中国科学: 信息科学 2019 年 第 49 卷 第 10 期: 1234–1266
SCIENTIA SINICA Informationis
《中国科学》杂志社
SCIENCE CHINA PRESS
软件自动化前沿进展专题 • 评述

Android 应用测试输入自动生成技术

王珏[1,2], 蒋炎岩[1,2], 许畅[1,2*], 马晓星[1,2], 吕建[1,2]

1. 计算机软件新技术国家重点实验室 (南京大学), 南京 210023
2. 南京大学计算机科学与技术系, 南京 210023
* 通信作者. E-mail: changxu@nju.edu.cn

收稿日期: 2019–01–14; 接受日期: 2019–03–27; 网络出版日期: 2019–10–16

国家重点研发计划 (批准号: 2017YFB1001801) 和国家自然科学基金 (批准号: 61690204, 61802165) 资助项目

摘要 软件自动化是提高软件生产率的根本途径. 由于 Android 应用快速迭代的开发模式, 其对于自动化协助应用开发, 尤其是自动化测试, 有很高的需求. 在自动化测试中, 测试输入自动生成是最为关键和资源消耗最大的步骤之一, 极大地影响自动化测试的有效性. 由于 Android 应用独有的特性, 自动为其生成测试输入存在独特的挑战. 为了应对这一挑战, 已有许多 Android 应用测试输入自动生成技术被提出. 本文提出 Android 应用测试输入自动生成技术的描述框架, 包括测试输入自动生成技术的 3 个维度 (分别为搜索空间的表示、候选输入的生成、候选输入的评估), 并讨论了测试输入自动生成技术在这 3 个维度采用策略的两个评价指标 (即充分性与高效性). 借助这一全新描述框架, 本文对已有技术进行分析和总结, 讨论现有技术的长处和不足, 并对未来可能的研究方向给予展望.

关键词 Android, 自动测试, 输入生成, 技术描述, 智能手机

1 引言

软件自动化的目的是在可能的范围内, 将软件开发过程中烦琐、不易、低效的部分委交机器完成, 以此减轻软件开发人员的负担, 同时依靠机器的高效性与可靠性加快开发进程, 保证软件质量 [1]. 其中, 由于软件测试在软件的开发过程中占有重要的位置, 而人工测试耗费大量人力且质量难以保证, 因而对软件测试的自动化引起了相关研究人员的广泛关注, 成为了目前软件自动化最为重要的发展方向之一 [2].

近年来, 业界对于软件自动化, 尤其是软件测试自动化的需求日益增加. 其中, 智能手机应用厂商, 尤其是 Android 手机应用厂商的这一需求格外强烈. 这主要由于激烈的市场竞争以及应用快速频繁迭代的需求. 以 Google Play 为例, 截至 2018 年 11 月, 其包含了超过 2500000 个 Android 应用, 并以

引用格式: 王珏, 蒋炎岩, 许畅, 等. Android 应用测试输入自动生成技术. 中国科学: 信息科学, 2019, 49: 1234–1266, doi: 10.1360/N112019-00003
Wang J, Jiang Y Y, Xu C, et al. Automatic test-input generation for Android applications (in Chinese). Sci Sin Inform, 2019, 49: 1234–1266, doi: 10.1360/N112019-00003

每个月 60000 余新增应用的速度增长[1)]. 由于智能手机平台的快速发展以及激烈的市场竞争, 智能手机应用需要进行快速频繁的迭代. 因而, 测试所需的资源如时间、人力等极度受限, 这使得通过人工进行有效的测试变得尤为困难. 自然地, 业界对于有效的智能手机应用测试自动化的需求日益增长. 而在软件测试中. 测试输入生成是最为关键, 资源消耗也最大的步骤之一. 测试输入的质量很大程度上影响着测试过程的有效性 [3]. 因而自动生成高质量的智能手机应用测试输入对实现有效的智能手机应用自动测试技术至关重要.

许多科研人员都已提出了不同的智能手机应用测试输入自动生成技术. 其中, 由于 Android 平台的开放性, 以及其是目前最为主要的智能手机平台, 大多数提出的技术都针对于 Android 应用. 这些技术采用各种各样的策略自动生成测试输入, 以服务具有不同目标的自动测试过程. 然而, 在最近 Choudhary 等 [4] 的实证性研究的比较实验中, 其收集的技术生成的测试输入平均仅能覆盖 50% 左右的应用代码. 这意味着现有的测试输入自动生成技术还不足以支持有效的 Android 应用自动测试. 因而, 目前亟需对现有技术进行分析, 探究其不足之处, 以在未来的发展中加以弥补.

国内外已有一些 Android 应用自动测试技术的讨论与总结 [3,5~7], 然而现有的研究通常广泛地考虑整个自动测试过程, 未能系统深入地分析测试输入自动生成的相关技术. Chouldhary 等 [4] 在 2015 年对截至当时的 Android 应用测试输入自动生成技术进行了简单总结与比较性实验, 但也未能对已有技术进行系统的探究与讨论. 特别地, 现有的研究未能在一个统一的框架下比较、分析、总结现有技术, 亦未在此基础上探究现有技术效果不够理想的原因, 以及由此带来的研究契机.

本文关注于 Android 应用测试输入自动生成这一重要问题, 提出一个可以统一描述现有技术的描述框架, 基于这一描述框架分析总结现有技术, 力图从更高层次分析讨论现有技术自动生成测试输入采取的策略, 做出的不同权衡以及原因, 在此基础上更好地探究现有技术的不足以及原因, 并进而讨论可能的未来发展方向. 本文主要内容包括:

(1) 简单介绍 Android 应用背景, 形式化定义其执行模型, 并在此模型基础上明确研究问题与挑战 (见第 2 节).

(2) 提出 Android 应用测试输入自动生成技术的描述框架, 包含 3 个维度以及两个评价指标 (见第 3 节).

(a) 3 个维度 (见 3.1 小节): 搜索空间表示、候选输入生成, 以及候选输入的评估. Android 应用测试输入自动生成技术通常会建立一个搜索空间, 借助这一搜索空间在一个循环过程中迭代生成候选输入, 并对其进行评估以选取高质量候选输入生成并发送给被测应用. 现有技术在这 3 个正交的维度上采取不同的策略;

(b) 两个评价指标 (见 3.2 小节): 充分性和高效性. 对于描述框架中的 3 个维度以及整体的测试输入自动生成过程, 均可以使用这两个评价指标进行评价. 充分性要求测试输入自动生成技术在这 3 个维度采取的策略能够帮助生成充分探索被测应用行为的测试输入, 以达到更好的测试效果. 而高效性要求这些策略能够更快速地生成测试输入, 以在有限的测试资源下完成测试输入生成过程.

(3) 基于第 3 节提出的描述框架, 系统化地分析总结现有技术 (见第 4 节). 首先对于描述框架的每个维度, 介绍常见的策略 (见 4.1~4.3 小节). 接着借助这些常见策略, 以搜索空间的表示对现有技术进行分类, 再系统总结现有技术采用的策略以及原因 (见 4.4 小节).

(4) 基于提出的描述框架以及第 4 节的总结, 探讨现有技术的不足以及相关的研究契机. 目前, 现有技术在描述框架的 3 个维度上的策略都存在不足之处, 严重影响自动生成的测试输入的质量 (见 5.1 小节). 在此基础上, 我们提出 5 个未来可能的研究方向 (见 5.2 小节): 搜索空间表示的融合、应用

1) Number of Android apps on Google play. https://www.appbrain.com/stats/number-of-android-apps.

运行环境输入的生成与评估、应用及运行环境自发输入 (即 ε – 输入) 的建模与控制、大数据支持的候选输入评估, 以及人类指导的测试输入自动生成.

2 问题定义

在这一节中, 对 Android 应用测试输入自动生成这一问题进行定义. 在 2.1 小节中, 简单介绍 Android 应用的背景, 并形式化定义其执行模型. 在 2.2 小节中, 定义 Android 应用测试输入自动生成这一问题.

2.1 Android 应用的执行模型

2.1.1 背景

Android 是目前最为流行的智能手机平台. 它为所有运行在 Android 系统上的应用都规定了统一的模型. 在这一应用模型下, 一个 Android 应用通常使用 Java 或 Kotlin 进行编写, 运行在 Dalvik 或 ART 虚拟机[2)]上, 并主要包含 4 类组件: (1) Activity 包含了用户图形界面 (GUI), 主要负责与用户进行图形化的交互; (2) Broadcast receiver 主要负责接受 Android 系统发来的信息并作出响应; (3) Service 主要负责在后台进行长时间的工作; (4) Content provider 主要负责管理应用的共享数据. 这 4 类主要的组件还可以进一步包含子组件来实现更复杂的应用逻辑, 如 Fragment 与 AsyncTask 等.

Android 应用的运行无法离开其运行环境 (简称环境). 运行环境是由 Android 平台提供的, Android 应用正常运行所必须的一组资源, 如时钟、文件系统、传感器等. 一个 Android 应用及其所需的运行环境共同组成了一个完整的应用运行系统 (简称应用系统).

在运行中, Android 应用接收外部使用者发送的各类输入, 如点击输入等, 并在接收到这些输入时发生状态的改变. 应用在每个时刻都有一组可以处理的输入, 并通过一组向系统注册的输入处理器来处理这些输入. 同样, 运行环境也可以接收外部使用者的输入, 如文件操作等, 从而发生状态的改变. 同时, 应用和运行环境自身也会自发地改变自己的状态, 如环境的时钟会自发地增加 1 s, 而应用会在处理接收到的输入时执行代码自发地改变状态. 我们定义这样自发的, 导致状态改变的行为为 ε – 输入. ε – 输入是自发的, 且无法被完全控制 [8]. 另外, 应用和环境的状态不是独立的, 一方在改变状态时可能也会改变另一方的状态. 因而在进行测试时, 必须同时考虑两者的状态.

2.1.2 执行模型

如 2.1.1 小节中所述, Android 应用的执行无法离开其运行环境. 因而我们的执行模型涉及这两个实体, 即整个应用运行系统. 我们使用一个四元组来形式化定义这一执行模型: $M = (S, S_0, I, \delta)$, 其中:

(1) 状态集合 $S = S_a \times S_e$ 是应用运行系统的状态空间. 应用运行系统的状态是应用的状态与运行环境的状态的组合. 对于应用, 定义其状态为运行该应用的虚拟机及进程所有与应用相关变量的瞬时取值, 用 S_a 表示其状态空间. 对于环境, 定义其状态为其包含的所有资源的瞬时取值, 用 S_e 表示其状态空间. 应用系统的状态空间 S 为 S_a 与 S_e 的笛卡尔积;

(2) 初始状态集合 $S_0 \in S$ 是应用运行系统的初始状态. 对于应用, 定义其初始状态集合为 S_{a0}. 对于环境, 定义其初始状态集合为 S_{e0}. 因而有 $S_0 = S_{a0} \times S_{e0}$;

2) Dalvik and ART. https://source.android.com/devices/tech/dalvik.

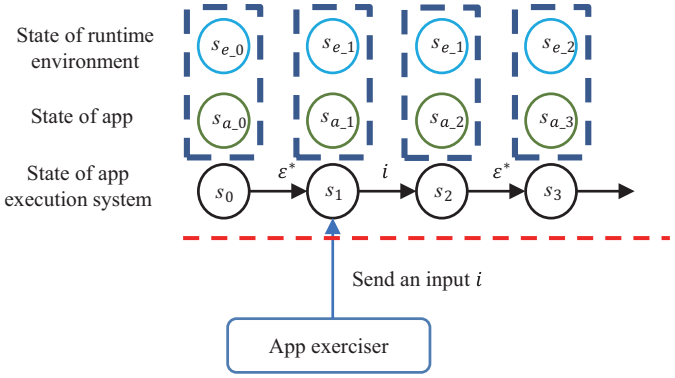


图 1 (网络版彩图) 执行模型状态转移

Figure 1 (Color online) State transitions of the execution model

(3) 输入集合 $I = I_a \cup I_e \cup I_\varepsilon$ 是所有可能的输入, 对每个 $i \in I$, 使用一个二元组 $i = (t, d)$ 表示. I 包含 3 类, 不同类别的输入的 t 与 d 有着不同的含义:

(a) I_a 是外部使用者向应用发送的输入, 包括按下、抬起等 GUI 交互输入, 以及语音等应用数据输入. 对于 GUI 交互输入, t 代表交互的类型, d 表示输入的屏幕坐标. 对于应用数据输入, t 代表输入的数据类型, d 为输入的具体数据, 如语音输入的 d 为具体输入的音频数据;

(b) I_e 是外部使用者向运行环境发送的输入. 对每个 $i_e \in I_e$, t 表示操作针对的环境资源以及要进行的操作, 如对文件系统进行增加/删除文件操作, 对时钟进行时间设置操作等, d 代表具体的输入参数或数据, 如增加的文件的具体数据, 以及要设置的目标时间等;

(c) I_σ 是所有可能的 ε – 输入. 对于应用来说, 定义每一条字节码指令的执行为一次 ε – 输入. 对于这一类 ε – 输入, t 表示要执行的字节码指令, d 表示导致的应用状态变化, 即相关变量的取值变化. 而对于环境来说, 定义一个资源一次自发的值变化为一次 ε – 输入. 对于这一类 ε – 输入, 其与向运行环境发送的输入具有相同的表示, 但其为自发发生, 如时钟自发地进行时间增加 1 s 的操作.

(4) 状态转移函数 $\delta : S \times I \to S$ 描述了在接收到外部使用者发送的或自发的输入后系统的状态转移.

注意, 在该模型中, 定义输入的发送和状态的转移都是瞬时的. 对于长时间的输入 (如长按输入), 可以使用多个瞬时输入表示, 如一个按下输入、多个时钟 ε – 输入, 以及一个抬起输入表示一个长按输入.

在该模型中, 使用状态转移图对应用系统进行建模. 由于我们定义一个应用系统的状态是应用瞬时的所有变量取值以及运行环境所有资源的取值, 其能够准确地描述真正影响应用行为的状态. 一些现有的技术通过其他方式对应用进行建模, 如代码执行路径空间等. 这些建模方法一般是在瞬时状态的基础上添加额外信息或提炼瞬时状态中的信息建立, 用以指导测试输入的生成. 在第 4 节中, 将对这些建模进行详细的讨论.

在这一模型中, 状态的转移既可以被外部使用者的输入触发, 也可以被 ε – 输入触发. 如图 1 所示, 从一个初始状态 $s_0 \in S_0$ 开始, 应用系统不断自发地发生 ε – 输入, 并不断发生状态的转移. 当一个外部使用者输入发送到应用时, 应用的状态 $s_{a\_1} \in S_a$ 会转移到一个新的状态 s_{a_2}, 此时处理该输入的处理器还未被执行 (但虚拟机的栈帧发生了变化导致状态变化), 且运行环境的状态也未变化. 之后一系列字节码执行即应用运行系统上的 ε – 输入发生, 处理器的代码被执行, 并使应用运行系统的状态不断转移.

2.2 Android 应用测试输入自动生成: 问题定义

Android 应用测试输入生成关注的是生成外部使用者的输入, 即 I_a 与 I_e 中的输入. 因而定义一个测试输入 $i \in I_a \cup I_e$. 特别地, 我们进一步将测试输入分为事件输入 (应用上的 GUI 交互输入) 与数据输入 (应用数据输入与环境输入). 事件输入会导致应用执行相应的输入处理器, 而数据输入除了触发输入处理器执行外, 有时也作为输入处理器执行中所需的数据.

定义测试输入生成过程为:

(1) 初始化一个运行环境 e, 将被测应用 P 在其上运行以到达一个初始状态 $s_0 \in S_0$;

(2) 循环地观测应用系统的状态, 并利用观测到的状态以及探索历史 hist 生成测试输入 (序列), 将其发送到应用系统上以探索应用系统的状态空间 S, 直至满足某种终止条件.

在这一过程中, 状态空间探索历史 hist 会包含已经生成的测试输入 (序列) 与相应观测到该测试输入 (序列) 探索到的状态序列, 即 $\text{hist} \subseteq (I_a \cup I_e)^+ \times S^+$. 注意这里保存的状态序列是测试输入生成技术观测到的状态. 由于 ε – 输入的存在, 应用系统会不停地自发改变状态, 因而测试输入生成技术在实际中很难观测到所有的状态. 同时, 在第 2 步的循环生成过程中, 测试输入自动生成技术可以生成重启输入, 通过不同的重启输入初始化不同的运行环境或使应用以不同的初始状态启动, 以探索不同的应用运行系统初始状态. 另外, 这一过程是一个动态的过程, 因而测试输入自动生成技术不仅要考虑测试输入的生成, 同样要考虑测试输入的发送.

