Beijing Forest Studio
北京理工大学信息系统及安全对抗实验中心

Android自定义权限及其设计缺陷

硕士研究生　陆永鑫

2022年04月17日

内容提要

- 背景简介
- 基本概念
- 算法原理
- 应用总结
- 参考文献

背景简介

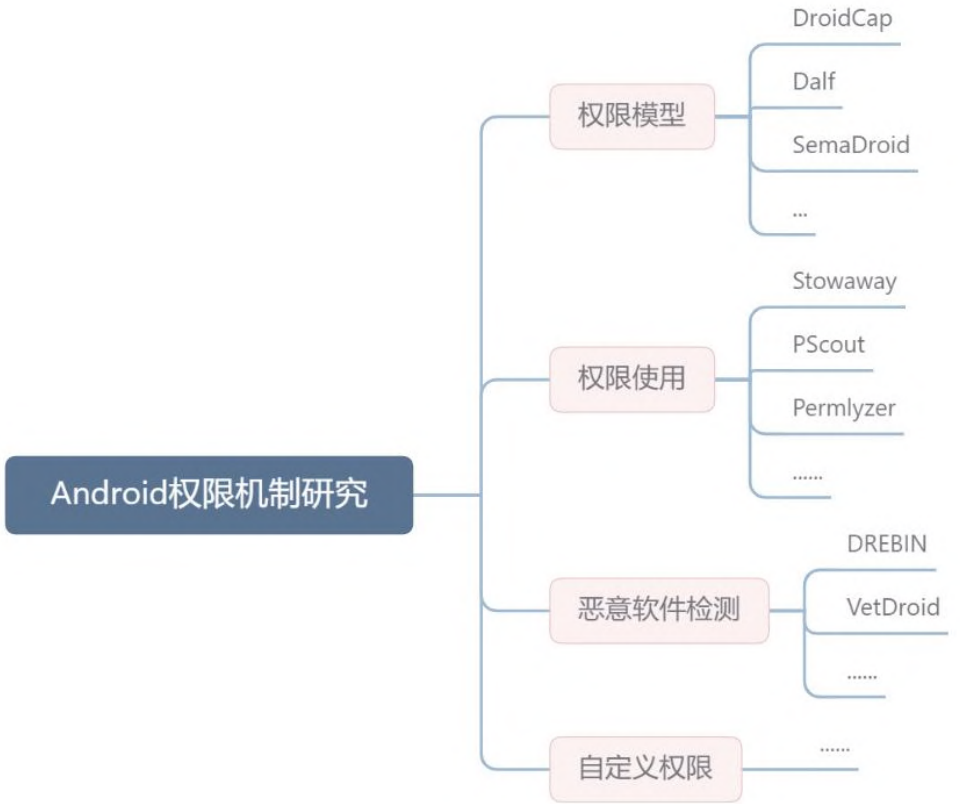
- 预期收获
  - 1. 了解Android权限机制研究历史和现状
  - 2. 了解权限机制及自定义权限基本概念
  - 3. 理解发现权限升级漏洞的模糊测试及分析方法
  - 4. 思考权限机制设计缺陷及缓解措施

背景简介

- 权限机制是Android安全的基础
  - 作为最流行的开源移动开发平台，Android提供了丰富的API和特性来支持第三方APP的开发。
  - 出于安全考虑，Android设计了一些机制来防止恶意行为，而其中**权限机制**就是基础之一。
  - 任何APP必须要拥有**特定权限**才能访问相应的**用户敏感数据**和**系统资源**。

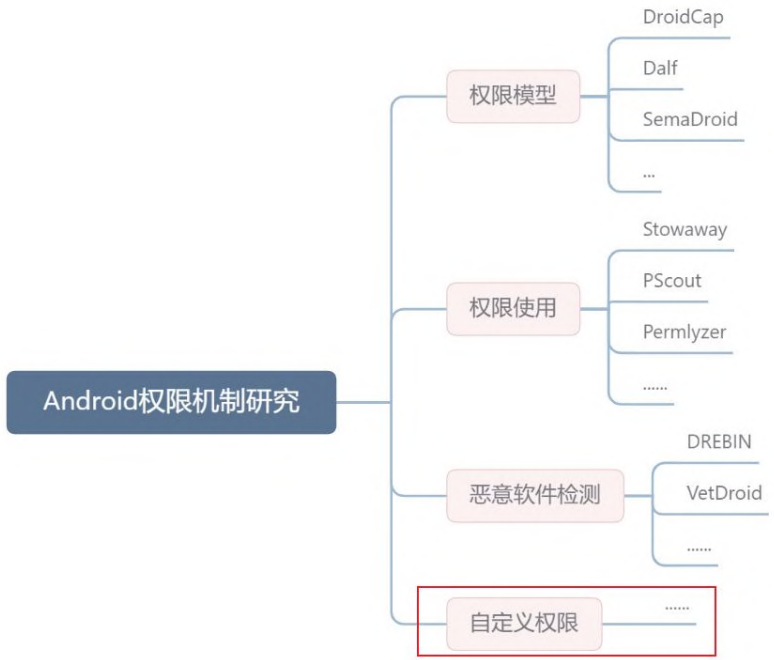
背景简介

- Android权限机制的重要性带来了许多相关研究
  - 权限模型：基于权限的安全模型设计，即分析权限模型，尝试改进当前的权限模型，以提高其安全性。
  - 权限使用：Android APP是否正确使用了权限
    - Stowaway，APP在使用过程中是否存在过度使用权限行为。
    - Pscout，静态分析Android OS源代码，分析权限规范所存在的问题。
  - 恶意软件检测
    - DREBIN，特征中包含为请求权限的API调用、请求的权限、使用的权限。
    - VetDroid，动态分析，基于权限使用行为，重建细粒度的恶意行为。



背景简介

- Android权限机制中的自定义权限
  - Android允许应用程序定义自己的权限
  - AndroZoo数据集（Google Play及第三方应用市场）随机选取约20万个APP，其中使用自定义权限的有25%；选取最热门免费应用，使用自定义权限的有70%。
  - 但有关于自定义权限安全性的研究却很少：
    - 第一个关于自定义权限安全的问题是在一篇2014年的博客中提到的“first one in wins”[1]，但Google的回应却是“这是权限工作的方式”；
    - 2018年，Tuncay等人发现自定义权限的两类漏洞，并构建攻击实例[2]。



