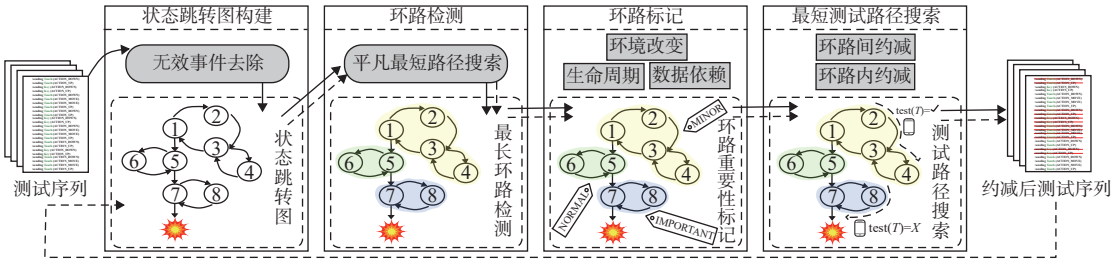

图 2 测试序列约减框架图 (→代表控件粒度上的约减流程, -→代表页面布局粒度上的约减流程)

为了寻找满足上述条件的最短路径, 我们首先找出单纯连接起始节点和结束节点的平凡最短路径, 并对此路径进行测试, 如果此路径能正确触发崩溃, 则认为我们找到了正确路径, 否则我们需要进一步搜索. 目前的路径无法正确触发崩溃的原因是因为其遗漏了某些测试步骤没有执行, 因此, 我们首先对于目前路径上的每个节点, 寻找以此节点为起始状态与结束状态的环路, 然后按照某些策略挑选环路并将其合并到目前的最短路径, 并测试新路

径能否触发崩溃. 特别的, 为了辅助进行高效环路搜索, 我们结合安卓系统的生命周期管理机制、操作步骤间的依赖关系对环路进行了不同重要性的标记, 这些重要性将会直接影响它们被选择的先后次序.

但本文中实现方法为局部最优, 约减后的测试序列符合当前粒度抽象构建的程序执行状态跳转图最短路径, 但并非触发程序崩溃绝对最短路径, 当抽象粒度越精细, 此粒度上的状态跳转图更能区分细微操作带来的影响, 这是在约减序列长度与约减时间的平衡, 我们将是否继续约减至绝对最优以及粒度抽象层次的选择权交给开发者.

3.1 状态跳转图构建

给定测试序列 $T=\{e_0,e_1,\ldots,e_i,\ldots,e_n\}$, e_i 代表测试序列中的第 i 个测试事件. 为了对测试序列执行过程中程序状态的变化进行准确表示, 我们为每个测试序列构建了状态跳转图, 其定义为 $G=(V,E)$, 其中 V 为节点集合, 每个节点代表一个应用界面状态, $E=\{e_0,e_1,\ldots,e_i,\ldots e_n\}$ 为边集合. 状态跳转图中的节点之间可以存在多条边, 并且会存在环. 在测试序列执行过程中, 我们在每个测试事件后插入监测事件以获得待测应用界面信息, 并通过对界面信息进行差异分析决定是否有新的节点产生. 在本文中, 我们关注的界面信息包括控件粒度以及页面布局粒度, 在控件粒度上我们关注页面上控件及其内容、属性的所有变化, 而在页面布局粒度上我们则只关注界面布局结构但忽略控件具体内容.

状态跳转图包含了一个起始节点 v_s 与终止节点 v_t, 起始节点对应的是待测应用的启动界面, 终止节点则代表了程序崩溃状态界面. 给定一个判定函数 $test$, 我们有 $test(T)=X$ 代表测试序列 T 触发了程序崩溃, 则我们的目标就变成了寻找状态跳转图中的路径 T', 其满足: 1) $T'\subseteq T$, 既 T' 为 T 的子测试序列; 2) T' 联通起始节点 v_s 与终止节点 v_t; 3) 触发程序崩溃, 即 $test(T')=X$; 4) 不存在路径 T'' 满足 $T''\subseteq T'$, 且 $test(T'')=X$.

此外, 以 Monkey 为代表的自动化测试工具中会有很多随机事件产生, 这些随机事情很多时候并不能真正作用在页面有效控件上, 针对这一点, 在构建状态跳转图过程中, 我们也即时进行无效事件约减. 在本文中, 我们定义无效事件为对测试状态转化不产生任何影响的事件, 比如在页面空白区域的点击事件等.

3.2 环路检测

在状态跳转图中寻找路径 T' 的最直观的做法是直接搜索起始节点 v_{initial} 与终止节点 v_{crash} 之间的最短路径 T_s, 但很多情况下最短路径 T_s 并不能触发程序崩溃, 例如图 1 中序列 (a), e_{47} 返回事件并没有导致应用程序界面发生任何变化, 序列 (b) 中 e_{113} 到 e_{115} 虽然使得应用程序界面发生了改变, 但这种改变没有在最短路径 T_s 上体现出来.

应用设置的改变仅体现在其所在的子路径上, 并没有在最短路径所在的主干路径上体现. 换句话说, 最短路径 T_s 无法触发程序崩溃是因为其缺失了某些环路. 在本文中, 我们定义 T_s 为平凡最短路径, 我们的目标则是寻找满足第 3.1 节中给出的 4 个条件的最短路径 T'.

在本文中, 我们提出了时序一致的最长环路检测算法, 其会在状态跳转图中寻找所有以最短路径 T_s 中的节点为起止点的环路. 这些检测到的环路在后续测试路径搜索算法中将会作为输入集, 依次被合并到 T_s 中并验证是否可以正确触发程序崩溃. 我们的环路检测算法有如下两个特点.

1) 事件执行顺序一致性: 在节点 v 上可能存在多条环路, 它们之间的执行顺序直接影响测试程序崩溃与否, 例如先有环路对应用程序某内部设置 (例如 language settings) 进行修改, 然后后续环路进行了这些设置影响到的某些操作, 在它们的共同作用下, 应用程序崩溃. 如果没有对测试序列中的原本执行顺序进行保持, 则无法成功复现这些崩溃错误;

2) 最长环路检测: 由于环路内容可能还存在一条甚至多条子环路, 若对其依次进行检测并验证是否可触发程序崩溃, 其平均效率为 $\mathrm{O}(n)$. 我们在检测环路时采用最长环路策略, 在此基础上, 后续测试路径搜索时可以采用二分方法对最长环路进行迭代验证, 将平均搜索效率降低到 $\mathrm{O}(\log(n))$.

其具体思路如算法 1 所示. 给定状态跳转图 G、起始节点 v_{initial} 到终止节点 v_{crash} 间的最短路径 T_s, 函数 findLoops 将会遍历路径 T_s 上经过的所有节点, 并检测出所有以此节点为起止状态的所有环路. 为了保持原测试序列中的事件执行顺序, 在检测每个节点上的环路时, 我们获得此节点最后一条入边的事件编号 *start* 与第 1 条出边的事件编号 *end* (第 5–8 行), 仅对它们之间的事件进行环路检测. 函数 findNodeLoops 负责检测节点 v 上的所有

环路, 其首先选取节点 v 上落在 *start* 、 *end* 之间的所有出边与入边, 这里我们还需保证入边事件一定发生在出边事件之前, 才能检测到环路 (第 14–16 行). 接下来, 依次遍历出边集 E_{out} 与入边集 E_{in}, 得到事件对 $(e_{\text{out}}, e_{\text{in}})$, 即可得到由 e_{out} 到 e_{in} 之间的所有事件组成的最长环路.

算法 1. 环路检测.

1. **Function** findLoops(G, T_s)
2. $L \leftarrow \varnothing$; //待返回的结果环路集合
3. **for** each event e on path T_s **do**
4. $v_s \leftarrow e.source$; // 事件 e 的源节点
5. $start \leftarrow 0$;
6. **if** $v_s \neq v_{\text{initial}}$ **then**
7. $start \leftarrow v_s.lastIncomingEvent.id$; //获取源节点的入边集中的最大 *id*
8. $end \leftarrow e.id$;
9. $L_{v_s} \leftarrow$ findNodeLoops($G, v_s, start, end$); // 检测以 v_s 作为起始及终止节点的环路
10. insert all loops in L_{v_s} into L;
11. **return** L;
12. **Function** findNodeLoops($G, v, start, end$)
13. $L \leftarrow \varnothing$; // 待返回的结果环路集合
14. $E_{\text{out}} \leftarrow$ filter($G.outgoingEvents(v), start, end$); // 选取 *id* 在 $[start, end]$ 之间的出边
15. $start \leftarrow \min(E_{\text{out}})$; // 重新设定 *start* 为所有出边中的最小 *id*
16. $E_{\text{in}} \leftarrow$ filter($G.incomingEvents(v), start, end$); // 选取 *id* 在 $[start, end]$ 之间的入边
17. **for** i in $[0, E_{\text{out}}.size)$ **do** // 此处 $E_{\text{out}}.size$ 不大于 $E_{\text{in}}.size$
18. $e_{\text{out}} \leftarrow E_{\text{out}}.item(i)$;
19. $e_{\text{in}} \leftarrow E_{\text{in}}.item(i)$;
20. $l \leftarrow \{e_k : e_{\text{out}}.id \leqslant e_k.id \leqslant e_{\text{in}}.id\}$; // 环路 l 包含 e_{out} 到 e_{in} 之间的所有事件
21. insert l into L;
22. **return** L;

3.3 环路标记

状态跳转图中包含的环路有时候会达到几十甚至上百, 此外, 有时需要在多个环路的共同作用下程序崩溃才会触发, 因此对所有环路及它们的组合进行验证是一项非常耗时的工作. 在本文中, 我们采用对标记环路重要性的策略, 将与程序崩溃触发最有关联的环路单独进行标记, 并在后续最短测试路径搜索过程中采用由高到低的方式依次对不同重要性环路进行搜索, 降低算法开销.