为了保证测试质量, 应用系统的状态空间 S 必须被有效地探索. 探索 S 的有效性定义为对如此探索 S 并进行测试的结果的信心. 因而测试输入自动生成的目标定义为在有限的测试资源内 (如时间等) 生成一组测试输入 (序列) $I_{\text{gen}} \subset (I_a \cup I_e)^+$ 并将其发送给应用系统, 以有效地遍历 S. 实现这一目标主要存在两个挑战.

- 状态空间 S 是无穷的. 这使得测试输入生成技术无法在有限的测试资源下遍历整个状态空间, 只能探索一部分状态. 由于不同的测试过程目的不同, 其对于不同的状态有着不同的重要性评估. 如对于检测应用并发错误的测试过程, 其自然认为探索涉及并发执行的状态更为重要. 如何生成测试输入以优先探索到所服务的测试过程认为更重要的状态, 以更好地服务测试过程, 是测试输入生成技术的一大挑战.

- 应用运行系统的执行模型 M 是未知的. 这主要体现在状态空间 S, S_0, 状态转移函数 δ 是未知的. 这一未知不仅包含已知的未知 (known unknowns), 即可以预知的未知信息, 如应用在每个状态的图形界面, 也包含大量未知的未知 (unknown unknowns), 即无法预知的未知信息, 如哪种 ε – 输入会在何时发生, 并导致何种状态转移. 由于这些未知的未知, 以及状态空间的无穷性, 很难在有限的测试资源内精准地刻画 S, S_0 与 δ. 由于未知性, 测试输入生成技术无法准确地获知生成什么测试输入可以进一步探索状态空间未被探索的部分. 有一些测试输入不会导致应用系统状态的转移, 如应用当前不可处理的输入. 而有一些测试输入只会探索已经探索到的状态. 由于无法准确识别这些输入, 状态空间探索容易困于局部, 影响生成的测试输入的有效性. 同时, 状态空间中存在一些深层状态, 需要特定的测试输入序列才能够探索到. 一些状态需要特定的数据输入 (如登录账号密码等) 才能够探索到, 而另一些状态需要一个特定的长测试输入序列 (如多次交替发送两个点击输入) 才能够探索到. 由于状态空间的未知性, 测试输入生成技术无法准确得知所需的测试输入, 从而很难探索到这些状态.

3 描述框架

本节给出描述 Android 应用测试输入自动生成技术的框架. 我们从 3 个维度描述一个测试输入自动生成技术: 搜索空间的表示、候选输入的生成, 以及对这些候选输入的评估. 另外, 使用两个评价指标来评价一个测试输入自动生成技术的这 3 个维度上的策略: 充分性和高效性. 3.1 小节对于测试输入自动生成技术的 3 个维度予以概述, 3.2 小节对两个评价指标予以概述.

注意, 本文提出的描述框架是现有技术框架之上的抽象, 能够普适地描述所有现有 Android 应用测试输入自动生成技术. 如现有的基于搜索的测试输入生成框架通常会建立测试输入序列的搜索空间, 并从这一空间中直接选取候选输入, 采用不同的方式进行评估. 而其他类的技术则会建立不同的搜索空间, 并采用不同的候选输入生成与评估策略. 由于我们的描述框架是在这些技术及技术框架之上的抽象, 因而能够描述这些技术与技术框架. 同时, 我们的描述框架包含了评估每个具体技术策略的指标: 充分性与高效性.

3.1 测试输入自动生成技术的 3 个维度

在 2.2 小节描述的测试输入循环生成过程中, 一个测试输入自动生成技术通常建立一个搜索空间, 借助这一搜索空间生成一组候选输入. 接着, 利用搜索空间与探索历史对这些候选输入进行评估, 选取一个候选输入将其发送到应用系统, 并通过观测应用系统的状态变化进一步优化搜索空间. 其中包含 3 个维度: 搜索空间的表示、候选输入的生成, 以及候选输入的评估. 每个测试输入自动生成技术在这 3 个维度上都有其独特的策略. 本小节剩余部分对这 3 个维度予以概述.

搜索空间的表示. 如 2.2 小节中所述, 应用系统的状态空间 S 是无穷且执行模型 M 是未知的. 无穷性与未知性使得 S, S_0, δ 无法被准确刻画, 从而无法为状态空间探索提供有效的信息. 为了应对这一问题, 测试输入自动生成技术通过建立一个搜索空间, 将对状态空间的探索转化为对搜索空间的探索. 搜索空间通常通过对 S 进行向细粒度或粗粒度的转化获得. 我们使用搜索空间中每个状态所包含的应用系统的信息来度量其粒度, 例如 S 中的每个状态包含了整个应用系统的全部瞬时状态, 如应用的所有变量取值、文件系统中当前存在的文件等. 对 S 进行向细粒度的转化形成的搜索空间会考虑更多的应用系统的相关信息, 如对一个状态 $s \in S$, 不仅考虑其瞬时的状态, 同时也考虑应用系统从初始状态到达这一状态所发生的状态转移路径. 由于可能存在不同的状态转移路径到达 s, 考虑更多信息会将 s 进一步分离为多个状态, 每个状态包含更多的信息. 通过这样向细粒度的转化使得搜索空间的每个状态包含更多的已知信息与更少的未知的未知, 使其更易被刻画和探索, 相应也可以更准确地获知如何进一步探索状态空间, 从而可以应对状态空间未知性的挑战. 对 S 进行向粗粒度的转化形成的搜索空间会考虑更少的应用系统的相关信息, 例如对一个状态 $s \in S$, 不考虑运行环境的瞬时状态 $s_e \in S_e$, 如文件系统的现有文件等, 只考虑应用的瞬时状态 $s_a \in S_a$. 由于可能存在多个应用系统状态具有相同的应用状态, 因而可以将这些状态进行合并. 通过这样向粗粒度的转化, 可以获得有效缩减的搜索空间, 同时忽略 M 中的一些未知信息, 从而可以应对 S 无穷性以及 M 未知性的挑战.

在测试输入生成过程中, 搜索空间的刻画也被动态地优化. 搜索空间通过对 S 进行转化取得. 由于 M 具有未知性, 搜索空间往往也包含未知的信息. 通过生成测试输入探索搜索空间并观测应用系统的状态, 测试输入自动生成技术可以获取搜索空间的更多信息, 从而可以动态地优化搜索空间的刻画, 以更好地指导测试输入的生成.

不同的测试输入生成技术会生成不同的搜索空间. 这与其生成测试输入所需的信息以及所服务的测试目的相关. 搜索空间的表示往往可以体现测试输入生成技术的内见. 在 4.1 小节中详细介绍现有

技术的搜索空间表示.

候选输入的生成. 通过对搜索空间及探索历史的分析, 测试输入生成技术会生成一组候选的测试输入. 这些候选输入可能组织成多个单个测试输入, 也可能组织成多个测试输入序列. 通过分析搜索空间以及探索历史, 测试输入生成技术判定这些候选输入都可以进一步探索搜索空间.

候选输入的评估. 在生成一组候选输入后, 测试输入自动生成技术会对这些候选输入进行评估, 选择预期收益最高的一个实际生成并发送给应用运行系统. 这些候选输入都可以进一步探索搜索空间. 然而, 由于搜索空间是通过转化状态空间得到, 其与状态空间存在一定差异. 因而, 能够进一步探索搜索空间的候选输入未必能够进一步探索状态空间. 同时, 不同的候选输入对于状态空间的探索效果也是不同的. 不同的候选输入会探索不同部分的状态空间与搜索空间, 从而带来不同的收益, 如所能够提供的搜索空间未知的信息, 所进一步探索到的状态的数量以及对测试过程的重要性等. 通过借助搜索空间和探索历史, 测试输入自动生成技术会综合预测候选输入的收益, 并根据预测的结果选择一个候选输入发送给应用系统.

候选输入的生成与评估是紧密相关的. 不同的测试输入生成技术会采取不同的候选输入生成与评估策略. 在 4.2 与 4.3 小节中详细介绍现有技术的候选输入生成与评估策略.

3.2 评价指标

对于测试输入自动生成技术在 3.1 小节中所描述的 3 个维度所采用的策略, 统一使用两个评价指标来评价: 充分性与高效性.

充分性反映了一个策略是否能够保证有效探索状态空间 S. 对于搜索空间的表示, 充分性反映对搜索空间有效的探索和对状态空间有效的探索的相关性. 搜索空间探索的有效性定义为对其的遍历的充分程度. 对于候选输入的生成, 充分性反映生成的候选输入集合是否充分包含了能够进一步探索搜索空间, 进而探索状态空间的输入. 对于候选输入的评估, 充分性反映对候选输入的收益预测是否能够体现候选输入对充分探索状态空间的贡献.

高效性反映了一个策略是否能够保证探索状态空间 S 有较快的速度. 对于搜索空间的表示, 高效性反映是否能够使用较少的测试输入来遍历整个搜索空间. 所需测试输入越少, 则所需的循环次数越少, 因而可以更高效地完成测试输入生成过程. 对于候选输入的生成和评估, 高效性反映生成及评估候选输入集合的速度. 生成及评估候选输入集合的速度越快, 则单次循环所需时间越少, 则测试输入生成过程更高效.

对于每一个维度的策略, 充分性和高效性都存在权衡关系. 一般而言, 为了保证充分性, 就需要更大的搜索空间、更多更复杂的候选输入, 以及更精细更耗时的收益评估过程. 而为了保证高效性, 则需要牺牲一些充分性以达到快速循环, 更快结束的目的. 具体合适的权衡与测试输入生成技术所服务的测试目标相关.

4 现有技术分析

本节利用第 3 节中提出的描述框架总结分析现有的测试输入自动生成技术. 对于框架中的 3 个维度, 目前都存在一些常用的策略. 图 2 展示了这些常用的策略以及其充分性/高效性权衡的倾向. 首先在 4.1~4.3 小节分别介绍常用的搜索空间表示与常见的候选输入生成和评估策略. 在 4.4 小节利用常用策略分析总结现有技术.

Search space representation	State transition trace space	Original app execution state space	Code execution trace space	App execution state equivalence class space
Candidate input generation strategy	All supported input generation	Global-analysis based generation	On-the-fly-observation-based generation	Existing input modification
Candidate input evaluation strategy	Rule-based evaluation	Black-box evaluation	Heuristically random evaluation	Uniformly random evaluation

图 2 (网络版彩图) 现有技术 3 个维度的常用策略

Figure 2 (Color online) Common strategies of dimensions of existing techniques

4.1 搜索空间表示

现有技术通常使用 4 种搜索空间的表示形式: (1) 状态转移路径空间, (2) 原始状态空间, (3) 应用代码执行路径空间, 以及 (4) 缩减的状态空间. 这 4 类搜索空间按顺序从倾向充分性逐渐更倾向高效性. 我们分别介绍这些搜索空间.

状态转移路径空间. 一个状态转移路径空间 ST 包含所有可能的从任一个初始状态开始的状态转移路径, 即 $\mathrm{ST} \subseteq S^+$, 且 $\forall \mathrm{st} = (s_0, s_1, \ldots, s_n) \in \mathrm{ST}.\ s_0 \in S_0 \wedge \exists is = (i_0, i_1, \ldots, i_{n-1}) \in I^*.\ \delta(s_0, i_0) = s_1 \wedge \delta(s_1, i_1) = s_2 \wedge \cdots \wedge \delta(s_{n-1}, i_{n-1}) = s_n$.

这一搜索空间通过将状态空间 S 向细粒度转化生成. 状态空间中, 状态转移路径之间可能共享状态转移路径片段, 通过将这些状态进行拆分, 则可以得到状态转移路径空间.

由于状态空间的无穷性与未知性, 这一搜索空间无法被测试输入生成技术准确地生成. 在实际中, 现有技术往往通过刻画测试输入序列与状态转移路径之间的关系搜索状态转移路径空间. 对每一条测试输入序列, 通过分析其探索历史可以获取其对应的状态转移路径的信息, 从而在生成测试输入的循环过程中动态地建立这一搜索空间. 注意, 一条测试输入序列由于没有考虑初始状态与 ε – 输入, 可能代表多条状态转移路径. 这也同时为状态转移空间进行了一定的缩减.

状态转移路径空间通过对状态空间向细粒度转化获得, 因而保留了状态空间的所有信息, 同时进一步突出了状态间转移的关系, 因而在理论上能够保证充分性. 然而, 由于其无法被准确地生成, 测试输入生成技术从其能够获取的信息有限, 且依旧受困于未知的未知等挑战, 因而使得候选输入生成与评估有较大的困难, 影响了整体的充分性. 同时, 由于其大小进一步增大, 同时具有冗余性, 因而高效性较差.

原始状态空间. 原始状态空间即为状态空间 S. 有些测试输入生成技术并不建立特别的搜索空间, 而是直接使用 S 作为搜索空间. 使用状态空间 S 作为搜索空间的优势在于其保留了整个应用系统的完整信息, 不会存在信息的丢失. 然而由于直接在状态空间 S 上进行探索, 测试输入生成技术无法通过搜索空间获得应对 S 无穷性与未知性的帮助. 使用这一类搜索空间的应用往往只能选取相对简单粗暴的候选输入生成与评估策略. 根据充分性定义, 原始状态空间同样具有较高的充分性, 同时其相对状态转移路径空间而言有较好的高效性.

应用代码执行路径空间. 应用代码执行路径空间包含了被测应用代码中执行路径的相关信息. 其

可以包含不同粒度的执行路径, 如表示为应用的调用图、控制流图或完整的字节码执行路径的集合等. 应用代码执行路径空间是状态空间向粗粒度的转化. 不同的状态转移路径可能会执行相同的应用代码, 通过将这些状态转移路径的合并, 就可以得到相应的应用代码执行路径空间.

应用代码执行路径空间主要通过静态分析代码获得. 然而对于 Android 应用来说, 其有多个组件构成, 并且与 Android 系统提供的执行框架紧密相关. 因而其调用关系以及控制流往往会流出应用自身进入 Android 系统, 再由 Android 系统流回应用. 这给静态分析获得执行路径空间带来了挑战. 现有技术一般以 3 种方法应对: (1) 仅使用输入处理器内部明确的控制流与调用关系, 为每个输入处理器建立应用代码执行路径空间; (2) 通过理解 Android 系统的机制建立静态分析应用组件间控制流调用关系的方法, 通过模仿 Android 系统的方式连接断开的控制流和调用关系, 以建立完整的执行路径空间; 以及 (3) 在发送测试输入到应用系统的过程中动态地收集应用的执行信息, 利用动态信息补全控制流与调用关系, 以建立完整的执行路径空间. 使用第 3 种方法时, 测试输入生成技术会动态地更新搜索空间. 有时, 测试输入生成技术会结合第 2 和 3 种方方法, 以获得更准确的执行路径空间 [9, 10].

应用的代码蕴含大量有关应用运行系统运行行为的信息. 其相比状态空间包含更多的已知信息, 甚至对理解未知的未知也提供了一些帮助, 如应用在何种状态下会被哪种 ε – 输入影响. 这些信息可以用来指导状态空间的探索. 如通过分析未被当前生成的测试输入覆盖的执行路径, 可以发现尚未被探索的状态, 并且获取探索这些状态的输入的信息 [11~22]. 通过建立应用代码执行路径空间, 测试输入生成技术将探索状态空间的问题转化为探索应用执行代码路径空间的问题, 并通过覆盖这一空间内的路径以探索状态空间: 由于其通过状态空间向粗粒度转化, 其相对状态空间大小更小, 因而高效性有所提升. 同时, 由于其合并了覆盖相同执行路径的状态转移路径, 其充分性相对下降. 特别地, 不同粒度的应用代码执行路径空间具有差异较大的充分性/高效性权衡, 如字节码执行路径相比控制流图更加倾向于充分性, 而调用图相比控制流图更加倾向于高效性.

状态等价类空间. 状态等价类空间是在原始的状态空间基础上, 使用一个等价标准 ρ 将相同的状态合并成状态等价类. 即将状态空间 S 转化为一个状态等价类空间 $\mathcal{S}$. 对于一个等价类, 只需要探索其中的一个状态. 显然, 状态等价类空间是状态空间向粗粒度转化获得.