基本概念

基本概念

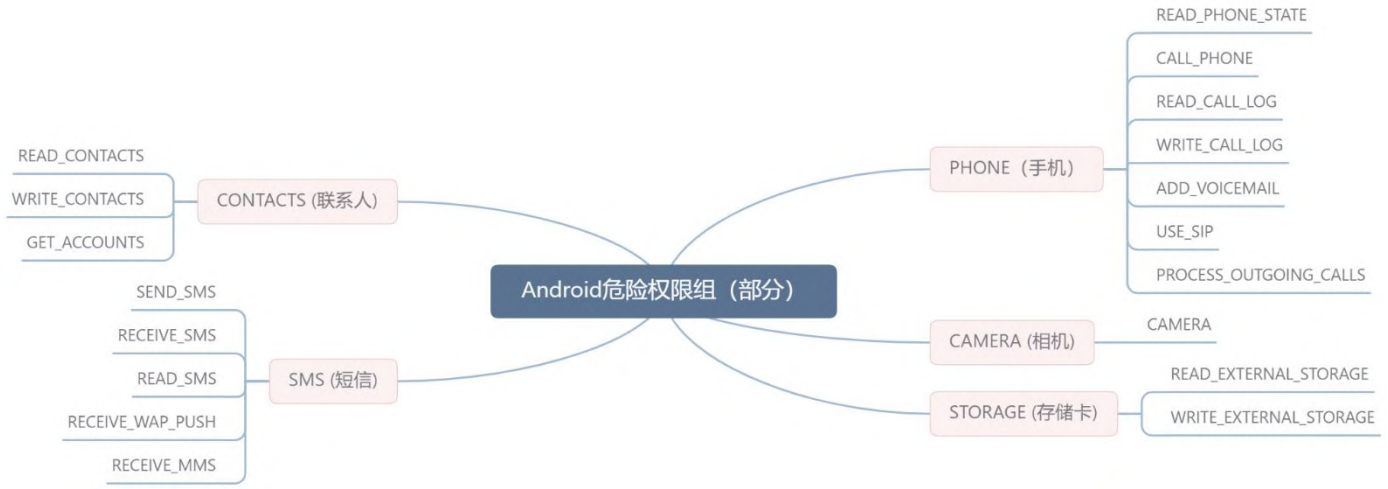
- Android权限机制
  - Android6.0之前，所有权限都是在应用安装时授予的；6.0版本之后，用户第一次使用某个APP时，自主选择是否授予APP危险级别的权限（运行时权限）。
  - 如果APP需要访问敏感API和系统资源，就必须要在其清单文件中声明相应的权限，并在使用时要求用户授权。

基本概念

- Android权限机制
  - Android权限保护的三个级别：
    - 普通（normal）：与签名权限组成安装时权限；
    - 签名（signature）：只能被由相同证书签名的APP使用；
    - 危险（dangerous）：运行时权限。
  - Android系统在安装APP时授予其普通权限和签名权限，且用户在安装时授予这两个权限后不能再撤销。
  - 用户可以在APP运行时选择授予或拒绝危险权限，且可以随时撤销。

基本概念

- Android权限机制
  - 所有危险权限都被分到一个对应的权限组中：
    - 例如READ_EXTERNAL_STORAGE和WRITE_EXTERNAL_STORAGE都属于STORAGE组。
  - 危险权限是基于组授予的：
    - 如果一个应用程序请求属于同一权限组的危险权限，一旦用户授予了其中一个，其他权限将自动被授予，而无需用户确认。
  - 权限的授予和撤销过程本质上就是改变相应的布尔类型（bool）变量。



基本概念

- 自定义权限
  - 系统权限目的是保护特定的系统资源，自定义权限则允许第三方APP定义自己的权限，以便保护和共享自己的资源或功能：
    - 例如同一公司开发的APP使用签名级别权限以保护平台资源。
  - 设计理念：
    - 在大多数使用场景中，Android不区分系统权限和自定义权限，包括保护级别、运行时权限控制、组管理过程，目的是为了统一和简化权限的控制。
  - Android本身也设计了一些机制来确保自定义权限不会影响系统权限的范围：
    - 自定义权限不能与系统权限同名；
    - 权限所有者优先设定为定义此权限的APP；
    - 系统APP先于第三方APP安装；
  - 两者本质上是不同的。

基本概念

- 清单文件（AndroidManifest.xml）
  - 描述APP包名、所使用的系统版本信息、权限定义与请求、暴露组件。
  - 权限定义、请求、使用：
    - 系统权限
      - 使用<uses-permission>请求。
    - 自定义权限
      - 先使用<permission>定义
        - » 权限名、保护级别、（权限组）
      - 再使用<uses-permission>请求。
    - 权限使用
      - 普通权限，允许安装则允许使用；
      - 签名权限，允许安装且签名一致则允许使用；
      - 危险权限，用户授予则允许使用，用户未授予则进行询问。

```
1  <permission
2  android:name="com.test.cp"
3  android:protectionLevel="dangerous"
4  android:permissionGroup="android.permission-
       group.UNDEFINED" />
5
6  <uses-permission android:name="android.
       permission.WRITE_EXTERNAL_STORAGE" />
7  <uses-permission android:name="android.
       permission.SEND_SMS" />
8  <uses-permission android:name="android.
       permission.CAMERA" />
9  ... <!--Omit lots of permission requests-->
10 <uses-permission android:name="android.
       permission.BODY_SENSORS" />
11 <uses-permission android:name="com.test.cp"
       />
```

<permission>用于定义自定义权限

<uses-permission>用于请求系统权限

自定义权限同样需要使用<uses-permission>进行请求

攻击案例

攻击案例

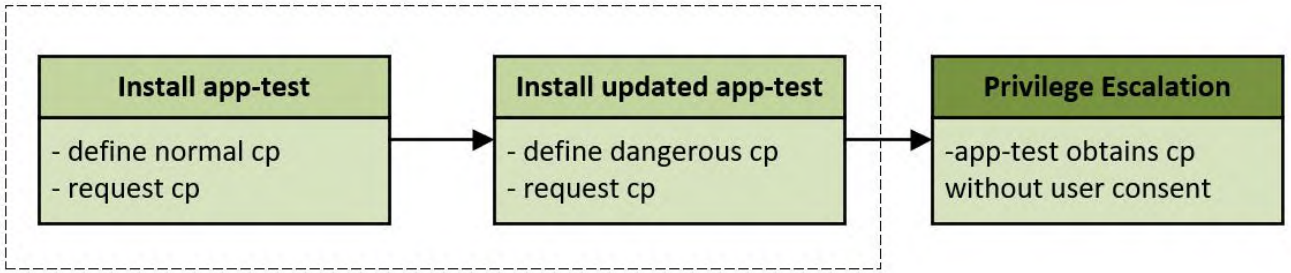
- Tuncay[2]等人发现自定义权限设计缺陷带来的漏洞并构建了攻击案例
  - 攻击条件
    - 攻击者可以爬取例如Google Play Store等APP应用商店的APP，并对其进行逆向工程，获得清单文件；
    - 攻击者分析自定义权限，构建并发布针对自定义权限漏洞的恶意APP。
  - 两类漏洞
    - 自定义权限升级；
    - 混淆代理。
  - 提出了防御措施：模块化设计方法Cusper
    - 隔离系统权限与自定义权限；
    - 自定义权限的命名约定。
  - 结果： Cusper与自定义权限设计理念冲突，会引发大量的逻辑和代码的修改，Google仅针对两个漏洞打补丁。

