Beijing Forest Studio

北京理工大学信息系统及安全对抗实验中心

Android第三方库检测

硕士研究生 陆永鑫

2022年11月20日

内容提要

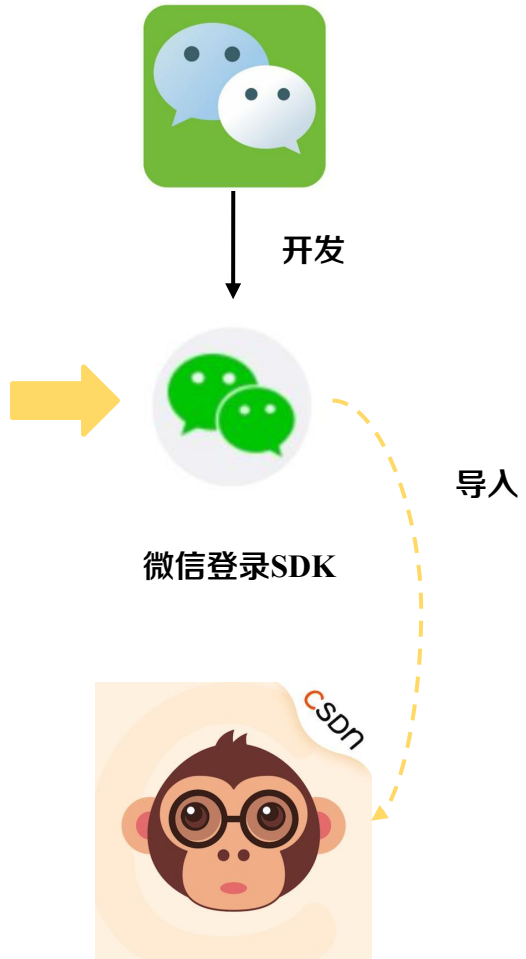
- 背景简介
- 基础概念
  - 白名单方法
  - 聚类方法
  - 相似性对比方法
- 算法原理
  - LibRoad
  - ATVHunter
- 应用总结
- 参考文献

背景简介

- 预期收获
  - 1. 了解Android第三方库的主要研究方向
  - 2. 了解Android第三方库检测的意义和基本方法
  - 3. 理解相似性对比方法的基本原理
  - 4. 理解APP代码混淆技术对第三方库检测的影响