我们将环路分为 3 个层次: 重要环路 (*IMPORTANT*)、普通环路 (*NORMAL*)、次要环路 (*MINOR*). 一条环路的优先级由此环路中所有事件的最高优先级决定. 其中次要事件主要是与应用逻辑无关的按键操作 (KeyEvent), 比如调整设备声音大小、屏幕亮度.

在本文中, 我们关注 3 个方面的重要事件, 分别是执行环境改变事件、生命周期相关事件、数据流依赖相关事件. 其中执行环境改变事件是指改变设备网络状态、位置服务状态、信号状态、电池状态、传感器数据设置等, 这些事件可直接通过对测试脚本中事件格式解析的方式进行识别.

安卓系统通过返回栈机制管理不同应用的活动, 活动在入栈、出栈过程中伴随了一系列生命周期状态的变化, 安卓组件需要在生命周期状态改变时调用不同回调函数来对布局、数据、资源进行妥善处理. 安卓中最常见的 Activity 活动组件提供了 7 个与生命周期管理相关的回调函数, 例如活动创建、暂停、销毁、重启等, 而 Fragment

组件则共有 11 个回调函数, 如果开发者没有正确处理这些回调函数, 会导致应用在设备旋转、应用切换等复杂交互中崩溃. 因此, 在本文中我们也将与组件生命周期变化相关的事件标记为重要事件, 包括应用切换事件、设备旋转事件、返回按键事件. 在实现上, 设备旋转事件、环境改变事件均可以通过测试脚本中事件格式解析的方式直接进行标记, 应用切换事件可通过状态跳转图中事件前后应用是否改变进行检测并标记.

此外, 某些操作事件对应用中变量进行修改, 这些修改虽然没有直接反映到 GUI 界面变化上, 但却会对应用逻辑产生影响, 继而触发程序崩溃. 例如图 1 序列 (b) 中操作事件 e_{113} 到 e_{115} 修改了 server 地址, 这个地址格式不正确时软件应用会崩溃. 为了检测这种类型的操作事件, 我们对崩溃 stack trace 进行分析, 获取崩溃错误类型、出错函数、错误信息, 并以出错函数为目标对应用进行后向数据流分析 (backward analysis)[28], 获取对出错函数调用产生影响的相关变量及变量值, 作为崩溃错误关键词, 对操作事件本身则通过 GUI 界面分析获取其作用到的控件, 得到控件上属性、内容相关关键词, 可以通过属性 text、resource-id、content-desc 等获取关键词, 而且平凡最短路径 T_s 的搜索也需遵循原测试序列中的事件时序. 最后通过对两类关键词进行相似度分析从而判断某操作事件是否应该被标记. 在本工作中, 采用轻量分析策略, 仅对 IllegalArgumentException、ArrayIndexOutOfBoundsException、NullPointerException 等几种与数据依赖最相关的崩溃错误进行依赖分析, 忽略了 NoClassDefFoundError、IllegalStateException 等仅与程序正确实现与否、程序栈状态是否正确等对数据流不敏感的错误, 并且采用严格关键词匹配策略, 只有当某一关键词完全匹配时才对操作事件进行标记, 这么做的原因是控制被标记的重要事件数量, 避免过多无关事件被标记从而影响后续最短路径搜索效率.

3.4 最短测试路径搜索

给定状态跳转图, 我们在图上进行起点到终点之间的最短路径 T' 的搜索, 其具体思路如算法 2 所示. 算法首先利用深度遍历的方式找到平凡最短路径 T_s[2] , 如果其能正确触发程序崩溃, 则直接终止搜索并将 T_s 作为找到的路径返回 (第 1–4 行). 若 T_s 无法正确触发程序崩溃, 算法检测在跳转图中所有以 T_s 上的节点为起止点的环路, 并对其重要性程度进行标记 (第 5, 6 行). 接下来算法将进行不同重要性层次上的环路集约减, 由于能影响程序崩溃是否触发的环路 (在本文中我们定义为必要环路) 更可能被标记为高重要性环路, 因此我们在约减时遵循重要性由高到低的顺序以便提升搜索效率 (第 7 行), 变量 *level* 代表了搜索算法的起始层次, 只有在所有高重要性的环路均无法正确触发程序崩溃时, 才会进入低重要性环路集约减, 举例来说, 当 *level* 为 *MINOR* 时, 说明搜索算法在 *IMPORTANT* 、 *NORMAL* 层次上均已失败, 要寻找的最短路径 T_s 中包含了 *MINOR* 必要环路.

算法 2. 最短测试路径搜索.

```
1. Function search(G)
2.    T_s ← shortestPath(G); //获取平凡最短路径
3.    if test(T_s) = X then
4.       return T_s;
5.    L ← findLoops(G, T_s); // 检测状态跳转图中所有环路
6.    L ← coloring(G, L); // 对所有环路进行重要性标记
7.    for level in {IMPORTTANT, NORMAL, MINOR} do
8.       level' ← level; // 起始搜索层次
9.       L'_changes ← ∅; // 存放所有比 level' 重要性低的必要环路
10.      while level' ⩽ IMPORTANT do
11.            T_base ← T_s + filterLoopsGreaterThan(L, level') + L'_changes; // 构建基础测试路径
12.            if test(T_base) = X then // 说明重要性为 level' 的环路均为冗余环路
13.                  T_base ← T_base − filterLoopsGreaterThan(L, level'); // 更新基础测试路径
14.                  if test(T_base) = X then // 说明更高重要性的所有环路均为冗余环路
```

15. $T' \leftarrow T_s +$ reduce($G, T_{\text{base}}, L'_{\text{changes}}$);
16. **return** T';
17. $level' \leftarrow level' + 1$; // 此重要性层次空间内, 未能搜索到最短测试路径
18. **continue**;
19. $L_{\text{changes}} \leftarrow$ filterLoops($L, level'$); // 获取当前重要性层次上的所有环路
20. $L'_{\text{changes}} \leftarrow L'_{\text{changes}} +$ ddmin($T_{\text{base}}, L_{\text{changes}}, 2$); // 对当前重要性层次上的所有环路进行约减
21. $T_{\text{base}} \leftarrow T_s +$ filterLoopsGreaterThan($L, level'$) $+L'_{\text{changes}}$;
22. **if** $level' = IMPORTANT$ **and** $test(T_{\text{base}}) = X$ **then**
23. $T' \leftarrow T_s +$ reduce($G, T_{\text{base}}, L'_{\text{changes}}$); // 成功搜索到最短路径
24. **return** T';
25. $level' \leftarrow level' + 1$;
26. **return** $G.longestPath$; // 路径搜索失败, 返回原测试路径

算法 2 中第 8–25 行代码在重要性层次 *level* 上对环路集 L 进行约减, 其中变量 $level'$ 指示当前搜索重要性层次, 变量 L'_{changes} 保存了所有更低重要性的已经约减成功的必要环路, 函数 filterLoopsGreaterThan($L, level'$) 则返回所有更高重要性的环路, 这两种环路在当前层次 $level'$ 上无需进行约减. 在进行约减时, 算法首先生成所有可保持原样的环路集合, 其由平凡最短路径 T_s 、更高层次环路、低层次必要环路组成, 定义为基础测试路径 (第 11 行), 若此基础测试路径无法正确触发程序崩溃, 则说明当前层次环路中存在必要环路, 算法将获取重要性为 $level'$ 的所有环路进行约减 (由函数 ddmin 完成), 并将约减后的结果更新到 L'_{changes} 中, 然后进行下一层次的搜索 (第 19–25 行). 若此基础测试路径可正确触发程序崩溃, 则说明当前层次环路均为冗余环路, 此时我们进一步对将所有更高重要性层次的环路也去除并检测能否正确触发程序崩溃, 如果可以, 则说明所有更高层次的环路也均为冗余环路, 可直接将目前获得的最短路径返回 (第 12–16 行), 否则, 继续进行下一层次的搜索 (第 17, 18 行).

算法 3 中的函数 ddmin(T_{base}, L, k) 利用分治策略[15]对环路集 L 进行约减 (第 1–13 行), 前提为 $test(T_{\text{base}}) \neq X$ 但 $test(T_{\text{base}} + L) = X$, 即基础测试路径 T_{base} 无法触发程序崩溃, 但合并上环路集 L 后可以触发. 其约减的基本思路为将环路集 L 均分为 k 份依次进行验证, 假设第 i 份子环路集可以成功触发程序崩溃, 则将其作为函数 ddmin 的输入继续进行分治 (第 4–7 行), 若第 i 份子环路集均无法触发程序崩溃, 则对它的补集进行验证、分治 (第 8–10 行). 若划分后所有子环路集、子环路补集均无法触发程序崩溃, 则增大划分粒度后继续约减, 直至粒度超过环路集 L 中包含的环路数量或环路集 L 中包含环路数量过少无法划分.

算法 3. 环路分治约减.