状态等价类空间可以静态或动态地获得. 通过分析代码, 可以静态地获得应用的信息, 以推测存在的等价类及之间的转移关系, 并建立相应的状态等价类空间. 如通过分析代码, 可以静态地获得应用的 Activity 转移图, 从而可以建立以当前活跃 Activity 为等价标准的状态等价类空间. 同样, 可以在生成测试输入的循环过程中动态地观测应用系统的状态, 并动态地建立缩减的状态空间.

状态等价类空间是对状态空间的抽象. 通过合并等价状态, 缩减的状态空间减少了未知的信息, 从而减少了候选输入生成与评估过程中未知信息的干扰. 同时, 缩减的状态空间仅保留了具有代表性的状态, 有效缩减了状态空间的大小, 具有很高的高效性, 很大程度上解决了状态空间无穷性的问题. 然而, 由于其极大地缩减了状态空间, 大量的信息被丢失, 使得实际探索过程中出现更多的不稳定性, 如一般的状态等价类空间忽略了 ε – 输入的影响, 从而可能导致状态等价类空间上预期的应用系统行为与实际的应用系统行为不一致, 如发送一个输入却没有探索到预期的状态等价类等. 同时, 由于其只要求同一个等价类内探索到一个状态, 导致大量的状态被忽略, 大幅降低了充分性.

不同的等价标准会生成截然不同的状态等价类空间, 其会具有不同的充分性/高效性权衡, 也会影响状态空间的探索效果 [23]. 常见的等价标准包括行为标准和界面标准两类. 行为标准主要按照应用系统 (1) 在当前状态可以处理的输入, 或 (2) 相应的状态转移情况对状态进行等价判断. 界面标准主要按照应用当前图形界面抽象后是否相同进行等价判断. 不同图形界面抽象也会导致不同的等价判断结果, 有时会极大地影响生成的状态等价类空间 [23]. 这两类等价标准有时也会组合使用 [24], 以获得确

定性较高的状态等价类空间, 同时也会增大空间的大小, 降低高效性. 如何选取合适的等价标准, 也是生成这一类搜索空间的测试输入生成技术关注的重点.

4.2 候选输入生成

现有技术通常有以下 4 种常用的候选输入生成策略: (1) 全部支持输入生成, (2) 全局分析生成, (3) 即时观测生成, 以及 (4) 已有测试输入修改. 这 4 种策略的充分性/高效性权衡按照顺序逐渐从倾向充分性向倾向高效性偏移. 我们分别介绍这 4 种策略.

生成全部支持输入. 每个测试输入生成技术都有其所支持生成的测试输入集合. 这一策略选取所有支持的测试输入集合或其形成的测试输入序列集合作为候选输入. 这一策略使得所有可能生成的输入都有机会被选取, 从而极大地保证了充分性. 然而, 由于其生成的候选输入集合很大, 同时给候选输入的评估带来较大负担, 因而高效性较差.

全局分析生成. 当使用这一策略时, 测试输入生成技术通过系统地分析整个搜索空间的信息, 以及对搜索空间与状态空间的所有探索历史, 借助这些全局信息确定哪一些测试输入 (序列) 可以进一步探索搜索空间, 并生成一组候选输入. 这通常是一个离线的过程, 即不考虑应用系统的当前状态, 只根据搜索空间及探索历史生成新的候选输入. 这一策略很大程度上依赖搜索空间所能提供的信息, 因而一般需要所能提供更多信息的搜索空间表示, 如应用代码执行路径空间. 通过系统的分析, 测试输入生成技术可以比较准确地选取能够进一步探索搜索空间的候选输入, 如符号执行技术通过分析完整执行路径的条件语句与控制流关系, 可以准确得到覆盖尚未覆盖的执行路径的测试输入 (序列) [11~22]. 因而这一类候选输入生成技术具有较高的充分性. 然而, 由于这一类候选输入生成技术需要系统地分析搜索空间以及探索历史, 其通常需要更长的时间和更多的资源, 如符号执行技术需要对执行路径进行约束求解. 这极大地降低了其高效性.

即时观测生成. 当使用这一策略时, 测试输入生成技术通过即时地观测应用系统的当前状态, 确定应用在当前状态可以处理的, 或者会引起应用系统状态改变的运行环境上的测试输入, 将其作为候选输入. 这是一个在线的即时过程, 很大程度上依赖于所能观测到的当前状态的信息. 通常观测的信息包括应用当前的图形界面、注册的输入处理器、运行环境的状态 (如传感器是否开启) 等. 这些信息同样常被用来建立状态等价类空间, 因而二者常配合使用. 在生成候选输入时, 有些技术也会借助静态分析的结果辅助决策, 如 GAT [25] 根据当前注册的手势输入处理器代码的静态分析结果确定哪些手势输入可以引起应用系统状态的变化, 并将这些输入作为候选输入.

这一策略只能考虑距离当前状态较近的状态空间, 相应对搜索空间的探索也有所受限, 因而其充分性较前两种生成方法有所下降. 同时, 由于其所考虑的空间范围较小, 可以较快地生成相对较少的候选输入, 因而高效性有所提升.

已有测试输入修改. 使用这一策略时, 现有技术一般通过对已有的测试输入 (序列) 进行修改, 以获一组取候选输入. 这一类技术通常使用简单的修改方法进行修改, 如随机删除一个输入序列的一个输入, 从而可以很快地得到所需的候选输入. 同时, 其相对于生成全部支持输入, 有效地缩减了候选输入数量, 因而具有较高的高效性. 然而, 由于在已有测试输入的基础上简单修改, 因而生成地候选输入探索搜索空间的能力较大程度上局限于已有测试输入的能力, 因而充分性较差. 另外, 这一类技术一般需要额外考虑如何获取初始的测试输入以进行之后的修改. 这一组初始测试输入会较大程度地影响后续生成的候选输入的探索能力. 获取初始测试输入 (序列) 的常见方法包括随机生成 [26~28], 或要求作为输入给出 [29~31].

4.3 候选输入评估

现有技术通常使用以下 4 类策略进行候选输入的评估: (1) 基于规则的评估, (2) 黑盒评估, (3) 启发式随机评估, 以及 (4) 均匀随机评估. 这 4 类策略的充分性/高效性权衡按顺序由倾向充分性向倾向高效性偏移. 我们分别介绍以下 4 种策略.

基于规则的评估. 使用这一策略的测试输入生成技术会根据一系列预先设定的规则预测候选输入的收益. 这些规则通常会借助搜索空间探索的历史对候选输入进行评估. 有时当前状态的信息也会被加以利用. 常见的规则一般有两类.

- 遍历优先规则. 这一类规则旨在选取候选输入以按照某种遍历顺序尽可能高效地遍历搜索整个搜索空间. 这一类规则的假设是搜索空间的遍历与状态空间探索的充分性有较高的正相关性, 因而能够更充分遍历搜索空间的候选输入也能够更充分地探索状态空间. 常见的规则包括深度优先、广度优先, 以及最少重启 (即回到初始状态) 等. 这一类规则一般需要当前状态、搜索空间的信息, 以及搜索空间的探索历史来进行预测. 其常与缩减的状态空间以及即时观测生成候选输入策略组合使用.
- 适宜性最高规则. 这一类规则旨在通过一个人工给定的适宜性函数评估候选输入发送到应用系统对探索状态空间有效性的提升以预测收益. 这一类规则通常通过预测候选输入所能覆盖的代码行数、执行路径等来评估其适宜性. 因而其通常需要结合应用代码执行路径空间进行评估. 其常与已有测试输入修改的生成策略组合使用, 通过评估已有测试输入的适宜性来预测候选输入的适宜性 [26~31].

这一策略会比较系统全面地评估候选输入对探索状态空间有效性的提升, 因而具有较好的充分性. 然而, 这一评估过程具有较大的计算量, 因而其高效性相对较差.

黑盒评估. 使用这一策略的测试输入生成技术通常使用一个预先设置好的黑盒函数来预测候选输入的收益. 与适宜性最高规则使用的人工给定的适宜性函数不同. 这一黑盒函数通常是一个事先训练好的机器学习模型. 通过预先的训练, 这一模型能够预测哪个候选输入可以对探索状态空间的有效性有较大的提升. 通过机器学习的模型, 其在预测时能够引入特有的领域知识与测试的普遍经验, 因而具有一定的充分性, 能够较为准确地预测候选输入的收益. 然而, 其无法全面地分析评估候选输入, 如可能会偏重于人类用户常生成的输入而忽视边界情况, 因而相比基于规则的评估充分性较低. 但其黑盒函数的设置一般为预先设置好, 无需在测试输入生成过程中进行调整, 因而可以较快地进行评估, 所以高效性有所提升.

启发式随机评估. 使用这一策略的测试输入生成技术通常使用一组简单的启发式规则指导一个随机预测的过程. 这些技术会建立一组启发式规则, 借助搜索空间与探索历史, 启发式地给予每个候选输入一个概率值. 之后按照这个概率值随机选取一个候选输入作为评估出的最优结果. 常见的启发式规则包括根据每个候选输入被选取的次数给予权值, 以及根据其组织的输入序列长度给予权值等. 这一策略可以很快地对候选输入进行评估, 有较好的高效性. 然而, 其启发式规则的合理性以及随机选择的随机性都影响了这一类评估策略的充分性, 使其充分性较差.

均匀随机评估. 这一策略均匀地随机选取一个候选输入作为预测最优的结果. 其省略了启发式随机评估策略概率值计算的步骤, 仅需一步随机选取, 因而具有最高的高效性. 相对应的, 其评估的结果不稳定, 且没有任何对实际效果的考虑, 因而具有最差的充分性.

4.4 现有技术分析与总结

本小节具体分析和总结现有技术. 我们发现, 搜索空间的表示往往可以展现出一个测试输入生成技术方法的内见. 因而, 我们按照现有技术搜索空间的表示将现有技术分类, 并分别进行分析.

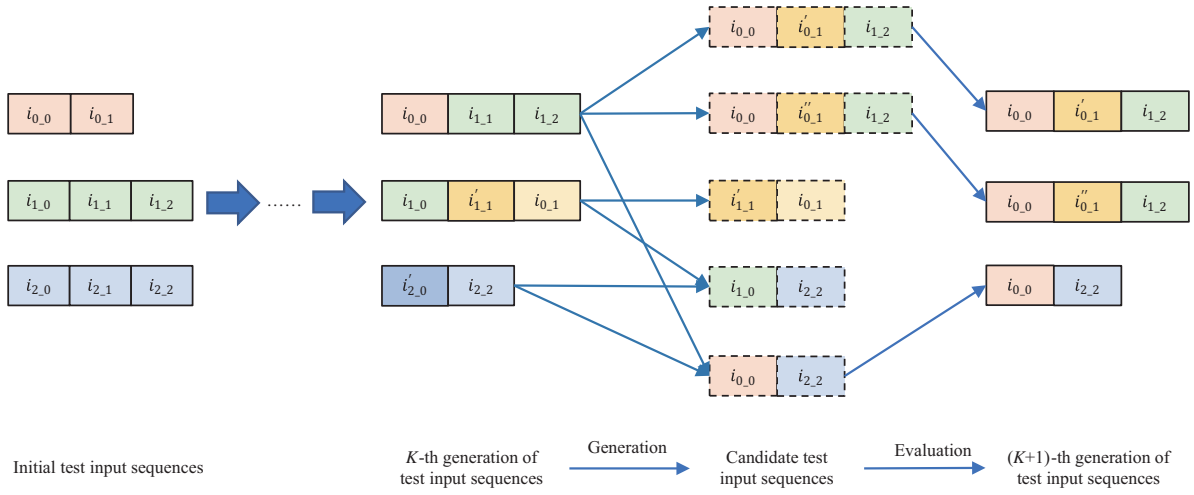


图 3 (网络版彩图) 基于遗传算法的测试输入生成图示

Figure 3 (Color online) Illustration of evolutionary algorithm-based input generation

为了进一步提高充分性或高效性, 一些现有技术会选取多种搜索空间表示以及候选输入的生成与评估策略, 在不同的情况下使用不同的组合, 以达到所需的效果以及更合适的充分性/高效性权衡. 在这些技术中, 有一些会选取一个主要使用的搜索空间表示, 并使用其他搜索空间进行辅助. 而另一些技术并列地使用多个搜索空间以探索状态空间的不同部分. 对于第 1 类技术, 按照其主要使用的搜索空间进行分类. 对于第 2 类技术, 在 4.4.5 小节中进行分析.

4.4.1 状态转移路径空间

使用这一类搜索空间表示的现有测试输入生成技术主要通过离线地生成测试输入序列, 将这些测试输入序列发送到应用运行系统并观测应用运行系统的状态变化, 以刻画状态转移路径空间, 进而探索状态空间. 按照候选输入生成与评估的方法分类, 常见的技术包括基于遗传算法 [32, 33] 进行测试输入生成的技术、基于机器学习算法进行测试输入生成的技术等.

基于遗传算法的现有技术. 遗传算法将样本按生成代进行划分, 初始样本为第 1 代. 通过交叉及变异上一代现有样本, 遗传算法可以获得新一代的样本, 并通过对这些新样本进行适宜性评估筛选出优质的样本予以保存, 基于此通过反复迭代最终获得最优质的一代样本. 如图 3 所示, 当应用在测试输入生成技术上时, 一个测试输入序列通常被当作一个样本. 通过交叉及变异现有的测试输入序列, 可以得到新的测试输入序列, 并使用一个适宜性函数评估其探索空间的能力, 选取最优的测试输入序列生成并发送到应用系统中去. 即在状态转移路径空间的基础上, 使用已有测试输入修改的候选输入生成策略, 并对候选输入进行基于规则 (适宜性最高规则) 的评估. 在评估候选输入时, 由于无法在实际生成其并发送到应用系统前准确获知其适宜性, 因而一般通过评估用于生成其的上一代已有测试输入序列的适宜性预测其适宜性. 同时, 为了保证新的测试输入序列的多样性, 在基于规则评估的基础上, 往往增加启发式随机评估的过程, 即根据预测的适宜性加权选取测试输入序列.

不同的基于遗传算法的测试输入技术有着不同的初代测试输入获取方式, 交叉/变异方法, 即修改已有测试输入的方式, 以及不同的适宜性函数. Sapienz [26] 通过生成全部支持输入与均匀随机选择组合的策略均匀随机生成初代测试输入序列. 它将每一代测试输入序列集合进一步划分为多个测试套

件, 每个测试套件包含相同数量的测试输入序列. 在修改已有测试输入时, Sapienz 采用 Pareto 最优机制 [26] 重新组织已有测试输入序列与套件, 而不引入新的测试输入. 在进行适宜性评估时, Sapienz 倾向于选择覆盖更多代码, 导致更多崩溃, 以及更短的测试套件. 值得注意的是, 在生成与修改测试输入序列时, Sapienz 引入了基序序列 [34] 的概念. 仿照生物学的概念, Sapienz 认为存在一些测试输入序列在应用实际使用中常由用户生成, 但在随机生成和修改过程中比较难以生成, 如在搜索框中输入相关字符串并点击搜索键, 即基序序列. Sapienz 人为地加入这些序列, 在随机生成初代测试输入序列时作为高阶单个输入生成. POLARIZ [28] 在 Sapienz 的基础上, 通过大量的实证研究进一步增加了基序序列的种类. Mahmood 等 [35] 于 2012 年提出的算法借助了应用代码执行路径空间. 其通过静态分析代码获取每个 Activity 的调用图, 并根据调用图为每个 Activity 生成单元测试的初代测试输入序列. 之后, 其随机修改已有测试输入的参数 (如点击输入的点击坐标等) 产生新一代的测试输入序列, 并根据其所覆盖的调用图路径进行评估. EvoDroid [27] 也借助了应用代码执行路径空间. 其通过静态分析建立了整个应用的调用/控制流图, 并根据测试输入序列覆盖路径的深度对其进行评估. 其特别之处在于, EvdDroid 根据测试输入序列不同子序列所覆盖的路径所属的组件将序列进行分段. 在对上一代测试序列进行修改时, 只修改最深处的一段, 而保留之前的分段, 以保证生成的测试输入序列保留上一代的效果. 通过逐段尝试覆盖, EvoDroid 对测试输入过程进行了有效地导向. TCM [30] 不进行初代测试输入序列的生成, 需要外界给定. 其通过增加循环的输入序列、应用暂停/恢复操作对、改变字符串输入值、增加执行环境操作、减少输入之间等待的时间等方式修改已有测试输入序列, 并借助缩减的状态空间模型 (借助其他技术获得) 的状态转移路径覆盖进行评估.