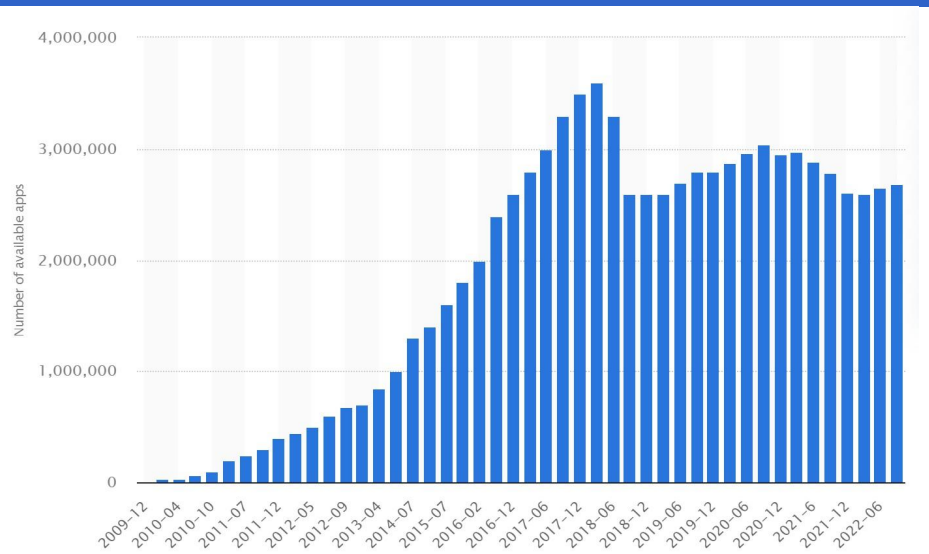
背景简介 第三方库

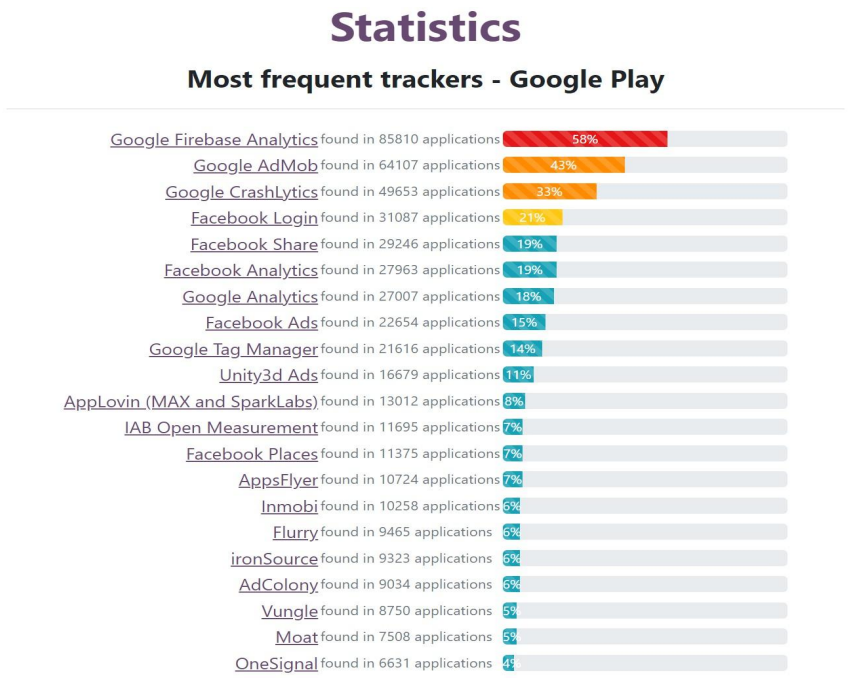
- 第三方库（Third-Party Library）
  - 功能
    - TPL提供许多可重用的功能
    - APP开发人员导入TPL并使用所需功能
      - 避免重复开发相同功能
      - 提升开发效率
  - 类型
    - 种类繁多的软件开发工具（SDK）
    - 丰富的UI
    - 各种社交媒体插件
    - 广告库
    - ……



背景简介 广泛使用

- Android APP数量急剧增加
  - 官方商城 Google Play [1] 中提供超过 250 万个APP
- Android TPL的广泛使用
  - Exodus Privacy[2]统计
    - Google Play 中大约 58% 的APP包含分析TPL
      - 例如，Google Firebase
  - 有研究表明
    - APP中超过 60% 的代码属于TPL



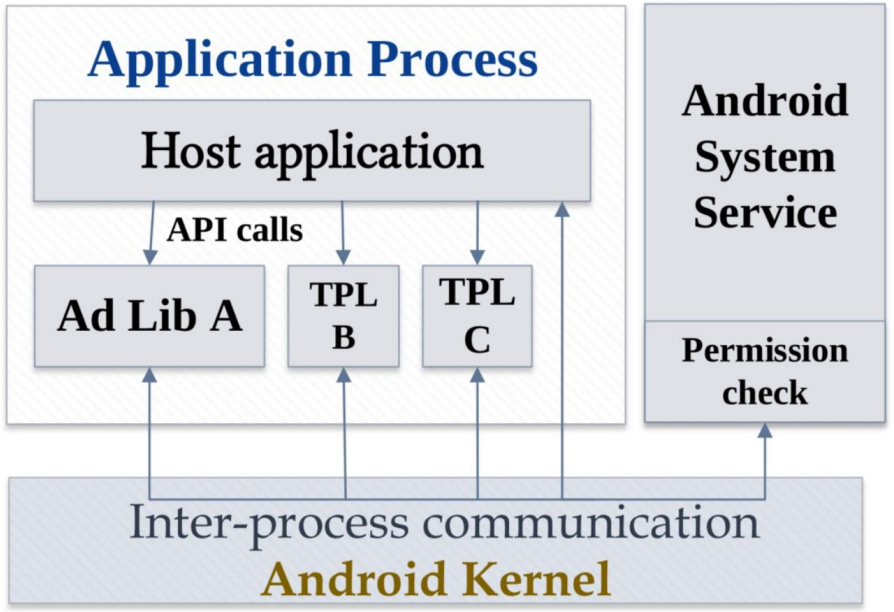


背景简介 安全隐患

- TPL的广泛使用带来安全问题
  - 库代码直接嵌入APP主程序会导致用户和安全分析人员无法了解其库成分
  - TPL存在缺陷和漏洞导致载体APP及用户面临安全威胁
    - 百度moplus SDK的wormhole、Chrome V8中的badkernel漏洞，可以被攻击者利用并远程控制手机
      - Android权限模型在应用程序级别起作用，TPL和主程序共享相同的权限