1. **Function** ddmin(T_{base}, L, k)
2. **if** $L.size = 0$ **or** $L.size = 1$ **then** // 无法进行约减, 直接返回
3. **return** L;
4. **for** i in $[0, k)$ **do**
5. $L_{\Delta i} \leftarrow$ calPartition(L, i); // 获取 L 的 k 均分后的第 i 个子环路集
6. **if** $test(T_{\text{base}} + L_{\Delta i}) = X$ **then**
7. **return** ddmin($T_{\text{base}}, L_{\Delta i}, 2$); // 对子环路集继续约减
8. $L_{\nabla i} \leftarrow L -$ calPartition(L, i); // 获取第 i 个子环路集补集
9. **if** $test(T_{\text{base}} + L_{\nabla i}) = X$ **then**
10. **return** ddmin($T_{\text{base}}, L_{\nabla i}, \max(k-1, 2)$); // 对子环路集补集继续约减
11. **if** $k < L.size$ **then**

12. **return** ddmin($T_{\text{base}}, L, \min(L.size, 2k)$) ; // 增大划分粒度
13. **return** L_{changes};

由于最初环路集 L 中的环路均为最长环路 (参见第 3.2 节), 对算法 2 中搜索到的必要环路, 我们还需对其进一步约减以去除它上面存在的冗余子环路, 具体思路如算法 4 所示. 算法依次遍历环路集 L 中的所有环路, 并检测其是否包含子环路, 若不包含子环路, 则直接返回 (第 4–6 行), 否则利用 reduceLoop 函数对其进一步约减 (第 7–9 行). 函数 reduceLoop 对环路 l 进行约减, 其利用 Dijkstra 最短路径算法获得环路 l 中的最短路径并进行验证, 若其可以正确触发程序崩溃, 则将其视为约减结果返回 (第 14–16 行), 否则同样利用 ddmin 函数对环路 l 中存在的所有子环路集进行分治约减 (第 17 行), 直至环路中不存在子环路 (第 12, 13 行) 或前后两次约减的结果相同 (第 18, 19 行).

算法 4. 环路内约减.

1. **Function** reduce(G, T_{base}, L)
2. $L' \leftarrow \varnothing$; // 约减后的路径集
3. **for** path l in L **do**
4. **if** l doesn't contain inner loop **do**
5. insert l into L' ; // 当前环路无子环路, 无需约减
6. **continue**;
7. $l' \leftarrow$ reduceLoop(G, $T_{\text{base}} + L - l, l$); // 对当前环路进行进一步约减
8. replace l of L with l'; // 用约减后的路径 l' 更新 L, 提升后续 *test* 函数验证效率
9. insert l' into L';
10. **return** L';
11. **Function** reduceLoop(G, T_{base}, l)
12. **if** l doesn't contain inner loop **do**
13. **return** l; // 当前环路无子环路, 无需约减
14. $l_s \leftarrow$ shortestPath(l); // 获取当前环路上起点到终止点之间的最短路径
15. **if** $test(T_{\text{base}} + l_s) = X$ **then**
16. **return** l_s; // 当前环路约减成功, 返回
17. $l' \leftarrow l_s + \text{ddmin}(T_{\text{base}} + l_s, \text{findLoops}(G, l_s), 2)$; // 检测当前环路中的所有子环路集, 并对其进行约减
18. **if** $l'.length = l.length$ **then**
19. **return** l'; // 当前环路已经约减成功, 返回
20. **return** reduceLoop(G, T_{base}, l'); // 对环路进行迭代约减

值得说明的是, 安卓测试序列执行非常耗时, 而在约减算法中则会反复对中间搜索到的测试路径进行执行以便验证是否可触发程序崩溃, 因此, 在算法实现过程中, 我们还进行了中间测试结果的缓存, 包括每次进行验证的路径及其验证结果, 避免了相同路径的多次执行, 提升搜索效率.

此外, 以上说明的是在控件粒度上的测试路径搜索, 对于页面布局结构粒度的测试路径搜索, 由于其输入数据 (即控件粒度约减后的测试序列) 已经体现了测试事件标记优先级的作用, 因此, 我们仅进行了简单的环路分治约减 (算法 3 中 ddmin 函数) 及环路内约减 (算法 4 中 reduce 函数).

4 实验分析

4.1 实验数据

为了获取测试数据, 我们采用与 Monkey 相同的随机测试策略, Monkey 是安卓系统内部提供的测试工具, 被很

多研究工作用来收集测试数据或对比测试工具. 但是随机测试对于与程序逻辑有关的操作无法高效生成, 例如示例中改变 server 值操作, 另外, 一些程序也要求进行用户登录才可进行后续操作. 为了发现更丰富、贴合实际的崩溃测试序列, 我们在完全随机的基础上加入了处理程序输入值、用户登录数据等模块, 辅助更高效的发现崩溃测试序列.

我们收集了来自 DroidDefects[4]与 Droixbench[29]两个数据集中的测试应用, 它们均提供了安卓应用上的测试崩溃错误, 被很多安卓测试方面的工作采用[30−35]. 此外, 我们通过对其提供的崩溃错误复现步骤进行分析, 获取了一系列测试输入值. 通过这种方式, 我们最终获得了来自 7 个应用的 66 个崩溃测试序列, 如表 1 所示, 这些应用来自 GitHub、FDroid 等软件代码管理平台, 包含了日常工具、游戏、新闻媒体等多种应用类型, 其测试序列长度范围为 8–797, 并且包含了用户登录、输入等复杂情景. 在之前的研究工作中, Echo[18]中仅提供了 5 个测试序列, Simply-Droid[16]中虽然提供了 95 个测试序列, 但仅 25 个测试序列来自真实错误, 其余均为变异错误, 相比之下, 我们的数据集不仅包含了更多的真实错误序列, 也反映了更丰富的测试场景.

表 1 实验数据集

应用程序	应用描述	应用版本	API版本	错误类型	序列数	序列长度
AntennaPod	播客管理	1.5.0.1	4.4	NoClassDefFoundError	1	75
ATimeTracker	时间管理	0.51.2	6.0	NullPointerException	8	64–146
ButterKnife	安卓工具	1.0.0	4.4	NullPointerException	3	8–13
CampFahrplan	日程浏览	1.32.2	4.4	IllegalArgumentException	7	49–79
LibreNews	新闻通知	1.4	6.0	NullPointerException ArrayIndexOutOfBoundsException	15	41–124
OpenSudoku	数独游戏	1.1.5	6.0	NullPointerException	24	66–797
Transistor	广播电台	1.1.5	4.4	IllegalStateException	8	97–350

4.2 方法实现

对于测试数据收集, 我们采用 Appium 测试框架完成随机测试策略与程序执行状态获取与分析, 我们共支持点击、拖拽、缩放、按键操作、用户等待、输入、设备旋转等多种事件, 为了验证测试序列的可复现性, 我们对每个测试序列重复执行了 5 遍确认其测试结果均可触发程序崩溃.

在方法实现方面, 我们利用 Soot[36]进行软件的基础解析, FlowDroid[37]进行静态数据流分析, FlowDroid 是目前对安卓应用进行静态污点分析的最好的工具之一, 已经被很多学术界与工业界的工作采用[38−42], 它完整建模了安卓的控件生命周期, 并且满足上下文敏感、流敏感、对象敏感、域敏感, 可以得到高精度的分析结果. 具体来说, 给定安卓 APK 程序, Soot 首先对程序进行解析获得 Jimple 中间码, FlowDroid 在此中间码基础上进行静态数据流分析, 因此本文方法即可对 Java 源代码进行分析, 也可以在缺乏源代码的情况下对 APK 程序进行分析. 本文实验均在处理器为 Intel i7 8750, 内存为 32 GB, 操作系统为 Windows 10 的计算机完成, 实验语言为 Java.

4.3 研究问题

为了验证 TREC 的效果, 我们将其与之前的两个最相近的工作进行对比: SimplyDroid[16]、ECHO[18], 这两个工作均提供了源代码, 我们直接采用其默认参数与实现进行实验, 此外, SimplyDroid 工作提供了 3 个算法版本, 我们采用其工作中介绍的效果最优的 LHDD 方法.

在评价指标上我们主要关注 3 个方面的表现: 约减结构正确性、约减序列长度、约减效率. 其中约减结果正确性是指方法能成功找到触发程序崩溃的序列, 这是基础与前提, 此外, 约减算法还应尽可能找到短的触发程序崩溃的序列, 序列越短, 则后续开发者或开发工具在进行错误理解与错误定位时, 需要关注的信息越精简, 无用干扰信息越少. 约减效率则不仅包括搜索最短序列的算法运行时间, 也包括前期对测试序列分析准备时间. 综上, 我们提出了如下 5 个研究问题.

- 研究问题 1: TREC、SimplyDroid、ECHO 约减后的测试序列是否可以正确触发程序崩溃?
- 研究问题 2: TREC 约减后的测试序列是否比其他方法约减后的测试序列更短?

- 研究问题 3: TREC 约减时间是否比其他方法更少?
- 研究问题 4: TREC 中控件粒度、页面布局粒度约减模块在约减效果与效率上分别表现如何?
- 研究问题 5: TREC 中环路间、环路内约减模块在约减效果上分别表现如何?

在研究问题 1 和 2 中, 我们对各方法约减后的测试序列进行验证, 检测其是否可正确触发程序崩溃, 并对各方法约减后的测试序列长度进行对比. 在研究问题 3 中, 我们对各方法生成约减序列的时间进行了记录, TREC 方法需要对测试序列进行一次事件标记, 涉及生命周期相关事件监测及应用静态数据流分析, 我们同样对其时间进行了记录. 我们还对每个方法约减时间进行了配对 t 检验以分析其是否在统计学意义上有显著差异, 其计算结果显著性 p 值越小, 说明样本数据代表的总体差异越大, 一般认为 $p < 0.05$ 时, 代表两者有显著差异. 此外, TREC 采用控件、页面布局两个精度结合的约减策略, 因此我们在研究问题 4 中对这两个模块在这 3 个指标上分别进行检验, 以分析每个约减模块是否有效, 从而检测 TREC 这种多精度结合的约减策略是否有效. 研究问题 4 对不同粒度约减效果进行了分析, 在此基础上, 我们在研究问题 5 中对每个粒度的约减模块内部的约减算法进行了进一步分析, 具体来说, 在第 3.4 节最短路径搜索中算法 3 及算法 4 分别进行了环路间约减以及环路内约减, 而能否在尽量少次的约减中去除无效环路会直接影响算法效率, 因此在此问题中我们将会对环路约减具体表现进行探讨. 最后, 由于 TREC 方法中包含了无效事件去除, 而测试序列长度会影响测试指标的结果 (尤其是测试效率), 因此, 为了公平性, 我们统一对测试序列中无效事件进行去除, 然后将测试序列输入给 TREC、SimplyDroid、ECHO 这 3 个方法进行约减.