攻击案例

- 自定义权限升级
  - 原因：当app-test更新修改了自定义权限声明，使保护级别从正常或签名变为危险时，系统错误地将这种情况视为系统应用程序的升级，并尝试升级现有的权限到运行时权限，而无需通过用户授权。
  - 补丁：系统可以以检查其源包是否是系统应用程序的方式判断一个权限是否是系统权限。若是自定义权限，则当权限保护级别从正常或签名变为危险时，系统将保持原有的保护级别。

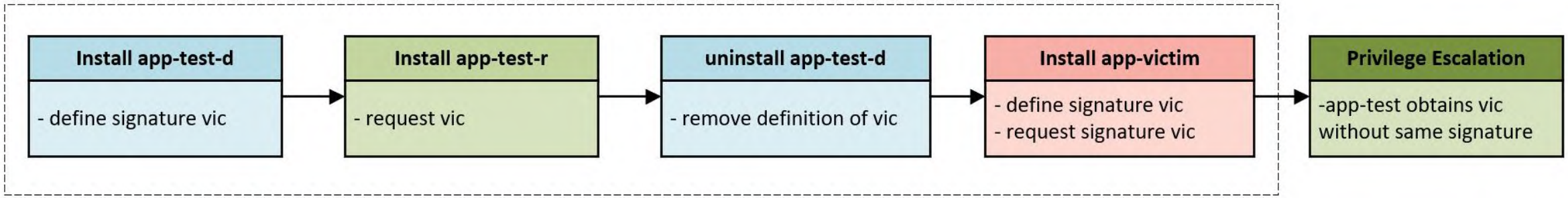
```
<!-- Define a custom permission -->
<permission
android:name="com.test.cp"
android:protectionLevel="normal"
android:permissionGroup="android.permission-
    group.PHONE"/>
< !-- Request a custom permission -->
<uses-permission android:name="com.test.cp"/
    >
```

"dangerous"



攻击案例

- 混淆代理
  - 原因：对自定义权限的名字没有足够约束。
  - 攻击条件：用户需要安装两个APP。
    - 应用内广告推荐达到两个APP的安装
    - 插件架构，以新应用的形式向用户揭示新功能
  - 补丁：授予签名权限之前加入检查过程，这一检查确保请求签名权限的应用程序与定义此权限的应用程序由相同的证书签名。



攻击案例

- Li[3]等人由此得到启发
  - 认为两个攻击案例可能只是冰山一角，需要一个自动分析工具。
  - 最终目标应该是识别存在于权限框架中的设计缺陷，而不仅仅是发现成功的攻击案例。
  - 受动机案例的启发，将分析过程抽象为特定APP、特定操作的执行顺序（序列），目的是寻找能触发权限升级漏洞的序列，而权限机制的内部操作可以被视作一个黑盒。

算法原理-CUPERFUZZER

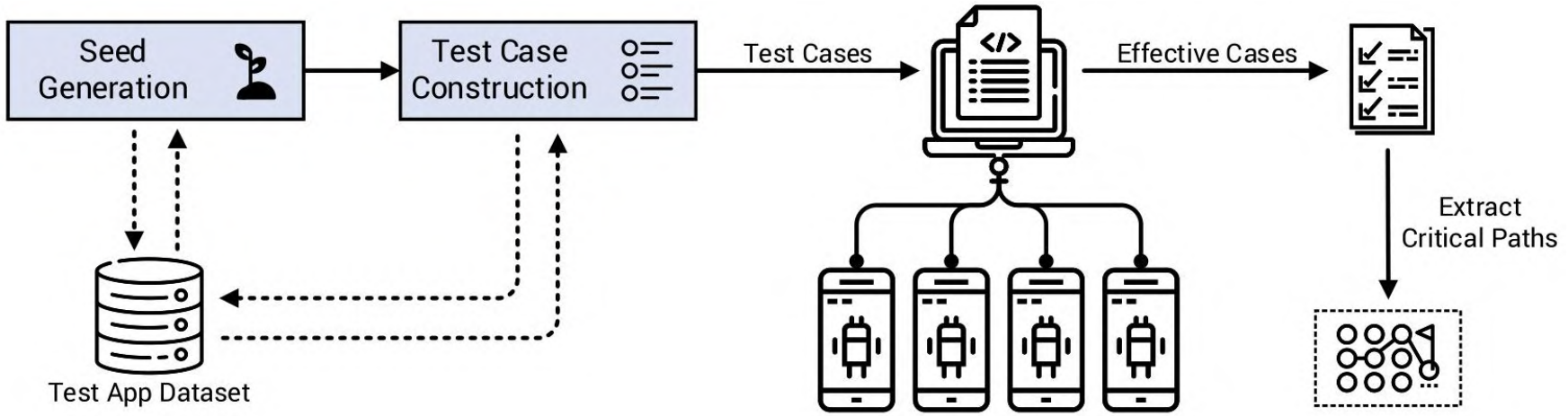
算法原理–CUPERFUZZER

T	发现自定义权限相关攻击案例并获取关键路径
I	测试APP + 权限相关操作组成的执行序列
P	1. 种子APP生成 2. 测试用例构建 3. 测试用例执行 4. 测试用例检查 5. 关键路径提取
O	有效攻击案例及关键路径

P	自动化发现及分析自定义权限设计缺陷引发的漏洞
C	整个模糊测试过程中，不主动授予测试APP任何危险级别权限；双应用模式下两个APP由不同的证书签名
D	如何权衡测试用例的多样性和有效性
L	CCF A类会议（S&P 2021）

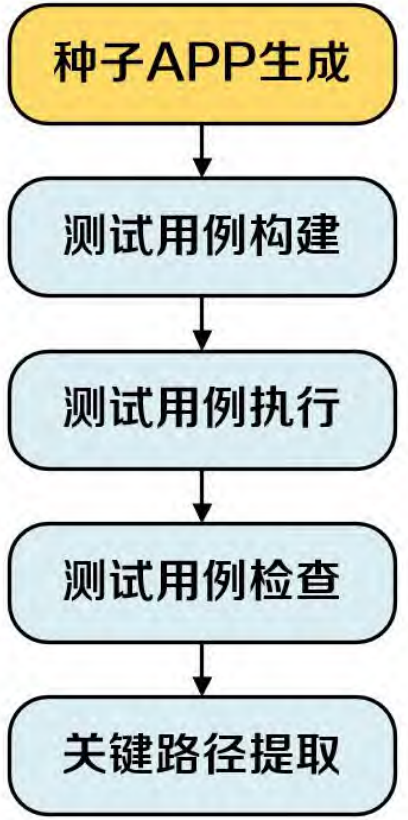
算法原理

- 算法原理图
  - 种子生成：生成一个测试APP作为种子，激活后续模糊测试过程；
  - 测试用例构建：每个测试用例本质上是一个由各种测试APP和权限相关操作组成的执行序列；
  - 测试用例执行：为了提高效率，在受控环境下对并行执行测试用例；
  - 测试用例检查：检查是否触发了权限升级问题；
  - 关键路径提取：自动过滤重复案例并识别关键路径。