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:21.0.3'
}
```

» APP开发导入TPL方法示例



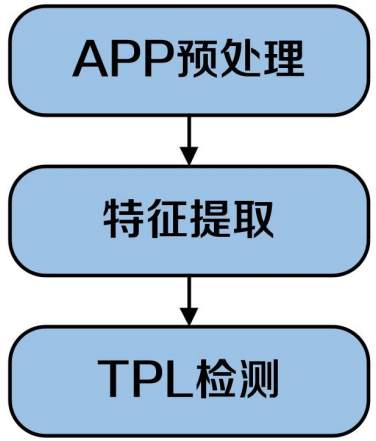
» APP安全机制示例

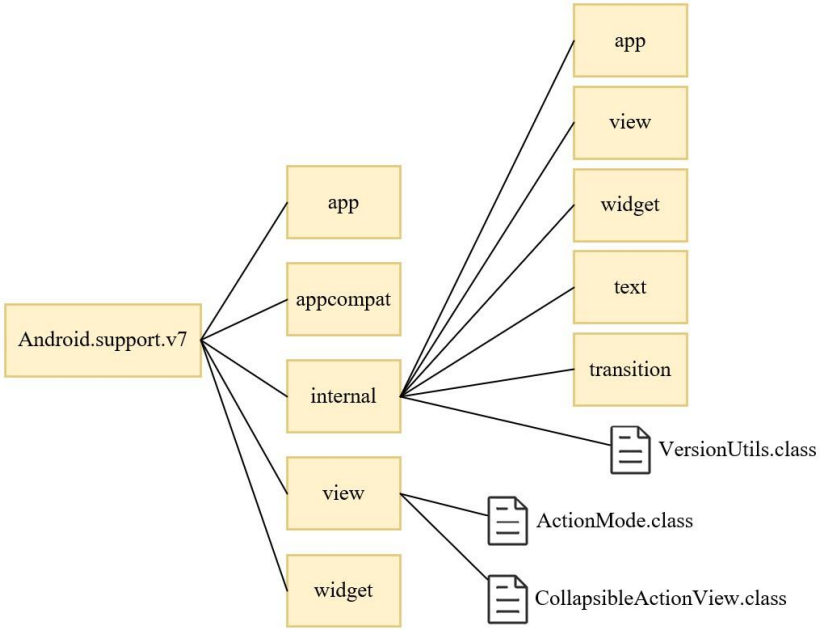
- 安卓TPL的相关研究[3]
  - TPL检测提升APP安全性
  - TPL检测是诸多领域的上游任务
    - TPL安全分析、TPL权限隔离、TPL更新
    - APP克隆检测、安卓恶意软件检测

基本概念

基础概念 TPL检测

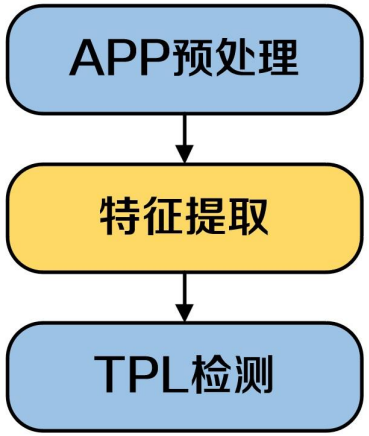
- TPL检测基本任务
  - 目标：检测APP使用的TPL
  - 输入：APP
  - 输出：TPL名称
- 与先前学术报告的2个区别（如同源漏洞检测）
  - 检测粒度：库级 vs 函数级