4.4 实验结果与分析

4.4.1 研究问题 1: 约减结果正确性分析

表 2 给出了 TREC 与两个对比方法约减后的序列能否触发程序崩溃的验证结果, 其中最左侧两列分别是每个测试序列的标号及序列长度 (即序列包含事件数量), 右侧“崩溃”中 Y 代表约减后序列可触发程序崩溃, N 则代表无法触发.

表 2 方法 TREC、ECHO、SimplyDroid 约减效果与效率表

ID	总长度	ECHO			SimplyDroid			TREC				
		长度	时间 (min)	崩溃	长度	时间 (min)	崩溃	长度	标记时间 (min)	约减时间 (min)	总时间 (min)	崩溃
1	98	6	22.31	N	12	949.79	Y	12	17.80	300.10	317.90	Y
2	98	6	34.79	N	14	646.63	Y	12	41.15	458.56	499.71	Y
3	141	6	38.18	N	20	13 804.50	Y	12	41.99	454.94	496.93	Y
4	106	6	23.89	N	78	21 864.08	N	12	32.51	255.26	287.77	Y
5	64	6	25.25	N	16	787.39	N	12	17.18	213.21	230.39	Y
6	104	6	24.76	N	60	10 783.38	N	12	30.54	296.10	326.64	Y
7	146	8	33.15	N	90	30 591.68	N	14	16.23	318.58	334.81	Y
8	101	6	28.88	N	17	4 360.93	Y	12	15.20	384.22	399.42	Y
9	73	2	19.67	N	4	123.90	Y	3	13.16	99.53	112.69	Y
10	41	3	21.40	N	5	771.37	Y	8	28.09	296.58	324.67	Y
11	49	2	22.70	N	9	600.53	Y	3	35.70	18.44	54.14	Y
12	48	2	22.22	N	3	145.79	Y	3	11.16	14.08	25.24	Y
13	70	2	23.93	N	3	9.80	Y	3	38.64	9.20	47.84	Y
14	68	2	21.90	N	3	24.62	Y	3	19.92	4.97	24.89	Y
15	87	2	20.82	N	3	149.34	Y	3	22.49	568.85	591.34	Y
16	66	7	37.03	N	3	412.91	Y	6	12.55	168.80	181.35	Y
17	124	2	20.60	N	8	1 794.57	Y	5	20.58	2 061.91	2 082.49	Y
18	61	5	27.62	Y	3	991.01	Y	5	14.33	50.09	64.42	Y
19	52	5	27.46	Y	6	116.47	Y	5	27.92	34.07	61.99	Y
20	50	5	30.36	Y	7	339.66	Y	5	14.16	41.81	55.97	Y
21	61	5	27.06	Y	6	602.01	Y	5	32.42	56.51	88.93	Y
22	83	5	39.86	Y	7	366.19	Y	5	37.20	37.97	75.17	Y

表 2 方法 TREC、ECHO、SimplyDroid 约减效果与效率表 (续)

ID	总长度	ECHO			SimplyDroid			TREC				
		长度	时间 (min)	崩溃	长度	时间 (min)	崩溃	长度	标记时间 (min)	约减时间 (min)	总时间 (min)	崩溃
23	63	5	27.74	N	3	2953.92	Y	5	27.98	180.64	208.62	Y
24	49	3	64.25	N	20	2762.18	N	9	24.12	232.08	256.20	Y
25	64	3	44.88	N	30	5190.21	N	9	26.57	156.28	182.85	Y
26	64	5	52.67	N	31	5792.33	N	11	34.45	164.97	199.42	Y
27	79	3	46.27	N	10	762.60	Y	9	35.49	256.38	291.87	Y
28	69	3	47.42	N	40	6963.39	N	9	29.22	213.31	242.53	Y
29	56	5	51.64	N	26	3432.85	N	11	38.50	145.32	183.82	Y
30	66	3	46.10	N	35	5542.74	N	9	29.44	186.67	216.11	Y
31	75	6	21.46	N	16	3151.00	Y	9	37.32	796.30	833.62	Y
32	90	6	19.92	N	37	5450.77	N	9	29.16	356.52	385.68	Y
33	66	14	32.02	N	23	2217.32	N	14	24.02	545.37	569.39	Y
34	76	13	29.53	N	9	1674.16	Y	14	20.46	936.73	957.19	Y
35	106	28	18.45	N	14	3005.76	Y	17	37.64	725.16	762.80	Y
36	312	80	131.11	N	18	2219.05	Y	9	31.68	5369.76	5401.44	Y
37	133	38	65.86	N	69	18055.44	N	24	43.41	1631.44	1674.85	Y
38	102	18	36.54	N	9	644.78	Y	9	42.88	293.12	336.00	Y
39	177	37	64.91	N	9	748.35	Y	9	18.82	870.15	888.97	Y
40	86	5	18.31	N	9	3465.46	Y	15	23.60	814.93	838.53	Y
41	71	16	34.35	N	18	2897.58	Y	13	25.16	363.01	388.17	Y
42	108	11	29.46	N	13	3371.75	Y	14	36.68	560.93	597.61	Y
43	107	24	46.01	N	13	2860.66	Y	19	33.24	2494.87	2528.11	Y
44	223	35	44.94	Y	17	10029.10	Y	14	27.07	615.76	642.83	Y
45	478	44	19.51	N	16	3927.86	Y	8	21.55	3813.72	3835.27	Y
46	79	6	19.76	N	10	523.07	Y	9	16.91	422.92	439.83	Y
47	103	14	30.59	N	9	815.97	Y	9	17.51	406.56	424.07	Y
48	201	41	69.05	N	13	2192.70	Y	10	18.24	6540.70	6558.94	Y
49	198	24	45.31	N	9	785.83	Y	9	20.22	656.54	676.76	Y
50	214	19	39.53	N	9	1133.61	Y	9	22.58	597.07	619.65	Y
51	359	24	46.25	N	13	4254.30	Y	9	40.78	797.41	838.19	Y
52	197	28	52.81	N	9	1039.30	Y	10	24.48	2563.99	2588.47	Y
53	376	30	54.42	N	13	2904.39	Y	10	35.41	946.66	982.07	Y
54	797	38	67.20	N	28	5776.50	N	9	36.13	1069.21	1105.34	Y
55	97	3	40.56	N	32	4447.66	N	7	0.63	332.76	333.39	Y
56	124	7	45.90	N	77	23736.22	N	10	0.23	730.09	730.32	Y
57	279	3	37.06	N	89	54593.30	N	7	0.31	730.99	731.30	Y
58	350	18	75.97	N	75	49846.06	N	7	0.56	912.52	913.09	Y
59	198	8	48.96	Y	107	40879.73	N	8	0.03	213.93	213.96	Y
60	224	3	37.24	N	144	78926.18	N	8	0.50	414.32	414.82	Y
61	108	3	37.29	N	37	6217.88	N	7	0.68	255.49	256.18	Y
62	181	7	47.64	N	96	32776.05	N	8	0.83	504.41	505.25	Y
63	9	1	25.01	Y	1	27.87	Y	1	0.00	11.45	11.45	Y
64	13	2	21.82	Y	2	27.75	Y	2	0.00	13.21	13.21	Y
65	8	1	10.70	Y	1	27.50	Y	1	0.00	8.68	8.68	Y
66	75	2	15.67	Y	5	1329.04	Y	2	0.00	13.29	13.29	Y

从表 2 中可以看出, ECHO 算法对 55 个序列均无法生成可触发程序崩溃的测试序列, 占到了总序列数的 83%, 就是因为 ECHO 直接将平凡最短路径作为最终约减结果返回, 并未考虑其是否可正确触发程序崩溃这一约束条件. 例如图 1 中给出的动机示例 (a), ECHO 算法仅寻找最短平凡路径 $e_{39} \rightarrow e_{48}$ 作为最终约减路径, 而关键事

件 e_{47} 由于未造成 GUI 界面的跳转变化被其忽略, 导致其约减后路径无法触发崩溃. SimplyDroid 算法约减后的结果中有 22 个测试序列无法触发崩溃, 达到了总序列数的 33%, 通过对其无法正确约减的序列进行分析, 我们发现其在 Activity 层次进行抽象, 导致约减策略过于粗略, 很多事件被合并到一起进行验证, 程序无法崩溃时则直接忽略这些事件, 导致其中一些原本对触发程序崩溃有作用的关键事件被去除, 例如序列 5 来自 ATimeTracker 应用, 在此序列中, 其触发崩溃的操作为创建活动并打开设置, 将其 sound 属性从 disabled 切换为 enabled, 然后返回活动列表并点击创建的活动, 其中切换 sound 属性的步骤仅改变了 sound 控件的状态, 而由于 SimplyDroid 在 Activity 层次进行建模, 忽略了此关键步骤, 导致其约减结果无法触发崩溃. 我们的方法 TREC 则对所有的测试序列均可返回正确触发程序崩溃的结果.