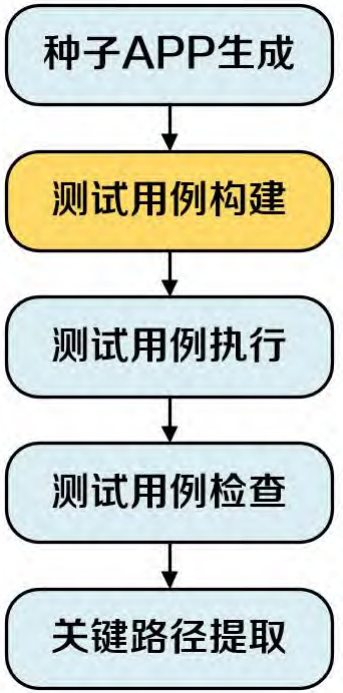
算法原理

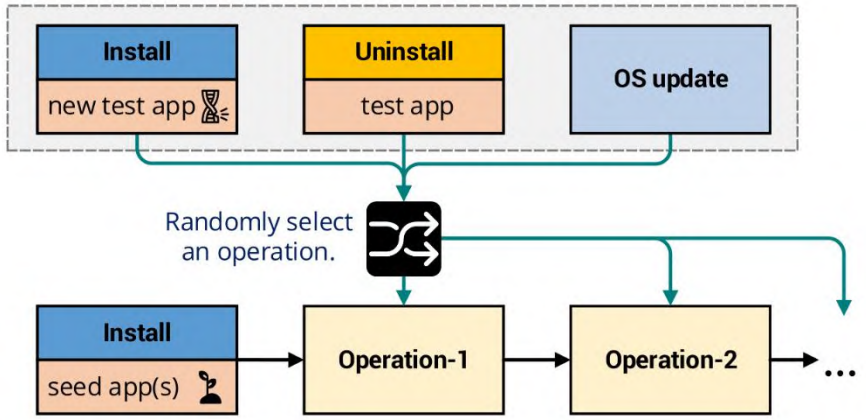
- 种子APP生成
  - 三个可变属性
    - 权限名称 —— 基于预定义列表
    - 权限等级 —— 普通 / 签名 / 危险
    - 权限组 —— 系统权限组 / 不定义
  - 生成模式
    - 单应用模式（signal app model）
    - 双应用模式（dual app model）
      - 定义权限和请求权限操作分开
      - 实现方式：在清单文件中将权限定义为“暴露的”
      - 限定：两个APP由不同的证书签名



算法原理

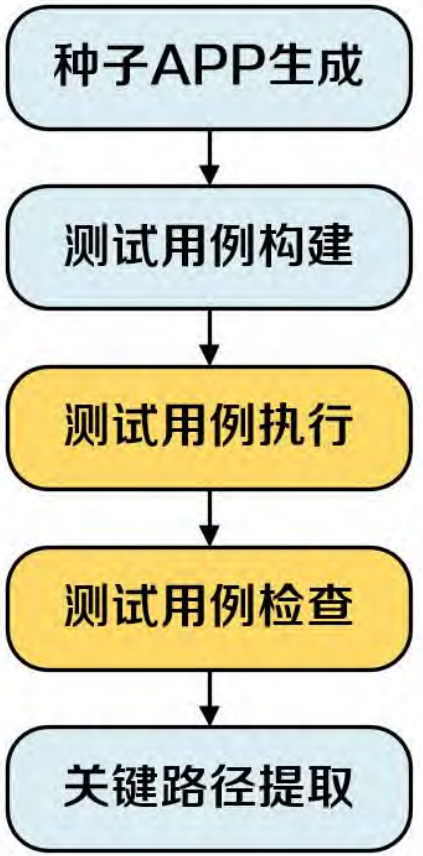
- 测试用例构建
  - 操作选择
    - 应用安装、应用卸载、应用更新、操作系统更新
  - 测试APP变异
    - 例如，新版本APP修改自定义权限的保护级别或权限组
  - 测试用例构建
    - 从{ 应用安装，应用卸载，操作系统更新 }中选择不定数量的操作，生成一个执行序列
      - 第一个操作是种子APP的安装
      - 执行卸载操作时，对应的APP必须存在
      - 再单应用模式下，最后一次操作后APP必须仍然存在
      - 操作系统更新在每个测试用例中只能执行一次