基于遗传算法的技术往往需要生成大量的测试输入序列, 并将其发送到应用运行系统以根据探索历史进行适宜性评估, 因而往往比较低效. 现有技术如 Sapienz 一般通过建立多个应用运行系统, 并发执行测试输入序列的方式应对低效的问题. 同时, 其一般选取具有较高充分性的策略 (状态转移路径空间与基于规则的评估策略), 因而一般能够比较有效地探索状态空间. 然而其修改已有测试输入序列往往较为随机, 因而严重依赖于适宜性函数的评估进行选择. 这要求这一类技术提供较为充分的适宜性评估函数, 而状态空间与搜索空间存在较多未知的挑战使得找到这样的评估函数尤为困难. 现有技术使用的评估函数能够一定程度上反映测试输入序列的适宜性, 但仍不能完全准确地判断其带来的探索状态空间有效性的提升 [28].

基于机器学习的现有方法. 有些技术将所有支持生成的测试输入作为候选输入, 并使用机器学习的方法进行黑盒评估. Liu 等 [36] 在 2017 年提出了针对字符串输入生成的技术. 其通过 RNN [37] 与 word2vec [38] 等机器学习技术, 采用所有支持输入生成与黑盒评估的策略, 在每个可以接收字符串输入的状态 (根据应用当前图形界面判断) 根据到达当前状态的状态转移路径以及应用在当前状态的图形界面信息, 生成一个字符串输入. MONKEYLAB [39] 同样使用机器学习的技术生成测试输入序列. 其同样采用所有支持输入生成与黑盒评估的策略. 通过预先对用户生成的输入序列导致的状态路径转移进行数据挖掘, MONKEYLAB 建立一个测试输入序列的语言模型. 在生成测试输入时, 其将所有可支持生成的测试输入作为候选输入, 并利用生成的语言模型, 根据到达当前状态的状态转移路径对候选输入进行黑盒评估. 在评估时, MONKEYLAB 可以选择严格遵循语言模型、完全违背语言模型, 以及混合 3 种方式, 分别用以生成探索常被用户探索状态空间区域的测试输入序列与探索边界状态的测试输入序列.

其他技术. 还有一些技术基于状态转移路径空间通过对已有的测试输入插入特定的环境输入生成新的测试输入. THOR [31] 采用已有测试输入修改与均匀随机评估的策略, 为已有的测试输入添加无效果的环境输入序列. THOR 定义无效果的环境输入序列为不会影响应用处理后续测试输入序列

的输入序列, 如旋转手机后再旋转回来. 其通过随机在已有输入序列中插入无效果环境输入序列的方式生成候选输入. 在对候选输入进行评估时, THOR 随机选取一个候选输入生成. 特别地, 其在执行这一候选输入时确保无效果的环境输入序列不会影响后续执行, 如等待处理前一个输入完成后再发送无效果的环境输入序列. Amalfitano 等 [40] 在 2013 年提出的方法与 THOR 类似, 但是其插入的环境输入可能会影响应用的状态.

4.4.2 原始状态空间

主要使用这一类搜索空间的测试输入生成方法一般不借助搜索空间的信息, 并采用较为简单的候选输入生成与评估策略. Monkey[3)]是 Google 在 Android 系统中附带的测试工具. 其通过全部支持输入生成与均匀随机评估的策略, 随机地快速生成大量的测试输入. Xdroid [41] 在 Monkey 的基础上, 增加了对简单环境输入生成的支持. 当应用执行过程中访问环境的资源时, Xdroid 判断应用所需的资源的值, 生成相应的环境输入, 或请求外部用户生成所需输入. Shahriar 等 [42] 为了检测 Android 应用的内存泄露问题, 在 Monkey 的基础上同样增加了对环境输入生成的支持. 其增加了包括重启应用、反复旋转屏幕、更换更大图片文件等环境输入的生成. DroidFuzzer [43] 采用类似的方法探索应用系统可能到达的初始状态. 所有应用均会声明开放的外界 Intent 接口, 并根据接收到的的 Intent 启动自身并进行相应操作, 到达一个初始状态. DroidFuzzer 通过修改测试用例修改的候选输入生成策略以及均匀随机评估策略生成 Intent 输入. 其半人工地生成一组 Intent 输入, 并按照预设的方式对其进行修改生成候选输入, 并每次随机选择一个候选输入发送给应用系统.

这一类技术的方法都较为简单, 因而充分性难以得到保证. 然而, 由于其方法简单, 其可以快速地生成大量的测试输入, 具有很高的高效性, 因而在实际中反而具有较好的效果. Choudhary 等 [4] 在 2015 年发表的实证研究的比较实验中, Monkey 在所有被选取的测试输入生成技术中具有最好的有效性. 另外, 由于这一类技术能够比较简单高效地生成测试输入, 常被其他技术用来作为辅助工具. 如 Sapienz [26] 就使用 Monkey 生成初代测试输入序列.

4.4.3 应用代码执行路径空间

主要使用这一类搜索空间表示的测试输入生成技术主要通过系统地全局分析搜索空间, 确定能够进一步覆盖应用执行代码的测试输入序列, 并生成其以探索状态空间. 常见的技术包括基于符号执行的技术与静态生成技术两种.

基于符号执行的技术. 符号执行 [44] 是模型验证 [45] 的一种方法. 其通过使用一条执行路径上的分支条件表示这条路径的方式将一个程序转化为一个逻辑公式集合. 通过约束求解, 其可以获得探索每条执行路径所需的输入, 从而能够系统地覆盖所有执行路径.

如图 4 所示, 许多 Android 应用测试输入生成技术借助符号执行生成测试输入. 符号执行属于全局分析生成与基于规则评估 (深度遍历优先) 的策略. 然而, 对 Android 应用进行符号执行有 3 点独特的挑战. (1) Android 应用是事件驱动的, 需要在不同的状态接收不同的事件输入才能执行不同的路径, 不能在程序开始时全部确定. 常规的符号执行无法生成这样的输入. (2) Android 应用严重依赖于 Android 系统控制其行为, 这使得应用的控制流经常流出应用进入 Android 系统, 之后又再流回应用. 由于破碎的控制流, 常规的符号执行无法直接处理 Android 应用. (3) Android 应用使用了大量的 Android 特有的库函数, 许多这样的函数都是使用 C/C++ 语言在底层编写. 对 Android 应用进行 Java 层符号执行的工具无法处理这些函数. 同时, 符号执行常见的问题如路径爆炸问题仍旧存在.

3) UI/application exerciser monkey. https://developer.android.com/studio/test/monkey.

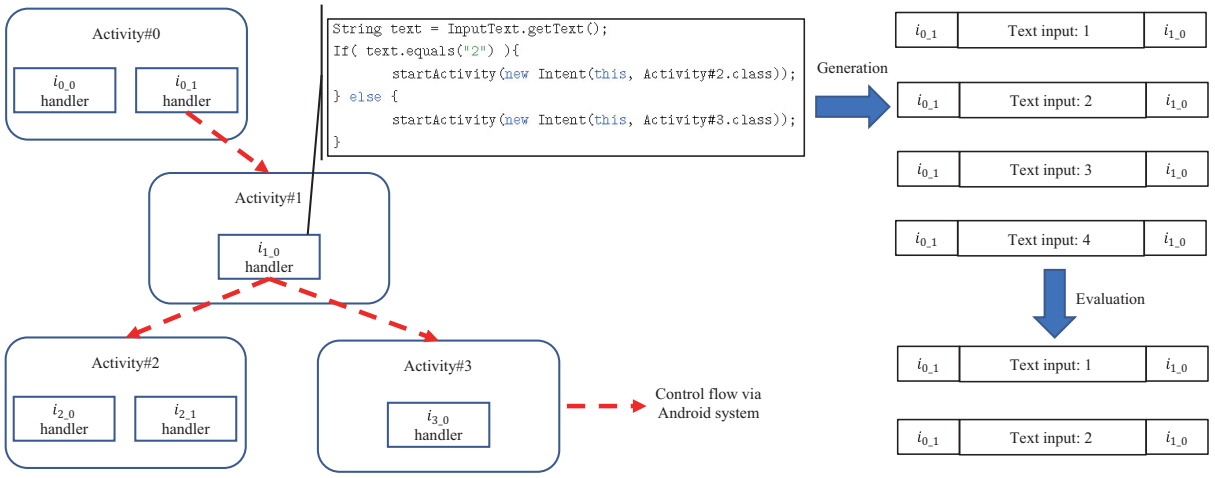


图 4　(网络版彩图) 基于符号执行的测试输入生成图示

Figure 4　(Color online) Illustration of symbolic execution-based input generation

现有技术均提出应对一个或多个这些挑战的方法.

一些技术通过模拟 Android 系统与库函数的方式解决控制流的问题, 并根据控制流及数据流信息获得覆盖每条执行路径所需的输入. Mirzaei 等 [20] 在 2012 年提出的方法采用全局分析生成的策略, 通过静态分析应用的全局调用图确定覆盖每条执行路径需要的事件输入, 并根据这一调用图生成候选输入. 对于数据输入, 其借助 Java PathFinder[4]的符号执行系统采用常规符号执行的方法生成. 为了解决控制流与库函数的问题, 其建立模拟的 Android 系统以及库函数. SIG-Droid [14] 通过建立更详细的控制流图, 并借助其生成事件输入, 对字符串输入进行动态符号执行. DroidPF [11] 则对每个组件分别建立控制流图. 通过增量式的扩展, DroidPF 不断对新遇到的组件建立控制流图, 以此刻画搜索空间, 并采用全局分析生成的策略生成事件输入. SynthesiSE [12] 采用类似 SIG-Droid 的方法进行输入生成. 不同的是, 其不是人工建立模拟的库函数, 而通过程序合成的方法, 在需要对一个库函数进行符号执行时, 通过多次使用满足该路径的不同输入执行的方式获得样本, 并根据样本生成相应的模拟库函数. 最后, GreenDroid 以及其扩展的技术 [18,19,46] 使用即时观测生成的策略以及基于规则评估的策略生成事件输入. 其建立一个控制器记录当前状态所注册的输入处理器, 将对应的输入作为候选输入, 并按照注册顺序选取一个尚未在当前状态发送的输入生成. 同时, GreenDroid 还建立了一个时序逻辑的 Android 系统模型以及模拟的 Android 库, 以应对破碎控制流的问题.

一些技术通过直接插装或扩展 Android 系统的方式解决控制流与库函数问题. Anand 等 [47] 使用直接插装 Android 模拟器及被测应用的方法进行动态符号执行. 其采用全局分析生成的策略生成候选事件输入. 通过插装 Android 模拟器与被测应用, 其通过符号化执行整个输入分发与处理的过程的方式, 确定应用在当前状态可以处理哪些输入, 将其作为候选输入. 在选取时, 其通过启发式随机评估随机选取一个还未在当前状态发送的, 且能覆盖不同执行路径的输入生成. 因为直接借助 Android 模拟器执行, 其不需要考虑破碎控制流的问题. CRAXDroid [48] 采用类似的方式处理控制流. 对于数据输入, 其使用 S^2E[5]符号执行相关的处理器生成所需测试输入, 再发送到应用系统. 然而对于事件输入,

4) Java PathFinder. http://javapathfinder.sourceforge.net/.
5) S^2E. http://s2e.systems/docs/.

其借助 Monkey 生成, 即采用全部支持输入生成与均匀随机评估的策略生成.

最后, 有一些技术选择控制流完整, 且仅需单个输入的搜索空间部分进行符号执行. SnowDrop [49] 针对 Service 组件生成 Intent 输入. 因为 Service 组件内部控制流完整且执行路径较短, 其可以通过常规符号执行获取所有可能的执行路径. 之后通过全局分析生成的策略, 其通过分析这些路径以及应用配置文件获取覆盖他们所需的 Intent 输入作为候选输入. 生成候选输入后, 其通过启发式随机评估随机选取一个还未在当前状态发送的, 且能覆盖不同执行路径输入生成. EOEDroid [13] 通过分析与处理网页输入相关的处理器以找到安全漏洞. 对于事件输入, 其借助 Monkey 进行生成. 当到达接收网页输入的状态后, EOEDroid 再进行符号执行. 因为这类处理器内部控制流完整保留在应用内部, 不需要考虑破碎控制流问题. 借助与安全相关的 API 列表, EOEDroid 只保留调用这些 API 的路径, 并生成相应的候选网页输入, 之后 EOEDroid 以启发式随机评估的方式随机选取一个能够覆盖新的执行路径的网页输入发送.

基于符号执行的测试输入生成技术能够系统全面地遍历整个搜索空间, 并生成相应的测试输入, 同时应用代码执行路径有较好的充分性, 因而能有比较有效地探索状态空间. 但是, 由于约束求解的复杂性, 以及路径爆炸等问题, 这一类方法通常需要较长的时间分析生成测试输入, 因而具有较差的高效性. 同时, 由于应用运行系统可以接收种类繁多的输入, 符号执行很难处理所有类型的输入, 这也影响了其有效性.

静态生成技术. 这一类技术一般通过静态生成建立控制流图、调用图等形式的应用代码执行路径空间, 采用全局分析生成策略生成候选输入, 并以基于规则的评估策略选择一个可以进一步覆盖执行路径的测试输入序列. APEChecker [15] 使用反向符号执行的方式, 生成测试输入序列探索特定可能导致异步错误的执行路径, 通过静态分析找到可能导致异步错误的方法. 之后从该方法起始, 其通过静态分析的方式反向建立调用图, 并最终回溯到应用入口 Activity. 然后再根据这一执行路径生成相应的候选输入. MAMBA [50] 通过静态地分析应用建立控制流图. 同时, 其通过静态分析找到代码中调用隐私敏感 API 的调用链. 之后, 其根据控制流图生成所有能够到达该调用链的候选输入, 并对这些候选输入进行基于规则的评估, 每次选取尚未选取的候选输入中最短的生成, 直至能够覆盖目标调用链. Griebe 等 [51] 在 2014 年提出的技术需要 Activity 的功能 UML 图作为输入. 其通过系统分析 UML 图, 将其转化为 Petri 网 [52], 通过计算该 Petri 网的可达性, 获取候选输入, 并以启发式随机评估的方式随机选取一个尚未发送的候选输入生成. PATDroid [29] 采用已有测试输入修改与基于规则评估的策略, 为已有的测试输入增加应用权限输入. 其通过组合所有可能的权限输入 (给予应用一个权限, 如读写通讯录), 并将这些输入序列作为已有测试输入的前缀以获得候选输入. 借助应用代码执行路径空间, 通过静态分析与动态观测已有测试输入覆盖的应用执行路径, PATDroid 利用遍历优先规则选取只包含能够影响应用接收到已有测试输入后执行的权限的候选输入. IntentFuzzer [53] 通过全局分析的生成策略, 以及启发式随机选择的策略为应用的 Service 和 Braodcast Receiver 生成 Intent 输入. 其通过分析应用的配置文件 (AndroidManifest 文件) 获取应用所能接收的 Intent 输入作为候选输入, 并随机选取这样的 Intent 输入. 另外, 它还借助应用代码执行路径空间, 通过动态跟踪应用处理接收到 Intent 的执行路径信息 (如分支条件等), 进一步生成可以覆盖不同执行路径的 Intent. LESDroid [54] 采用类似的方法为所有组件生成 Intent 输入. 其提供所有支持输入生成与全局分析生成两种候选输入生成策略. LESDroid 可以完全随机地生成候选输入或只通过分析应用配置文件生成候选输入. 同时, 其还可以通过借助应用代码执行路径空间, 通过系统分析与 Intent 输入相关的变量与域, 生成可能导致不同执行结果的 Intent 输入. 其同样采用均匀随机评估的策略.