4.4.2 研究问题 2: 约减序列长度分析

表 2 中右侧第 3、6、9 列给出了每个算法对各个测试序列约减后的序列长度. 由表格可以看出, 在 ECHO 给出的测试序列能够触发崩溃的情况下, 仍无法保证其给出的是最短测试序列. 例如对于序列 44, ECHO 约减后序列长度为 35, 而 TREC 和 SimplyDroid 则分别给出了长度为 14 和 17 的测试序列. 这是因为 ECHO 仅仅在控件层次进行约减, 抽象粒度较低, 当测试序列一直在同一 Activity 中进行细微探索时, 导致测试状态跳转图中最短路径也很长, ECHO 生成的测试序列也会很长. 而 SimplyDroid 本身在 Activity 层面进行抽象、TREC 采用控件与页面布局结合的方式进行抽象, 均可规避这个问题.

对比 SimplyDroid 与 TREC 的约减结果, 在 57 个测试序列上, TREC 约减后的序列长度比 SimplyDroid 得到的结果更短或相等, 占到总序列的 86%, 而在这其中二者给出的序列长度相差大于 3 的达到了 30 个. 此外, 对于标号 57–60 的 4 个序列, SimpyDroid 给出的序列仍包括几十甚至上百个事件, 与 TREC 给出序列长度相差 10 倍以上, 通过进一步分析发现, 此缺陷来源于 Transistor 应用, 其是一款广播电台收听与管理应用, 其崩溃触发操作序列包括了应用切换、输入等复杂操作, 且包括了对创建的电台的图标切换操作, 此操作无法在 Activity 层次上进行体现, 导致 SimplyDroid 无法有效约减, 约减后序列仍旧包括很多冗余, 我们认为在这种情况下 SimplyDroid 给出的测试序列对开发者进行后续的测试理解或复现没有任何帮助.

TREC 仅在 10 个 (15%) 上生成了更长的测试序列, 而在这其中二者给出的序列长度相差大于 3 的仅有 3 个 (序列 34、40、43), 经过对这些序列进行分析, 我们发现这 3 个案例均来自 OpenSudoKu 游戏软件, 包括了简单、中等、复杂这 3 个难度的数独游戏界面, 它们均采用了相同的布局结构, 导致 TREC 在布局粒度上约减时无法区分. 此外, TREC 在这 3 个测试序列的约减结果长度分别为 14、15、19, 约减掉了原序列的 82%、83%、82%, 我们认为此约减结果已经足够精简, 可以辅助开发者进行后续测试理解或复现.

4.4.3 研究问题 3: 约减效率分析

表 2 中右侧第 4、7、10–12 行分别给出了 ECHO、SimplyDroid、TREC 的约减时间, 其中标记时间为 TREC 对每个事件进行标记花费的时间, 约减时间代表了具体约减算法执行时间. 由表格可以看出, 在 ECHO 能够成功约减的情况下, 其与 TREC 花费的时间相差不大, 我们进行了二者运行时间的配对 t 检验, 其显著性 p 值为 0.072 39, 代表二者之间没有显著差别, 这也与二者均是先发现平凡最短路径的设计策略相匹配.

对于 SimplyDroid, 其在 58 个测试序列上的约减时间均高于 TREC, 占到了总序列的 87.88%, 而对于其能成功约减的序列, TREC 则在 36 个案例上能在更短时间内约减, 占到了总成功序列的 81.82%. 此外我们计算了二者在成功案例上的配对 t 检验, 其结果为 0.007 9, 说明 TREC 约减效率明显优于 SimplyDroid 方法.

为了更直观地展示二者的区别, 我们计算出二者在每个测试序列上的总执行时间相差数值, 并绘制了柱状图, 其结果展示在图 3 中, 其横坐标为测试序列标号, 纵坐标为运行时间差, 数值为正表示 TREC 约减效率更高, 此外, 黄色柱形代表 SimplyDroid 在此测试序列上的约减结果无法触发测试崩溃. 从图 3 可以看出在绝大多数测试序列上, TREC 的约减时间均明显小于 SimplyDroid, 其中很多测试序列二者相差时间超过 1 h, 二者最高时间相差 1 308.52 min, 超过 21 h. 此外, 二者时间相差超过 1 h 的序列中, SimplyDroid 在绝大多数案例上均无法成功约减, 这严重限制了其在实际使用的价值, 开发者可能等待 1 h 仍无法得到可成功触发崩溃的约减结果. 例如序列 37 源于 OpenSudoKu 应用, 包括了编辑单个数独游戏名称然后取消、删除单个数组游戏等操作, 这些操作由于均在弹

出框上进行, 在 SimplyDroid 的结构抽象层次树中被视为大枝干中的独立小枝干, 无法进行组合操作的有效区分, 导致最终约减效率低下, 用时超过 5 h 且未约减成功.

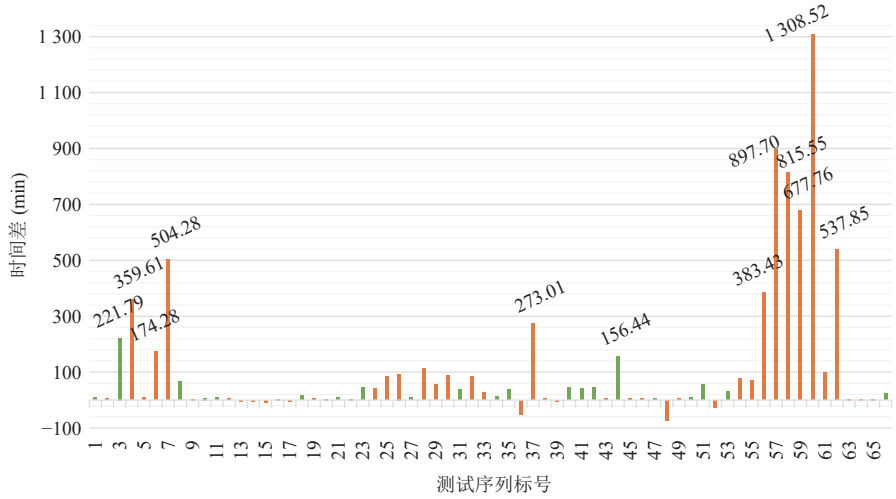

图 3 SimplyDroid、TREC 运行时间差展示图

此外, 表 2 中也给出了 TREC 对每个测试序列的事件标记执行时间, 对于一些在控件粒度及页面布局粒度上均通过最短路径触发程序崩溃的案例 (序列 63–66), 无需进行事件标记. 对于无需进行数据流依赖分析的测试序列 (序列 55–62), 由于其仅需要通过分析测试脚本格式对环境改变、生命周期相关事件进行识别, 耗时很短, 基本可以忽略. 而对于需要进行数据流依赖分析的测试序列, 其时间也基本小于 1 min, 对于整个测试序列约减流程效率影响很小.

4.4.4 研究问题 4: 多粒度约减效果分析

表 3 给出了 TREC 分别在控件粒度约减、页面布局粒度约减上的表现, 其中右侧 3、5 列分别为约减后序列长度, 括号内给出了约减比例, 右侧 4、6 代表约减总时间. 为了方便说明, 我们后续将控件粒度定义为前期约减, 页面布局粒度定义为后续约减. 可以看出, 测试序列在经过前期约减后, 一般可以去除 69.01%–97.33% 的事件, 而后续约减模块则继续成功对 32 个 (占总数的 48.5%) 的序列进一步约减, 约减比例为 7.69%–89.41%. 特别的, 序列 36、45 在前期约减后依然较长, 分别包括了 85 个和 51 个事件, 它们在后续约减中又被去除 76 和 43 个事件, 大大降低了开发者后续程序理解及复现的成本.

表 3 TREC 中控件粒度、页面布局粒度约减效果与效率表

ID	总长度	控件粒度		页面布局粒度		ID	总长度	控件粒度		页面布局粒度	
		长度	时间 (min)	长度	时间 (min)			长度	时间 (min)	长度	时间 (min)
1	98	12 (87.76%)	103.27	12 (0)	196.83	14	68	3 (95.59%)	4.89	3 (0)	0.08
2	98	13 (86.73%)	94.64	12 (7.69%)	363.92	15	87	12 (86.21%)	490.86	3 (75%)	77.99
3	141	12 (91.49%)	276.06	12 (0)	178.87	16	66	11 (83.33%)	77.07	6 (45.45%)	91.73
4	106	12 (88.68%)	96.49	12 (0)	158.77	17	124	8 (93.55%)	2 007.09	5 (37.5%)	54.82
5	64	12 (81.25%)	81.59	12 (0)	131.62	18	61	5 (91.8%)	25.05	5 (0)	25.05
6	104	12 (88.46%)	165.38	12 (0)	130.72	19	52	5 (90.38%)	21.43	5 (0)	12.64
7	146	14 (90.41%)	151.71	14 (0)	166.87	20	50	5 (90%)	19.52	5 (0)	22.29
8	101	12 (88.12%)	191.02	12 (0)	193.20	21	61	5 (91.8%)	17.12	5 (0)	39.38
9	73	3 (95.89%)	99.11	3 (0)	0.41	22	83	5 (93.98%)	15.69	5 (0)	22.29
10	41	9 (78.05%)	220.65	8 (11.11%)	75.93	23	63	8 (87.3%)	43.99	5 (37.5%)	136.65
11	49	3 (93.88%)	18.36	3 (0)	0.08	24	49	9 (81.63%)	229.45	9 (0)	2.63
12	48	3 (93.75%)	13.99	3 (0)	0.08	25	64	9 (85.94%)	155.52	9 (0)	0.76
13	70	3 (95.71%)	9.13	3 (0)	0.07	26	64	11 (82.81%)	163.99	11 (0)	0.98

表 3 TREC 中控件粒度、页面布局粒度约减效果与效率表 (续)