    - 某些函数相同 ≠ 使用了某个TPL
      - TPL之间存在嵌套
      - 不同TPL可能有相同的通用函数
  - 源码语言：Java/Kotlin vs C/C++
    - 反汇编难度：Java/Kotlin < C/C++
    - Java基于类和包组织代码
      - 主包：com.android（反向域名）

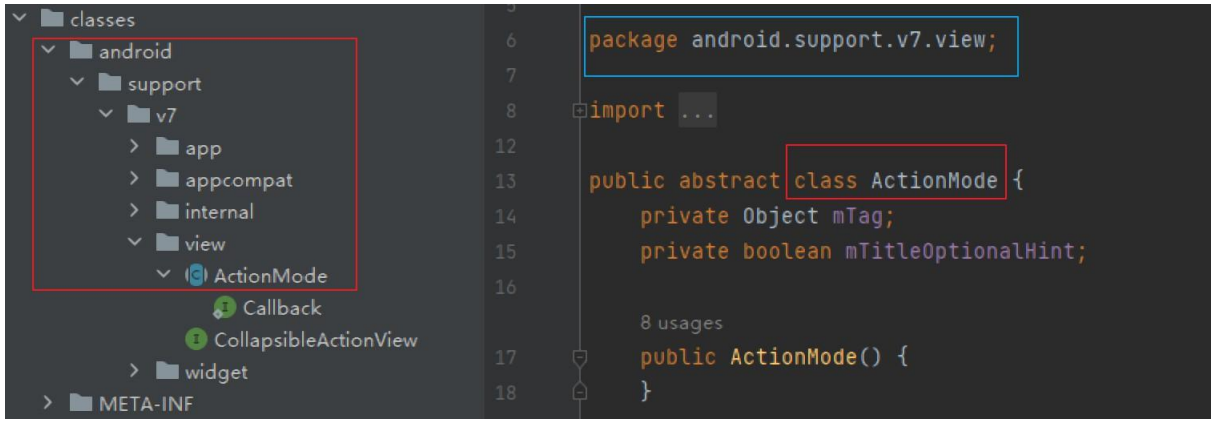




基础概念 白名单方法

- 思考：
  - 假设APP导入TPL时不改变文件结构和名称，如何检测APP中所使用的TPL?



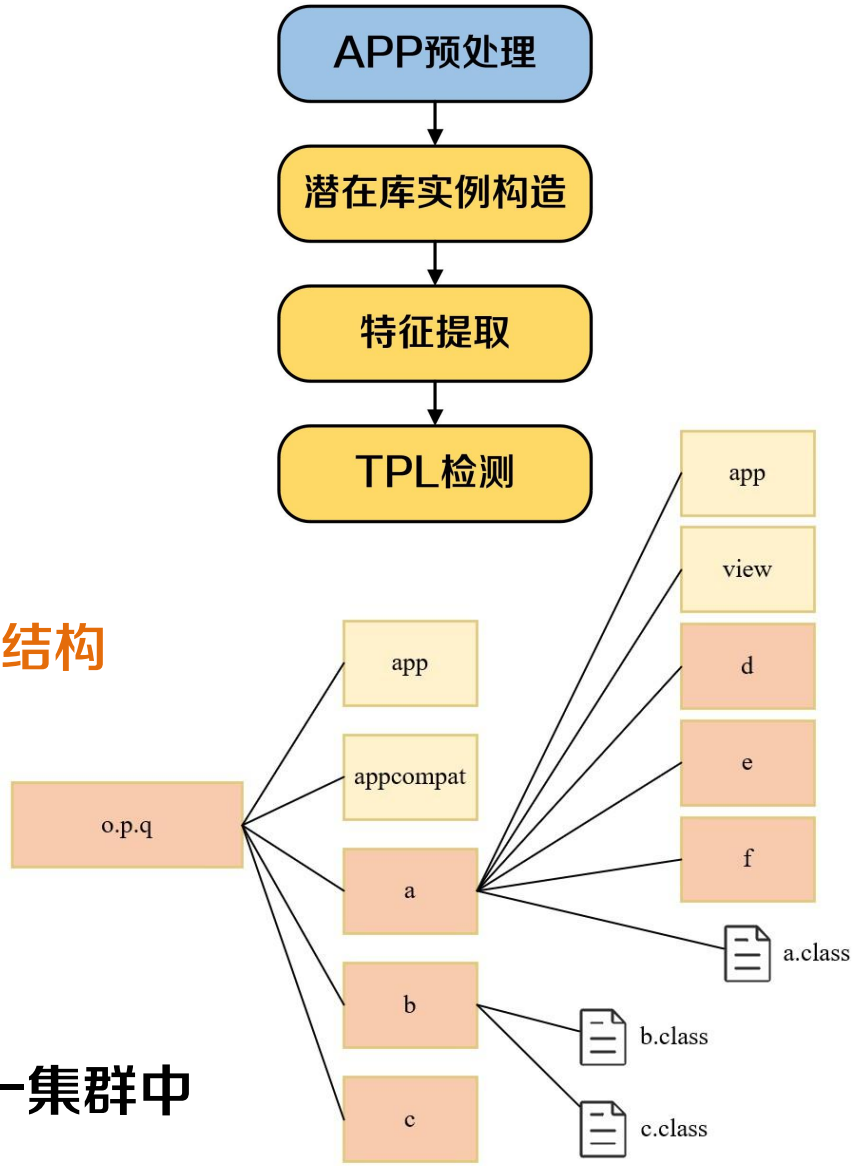


» APP导入TPL的文件结构

» APP反编译获得的文件结构

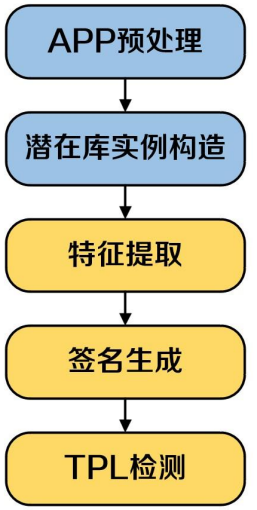
基础概念 聚类方法

- 标识符重命名混淆（混淆Ⅰ）
  - 包级别：包名称混淆
  - 代码级别：类/函数/变量名称混淆