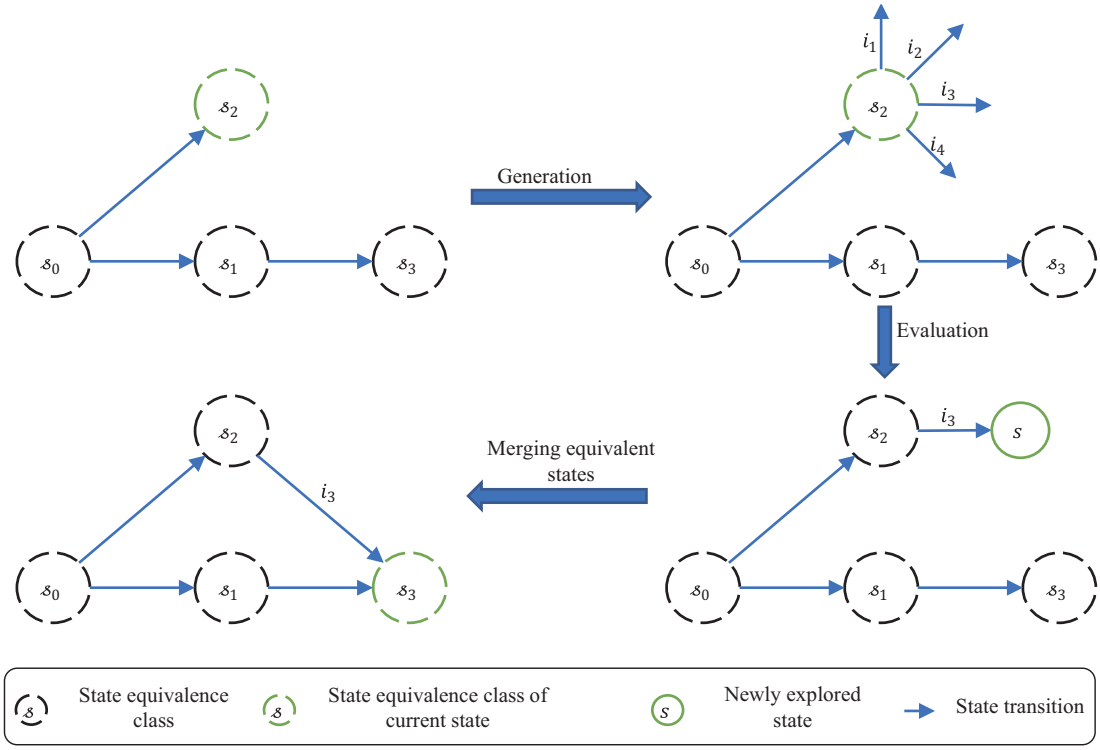


图 5 (网络版彩图) 动态建立状态等价类空间图示

Figure 5 (Color online) Illustration of dynamic generation of reduced state space

4.4.4 状态等价类空间

大部分的测试输入生成技术使用状态等价类空间作为搜索空间. 根据其建立搜索空间的方式, 可以分类动态建立技术与静态建立技术.

动态建立状态等价类空间技术. 这一类技术在开始时搜索空间仅包含初始状态. 如图 5 所示, 在发送每个测试输入后, 观测应用系统的当前状态, 根据观测的结果进一步扩展细化搜索空间, 逐步建立完整的状态等价类空间 $\mathcal{S}$. 因而其一般采用即时观测生成的候选输入生成策略. 这些技术的不同点主要在于如何进行候选输入评估、使用何种等价标准 ρ, 以及支持生成哪些测试输入.

基于规则的评估策略. 大多数技术采用基于规则的评估策略, 特别是遍历优先规则. 一些技术使用深度优先的规则进行候选输入评估, 它们优先选择能够探索离初始状态更远的状态的候选输入, 且在同一状态, 除了回溯输入 (重启应用或点击返回按钮) 外, 不会生成同一个输入两次. 在没有可选的候选输入后, 它们会选择回溯输入生成. A^{2}T^2 [55] 即采用这样的评估策略. 它的等价标准 ρ 为相同的图形界面与注册的输入处理器, 并支持生成图形界面上的事件输入. A^{3}E [56], EHBDroid [57], 以及 GAT [25] 采用相同的策略, 且其等价标准 ρ 均为相同的当前活跃 Activity. 不同的是, A^{3}E 同时借助静态分析, 确定哪些 Activity 可以作为应用入口, 通过分别启动这些 Activity 探索到不同的初始状态, 在此基础上建立多初始状态的状态等价类空间. EHBDroid 通过注册的输入处理器获取应用当前可处理的输入. 其通过直接向这些处理器发送输入的方式, 确保能够切实改变应用系统的状态. GAT 在 A^{3}E 的基础上, 通过静态分析获取可以改变应用当前状态的手势输入, 将其添加至候选输入. AATT+ [9,10] 也采用深度优先的规则进行评估. 不同的是其在不同的状态空间部分采用不同的等价标准 ρ. 其首先通过

静态/动态混合分析 [9], 或动态符号执行 [10] 获取可能导致并发错误的测试输入. 在到达能够处理这些输入的状态前, 其使用相同的 GUI 作为 ρ, 并生成图形界面上的事件输入. 当到达一个这样的状态后, 其将等价标准 ρ 细化为相同的应用上可能发生的执行多线程共享资源的读写操作的 ε – 输入集合, 将其作为候选输入. 通过深度优先地生成这些 ε – 输入, 其生成所有可能的共享资源读写序列, 以触发潜在的并发错误. RacerDroid [58] 使用类似的策略.

一些技术使用广度优先的规则进行候选输入评估. 它们优先选择探索离初始状态更近的状态的候选输入, 且在同一状态, 除了回溯输入 (重启应用或点击返回按钮) 外, 不会生成同一个输入两次. 在每发送一个测试输入并到达一个新状态后, 测试输入技术会检查是否存在离初始状态更近的, 应用系统尚未接收到所有可处理候选输入的状态. 是则其选择回溯输入. 这通常导致其需要经常回溯应用系统到之前的状态. 这对于 Android 应用来说并不是一个简单的任务 [10]. MobiGUITAR [59] 即采用这样的评估策略. 其使用相同的图形界面作为等价标准, 并支持生成图形界面输入. 特别的是, 其在动态建立状态等价类空间之后, 会采用全局分析生成以及启发式随机评估的策略, 生成满足在状态等价类空间上边对覆盖标准 (pair-edge coverage criteria, 其中边为状态空间中的状态转移) 的测试输入序列. CRASHSCOPE [60] 采用相似的等价标准与候选输入生成评估策略探索状态等价类空间. 不同的是, 其考虑了环境输入. 通过预先的静态分析, CRASHSCOPE 确定应用不同 Activity 调用的环境资源访问方法, 并进而获得能影响应用状态的环境输入. 当观测应用系统当前状态时, 其获取当前活跃 Activity, 并将相应的环境输入添加到候选输入. AMOLA [23] 同样采用与 MobiGUITAR 类似的策略探索状态等价类空间. 其使用相同的图形界面作为等价标准. 特别的是, 其为图形界面的等价判断提供了不同的标准, 通过在不同的抽象层面上判断其是否等价, 以应对不同的需求. FraudDroid [61] 在 MobiGUITAR 的策略基础上, 给与广告相关的输入更高的收益预测. 其主要服务于检测应用广告欺诈行为. FraudDroid 通过分析应用当前图形界面上各个组件的类型、属性、位置判断其是否为广告组件, 并给出与之相关的输入更高的收益预测.

最后, 一些技术使用特定的图遍历优先规则以提升状态等价类空间的高效性或充分性. SwiftHand [24] 通过对整个当前建立的状态等价类空间进行搜索, 以找到能够最少重启应用的探索顺序, 并相应评估候选输入. 其使用相同的图形界面与相同的图形界面事件输入状态转移情况 (即 $\forall s_1 \in S$, $s_2 \in S$. $\rho(s_1) = \rho(s_2) \rightarrow (\forall i \in I_{\mathrm{GUI}}.\ \rho(s_1, i) = \rho(s_2, i))$. 其先通过图形界面判定状态是否等价. 当在之后的探索过程中发现状态转移情况出现了非确定性时, 采用被动学习的方法重新建立状态等价类空间. DroidDEV [62] 同样试图寻找最快的探索顺序. 其使用相同的图形界面组件属性以及类别作为等价依据. 并在进行候选输入评估时给与导致最少回溯的输入更高的收益预测. 另外, 其考虑了字符串输入: 在生成事件输入前, 先根据图形界面上字符串相关组件的属性, 为其生成预设的字符串输入. Autodroid [63] 在进行候选输入评估时, 根据每个候选输入所能增加的测试输入组合数预测其收益. 优先选取最能够增加测试输入组合数目的测试输入.

黑盒评估策略. 一些技术会借助机器学习等方法对候选输入进行黑盒评估策略. QBE [64] 使用预先训练的 Q – 学习模型 [65] 对候选输入进行评估, 以预测其增加代码覆盖率以及进一步探索状态等价类空间的收益. 其使用相同应用当前可处理的输入作为等价标准. AimDroid [66] 结合广度优先规则评估与黑盒评估策略, 并配合使用两种不同的等价标准. 在外层, AimDroid 使用相同当前活跃 Activity 作为等价标准. 对每个等价类内部, 其使用相同应用当前可处理的输入作为等价标准. 对于每个等价类内部, 其将应用所有可处理的输入作为候选输入, 并使用在线学习的 Q – 学习模型进行评估. 当发送的输入导致当前活跃 Activity 变化, 即离开该等价类时, AimDroid 阻止该变化, 并将该输入 (序列) 作为外层的状态等价类空间上该等价类所能够处理的候选输入. 当等价类内部状态等价类空间探索完毕

后, 外层状态等价类空间按照广度优先规则评估所有候选输入.

启发式随机评估策略. 一些技术会使用启发式随机的策略对候选输入进行评估. DynoDroid [67] 使用相同注册输入处理器集合作为等价标准. 其根据应用系统当前状态, 即注册的输入处理器生成所有对应的候选输入. 在进行评估时, 其启发式地按照每个候选输入在当前状态等价类被选取的频率给与每个候选输入概率, 并随机选取一个生成. 越少被发送的候选输入越有可能被选取. PUMA [68] 在其提出的可编程自动测试平台上实现了同样的候选输入生成与评估策略. LeakDAF [69] 在 DynoDroid 的基础上, 当被测应用接收一个测试输入后其活跃 Activity 发生变更, LeakDAF 会固定生成一个回溯事件回到之前的状态, 并再次发送相同的测试输入, 然后旋转屏幕, 以此尝试触发应用的内存泄漏. Caiipa [70] 在 DynoDroid 的基础上, 更多地考虑环境输入. 其建立一个存有大量应用测试记录的云平台. 通过对被测应用访问环境资源的方式, Caiipa 找到一组与之相似的以往被测应用. 通过考察哪些环境输入更好地探索了这些以往被测应用的状态空间, 其为当前被测应用选定一组环境输入, 并加入到候选输入中. Cadage [71] 也采用启发式随机的策略评估候选输入. 其使用相同图形界面作为等价标准 ρ. 其通过分析图形界面确定应用在当前状态能够处理的输入, 将其作为候选输入. 进行候选输入评估时, 其搜索状态等价类空间以判定哪些候选输入更有可能使应用状态向未被探索的状态转移, 并结合事件的发送频率给与每个候选输入概率. ACAT [72] 使用一个预先训练好的分类器, 根据应用在当前状态的活跃 Activity 将该状态归入其提出的 7 个等价类中, 并以此作为等价标准. 对于每一个等价类, 其生成一组预设的候选输入, 并启发式随机选取一个未被选取的候选输入发送到应用系统.

静态建立搜索空间技术. 一些技术通过静态地分析应用代码建立状态等价类空间, 并根据其生成测试输入. 其与使用应用代码执行路径空间的技术相比, 生成的搜索空间更加抽象, 不包含详细的控制流或调用关系信息. Yang 等 [73] 在 2013 年提出的方法通过静态分析应用代码, 找到所有 Activity 及其所能处理的输入, 即获得所有的状态等价类. 接着从初始状态开始, 动态地观测应用当前状态以获取活跃的 Activity, 并根据静态获得的状态等价类空间信息生成候选输入, 并以深度优先规则的策略评估候选输入. FragDroid [74] 通过静态分析应用代码, 建立完整地 Activity/Fragment 转移图, 以作为缩减的状态空间. 其通过系统分析生成的状态等价类空间, 将所有覆盖不同状态转移路径的测试输入序列作为候选输入, 并使用广度优先规则进行评估选取. 特别地, 每当一个测试输入序列被发送到应用系统, FragDroid 会观测其探索到的状态, 并相应地更新细化静态生成的状态等价类空间. Sentinel [75] 通过静态分析建立应用的 Activity 启动界面、对话框界面, 以及菜单界面的转移图, 以作为缩减的状态空间. 同时, 其通过静态分析获得每个 Acivity 的传感器使用情况. 接着, Sentinel 通过使用将状态等价类空间中状态转移路径映射到上下文无关语言的方法, 通过结合传感器使用情况, 获得可能导致传感器泄露的状态转移路径, 并为这些路径生成测试输入序列. 这属于全局分析生成与启发式随机选择的评估策略.

4.4.5 组合的搜索空间表示

大部分技术只使用或主要使用一种搜索空间的表示. 然而如 4.1 小节所述, 单一的搜索空间表示都有其不足之处, 难以应对探索整个状态空间的无穷性与未知性. 因而近年来, 越来越多的技术开始尝试结合使用多种搜索空间表示, 结合其长处以提高探索的有效性. 本小节列出一些代表性的工作.

EventBreak [76] 结合使用了原始状态空间、状态转移路径空间, 以及状态等价类空间. 其希望探索可能存在性能问题的深层状态. 特别地, EventBreak 希望找到两对状态 – 输入对, 在一个状态上处理一个输入会增加在另一个状态上处理另一个输入的时间, 即增加应用 ε – 输入的数量. 首先, 其使用原始状态空间, 通过所有支持输入生成与均匀随机评估的策略随机地探索状态空间. 在探索过程

中, 其记录处理每个输入的 ε – 输入的数量 (以执行的条件分支语句与方法调用语句数量表征). 接着, EventBreak 使用状态转移路径空间, 通过分析记录的数据, 找到可能互相影响的状态转移. 最后, EventBreak 借助已有的技术建立状态等价类空间, 通过全局分析生成的策略, 找到能够交替到达两个可疑状态的所有候选输入. 接着, EventBreak 使用启发式随机的评估策略, 根据每个候选输入的长度给予不同的概率 (越短则概率越高), 并随机选取生成, 以此探索深层空间, 以检测可能的性能问题.

Banerjee 等 [77] 在 2014 年提出的方法中结合使用了状态等价类空间与状态转移路径空间. 其测试输入生成方法服务于其在文中提出的应用能耗错误与浪费的自动检测. 因而其希望探索能够触发这类问题的状态. 首先, 其使用缩减的状态空间, 借助 DynoDroid [67] 进行测试输入生成. 在这一过程中, 其将生成的测试输入转化为输入流图, 每个输入作为一个点 (相同输入合并), 而生成的先后顺序作为边. 通过这样的输入流图, 其将状态等价类空间转变为状态转移路径空间. 接着, 其首先使用全局分析生成的策略, 生成输入流图中所有的测试输入序列作为候选输入. 采用基于规则的评估, 依次将所有长度小于等于常数 k 的测试输入序列从初始状态开始发送到应用运行系统. 在应用系统处理这些输入时, 记录系统方法的调用, 并以此找到潜在的能耗问题. 然后, 对于剩余的候选输入, 采用基于规则的评估策略, 借助探索历史 (1) 预测每个候选输入能够触发的系统调用, (2) 评估每个候选输入与已知触发能耗问题的测试输入序列的相似性, 以及 (3) 预测每个候选输入可能触发的前所未见的系统调用. 根据这些评估与预测结果, 其选取预测收益最高的 (能够触发尽可能多的系统调用, 且与已知能耗问题触发测试输入序列相似) 候选输入生成并发送.

AGRippin [78] 同样结合了状态等价类空间与状态转移路径空间. 其特点在于, 它的状态等价类空间依赖于状态转移路径空间. AGRippin 的状态等价类空间使用相同的图形界面作为等价标准 ρ. 其首先使用状态转移路径空间, 采用遗传算法的方式, 生成测试用例. 在进行候选输入生成时, 采用已有测试输入修改的方式, 修改测试输入序列中数据输入的值, 以及交换已有序列的后缀. 进而, 在进行候选输入评估时, 采用适宜性最高原则, 考虑候选输入的代码行数覆盖以及与探索历史中测试输入序列的相似性. 在发送选取的候选输入时, AGRippin 观测应用的图形界面. 当遇到不属于任何已知等价类的图形界面时, AGRipping 会相应更新状态等价类空间, 并停止这一轮基于遗传算法的生成. 接着, 其采用全局分析生成的策略, 将状态等价类空间上所有可能的测试输入序列作为候选输入, 并使用基于规则的评估策略, 选择所有到达新等价类的候选输入生成, 将这些测试输入序列作为新一轮基于遗传算法生成的初代测试输入, 并继续基于遗传算法的生成. 通过两种搜索空间以及候选输入生成评估策略的迭代, AGrippin 能够较为充分地探索状态空间.