ID	总长度	控件粒度		页面布局粒度		ID	总长度	控件粒度		页面布局粒度	
		长度	时间 (min)	长度	时间 (min)			长度	时间 (min)	长度	时间 (min)
27	79	9 (88.61%)	255.75	9 (0)	0.62	47	103	18 (82.52%)	370.86	9 (50%)	35.71
28	69	9 (86.96%)	212.69	9 (0)	0.62	48	201	44 (78.11%)	6 336.83	10 (77.27%)	203.87
29	56	11 (80.36%)	144.43	11 (0)	0.89	49	198	33 (83.33%)	502.96	9 (72.73%)	153.59
30	66	9 (86.36%)	186.05	9 (0)	0.62	50	214	23 (89.25%)	488.95	9 (60.87%)	108.12
31	75	9 (88%)	727.93	9 (0)	68.37	51	359	32 (91.09%)	516.34	9 (71.88%)	281.08
32	90	9 (90%)	292.70	9 (0)	63.82	52	197	54 (72.59%)	1 386.47	10 (81.48%)	1 177.52
33	66	17 (74.24%)	34.65	14 (17.65%)	510.72	53	376	34 (90.96%)	681.82	10 (70.59%)	264.84
34	76	16 (78.95%)	614.88	14 (12.5%)	321.85	54	797	42 (94.73%)	878.18	9 (78.57%)	191.03
35	106	31 (70.75%)	164.63	17 (45.16%)	560.52	55	97	8 (91.75%)	229.52	7 (12.5%)	103.24
36	312	85 (72.76%)	4 863.41	9 (89.41%)	506.35	56	124	10 (91.94%)	393.51	10 (0)	336.58
37	133	42 (68.42%)	491.60	24 (42.86%)	1 139.84	57	279	8 (97.13%)	665.15	7 (12.5%)	65.84
38	102	22 (78.43%)	121.26	9 (59.09%)	171.87	58	350	22 (93.71%)	393.78	7 (68.18%)	518.75
39	177	41 (76.84%)	559.19	9 (78.05%)	310.96	59	198	8 (95.96%)	50.16	8 (0)	163.77
40	86	15 (82.56%)	558.68	15 (0)	256.25	60	224	8 (96.43%)	383.06	8 (0)	31.26
41	71	22 (69.01%)	103.74	13 (40.91%)	259.27	61	108	8 (92.59%)	201.73	7 (12.5%)	53.76
42	108	29 (73.15%)	362.40	14 (51.72%)	198.53	62	181	9 (95.03%)	427.83	8 (11.11%)	76.58
43	107	29 (72.9%)	1 573.04	19 (34.48%)	921.84	63	9	1 (88.89%)	11.45	1 (0)	0.00
44	223	35 (84.3%)	80.62	14 (60%)	535.14	64	13	2 (84.62%)	13.14	2 (0)	0.07
45	478	51 (89.33%)	3 697.73	8 (84.31%)	115.99	65	8	1 (87.5%)	8.68	1 (0)	0.00
46	79	24 (69.62%)	361.14	9 (62.5%)	61.78	66	75	2 (97.33%)	13.21	2 (0)	0.07

在约减效率方面, 表 3 中同样给出了两粒度约减算法运行时间, 可以看出对于大多数案例, 后续约减模块花费更少时间. 我们对二者执行时间进行了配对检验, 其结果为 0.005 22, 说明这种差异具有统计学显著意义. 造成这种差异的原因有两个, 首先在前期约减上已经去除了大部分对触发崩溃无用的事件, 每轮次执行时由于序列较短, 即使后续约减执行约减轮次较多, 总体花费时间也不会很长, 其次后期约减可利用前期约减的执行结果历史缓存.

此外, 我们对二者在整体约减中耗费的时间比例以柱状图形式进行说明, 如图 4 所示, 其横坐标为测试序列标号, 纵坐标前后期约减运行时间占总时间的百分比. 可以看出, 在大部分测试序列上, 前期约减花费了大部分时间, 这其中甚至一些序列上后期约减运行时间基本可以忽略, 这也再次验证了后期约减花费时间更少的结论.

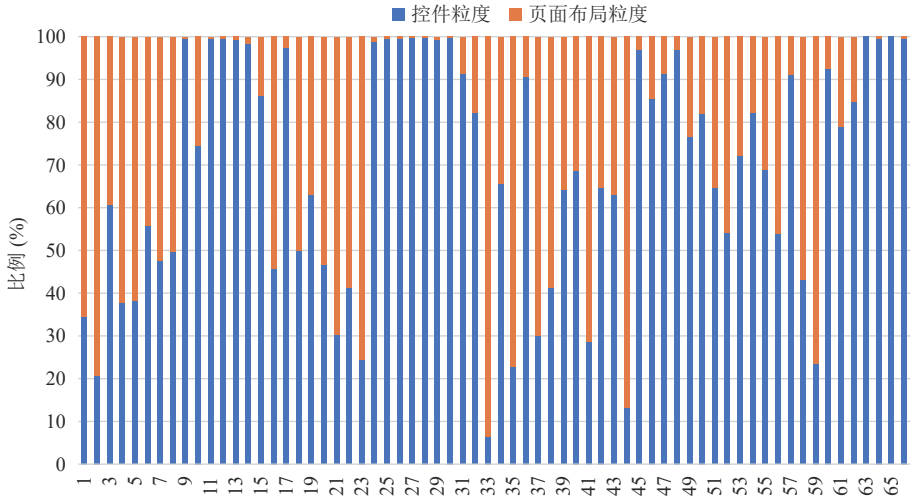

图 4 TREC 控件粒度、页面布局粒度约减时间对比图

以上结果说明了两个粒度的约减分别起到了粗略筛选与精细缩减的作用, 前期约减首先去除大部分对触发程序崩溃不起作用事件, 而后期约减则可进一步精简序列, 但又不会花费很长时间, 这也验证了 TREC 多粒度结合的约减机制的有效性.

4.4.5 研究问题 5: 环路约减效果分析

在此研究问题中, 我们对 TREC 算法在两个粒度上的环路约减效果进行分析, 具体包括了环路间约减及环路内约减. 表 4 给出了具体结果, 在控件粒度和页面布局粒度上分别统计了当前粒度状态跳转图上的环路数量 (第 2、8 列) 及环路深度 (第 3、9 列), 以及分别在环路间约减及环路内约减上算法执行的轮次数目 (第 4、6、10、12 列) 及约减后的序列长度 (第 5、7、11、13 列). 轮次数目代表了算法可以在多少次搜索、验证后完成路径约减, 是 TREC 算法效率高低的直接体现. 特别的, 表 4 的环路数量为最长环路的数量, 此外, 环路深度并非为所有环路的深度, 而是 TREC 方法最终搜索出来的路径中的环路深度.

表 4 TREC 中环路间约减与环路内约减模块效果分析表

ID	控件粒度						页面布局粒度					
	环路数量	环路深度	环路间约减		环路内约减		环路数量	环路深度	环路间约减		环路内约减	
			轮次	长度	轮次	长度			轮次	长度	轮次	长度
1	13	0	4	12 (87.76%)	0	12 (0)	3	0	10	12 (0)	0	12 (0)
2	9	0	4	13 (86.73%)	0	13 (0)	4	1	9	12 (7.69%)	5	12 (0)
3	12	2	3	15 (89.36%)	3	12 (20.00%)	3	0	7	12 (0)	0	12 (0)
4	48	1	3	12 (88.68%)	2	12 (0)	3	0	6	12 (0)	0	12 (0)
5	3	0	4	12 (81.25%)	0	12 (0)	3	0	6	12 (0)	0	12 (0)
6	7	1	5	12 (88.46%)	2	12 (0)	3	0	6	12 (0)	0	12 (0)
7	17	1	3	14 (90.41%)	2	14 (0)	3	0	6	14 (0)	0	14 (0)
8	10	1	3	12 (88.12%)	2	12 (0)	3	0	8	12 (0)	0	12 (0)
9	7	2	3	4 (94.52%)	2	3 (25.00%)	1	0	1	3 (0)	0	3 (0)
10	4	0	8	9 (78.05%)	0	9 (0)	3	0	3	8 (11.11%)	0	8 (0)
11	3	0	2	3 (93.88%)	0	3 (0)	0	0	0	3 (0)	0	3 (0)
12	2	0	2	3 (93.75%)	0	3 (0)	0	0	0	3 (0)	0	3 (0)
13	4	0	2	3 (95.71%)	0	3 (0)	0	0	0	3 (0)	0	3 (0)
14	4	0	2	3 (95.59%)	0	3 (0)	0	0	0	3 (0)	0	3 (0)
15	5	7	3	17 (80.46%)	9	12 (29.41%)	2	2	1	12 (0)	2	12 (0)
16	4	2	2	12 (81.82%)	1	11 (8.33%)	4	2	3	7 (36.36%)	1	6 (14.29%)
17	3	3	46	25 (75.00%)	2	8 (68.00%)	3	2	3	7 (12.50%)	2	5 (28.57%)
18	4	0	1	5 (91.80%)	0	5 (0)	0	0	2	5 (0)	0	5 (0)
19	3	0	1	5 (90.38%)	0	5 (0)	0	0	2	5 (0)	0	5 (0)
20	1	0	1	5 (90.00%)	0	5 (0)	0	0	1	5 (0)	0	5 (0)
21	2	0	1	5 (91.80%)	0	5 (0)	0	0	2	5 (0)	0	5 (0)
22	2	0	1	5 (93.98%)	0	5 (0)	0	0	1	5 (0)	0	5 (0)
23	2	0	2	8 (87.30%)	0	8 (0)	3	2	3	6 (25.00%)	5	5 (16.67%)
24	2	0	4	9 (81.63%)	0	9 (0)	0	0	0	9 (0)	0	9 (0)
25	2	0	3	9 (85.94%)	0	9 (0)	0	0	0	9 (0)	0	9 (0)
26	7	0	3	11 (82.81%)	0	11 (0)	0	0	0	11 (0)	0	11 (0)
27	2	0	4	9 (88.61%)	0	9 (0)	0	0	0	9 (0)	0	9 (0)
28	2	0	4	9 (86.96%)	0	9 (0)	0	0	0	9 (0)	0	9 (0)
29	2	0	4	11 (80.36%)	0	11 (0)	0	0	0	11 (0)	0	11 (0)
30	2	0	4	9 (86.36%)	0	9 (0)	0	0	0	9 (0)	0	9 (0)
31	2	0	20	9 (88.00%)	0	9 (0)	2	0	3	9 (0)	0	9 (0)
32	12	5	5	15 (83.33%)	7	9 (40.00%)	2	0	3	9 (0)	0	9 (0)
33	2	0	3	17 (74.24%)	0	17 (0)	5	2	12	15 (11.76%)	5	14 (6.67%)
34	8	3	14	23 (69.74%)	3	16 (30.43%)	4	1	9	14 (12.50%)	2	14 (0)
35	3	0	5	31 (70.75%)	0	31 (0)	6	1	7	17 (45.16%)	9	17 (0)