- 聚类方法
  - 潜在库实例构造
    - 包名称被混淆，但包结构依然保留
      - Android.support.v7.view与o.p.q.b同结构
  - 特征提取
    - 提取API调用频率等特征
  - TPL检测
    - 同一TPL会被许多APP使用
    - 具有相同特征的潜在库实例，会被聚类到同一集群中

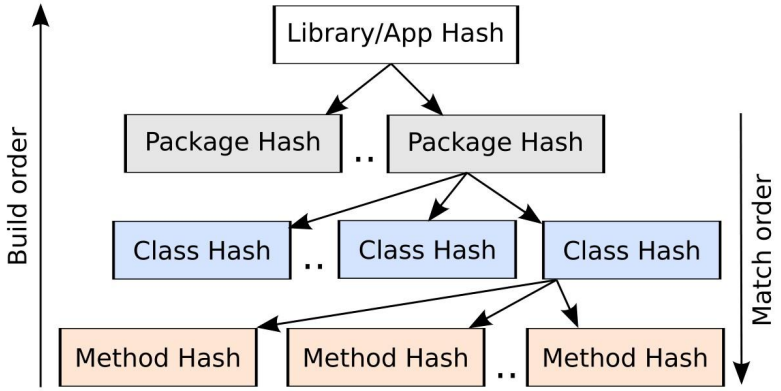


基础概念 相似性对比方法

- 聚类方法存在的问题
  - “同一TPL会被许多APP使用”
    - 会遗漏不流行的TPL
    - 不适合检测特定APP中TPL的任务
- 相似性对比方法
  - 特征提取
    - 模糊方法（函数）签名
  - 签名生成
    - 字符串哈希
    - 函数签名 → 类签名 → 包签名 → 库签名
  - TPL检测
    - 构建TPL签名集
    - 比较原始TPL和潜在库实例签名相似度