TrimDroid [79] 也同样结合了状态等价类空间与状态转移路径空间, 旨在通过分析应用处理不同输入之间的数据依赖, 以探索深层状态. 其首先通过静态分析数据流, 找到同一 Activity 下或存在直接通信的 Activity 对上存在数据依赖的输入处理器, 并记录相应输入对. 接着, TrimDroid 使用静态建立搜索空间的技术建立以相同 Activity 作为等价标准 ρ 的状态等价类空间. 通过全局分析生成的策略, TrimDroid 生成所有缩减空间上的测试输入序列作为候选输入, 并使用启发式随机的策略随机选择一条到达能够处理记录的输入对的状态的候选输入. 接着, TrimDroid 使用状态转移路径空间, 利用组合测试的方式生成所有记录的当前状态可以处理的数据依赖输入对.

最后, Stoat [8] 也结合了状态等价类空间与状态转移路径空间. 首先, 其仿照 DynoDroid 的方式, 采用动态建立搜索空间的方式建立缩减的状态空间, 采用相同图形界面作为等价标准. 其次, Stoat 借助状态转移路径空间, 选择空间中所有可能的状态转移路径生成对应的候选输入. 然后, 其使用启发式随机的评估策略, 随机给状态等价类空间上每一个状态转移 (即一个状态 – 输入 – 状态组合) 一个概率, 并依据此概率随机选择候选输入生成并发送, 采用 MCMC [80] 的方式考虑测试输入序列覆盖的

代码以及状态等价类空间的覆盖, 以决定是否接受这些测试输入序列. 借助 MCMC, Stoat 通过调整概率, 提升了小概率发生的状态转移路径对应的测试输入序列被生成的概率.

可以看出, 现有的使用组合搜索空间的技术中, 较多地选取状态等价类空间与状态转移路径空间进行组合, 以平衡充分性与高效性的权衡. 通过状态等价类空间, 测试输入生成技术可以对状态空间的主体框架进行刻画与认识. 而通过状态转移路径空间, 可以在主体框架的基础上以较细的粒度探索状态空间的其他部分, 这样既可以避免单独使用状态转移路径空间时缺乏足够信息的问题, 又可以应对状态等价类空间粒度过粗导致的大量状态被忽略的问题. 同时, 在小范围使用状态转移路径空间, 可以较好地控制其大小, 从而一定程度上解决使用其作为搜索空间的技术低效的问题.

5 讨论: 现有技术不足与研究契机

本文提出了 Android 应用测试输入自动生成技术的 3 个维度: 搜索空间的表示、候选输入的生成, 以及对这些候选输入的评估. 如 4.4 小节所述, 现有技术都在这 3 个问维度上采用了不同的策略, 且都能一定程度上达到探索状态空间的目的. 然而, 已有的许多综述以及实证研究 [3, 4, 7] 均表明, 现有的技术仍旧存在各种各样的问题. 在 Choudhary 等 [4] 发表的实证研究的对比实验中, 没有一个技术能够在所有测试用例上达到 50% 的代码覆盖率. 本节从描述框架的 3 个维度, 讨论现有技术的不足, 以及可能的研究契机.

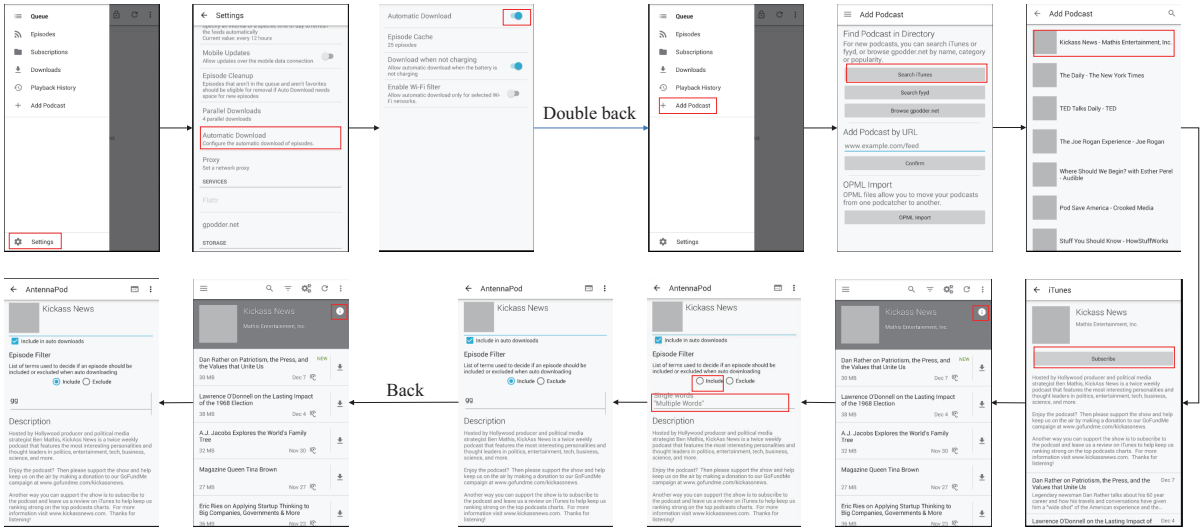
5.1 现有技术不足

从描述框架的 3 个维度分别讨论现有技术的不足.

搜索空间的表示. 如图 2 以及 4.1 小节所述, 常见的搜索空间表示包括状态转移路径空间、原始状态空间、应用代码执行路径空间, 以及缩减状态空间. 测试输入自动生成技术建立这些搜索空间的初衷, 是通过将状态空间 S 向粗粒度或细粒度转化, 以获得更多的已知信息, 或缩减状态空间的大小, 以更好地指导测试输入的生成. 然而现有的搜索空间表示并不能很好地达到这一目的.

从获得更多已知信息, 并减少未知的未知 (即无法预知的未知信息, 如哪种 ε – 输入会在何时发生) 的角度来讲, 原始状态空间显无法提供任何有效的帮助. 状态转移路径空间由原始状态空间向细粒度转化生成, 其额外包含状态转移路径的信息. 然而, 由于 ε – 输入的存在, 状态转移路径中也包含着未知信息, 这进一步加剧了未知性的干扰, 使得测试输入生成技术更难进行准确的决策. 对于应用代码执行路径空间与缩减状态空间, 由于其均为状态空间 S 向粗粒度转化, 其忽略了 S 中的一些未知信息, 如 ε – 输入发生的时机等, 一定程度上能够在指导测试输入生成时减少未知信息的干扰. 然而, 由于其对 S 进行了向粗粒度的转化, 其丢失了很多原本 S 中蕴含的信息, 从而在很多情况下同样很难提供有效的帮助. 图 6 给出了开源应用 AntennaPod[6)] 中一个深层状态的例子. 现有的搜索空间表示均很难为探索这一深层状态提供帮助. AntennaPod 是一个开源的广播收听应用. 其可以在每个节目中设置相应的过滤机制, 以选择节目的某些集进行自动下载. 在每个节目的设置页面可以看到设置的过滤机制, 且必须在总设置中启用自动下载才能设置. 要探索到显示设置好的过滤机制的设置页面的应用运行系统状态 (以执行图 6(b) 所示代码为表征), 需要一个特定的长输入序列: Z (1) 点击进入设置页面, (2) 点击自动下载设置分页面, (3) 启用自动下载, (4) 两次返回到达主页面, (5) 点击添加广播, (6) 使用合法的输入序列打开一个广播的信息界面 (图 6(a) 中给出一个例子), (7) 点击右上角进入设置页面, (8) 设置过滤机制, 以及 (9) 离开设置页面再次进入. 这一测试输入序列具有很长的

6) AntennaPod. https://github.com/AntennaPod/AntennaPod.



(a)

```
1  public class FeedInfoActivity extends AppCompatActivity {
2      ...
3      @Override
4      protected void onCreate(Bundle savedInstanceState) {
5          ...
6          FeedFilter filter = prefs.getFilter();
7          if (filter.includeOnly()) {
8              etxtFilterText.setText(filter.getIncludeFilter());
9              rdoFilterInclude.setChecked(true);
10             rdoFilterExclude.setChecked(false);
11             filterInclude = true;
12         } else if (filter.excludeOnly()) {
13             etxtFilterText.setText(filter.getExcludeFilter());
14             rdoFilterInclude.setChecked(false);
15             rdoFilterExclude.setChecked(true);
16             filterInclude = false;
17         }
18         ...
19     }
20     ...
21 }
```

(b)

图 6 (网络版彩图) AntennaPod 广播设置页面过滤机制相关状态探索图示 (a) 及代码片段 (b)

Figure 6 (Color online) State exploration illustration of feed setting page with filters (a) and code snippet of AntennaPod (b)

长度, 且需要多次探索近似的状态 (如显示主界面及广播设置的状态). 对于状态转移路径空间与原始状态空间, 虽然其包含相关信息, 但测试输入自动生成技术无法从搜索空间中获取这样的信息. 对于缩减状态空间, 由于其将状态合并为等价类, 因而过程中多个状态一般会被合并到等价类中, 如两次到达主界面与广播设置界面的状态, 因而目标深层状态被丢失. 这也是缩减状态空间固有的问题, 深层状态极易被等价类中其他状态覆盖, 从而无法被探索到. 对于应用代码执行路径空间, 其在详细完整执行路径中包含了这一输入序列的信息. 然而, 由于其包含多个循环的输入序列 (如两次进入广播设置界面), 且其具有相对复杂的数据依赖 (如设置中的启用自动下载会影响离其较远的广播设置页面的行为), 提

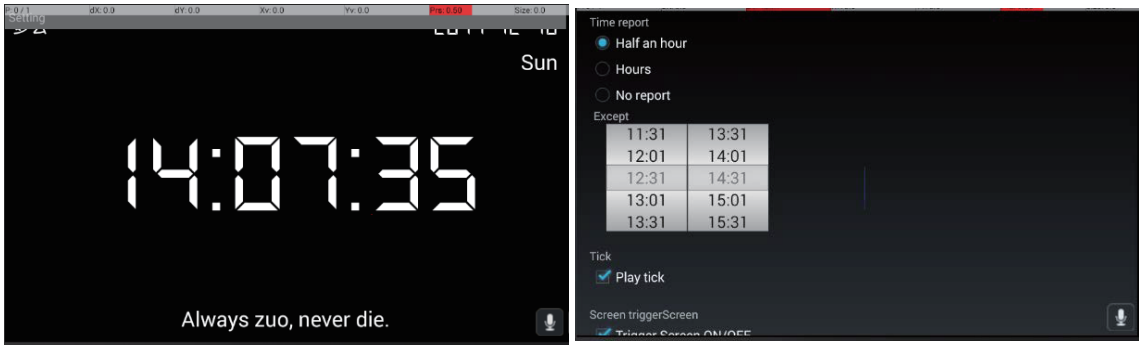


图 7 (网络版彩图) CoolClock 时间输入相关图形界面

Figure 7 (Color online) Time input related GUIs of CoolClock

取这一信息需要复杂且耗时的计算, 并且受制于数据流控制流分析手段, 如常见的重名 (aliasing) 问题与 Android 应用特有的破碎控制流问题.

同时, 为了减少未知的未知干扰, 应用代码执行路径空间以及缩减状态空间忽略了大量相关的状态转移, 如环境 ε – 输入导致的状态转移, 以及多线程执行任务时应用 ε – 输入导致的状态转移等. 这导致根据搜索空间生成的测试输入很难稳定地探索预期探索的状态, 同时也使得回溯状态等操作变得尤为困难. 例如, 许多应用通常在使用过程中随机地出现广告. 这些广告会导致应用界面发生变化. 现有的搜索空间表示无法提供这样的变化出现的时机的信息, 这导致当出现界面变化时, 生成的测试输入无法触发期望触发的处理器, 如在发送点击输入瞬间应用界面变化, 导致点击输入点击到空白区域. 由于搜索空间表示的局限性, 测试输入技术无法有效地应对这样未知的未知的干扰, 从而大大影响了探索状态空间的有效性.

从缩减状态空间大小的角度来讲, 状态转移路径空间与原始状态空间没有任何作用. 对于应用代码执行路径空间与缩减状态空间, 其已经牺牲了较大的充分性以追求高效性, 有效地缩减了状态空间. 然而, 目前的搜索空间仍旧面临空间爆炸的问题 [11, 59], 许多技术仍旧需要专门应对这一问题 [59, 66]. 这说明目前高效性最好的缩减状态空间依旧存在过于低效的问题, 测试输入生成技术仍旧需要更高效的搜索空间表示. 然而缩减状态空间已经丢失了大量的信息, 导致了较差的充分性. 找到真正合适的搜索空间表示, 以达到更好的充分性/高效性权衡, 也是现有技术尚未完全应对的挑战.

候选输入生成. 如图 2 及 4.2 小节描述, 常见的候选输入生成策略包括全部支持输入生成, 全局分析生成, 即时观测生成, 以及现有测试输入修改. 尽管现有技术都提出了许多属于这 4 类的候选输入生成策略, 其仍旧难以满足有效探索状态空间的需求.

首先, 大部分测试输入生成技术都只能支持一小部分测试输入的生成. 如 2.2 小节所述, 测试输入 $i \in I_a \cup I_e$ 为所有可能的外部使用者输入. 不同于一般的软件, 应用及运行环境可以接收大量不同形式的输入. 然而, 大部分现有技术只能支持生成一部分应用上的输入. 特别地, 许多技术只支持一小部分图形界面输入, 如点击事件等. 这极大地影响了状态空间探索的有效性. 图 7 给出了开源应用 CoolClock[7)] 中的一个例子. CoolClock 是一个开源的时钟应用, 其在系统时钟到达半点或整点的时候, 会进行报时. 同时, 可以通过设置, 关闭其半点或整点的报时, 甚至设定固定的关闭时段. 然而, 要探索 CoolClock 报时或进行不报时判断的状态, 需要特定的时钟时间予以配合. 如果现有技术不能生成相应的时钟操作, 则很难在测试的过程中探索到这一状态. 注意, 这里同样涉及到了未知的未知的

7) CoolClock. https://github.com/socoolby/CoolClock.

干扰. 由于时钟会自发地改变时间, 测试输入过程中可能会碰巧出现合适的时间, 探索到相应的状态. 然而由于缺乏确定的时钟操作配合, 其很难被稳定探索到.

虽然如一些技术所称, 一些输入如长按输入能够被比较简单地扩展支持, 许多输入需要比较复杂的手段才能生成. 例如许多应用支持文件的输入, 如 Aard2[8], 一个开源词典应用, 需要通过文件输入提供一个特定的词典, 才能正常使用其大部分功能. 生成一个合适的文件输入是非常困难的, 目前只有很少的技术涉及这一问题 [48].

同时, 现有技术的候选输入生成策略自身也很难满足有效探索状态空间的目标. 全部支持输入生成会生成一个极大的候选输入集合, 导致后续的测试输入评估没有足够的测试资源有效地评估这些候选输入, 从而只能选择均匀随机等简单省时的策略, 很大程度上影响了最终生成的测试输入的有效性. 同时, 由于现有技术支持生成的输入很少, 其依旧不具有足够好的充分性. 全局分析生成对搜索空间的依赖较为严重, 由于现有技术搜索空间表示存在局限性, 其所能分析得到的信息有限. 同时, 全局分析生成需要较长的分析时间 (如约束求解), 其过低的效率导致了其不能在有限时间内全面地考虑所有信息, 从而影响了其充分性. 如图 6 中的例子, 由于其存在较多输入且跨度较大, 全局分析很难有足够的时间在搜索空间中挖掘出探索这一状态的测试输入序列作为候选输入. 即时观测生成只能考虑当前状态临近的状态空间以及搜索空间, 因而难以用于生成较长的测试输入序列, 这导致需要较多的循环才能有效探索状态空间. 同时, 现有技术没有或无法准确观测应用运行系统完整的状态信息, 只收集一部分信息, 这也影响了根据观测结果生成的候选输入的充分性. 最后, 现有测试输入修改很大程度上依赖于现有测试输入的质量, 这使得初始的测试输入变得尤为重要. 然而, 现有技术很少关注于初始测试输入的获取. 另外, 如何修改已有测试输入, 以在保留其探索状态空间能力的同时探索到未探索过的状态空间, 也很大程度上决定了候选输入的质量. 现有技术一般只采用简单的修改策略 [26, 31], 这也影响了最终的有效性.