表 4 TREC 中环路间约减与环路内约减模块效果分析表 (续)

ID	控件粒度						页面布局粒度					
	环路数量	环路深度	环路间约减		环路内约减		环路数量	环路深度	环路间约减		环路内约减	
			轮次	长度	轮次	长度			轮次	长度	轮次	长度
36	16	2	49	131 (58.01%)	3	85 (35.11%)	12	3	3	51 (40.00%)	6	9 (82.35%)
37	6	0	7	42 (68.42%)	0	42 (0)	8	5	9	35 (16.67%)	15	24 (31.43%)
38	3	0	3	22 (78.43%)	0	22 (0)	6	3	3	12 (45.45%)	6	9 (25.00%)
39	7	3	3	68 (61.58%)	4	41 (39.71%)	9	4	3	29 (29.27%)	7	9 (68.97%)
40	2	0	16	15 (82.56%)	0	15 (0)	3	0	11	15 (0)	0	15 (0)
41	4	1	3	22 (69.01%)	2	22 (0)	3	2	9	14 (36.36%)	1	13 (7.14%)
42	4	3	3	33 (69.44%)	6	29 (12.12%)	7	3	3	21 (27.59%)	5	14 (33.33%)
43	11	4	19	46 (57.01%)	8	29 (36.96%)	7	2	20	20 (31.03%)	3	19 (5.00%)
44	7	2	3	51 (77.13%)	1	44 (13.73%)	11	3	3	27 (38.64%)	17	14 (48.15%)
45	6	1	104	51 (89.33%)	1	51 (0)	11	3	7	15 (70.59%)	6	8 (46.67%)
46	2	0	10	24 (69.62%)	0	24 (0)	3	1	3	9 (62.50%)	2	9 (0)
47	12	4	6	23 (77.67%)	10	18 (21.74%)	4	1	3	9 (50.00%)	2	9 (0)
48	18	0	71	44 (78.11%)	0	44 (0)	8	2	3	32 (27.27%)	4	10 (68.75%)
49	8	3	3	47 (76.26%)	4	33 (29.79%)	8	3	3	17 (48.48%)	5	9 (47.06%)
50	6	3	3	78 (63.55%)	4	23 (70.51%)	6	3	3	12 (47.83%)	5	9 (25.00%)
51	4	2	3	36 (89.97%)	2	32 (11.11%)	7	3	3	18 (43.75%)	5	9 (50.00%)
52	9	5	3	78 (60.41%)	12	54 (30.77%)	10	2	19	11 (79.63%)	7	10 (9.09%)
53	5	3	3	64 (82.98%)	4	34 (46.88%)	4	2	8	11 (67.65%)	3	10 (9.09%)
54	14	3	7	121 (84.82%)	4	42 (65.29%)	6	3	3	12 (71.43%)	5	9 (25.00%)
55	8	1	3	8 (91.75%)	2	8 (0)	3	1	4	7 (12.50%)	1	7 (0)
56	7	2	3	24 (80.65%)	2	10 (58.33%)	4	0	11	10 (0)	0	10 (0)
57	15	3	3	52 (81.36%)	5	8 (84.62%)	3	1	4	7 (12.50%)	1	7 (0)
58	21	1	3	22 (93.71%)	2	22 (0)	5	3	8	9 (59.09%)	9	7 (22.22%)
59	9	0	1	8 (95.96%)	0	8 (0)	3	0	8	8 (0)	0	8 (0)
60	15	2	3	19 (91.52%)	4	8 (57.89%)	3	0	5	8 (0)	0	8 (0)
61	13	1	3	8 (92.59%)	2	8 (0)	3	1	4	7 (12.50%)	1	7 (0)
62	7	3	3	23 (87.29%)	4	9 (60.87%)	3	0	6	8 (11.11%)	0	8 (0)
63	4	0	1	1 (88.89%)	0	1 (0)	0	0	0	1 (0)	0	1 (0)
64	5	0	1	2 (84.62%)	0	2 (0)	0	0	0	2 (0)	0	2 (0)
65	1	0	1	1 (87.50%)	0	1 (0)	0	0	0	1 (0)	0	1 (0)
66	2	0	1	2 (97.33%)	0	2 (0)	0	0	0	2 (0)	0	2 (0)

从表 4 中可以看出, 在控件粒度上, 所有序列上的环路数量最少的为 1 个, 最多的则包括 48 个环路, 其中 53.03% 的序列包括的环路数量不超过 5 个, 而在页面布局粒度上的环路数量范围则是 0–12, 不超过 5 个环路数量的序列比例上升至 77.27%, 且有 20 个 (30.3%) 序列不再包含任何环路. 表 4 结果表明, 虽然页面布局粒度本身更粗糙, 理论上环路数量会更多, 但经过 TREC 算法在控件粒度上的约减后, 大量冗余事件被去除, 为页面布局粒度约减提供了良好基础, 降低了状态跳转图中的环路数. 此外, TREC 算法在控件粒度上的环路间约减可以去除 57.01%–97.33% 的冗余事件, 而在此基础上, 有 23 个约减后序列 (34.85%) 仍旧包括环路并且环路可被进一步约减, 在页面布局粒度上的环路间约减则最高可去除 79.63% 的冗余事件, 且 21 个约减后序列 (31.82%) 仍旧包括可进一步被约减的环路.

此外, 对表 4 进行分析可发现, 环路间、环路内约减的轮次数与环路本身结构相关. 对于扁平结构环路, 即环路本身很长, 不包括内环路或仅包含单层内环路, 环路间约减的轮次数会比较多, 例如第 45 个测试序列, 其在控件粒度上的状态跳转图仅包括了 6 个环路, 但由于其环路很长, 一共进行了 104 次搜索才完成了环路间约减, 而对于纵深结构环路, 即环路内嵌套了两层及以上内环路, 则环路内约减的轮次数会比较多, 例如第 32 个测试序列, 其在控件粒度上的状态跳转图虽然包括了 12 个环路, 但每个环路很短, 仅进行了 5 次搜索就完成了环路间约减, 但由

于其环路很深, 嵌套了 5 层环路, 环路内进行了 7 次搜索才完成约减过程.

上述结果表明 TREC 算法的环路间约减重点作用于扁平结构环路, 去除大部分冗余事件, 而环路内约减则进一步降低环路深度, 缩短最终序列, 二者相互配合, 对测试序列的有效约减均起到必要作用.

5 总 结

测试序列约减工作对于辅助自动化测试回放以及缺陷理解、修复具有重要意义, 但目前安卓测试序列约减工作仅仅关注应用界面状态变化, 并且在单一粒度上对程序执行状态进行抽象, 造成其约减结果序列过长或约减效率低下. 本文提出了基于事件标记的多粒度结合的安卓测试序列约减方法, 对测试序列中触发程序崩溃的关键事件进行标记, 缩小搜索空间, 并实现了低粒度粗筛选、高粒度细约减的约减策略. 本文工作仅支持确定性崩溃缺陷, 并且仅支持原测试序列约减, 不支持自主生成更短测试路径以满足不确定性崩溃缺陷复现要求, 此外, 本文工作虽然涉及不同应用切换、应用内输入等复杂操作与场景, 但支持范围有限, 仍缺乏与系统设置、网络环境改变有关的崩溃错误的支持, 我们将在未来工作中对其进行探索.