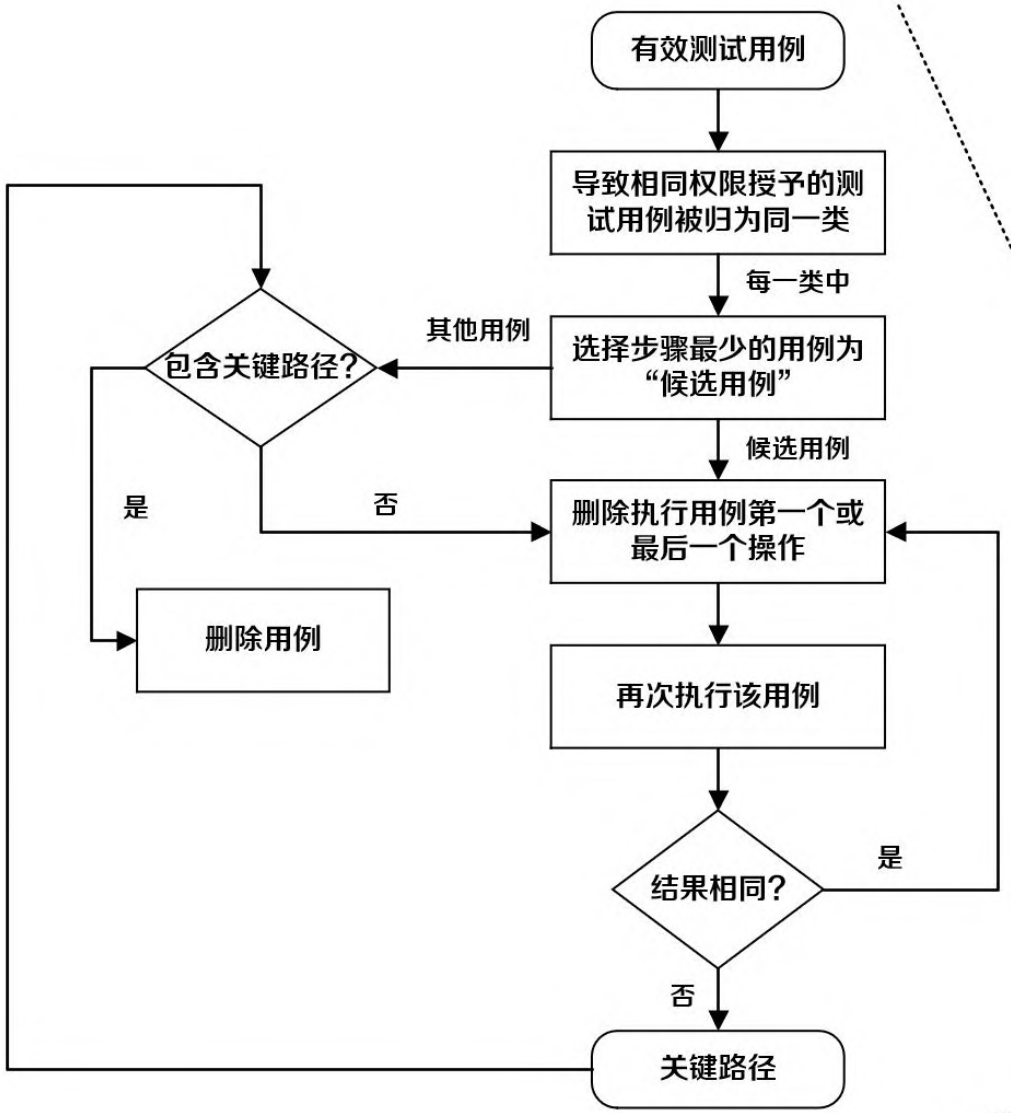
算法原理

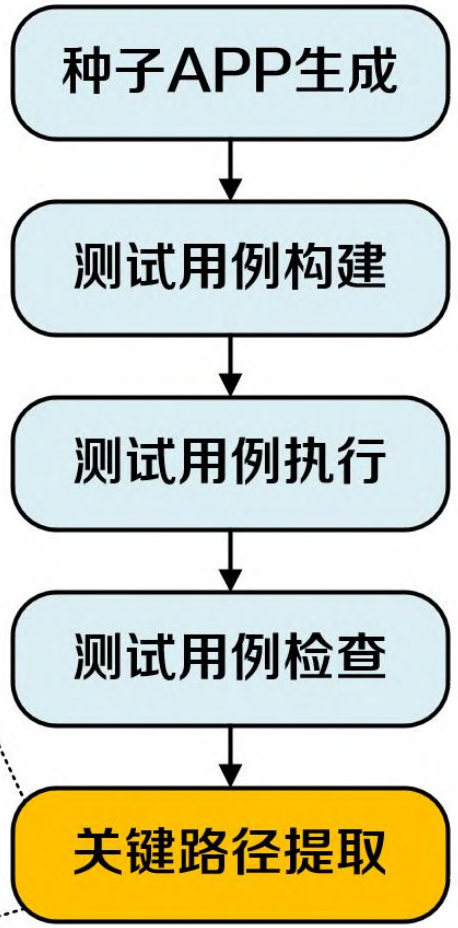
- 测试用例执行
  - 在真实的物理机上测试
  - 并行的用例执行
  - 环境重置：每进行一次用例执行，手机就会被恢复出厂设置
- 测试用例检查
  - 检查测试APP或双应用模式的请求权限APP是否实现了权限升级
  - 检查依据（权限升级情况）
    - 测试APP或双应用模式的请求权限APP在未经用户授权情况下获得了危险级别的权限
    - 测试APP或双应用模式的请求权限APP获得了签名权限，但其与定义该权限的APP是由不同证书签名的
  - 限定：整个模糊测试过程中，不主动授予测试APP任何危险级别权限



算法原理

- 关键路径提取


有效测试用例
导致相同权限授予的测试用例被归为同一类
每一类中
选择步骤最少的用例为“候选用例”
其他用例
包含关键路径?
是
否
删除用例
候选用例
删除执行用例第一个或最后一个操作
再次执行该用例
结果相同?
是
否
关键路径


种子APP生成
测试用例构建
测试用例执行
测试用例检查
关键路径提取

算法原理

- 测试用例优化
  - 由于动态分析耗时长，需要对测试用例进行优化
  - 最多包含5个操作
  - 请求系统或签名权限，随机选择一个；请求自定义权限，统一命名为com.test.cp
  - 2种生成模式、3个可用权限名、3种保护级别、12个系统权限组
    - 种子APP数量 = 2 × 3 × 3 × (12 + 1) = 234
    - 每个种子突变APP数量 = 3 × 13 + 1 = 40
    - 若存在3步APP安装（更新）操作，测试用例数量 = 40 × 40 × 40 = 64000
    - 限定：突变APP只能修改一个属性（保护级别或权限组）
      - 40 → 15；64000 → 3375

- 效率
  - 4部Pixel 2手机，40195个测试案例，耗时319.3小时(约13.3天)
  - 理想情况下耗时

TABLE III: Average execution time of the operation.

Operation Type	Operation	Time Cost (second)
Case execution	App installation	1.1
	App uninstallation	0.5
	OS update	109.8
Environment reset	Factory reset	60.2
	OS downgrade	129.5

  - 多台设备同时进入fastboot模式（OS更新工具）时，可能出现读写报错的情况
    - “status read failed (too many links)”
    - “command write failed (unknown error)”
    - 此时，为降低执行逻辑复杂性，减少耗时CUPERFUZZER会直接跳过这些测试用例