最后, 现有技术的候选输入生成策略也没有考虑未知的未知的干扰. 特别地, 其没有考虑 ε – 输入的影响. 如即时观测生成策略, 当测试输入自动生成技术根据观测的应用系统状态生成候选输入时, 应用系统可能已经由于 ε – 输入的影响发生了状态的改变, 从而可能导致生成的候选输入无效.

候选输入评估. 如图 2 及 4.3 小节所述, 现有技术的候选输入策略包括基于规则评估、黑盒评估、启发式随机评估, 以及均匀随机评估. 测试输入自动生成技术对候选输入进行评估的目的, 是预测候选输入对于探索状态空间有效性的提升, 即收益. 因而候选输入评估应尽可能准确地做出判断, 从而能够最有效地探索状态空间. 然而, 现有的候选输入评估手段一般难以达到这一目标. 现有技术的候选输入评估策略不能深入地分析候选输入, 导致错误地预测候选输入的收益. 仍以 AntennaPod 为例, 如图 8 代码所示, 当其正在播放广播时, 若有电话打入, 则其会根据设置暂停播放或降低声音. 在未播放广播时, 来电并不会影响其行为. 对于来电这一候选输入的评估, 需要根据应用系统的当前状态, 准确地做出判断. 现有技术的候选输入评估策略尚不能在这一情况下做出准确的判断. 这是由于现有技术的候选输入评估策略仍较为简单, 没有足够深入地分析应用系统的当前状态以及搜索空间的信息.

均匀随机评估完全没有考虑对收益进行预测. 其完全随机的特性, 使得使用这一策略的测试输入生成技术没有有效性保证. 其次, 启发式随机的评估策略同样受制于随机带来的不稳定性. 其尝试使用一些启发式规则控制随机的影响, 但现有技术的规则一般较为简单, 且不够全面, 因而难以准确预测收益.

黑盒评估的初衷是借助人类外部使用者探索状态空间的经验, 对候选输入进行评估. 使用黑盒评估策略的技术有些需要人类使用者提供的输入序列作为训练样本, 通过预先的训练获得黑盒函数. 这

8) Aard2 for Android. https://github.com/itkach/aard2-android.

```
public class LocalPSMP extends PlaybackServiceMediaPlayer{
    private final AudioManager.OnAudioFocusChangeListener audioFocusCha ngeListener = new AudioManager.
        OnAudioFocusChangeListener() {
        @Override
        public void onAudioFocusChange(final int focusChange) {
            ...
            else if (focusChange == AudioManager.AUDIOFOCUS_LOSS_TRANSIENT _CAN_DUCK) {
                if (playerStatus == PlayerStatus.PLAYING) {
                    if (!UserPreferences.shouldPauseForFocusLoss()) {
                        Log.d(TAG, "Lost audio focus temporarily. Ducking...");
                        final float DUCK_FACTOR = 0.25f;
                        setVolumeSync(DUCK_FACTOR * UserPreferences.getLeftVolume(),
                        DUCK_FACTOR * UserPreferences.getRightVolume());
                        pausedBecauseOfTransientAudiofocusLoss = false;
                    } else {
                        Log.d(TAG, "Lost audio focus temporarily. Could duck, but won't, pausing...");
                        pause(false, false);
                        pausedBecauseOfTransientAudiofocusLoss = true;
                    }
                }
            }
        }
        ...
    }
    ...
}
```

图 8　(网络版彩图) AntennaPod 来电输入相关代码片段

Figure 8　(Color online) Phone-call handling code snippet of AntennaPod

些训练样本的质量对训练出的黑盒预测的准确性有极大的影响. 然而, 在缺乏有效的提示与指导下, 人类使用者很难提供全面的训练样本, 反而往往关注于常使用的部分, 因而对于探索不常探索到的状态反而起到了反向的作用. 同时, 训练样本往往来自人类使用者使用其他应用时产生的测试输入, 由于不同 Android 应用之间的使用模式差别较大, 训练样本中能够普适使用的经验较少, 因而难以提供全面准确的评估. 同时, 有些技术的黑盒评估策略采用线上增强学习的形式, 通过即时反馈训练. 这样的策略需要较长的时间才能够得到较为准确的预测, 同时需要花费较多测试时间进行训练, 因而高效性较差.

基于规则的评估策略通常以遍历优先或适宜性最高优先作为规则进行预测. 这些规则受制于搜索空间的局限性, 往往不能准确地预测发送候选输入的收益. 同样以图 6 为例. 对于探索这一深层状态的测试输入序列, 由于其包含重复到达广播设置页面, 现有技术的遍历优先规则不会预测其能够探索新的状态, 因而不会选取其生成. 而对于适宜性最高规则, 由于其无法在候选输入被真实发送到应用系统前准确获知其收益性, 往往需要通过对探索历史中与候选输入相关的测试输入进行评估的方式预测候选输入的适宜性, 如遗传算法为基础的技术. 然而, 候选输入的适宜性与这些已有的测试输入的适宜性是否具有很强的相关性是无法准确判定的. 由于 Android 应用事件驱动的机制, 测试输入简单的差异往往就会导致应用运行系统完全不同的状态转移, 使得之后的测试输入即使相同也可能会探索到完全不同的状态. 同时, 由于 ε – 输入的存在, 测试输入的状态探索效果存在不确定性, 这导致已有测试输入的适宜性具有不稳定性, 这也会导致错误地预测候选输入的适宜性.

最后, 现有技术的候选输入评估策略也没有考虑未知的未知的干扰. 由于 ε – 输入的影响, 应用

运行系统的状态在不断自发地变化因而候选输入探索状态空间的收益也在不断变化. 这为候选输入的评估增加了新的不确定性. 同样以图 6 为例, 由于 ε – 输入的影响, AntennaPod 会逐渐播放完成一集广播, 之后会停止播放. 因而在其播放时预测的来电输入的收益, 在实际生成发送时可能会大幅降低, 从而影响了最终的有效性.

5.2 研究契机

针对 5.1 小节提出的现有技术的不足, 本小节将一些可能的发展方向与研究契机加以展望.

搜索空间表示的融合. 如 5.1 小节所述, 现有技术的搜索空间表示都有其不足之处, 急需具有更好充分性与高效性的搜索空间表示. 而从另一个方面讲, 现有的搜索空间表示在探索特定部分的状态空间时, 都具有一定的长处. 因而通过融合搜索空间的表示, 博取现有搜索空间表示的长处, 可以促进搜索空间表示向更有效的方向发展. 如 4.4.5 小节所述, 已有一些技术组合地使用不同搜索空间的表示 [8,76~79], 也都具有较好的有效性. 这些技术一般都选择缩减状态空间与状态转移路径空间结合, 通过最倾向于充分性的搜索空间表示与最倾向于高效性的搜索空间表示进行组合, 以期得到较好的充分性/高效性权衡. 缩减状态空间能够一定程度上刻画状态空间 S 的主体框架, 且将状态进行了等价分类, 比较适宜在其基础上进行扩展. 然而, 状态转移路径空间所能提供的信息有限, 因而影响了组合的状态空间的效果. 与之相比, 应用代码执行路径空间可以提供更多的信息, 因而也可以更好地在缩减状态空间的基础上进行扩展. 例如, 可以在缩减状态空间的每个等价类上, 建立该等价类内状态空间的应用代码执行路径空间. 由于缩减状态空间的状态等价类内部状态较为相似, 如具有相同的当前活跃 Activity, 应用代码执行路径空间可以只关注于某个特定应用组件的执行路径, 如仅建立活跃 Activity 的代码执行路径空间. 这样即扩展了缩减状态空间, 避免了大量深层状态被等价类内其他状态覆盖的问题, 又细化分割了应用代码执行路径空间, 帮助其解决了破碎控制流, 空间过大分析耗时高效性较低的问题.

同时, 同一类的搜索空间表示也可以进行融合. 如缩减状态空间, 根据不同等价标准建立的缩减状态空间在充分性/高效性权衡上会具有完全不同的效果 [23]. 因而在探索不同的状态空间部分时, 可以根据状态空间探索的情况, 动态决策使用不同的等价标准, 建立融合的缩减状态空间. 现有一些技术如 AimDroid [66] 等, 已经使用了混合标准的思路. 然而其依旧在固定的状态空间部分使用预设的等价标准, 没有考虑实际的状态空间探索情况.

环境输入生成与评估. 如 5.1 小节所述, 很多环境输入现有技术都不能支持生成, 这严重影响了生成的测试输入的有效性. 探究如何生成及评估这些环境输入, 是当前亟需的研究方向. 由于 Android 平台的特性, 环境输入的形式多种多样, 如重力感应输入、加速输入、传感器输入、文件输入等. 这些环境输入由于其机制的复杂性, 生成其需要额外的考虑.

首先, 具体需要的环境输入需要依据被测应用的特性进行判定. 如文件输入, 其包含可能的文件种类多种多样, 如音频文件、文本文件、数据库文件等. 不同的应用需要不同类型的文件输入, 如 Aard2 需要特定格式的字典数据库文件输入, AntennaPod 接收一组不同格式的音频或视频文件输入等. 如何自动确定所需的文件输入是自动生成文件输入的一大难点.

另外, 如何生成能够探索不同状态的环境输入, 也是需要考虑的重要因素. 不同的环境输入能够探索到应用系统的不同状态, 如不同的传感器输入能够触发应用处理器不同的处理逻辑. 有些类型的环境输入, 如旋转手机输入能够比较简单地判定其所能探索的状态, 而有些类型的环境输入则比较困难. 如 CoolClock 可以设置定时事件在整点时进行报时, 其设定报时时间是在运行中动态设定, 即根据当前时间推算下一次报时的时间, 并设定闹钟. 测试输入自动生成技术要理解这一逻辑, 确定所需的时

间输入, 需要对应用进行精确的数据流与控制流分析. 如果能够准确获得这一类的信息, 并生成相应的环境输入, 能够极大提高探索状态空间的有效性.

最后, 在评估环境输入时, 需要考虑其合法性与异常性. Android 应用运行系统与真实世界的交互更为紧密, 如传感器输入是感知真实世界所形成的输入. 由于真实世界存在一些特定的规律, 相应的环境输入也应当遵循这些规律, 如海拔传感器输入的变化应当考虑真实可能的情况. 然而, 由于 Android 系统感知真实世界的硬件存在偏差, 也可能存在不遵循真实世界规则的数据, 如海拔传感器输入发生跳变. 但是在正常的 Android 系统中, 这一类输入不会是高频发生的输入. 测试输入自动生成技术应当慎重考虑这两种输入的关系, 在评估时考虑在真实使用场景中可能出现的环境输入, 而不是仅考虑状态空间的探索而生成不切实际的输入, 如仅选择完全正常的, 或完全无规律跳变的海拔传感器输入.

ε – 输入的建模与控制. 应用运行系统的执行模型 M 中, 未知的未知, 尤其是 ε – 输入, 极大地影响了测试输入探索状态空间的有效性. 现有的测试输入只考虑外部使用者的输入, 即 $I_a \cup I_e$. 并且, 现有技术少有考虑 ε – 输入的影响. 对 ε – 输入进行建模与控制, 可以将测试输入的范围扩展到整个输入集合 I. 此时应用运行系统会完全可控, 则可以更有效地探索状态空间.

现有技术已有一些建模与控制 ε – 输入的尝试. 对于应用上的 ε – 输入, 一般技术在发送新的测试输入前, 都会等待一段时间, 以使处理上一个输入的 ε – 输入发生完毕. 这样, 其控制了处理两个输入的 ε – 输入具有严格的顺序. 同时, 对于一些关注于应用并发错误的技术 [10, 58], 其会通过插装应用的方式, 控制并发执行时共享资源读写 ε – 输入的执行顺序. 另一方面, 对于环境上的 ε – 输入, 现有技术也通过诸如等待的方法尝试控制 ε – 输入.

然而, 现在仍然缺乏对 ε – 输入明确的建模与控制. 首先, 对于不同的被测应用, 同一 ε – 输入会对其造成不同的影响. 如 CoolClock 会受到时钟 ε – 输入的影响, 而 Aard2 则完全不会. 同时, 应用运行系统在不同的状态发生相同的 ε – 输入也会导致其不同的状态转移. 因而, 测试输入自动生成技术需要对 ε – 输入进行更深入的分析, 以获得针对被测应用的 ε – 输入模型. 其次, ε – 输入的控制很难在合理的资源开支下做到精确. 对于应用上的 ε – 输入, 精确的控制需要对每一行指令的执行都进行控制, 会带来很大的额外开销. 对于环境上的 ε – 输入, 由于其始终在进行自发的 ε – 输入, 需要连续的观测与控制, 或者直接修改 Andorid 系统甚至内核, 这也是一个艰巨的挑战.

大数据支持的候选输入评估. 如 5.1 小节所述, 现有的候选输入评估策略很难做出准确的收益预测. 其中较大一部分原因是这些评估策略受制于制定者或训练样本有限的数据与经验. 对于基于规则的评估与启发式随机评估, 其均由研究人员根据自身的经验设定. 而对于黑盒评估, 现有技术一般使用较少的训练样本进行训练. 现如今, 对于大数据的支持与信息挖掘的技术已经十分成熟, 从大量的 Android 应用测试记录中, 挖掘出更丰富的经验, 以制定评估的规则, 或进行黑盒函数的训练, 都能够有效地提升评估的准确性.

目前已经有一些技术涉及大数据的评估. 许多技术, 如 POLARIZ [28], 提出了众包的测试平台, 通过收集大量的人类用户提供的数据并进行分析, 获取普适的经验. 一些技术如 Caiipa [70] 则通过大量应用的自动测试输入记录, 不借助人类地分析探索历史, 以获取更优地评估策略. 如何更好地利用这两方面经验, 以提升评估策略的准确性, 也是值得关注的方向.

人类指导的测试输入自动生成. 现有技术如 5.1 小节所述, 仍旧存在很多的不足. 上述研究方向一定程度上都能够应对这些不足, 但从目前来看, 较短时间内难以有较大的突破. 然而, 人类往往可以相对容易地应对这些问题. 人类, 尤其是专业测试人员, 能够较快较全面地理解应用的意图与逻辑, 从而快速建立足够精确的搜索空间. 其次, 人类能够更全面地了解应用运行系统的状态, 从而生成能够更有效探索状态空间的测试输入. 同时, 人类能够比较好地理解 ε – 输入的机制. 最后, 人类能够生成

大部分测试输入.

因而, 人类通过参与测试输入自动生成的过程, 可以对搜索空间的表示, 测试输入的生成与评估都起到指导的作用. 目前已有一些技术采用了类似的方法. Griebe 等 [51] 在 2014 年提出的方法要求给定 Activity 的功能 UML 图作为输入, 这实际上通过人类的指导建立了搜索空间. UGA [81] 则首先让人类生成测试输入探索状态空间, 通过人类的测试输入序列建立状态空间的主体框架, 然后再借助测试输入自动生成工具根据主体框架进一步探索状态空间. 这类似于前文提及的缩减状态空间与其他搜索空间的融合, 但使用人类代替了缩减状态空间, 具有更好的有效性.

然而, 现有技术使用人类的方式还比较简陋. 一方面, 现有技术并没有为人类足够的信息帮助, 这导致人类只能盲目地探索状态空间. 这会导致人类由于缺乏指导, 可能不能有效地探索状态空间, 从而导致不能为自动过程提供有效的指导. 如图 6 中的例子, 人类同样比较难以找到这样的测试输入序列. 同时, 由于缺乏有效信息, 人类无法得知测试输入自动生成技术的不足之处, 因而可能浪费宝贵的人力资源探索到自动技术也能探索到的状态, 并且无法为自动技术提供额外的指导. 另一方面, 现有技术使用人类提供的测试输入的方式还较为简单, 通常为从中挖掘常见的使用模式或直接使用. 这样大量的相关信息被丢失, 如测试输入序列中每个输入之间的关系, 使用模式的应用场景等, 导致许多有用的信息无法被利用. 基于此, 如何有效地提供信息指导用户花费较少的资源提供有价值的测试输入序列, 以及如何利用这些测试输入序列, 是这一方向值得关注的问题.