References:

[1] Ceci L. Google Play: Number of available APPs as of Q3 2022. 2023. https://www.statista.com/statistics/289418/number-of-available-apps-in-the-google-play-store-quarter/

[2] Ceci L. Number of iOS and Google Play APP downloads as of Q1 2023. 2023. https://www.statista.com/statistics/695094/quarterly-number-of-mobile-app-downloads-store/

[3] Khalid H, Shihab E, Nagappan M, Hassan AE. What do mobile APP users complain about? IEEE Software, 2015, 32(3): 70–77. [doi: 10.1109/MS.2014.50]

[4] Su T, Fan LL, Chen S, Liu Y, Xu LH, Pu GG, Su ZD. Why my APP crashes? Understanding and benchmarking framework-specific exceptions of Android APPs. IEEE Trans. on Software Engineering, 2022, 48(4): 1115–1137. [doi: 10.1109/TSE.2020.3013438]

[5] Wei LL, Liu YP, Cheung SC. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android APPs. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: ACM, 2016. 226–237. [doi: 10.1145/2970276.2970312]

[6] Android Developers. UI/application exerciser monkey. 2023. https://developer.android.com/studio/test/monkey

[7] Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Memon AM. Using GUI ripping for automated testing of Android applications. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. Essen: ACM, 2012. 258–261. [doi: 10.1145/2351676.2351717]

[8] Machiry A, Tahiliani R, Naik M. DynoDroid: An input generation system for Android APPs. In: Proc. of the 9th ACM Joint Meeting on Foundations of Software Engineering. Saint Petersburg: ACM, 2013. 224–234. [doi: 10.1145/2491411.2491450]

[9] Choi W, Necula G, Sen K. Guided GUI testing of Android APPs with minimal restart and approximate learning. ACM SIGPLAN Notices, 2013, 48(10): 623–640. [doi: 10.1145/2544173.2509552]

[10] Mao K, Harman M, Jia Y. Sapienz: Multi-objective automated testing for Android applications. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 94–105. [doi: 10.1145/2931037.2931054]

[11] Su T, Meng GZ, Chen YT, Wu K, Yang WM, Yao Y, Pu GG, Liu Y, Su ZD. Guided, stochastic model-based GUI testing of Android APPs. In: Proc. of the 11th ACM Joint Meeting on Foundations of Software Engineering. Paderborn: ACM, 2017. 245–256. [doi: 10.1145/3106237.3106298]

[12] Choudhary SR, Gorla A, Orso A. Automated test input generation for Android: Are we there yet? In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering. Lincoln: IEEE, 2015. 429–440. [doi: 10.1109/ASE.2015.89]

[13] Li C, Jiang YY, Xu C. GUI event-based record and replay technologies for Android APPs: A survey. Ruan Jian Xue Bao/Journal of Software, 2022, 33(5): 1612–1634 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6551.htm [doi: 10.13328/j.cnki.jos.006551]

[14] Zhong Y, Shi MY, Fang CR, Zhao ZH, Chen ZY. Towards comprehensive evaluation for Android automated testing tools. Ruan Jian Xue Bao/Journal of Software, 2023, 34(4): 1630–1649 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6701.htm [doi: 10.13328/j.cnki.jos.006701]

[15] Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. IEEE Trans. on Software Engineering, 2002, 28(2): 183–200.

[doi: 10.1109/32.988498]

[16] Jiang B, Wu YX, Li T, Chan WK. SimplyDroid: Efficient event sequence simplification for Android application. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering. Urbana-Champaign: IEEE, 2017. 297–307.

[17] Clapp L, Bastani O, Anand S, Aiken A. Minimizing GUI event traces. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 422–434. [doi: 10.1145/2950290.2950342]

[18] Sui YL, Zhang YF, Zheng W, Zhang MQ, Xue JL. Event trace reduction for effective bug replay of Android APPs via differential GUI state analysis. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 1095–1099. [doi: 10.1145/3338906.3341183]

[19] Yan JW, Zhou H, Deng X, Wang P, Yan RJ, Yan J, Zhang J. Efficient testing of GUI applications by event sequence reduction. Science of Computer Programming, 2021, 201: 102522. [doi: 10.1016/j.scico.2020.102522]

[20] Misherghi G, Su ZD. HDD: Hierarchical delta debugging. In: Proc. of the 28th Int'l Conf. on Software Engineering. Shanghai: ACM, 2006. 142–151. [doi: 10.1145/1134285.1134307]

[21] Herfert S, Patra J, Pradel M. Automatically reducing tree-structured test inputs. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering. Urbana: IEEE, 2017. 861–871. [doi: 10.1109/ASE.2017.8115697]

[22] Regehr J, Chen Y, Cuoq P, Eide E, Ellison C, Yang XJ. Test-case reduction for C compiler bugs. In: Proc. of the 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation. Beijing: ACM, 2012. 335–346. [doi: 10.1145/2254064.2254104]

[23] Kanstrén T, Chechik M. Trace reduction and pattern analysis to assist debugging in model-based testing. In: Proc. of the 2014 IEEE Int'l Symp. on Software Reliability Engineering Workshops. Naples: IEEE, 2014. 238–243. [doi: 10.1109/ISSREW.2014.9]

[24] Wang J, Dou WS, Gao CS, Gao Y, Wei J. Context-based event trace reduction in client-side JavaScript applications. In: Proc. of the 11th IEEE Int'l Conf. on Software Testing, Verification and Validation. Västerås: IEEE, 2018. 127–138. [doi: 10.1109/ICST.2018.00022]

[25] Gharachorlu G, Sumner N. Avoiding the familiar to speed up test case reduction. In: Proc. of the 18th IEEE Int'l Conf. on Software Quality, Reliability and Security. Lisbon: IEEE, 2018. 426–437. [doi: 10.1109/QRS.2018.00056]

[26] Choi W, Sen K, Necul G, Wang WY. DetReduce: Minimizing Android GUI test suites for regression testing. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: IEEE, 2018. 445–455. [doi: 10.1145/3180155.3180173]

[27] Jiang B, Wang XY, Xu HQ, Wang H, Zhang CY. Nondeterministic event sequence reduction for Android applications. In: Proc. of the 5th Int'l Conf. on Dependable Systems and Their Applications. Dalian: IEEE, 2018. 96–101. [doi: 10.1109/DSA.2018.00026]

[28] Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In: Proc. of the 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. San Francisco: ACM, 1995. 49–61. [doi: 10.1145/199448.199462]

[29] Tan SH, Dong Z, Gao X, Roychoudhury A. Repairing crashes in Android APPs. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering. Gothenburg: IEEE, 2018. 187–198. [doi: 10.1145/3180155.3180243]

[30] Pan MX, Xu TT, Pei Y, Li Z, Zhang T, Li XD. GUI-guided test script repair for mobile APPs. IEEE Trans. on Software Engineering, 2022, 48(3): 910–929. [doi: 10.1109/TSE.2020.3007664]

[31] Su T, Wang J, Su ZD. Benchmarking automated GUI testing for Android against real-world bugs. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 119–130. [doi: 10.1145/3468264.3468620]

[32] Fazzini M, Xin Q, Orso A. Automated API-usage update for Android APPs. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 204–215. [doi: 10.1145/3293882.3330571]

[33] Lin JW, Salehnamadi N, Malek S. Test automation in open-source Android APPs: A large-scale empirical study. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: IEEE, 2020. 1078–1089.

[34] Xia H, Zhang Y, Zhou YT, Chen XT, Wang Y, Zhang XY, Cui SS, Hong G, Zhang XH, Yang M, Yang ZM. How Android developers handle evolution-induced API compatibility issues: A large-scale study. In: Proc. of the 42nd IEEE/ACM Int'l Conf. on Software Engineering. Seoul: IEEE, 2020. 886–898.

[35] Wang HY, Xia X, Lo D, Grundy J, Wang XY. Automatic solution summarization for crash bugs. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 1286–1297. [doi: 10.1109/ICSE43902.2021.00117]

[36] Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot: A Java bytecode optimization framework. In: Proc. of the 2010 CASCON 1st Decade High Impact Papers. Toronto: IBM, 2010. 214–224. [doi: 10.1145/1925805.1925818]

[37] Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android APPs. ACM SIGPLAN Notices, 2014, 49(6): 259–269. [doi: 10.1145/2666356.2594299]

[38] Lin L, Liao XF, Jin H, Li P. Computation offloading toward edge computing. Proc. of the IEEE, 2019, 107(8): 1584–1607. [doi: 10.1109/

JPROC.2019.2922285]

[39] Pan Y, Ge XT, Fang CR, Fan Y. A systematic literature review of Android malware detection using static analysis. IEEE Access, 2020, 8: 116363–116379. [doi: 10.1109/ACCESS.2020.3002842]

[40] Yang W, Xiao XS, Andow B, Li SH, Xie T, Enck W. AppContext: Differentiating malicious and benign mobile APP behaviors using context. In: Proc. of the 37th IEEE Int'l Conf. on Software Engineering. Florence: IEEE, 2015. 303–313. [doi: 10.1109/ICSE.2015.50]

[41] Chen S, Xue MH, Fan LL, Hao S, Xu LH, Zhu HJ, Li B. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. Computers & Security, 2018, 73: 326–344. [doi: 10.1016/j.cose.2017.11.007]

[42] Miao XC, Wang R, Xu L, Zhang WF, Xu BW. Security analysis for Android applications using sensitive path identification. Ruan Jian Xue Bao/Journal of Software, 2017, 28(9): 2248–2263 (in Chinese with English abstract). [doi: 10.13328/j.cnki.jos.005177]

附中文参考文献:

[13] 李聪, 蒋炎岩, 许畅. 基于 GUI 事件的安卓应用录制重放关键技术综述. 软件学报, 2022, 33(5): 1612–1634. http://www.jos.org.cn/1000-9825/6551.htm [doi: 10.13328/j.cnki.jos.006551]

[14] 钟怡, 石孟雨, 房春荣, 赵志宏, 陈振宇. 面向安卓自动化测试工具综合评估. 软件学报, 2023, 34(4): 1630–1649. http://www.jos.org.cn/1000-9825/6701.htm [doi: 10.13328/j.cnki.jos.006701]

[42] 缪小川, 汪睿, 许蕾, 张卫丰, 徐宝文. 使用敏感路径识别方法分析安卓应用安全性. 软件学报, 2017, 28(9): 2248–2263. [doi: 10.13328/j.cnki.jos.005177]

郝蕊(1991－), 女, 博士, 主要研究领域为移动应用测试, 缺陷理解.

李玉莹(1994－), 女, 博士, 主要研究领域为众包测试, 缺陷理解.

冯洋(1988－), 男, 博士, CCF 高级会员, 主要研究领域为复杂系统质量保障.

陈振宇(1978－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为智能软件工程.