算法原理 实验结果

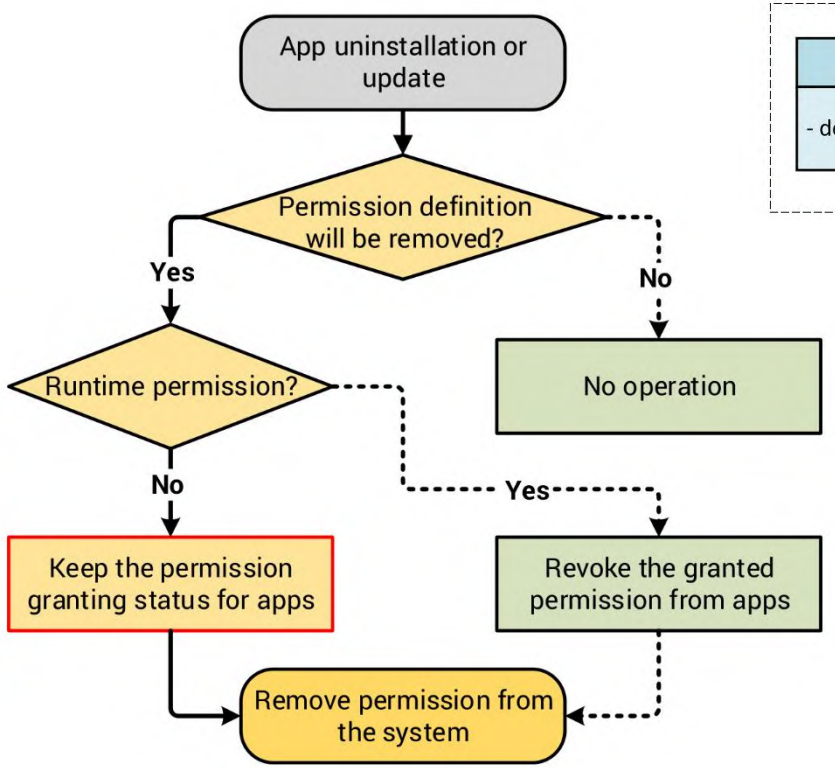
- 结果
  - 2384个有效测试用例，均符合情况1
    - 混淆代理补丁使得情况2难以实现
  - 自定义权限升级补丁在双应用模式或单应用模式 + OS升级时失效

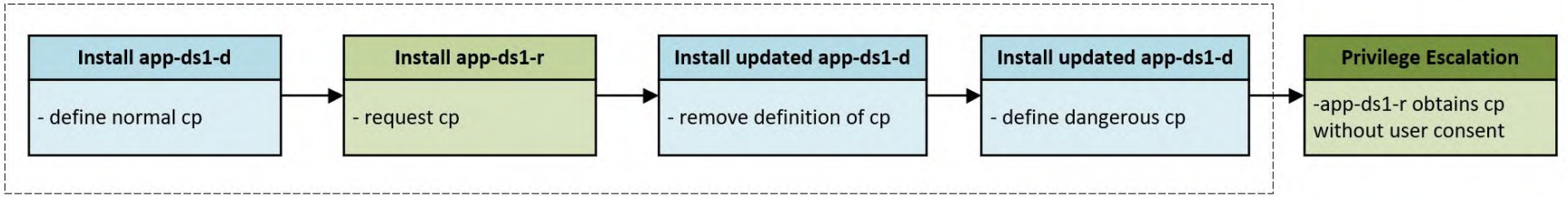
TABLE IV: Discovered critical paths in our experiments.

No.	Effective Cases	Seed Mode	Critical Path†	Privilege Escalation (Granted Permissions)	Flaw
1 ②	1,904	single-app dual-app	Installation [ACTIVITY_RECOGNITION, normal, NULL] → OS-update	ACTIVITY_RECOGNITION	DS#3
2	3	dual-app	Installation [com.test.cp, normal, NULL] → Installation [NULL, NULL, NULL] → Installation [com.test.cp, dangerous, NULL]	com.test.cp	DS#1
3	4	single-app dual-app	Installation [com.test.cp, normal, NULL] → Installation [com.test.cp, dangerous, NULL] → OS-update	com.test.cp	DS#4
4 ③	92	dual-app	Installation [com.test.cp, normal, NULL] → Uninstallation → Installation [com.test.cp, dangerous, NULL]	com.test.cp	DS#1
5-15‡	44	single-app dual-app	Installation [com.test.cp, normal, {Group}] → Installation [com.test.cp, dangerous, {Group}] → OS-update ①	com.test.cp system permissions in {Group}	DS#4
16	4	single-app dual-app	Installation [com.test.cp, normal, UNDEFINED] → Installation [com.test.cp, dangerous, UNDEFINED] → OS-update	com.test.cp READ_CONTACTS ... (30 dangerous system permissions in total)	DS#2
17-27‡	304	dual-app	Installation [com.test.cp, normal, {Group}] → Uninstallation → Installation [com.test.cp, dangerous, {Group}]	com.test.cp system permissions in {Group}	DS#1
28	27	dual-app	Installation [com.test.cp, normal, UNDEFINED] → Uninstallation → Installation [com.test.cp, dangerous, UNDEFINED]	com.test.cp READ_CONTACTS ... (30 dangerous system permissions in total)	DS#2
29	1	dual-app	Installation [com.test.cp, normal, NULL] → OS-update → Installation [NULL, NULL, NULL] → Installation [com.test.cp, dangerous, NULL]	com.test.cp	DS#1
30	1	dual-app	Installation [com.test.cp, normal, NULL] → OS-update → Uninstallation → Installation [com.test.cp, dangerous, NULL]	com.test.cp	DS#1

算法原理 实验结果

- 手工分析30条关键路径，获得Android权限框架的4个致命设计缺陷，获得官方认可的4个高危漏洞
  - DS#1，悬空的自定义权限（dangling custom permission）