6 总结

本文讨论了 Android 应用测试输入自动生成技术的描述框架, 包括 3 个维度 (分别是搜索空间的表示、候选输入的生成, 以及候选输入的评估) 与两个评价指标 (即充分性与高效性). 基于这一描述框架, 本文分析总结了现有技术, 理解现有技术采用的策略以及在评价指标上的权衡. 此外, 基于描述框架与分析总结, 本文讨论了现有技术的不足, 指出了未来发展的可能方向, 也为未来工作的开展奠定了基础. 希望本文能进一步促进 Android 应用测试输入自动生成技术的研究发展, 为自动化的 Android 应用质量保障提供坚实的基础.

致谢 感谢江苏省软件新技术与产业化协同创新中心.

参考文献

1 Xu J F. Software automation. J Comput Res Develop, 1994, 11: 9–15
2 Yao L, Shu Y A. Research on key techniques for software automatic testing. J Anhui Univ Natl Sci Edit, 2003, 27: 14
3 Rubinov K, Baresi L. What are we missing when testing our Android apps? Computer, 2018, 51: 60–68
4 Choudhary S R, Gorla A, Orso A. Automated test input generation for Android: are we there yet? In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, 2015. 429–440
5 Sahinoglu M, Incki K, Aktas M S. Mobile application verification: a systematic mapping study. In: Proceedings of International Conference on Computational Science and its Applications, Banff, 2015. 147–163
6 Zein S, Salleh N, Grundy J. A systematic mapping study of mobile application testing techniques. J Syst Softw, 2016, 117: 334–356
7 Kong P F, Li L, Gao J, et al. Automated testing of Android apps: a systematic literature review. IEEE Trans Rel, 2019, 68: 45–66
8 Su T, Meng G Z, Chen Y T, et al. Guided, stochastic model-based GUI testing of Android apps. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017), New York, 2017. 245–256

9 Li Q W, Jiang Y Y, Gu T X, et al. Effectively manifesting concurrency bugs in Android apps. In: Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC), Hamilton, 2016. 209–216

10 Wang J, Jiang Y Y, Xu C, et al. AATT+: effectively manifesting concurrency bugs in Android apps. Sci Comput Program, 2018, 163: 1–18

11 Bai G D, Ye Q Q, Wu Y Z, et al. Towards model checking Android applications. IEEE Trans Softw Eng, 2018, 44: 595–612

12 Gao X, Tan X H, Dong Z, et al. Android testing via synthetic symbolic execution. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018), New York, 2018. 419–429

13 Yang G L, Huang J, Gu G F. Automated generation of event-oriented exploits in Android hybrid apps. In: Proceedings of Network and Distributed System Security Symposium, San Diego, 2018

14 Mirzaei N, Bagheri H, Mahmood R, et al. SIG-Droid: automated system input generation for Android applications. In: Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE), Gaithersbury, 2015. 461–471

15 Fan L L, Su T, Chen S, et al. Efficiently manifesting asynchronous programming errors in Android apps. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018), New York, 2018. 486–497

16 Jensen C S, Prasad M R, Møller A. Automated testing with targeted event sequence generation. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2013), New York, 2013. 67–77

17 Anand S, Naik M, Harrold M J, et al. Automated concolic testing of smartphone apps. In: Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12), New York, 2012

18 Liu Y P, Xu C, Cheung S C, et al. GreenDroid: automated diagnosis of energy inefficiency for smartphone applications. IEEE Trans Softw Eng, 2014, 40: 911–940

19 Wang J, Liu Y P, Xu C, et al. E-greenDroid: effective energy inefficiency analysis for Android applications. In: Proceedings of the 8th Asia-Pacific Symposium on Internetware (Internetware'16), Beijing, 2016. 71–80

20 Mirzaei N, Malek S, Păsăreanu C S, et al. Testing Android apps through symbolic execution. SIGSOFT Softw Eng Notes, 2012, 37: 1–5

21 Liu Y, Wang J, Xu C, et al. NavyDroid: detecting energy inefficiency problems for smartphone applications. In: Proceedings of the 9th Asia-Pacific Symposium on Internetware (Internetware'17), Shanghai, 2017

22 Zhang L L, Liang C M, Liu U X, et al. Systematically testing background services of mobile apps. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017), Piscataway, 2017. 4–15

23 Baek Y M, Bae D H. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016), New York, 2016. 238–249

24 Choi W, Necula G, Sen K. Guided GUI testing of Android apps with minimal restart and approximate learning. In: Proceedings of ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13), New York, 2013. 623–640

25 Wu X Y, Jiang Y Y, Xu C, et al. Testing Android apps via guided gesture event generation. In: Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC), Hamilton, 2016. 201–208

26 Mao K, Harman M, Jia Y. Sapienz: multi-objective automated testing for Android applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016), New York, 2016. 94–105

27 Mahmood R, Mirzaei N, Malek S. EvoDroid: segmented evolutionary testing of Android apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), New York, 2014. 599–609

28 Mao K, Harman M, Jia Y. Crowd intelligence enhances automated mobile testing. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017), Piscataway, 2017. 16–26

29 Sadeghi A, Jabbarvand R, Malek S. PATDroid: permission-aware GUI testing of Android. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017), New York, 2017. 220–232

30 Koroglu Y, Sen A. TCM: test case mutation to improve crash detection in Android. In: Proceedings of International Conference on Fundamental Approaches to Software Engineering, Thessaloniki, 2018. 264–280

31 Adamsen C Q, Mezzetti G, Møller A. Systematic execution of Android test suites in adverse conditions. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2015), New York, 2015. 83–93

32 Wegener J. Evolutionary testing techniques. In: Proceedings of Stochastic Algorithms: Foundations and Applications, Berlin, 2005. 82–94

33 Fudenberg D, Tirole J. Game Theory. Cambridge: MIT Press, 1991

34 Stormo G D. DNA binding sites: representation and discovery. Bioinformatics, 2000, 16: 16–23

35 Mahmood R, Esfahani N, Kacem T, et al. A whitebox approach for automated security testing of Android applications on the cloud. In: Proceedings of the 7th International Workshop on Automation of Software Test (AST), Zurich, 2017. 22–28

36 Liu P, Zhang X Y, Pistoia M, et al. Automatic text input generation for mobile testing. In: Proceedings of the 39th International Conference on Software Engineering (ICSE'17), Piscataway, 2017. 643–653

37 Schmidhuber J. Deep learning in neural networks: an overview. Neural Netw, 2015, 61: 85–117

38 Goldberg Y, Levy O. word2vec explained: deriving Mikolov et al.'s negative-sampling word-embedding method. 2014. ArXiv:1402.3722

39 Linares-Vásquez M, White M, Bernal-Cárdenas C, et al. Mining Android app usages for generating actionable GUI-based execution scenarios. In: Proceedings of the 12th Working Conference on Mining Software Repositories (MSR'15), Piscataway, 2015. 111–122

40 Amalfitano D, Fasolino A R, Tramontana P, et al. Considering context events in event-based testing of mobile applications. In: Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops, Luxembourg, 2013. 126–133

41 Cao C, Meng C L, Ge H J, et al. Xdroid: testing Android apps with dependency injection. In: Proceedings of the 41st Annual Computer Software and Applications Conference (COMPSAC), Turin, 2017. 214–223

42 Shahriar H, North S, Mawangi E. Testing of memory leak in Android applications. In: Proceedings of the 15th International Symposium on High-Assurance Systems Engineering, Miami Beach, 2014. 176–183

43 Ye H, Cheng S Y, Zhang L B, et al. DroidFuzzer: fuzzing the Android apps with intent-filter tag. In: Proceedings of International Conference on Advances in Mobile Computing & Multimedia (MoMM'13), New York, 2013

44 Clarke L A. A program testing system. In: Proceedings of Annual Conference, Houston, 1976. 488–491

45 Lam W K. Hardware Design Verification: Simulation and Formal Method-Based Approaches. Upper Saddle River: Prentice Hall, 2008

46 Liu Y, Wang J, Xu C, et al. NavyDroid: an efficient tool of energy inefficiency problem diagnosis for Android applications. Sci China Inf Sci, 2018, 61: 050103

47 Anand S, Naik M, Harrold M J, et al. Automated concolic testing of smartphone apps. In: Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12), New York, 2012

48 Yeh C C, Lu H L, Chen C Y, et al. CRAXDroid: automatic Android system testing by selective symbolic execution. In: Proceedings of the 8th International Conference on Software Security and Reliability-Companion, San Francisco, 2014. 140–148

49 Zhang L L, Liang C M, Liu Y X, et al. Systematically testing background services of mobile apps. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017), Piscataway, 2017. 4–15

50 Keng J C J, Jiang L X, Wee Y K, et al. Graph-aided directed testing of Android applications for checking runtime privacy behaviours. In: Proceedings of the 11th International Workshop on Automation of Software Test (AST'16), New York, 2016. 57–63

51 Griebe T, Gruhn V. A model-based approach to test automation for context-aware mobile applications. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14), New York, 2014. 420–427

52 Peterson J L. Petri nets. ACM Comput Surv, 1977, 9: 223–252

53 Yang K, Zhuge J W, Wang Y K, et al. IntentFuzzer: detecting capability leaks of Android applications. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIA CCS'14), New York, 2014. 531–536

54 Ma J, Liu S C, Jiang Y Y, et al. LESdroid: a tool for detecting exported service leaks of Android applications. In: Proceedings of the 26th Conference on Program Comprehension (ICPC'18), New York, 2018. 244–254

55 Amalfitano D, Fasolino A R, Tramontana P. A GUI crawling-based technique for Android mobile application testing. In: Proceedings of the 4th International Conference on Software Testing, Verification and Validation Workshops, Berlin, 2011. 252–261
56 Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of Android apps. SIGPLAN Not, 2013, 48: 641–660
57 Song W, Qian X X, Huang J. EHBDroid: beyond GUI testing for Android applications. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering, 2017. 27–37
58 Tang H Y, Wu G Q, Wei J, et al. Generating test cases to expose concurrency bugs in Android applications. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering, Singapore, 2016. 648–653
59 Amalfitano D, Fasolino A R, Tramontana P, et al. MobiGUITAR – a tool for automated model-based testing of mobile apps. IEEE Softw, 2015, 32: 53–59
60 Moran K, Linares-Vásquez M, Bernal-Cárdenas C, et al. Automatically discovering, reporting and reproducing Android application crashes. In: Proceedings of IEEE International Conference on Software Testing, Verification and Validation, Kyoto, 2016. 33–44
61 Dong F, Wang H Y, Li L, et al. FraudDroid: automated ad fraud detection for Android apps. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, 2018. 257–268
62 Arnatovich Y L, Ngo M N, Tan H B K, et al. Achieving high code coverage in Android UI testing via automated widget exercising. In: Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC), Hamilton, 2017. 193–200
63 Adamo D, Nurmuradov D, Piparia S, et al. Combinatorial-based event sequence testing of Android applications. Inf Softw Tech, 2018, 99: 98–117
64 Koroglu Y, Sen A, Muslu O, et al. QBE: qlearning-based exploration of Android applications. In: Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST), Vasteras, 2018. 105–115
65 Watkins C J C H. Learning from delayed rewards. Dissertation for Ph.D. Degree. Cambridge: Cambridge University, 1989
66 Gu T X, Cao C, Liu T C, et al. AimDroid: activity-insulated multi-level automated testing for Android applications. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution, Shanghai, 2017. 103–114
67 Machiry A, Tahiliani R, Naik M. Dynodroid: an input generation system for Android apps. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013), New York, 2013. 224–234
68 Hao S, Liu B, Nath S, et al. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In: Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'14), New York, 2014. 204–217
69 Ma J, Liu S, Yue S T, et al. LeakDAF: an automated tool for detecting leaked activities and fragments of Android applications. In: Proceedings of the 41st Annual Computer Software and Applications Conference (COMPSAC), Turin, 2017. 23–32
70 Liang C M, Lane N D, Brouwers N, et al. Caiipa: automated large-scale mobile app testing through contextual fuzzing. In: Proceedings of the 20th Annual International Conference on Mobile Computing and Networking (MobiCom'14), New York, 2014. 519–530
71 Zhu H W, Ye X J, Zhang X J, et al. A context-aware approach for dynamic GUI testing of Android applications. In: Proceedings of the 39th Annual Computer Software and Applications Conference, Taichung, 2015. 248–253
72 Rosenfeld A, Kardashov O, Zang O. Automation of Android applications testing using machine learning activities classification. In: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, Montréal, 2018. 122–132
73 Yang W, Prasad M R, Xie T. A grey-box approach for automated GUI-model generation of mobile applications. In: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE'13), Berlin, 2013. 250–265
74 Chen J, Han G, Guo S Q, et al. FragDroid: automated user interface interaction with activity and fragment analysis in Android applications. In: Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable

Systems and Networks (DSN), Luxembourg City, 2018. 398–409
75 Wu H W, Wang Y, Rountev A. Sentinel: generating GUI tests for Android sensor leaks. In: Proceedings of the 13th International Workshop on Automation of Software Test (AST'18), New York, 2018. 27–33
76 Pradel M, Schuh P, Necula G, et al. EventBreak: analyzing the responsiveness of user interfaces through performance-guided test generation. In: Proceedings of ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14), New York, 2014. 33–47
77 Banerjee A, Chong L K, Chattopadhyay S, et al. Detecting energy bugs and hotspots in mobile apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), New York, 2014. 588–598
78 Amalfitano D, Amatucci N, Fasolino A R, et al. AGRippin: a novel search based testing technique for Android applications. In: Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile (DeMobile 2015), New York, 2015. 5–12
79 Mirzaei N, Garcia J, Bagheri H, et al. Reducing combinatorics in GUI testing of Android applications. In: Proceedings of the 38th International Conference on Software Engineering (ICSE'16), New York, 2016. 559–570
80 Jespersen N S. An introduction to Markov chain Monte Carlo. Soc Sci Electron, 2010, 11: 395–402
81 Li X J, Jiang Y Y, Liu Y P, et al. User guided automation for testing mobile apps. In: Proceedings of the 21st Asia-Pacific Software Engineering Conference, Jeju, 2014. 27–34

Automatic test-input generation for Android applications

Jue WANG[1,2], Yanyan JIANG[1,2], Chang XU[1,2*], Xiaoxing MA[1,2] & Jian LÜ[1,2]

1. *State Key Lab for Novel Software Technology, Nanjing University, Nanjing* 210023*, China*;
2. *Department of Computer Science and Technology, Nanjing University, Nanjing* 210023*, China*
* Corresponding author. E-mail: changxu@nju.edu.cn

Abstract Automatic test-input generation is an important and expensive testing activity that greatly impacts the effectiveness of automatic testing. There are unique challenges to automatically generating test inputs for Android applications (apps) due to the unique mechanism of the Android platform. Therefore, numerous automatic test-input generation methods for Android apps have been proposed. This study proposes a description framework to demonstrate the key issues in automatic test-input generation and includes three dimensions to describe the technique (representation of search space and the generation and evaluation of candidate test inputs) and two performance metrics for the dimensions (thoroughness and efficiency). Furthermore, existing techniques, as well as potential future work, are discussed.

Keywords Android, automated testing, input generation, description of technology, smartphones

Jue WANG was born in 1993. He received his B.Sc. degree in Department of Computer Science and Technology from Nanjing University, Nanjing, China, in 2016. He is currently a research postgraduate student at the State Key Laboratory for Novel Software Technology pursuing his Ph.D. degree in computer science and technology at Nanjing University. His research interests include but are not limited to Android app testing and analysis. His current work focuses on automatically generating high-quality test inputs for Android apps.

Chang XU was born in 1977. He received his Ph.D. degree in computer science and engineering from the Hong Kong University of Science and Technology and is now a full professor at the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology, Nanjing University. His research interests include big-data software engineering, intelligent software testing and analysis, and adaptive and autonomous software systems.

Yanyan JIANG was born in 1988. After obtaining a Ph.D. degree from Nanjing University in 2017, he joined the Department of Computer Science and Technology, Nanjing University, as an assistant researcher. His research interests include software testing and analysis, run-time systems, and program synthesis.

Xiaoxing MA was born in 1975. He is a professor and the deputy director of the Institute of Computer Software at Nanjing University. He received his Ph.D. degree from Nanjing University in 2003. His research interests include self-adaptive software systems, software architectures, and middleware systems.