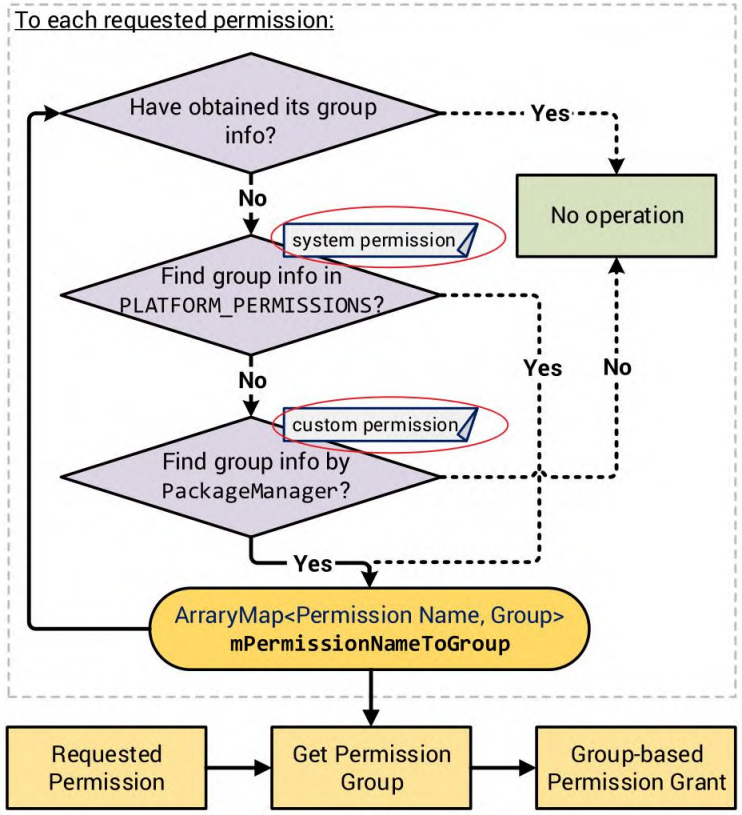


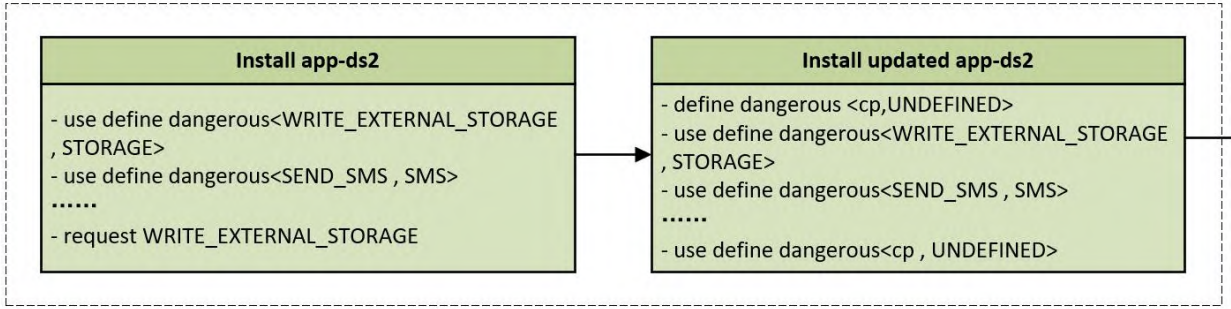
解析：

当一个APP卸载或更新，PackageManager（PMS）刷新所有权限 的注册和授予状态；如果一个危险级别的自定义权限定义被删除，系统也会从APP中撤销它的授权；

但是，如果被移除的自定义权限是安装时权限（普通 / 签名），则会保留相应的应用权限授予状态，导致权限悬空。

– DS#2，不一致的权限组映射（ inconsistent permissiongroup mapping）






```
1 <WRITE_EXTERNAL_STORAGE , STORAGE>
2 <SEND_SMS , SMS>
3 <CAMERA , CAMERA>
4 ...
5 <BODY_SENSORS , SENSORS>
```

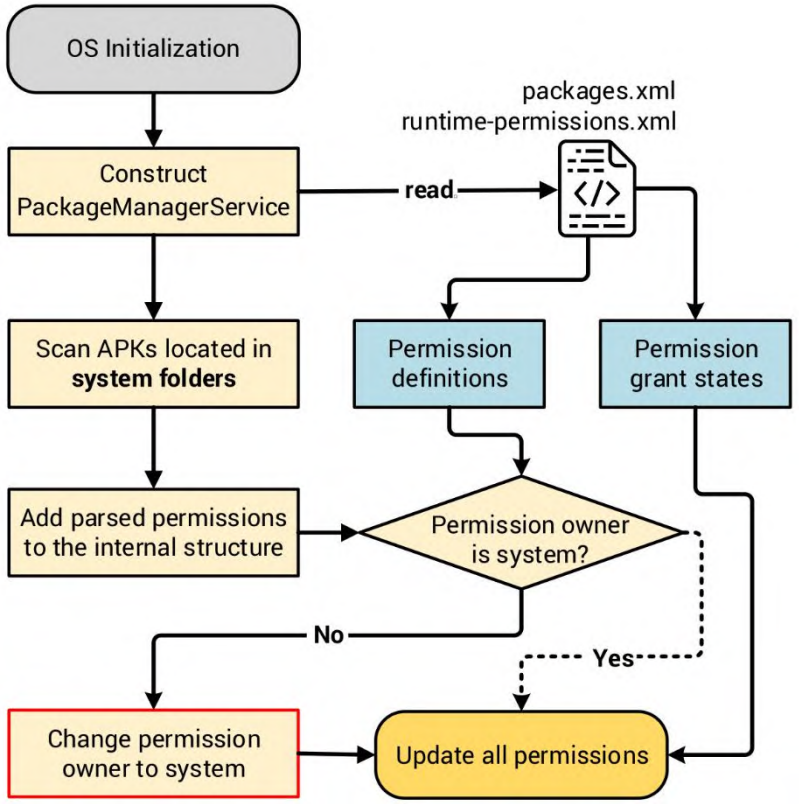
```
1 <WRITE_EXTERNAL_STORAGE , UNDEFINED>
2 <SEND_SMS , UNDEFINED>
3 <CAMERA , UNDEFINED>
4 ...
5 <BODY_SENSORS , UNDEFINED>
```

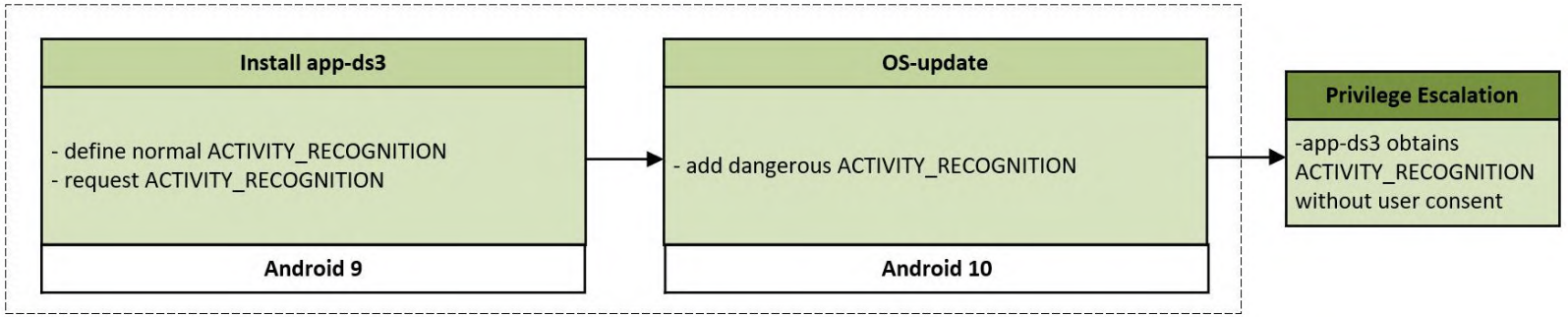
解析：

系统权限和自定义权限依赖于不同的来源来获取映射关系，可能存在不一致的定义；

在AndroidManifest.xml中，所有危险的系统权限都被放在一个特殊的权限组中，命名为android.permission-group.UNDEFINED。

– DS#3，自定义权限提升（custom permission elevating）



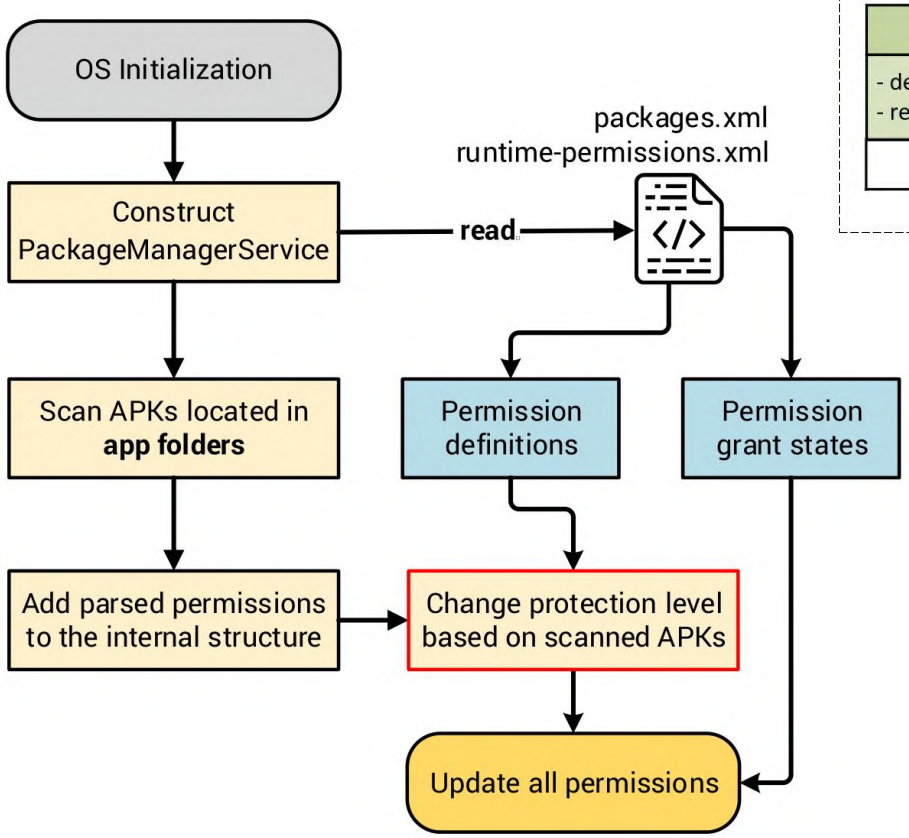


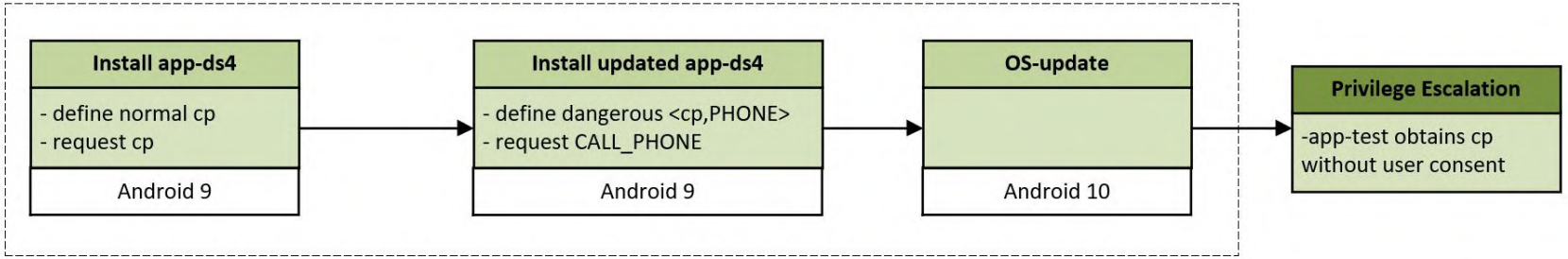
解析：

在Android操作系统初始化期间，PMS将被构建，用于APP安装卸载操作；PMS读取packages.xml和runtime-permissions.xml以获得存储的权限声明信息和授予状态；PMS扫描位于系统文件夹中的APKs，然后将解析的权限添加到内部结构中；

需要注意的是，如果权限的当前所有者不是系统，则该权限将被覆盖。

– DS#4，不一致的权限定义（inconsistent permission definition）





解析：

一个APP安装操作也可以更新它自己定义的现有自定义权限。在此过程中，当保护级别从正常或签名变为危险时，系统将保持原有的保护级别。这样的设计是为了阻止[2]中的权限升级攻击。可能存在此情况：系统持有的权限定义与所有者APP提供的权限定义不同，即权限定义不一致；

如果系统中存在基于源包刷新权限授予状态的逻辑，如OS升级过程，可能会出现权限升级问题。

- 缓解措施
  - 最小修复
    - 对DS#1，当系统删除自定义权限时，它对所有APP的授权应该被撤销
    - 对DS#2，删除未定义权限组，android.permission-group.UNDEFINED
    - 对DS#3，在OS更新时，如果权限的当前所有者不是系统，对应权限的授权应该被撤销
    - 对DS#4，当权限定义更新，APP对应权限的授权应该被撤销
  - 通用的指导方针
    - 不隔离系统权限和自定义权限的情况下，解决上述设计缺陷
    - Guideline#1，如果权限的定义被更改，应用程序的相应授权应该被撤销
      - 解决DS#1、DS#3、DS#4、[2]中的攻击
      - “定义被更改”包括权限所有者、权限组、保护等级
    - 销Guideline#2，系统所持有的权限的定义应该与权限所有者的定义（声明）一致
      - 解决DS#2（<权限，组>不一致）、DS#4（保护级别不一致）

优劣分析

- 优势
  - 实现了自动化有效攻击案例的发现
  - 对重复案例进行了过滤和关键路径提取
  - 缓解措施中，不需要隔离系统权限和自定义权限，遵从Android自定义权限设计理念，具有向后兼容性
- 局限性
  - 部分案例需要用户多次交互，在现实情况下不一定具有意义
  - 测试用例的多样性和有效性需要进一步提升

应用总结

应用总结

- 算法的应用领域
  - Android权限机制设计缺陷检测
  - Android OS更新检查
  - 用户设备隐患评估
- 未来的发展
  - 寻求更多设备并行执行的可行方法，受控环境、新的OS刷新工具
  - 寻找更有效分析方法，在生成测试用例或提取关键路径环节引入机器学习方法

参考文献

[1] M. L. Murphy. (2014) Vulnerabilities with Custom Permissions. Vulnerabilities with Custom Permissions (commonsware.com)

[2] G. S. Tuncay, S. Demetriou, K. Ganju, and C. A. Gunter. Resolving the Predicament of Android Custom Permissions[C]// Proceedings of the 25th Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 18-21, 2018, 2018.

[3] Li R, Diao W, Li Z, et al. Android Custom Permissions Demystified: From Privilege Escalation to Design Shortcomings[C]//2021 IEEE Symposium on Security and Privacy (SP). IEEE, 2021: 70-86.

知人者智，自知者明。胜人者有力，自胜者强。知足者富。强行者有志。不失其所者久。死而不亡者，寿。

谢谢！