```
signature: com.fb.Session.openActiveSession(android.app.Activity,bool,com.fb.Session$StatusCallback)com.fb.Session
descriptor:                 (android.app.Activity,bool,com.fb.Session$StatusCallback)com.fb.Session
fuzzy descriptor:           (android.app.Activity,bool,X)X
```



LibRoad

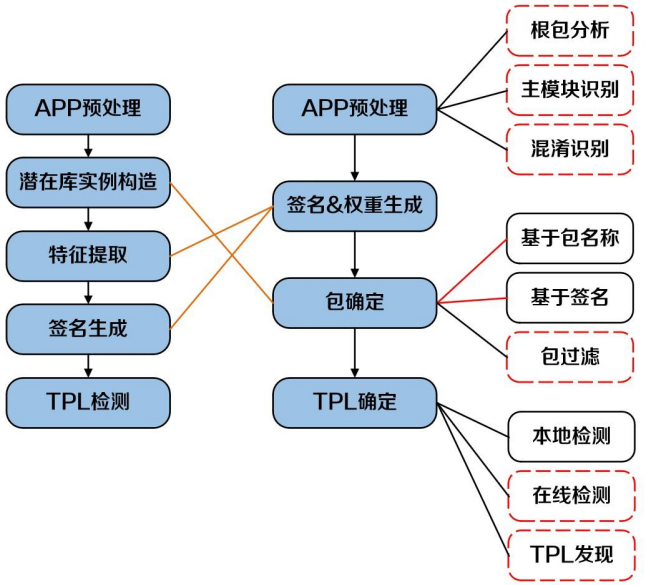
LibRoad TIPO PCDL

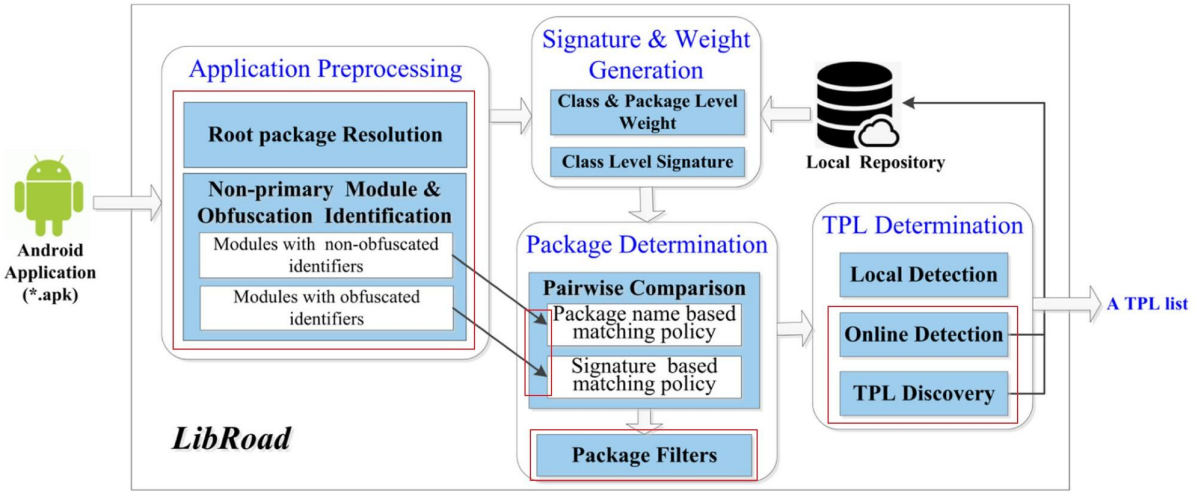
T	检测Android APP中所使用的TPL
I	Android APP、原始TPL文件
P	1. APP 预处理 2. 签名和权重生成 3. 包确定 4. TPL确定
O	TPL名称

P	提高检测效率、降低假阳性和假阴性
C	原始TPL文件可获得、原始TPL包结构未改变
D	如何充分利用APP中包含的原始TPL信息
L	SCI 2区期刊（TMC 2020）

LibRoad 方法框架

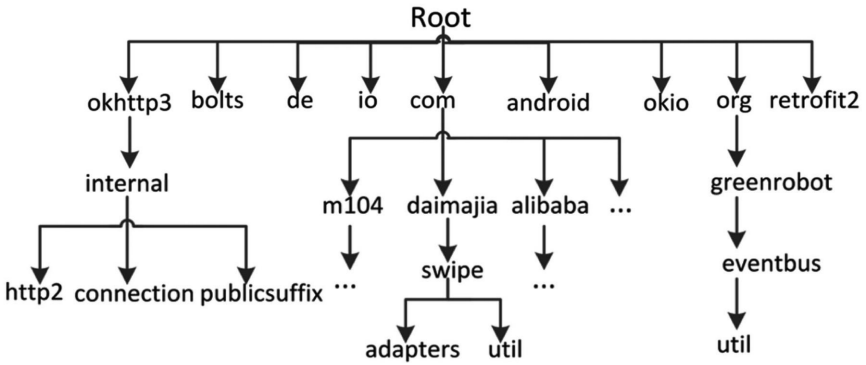
- Framework
  - APP 预处理
    - 根包分析：得到APP的所有ARP
    - 主模块识别：识别APP中的主ARP
    - 混淆识别：将非主ARP分为混淆ARP/非混淆ARP
  - 签名&权重生成
    - 生成类签名/权重，包签名/权重
  - 包确定
    - 基于包名称/签名相似度进行包匹配
  - TPL确定
    - 基于包相似度进行本地检测
    - 基于爬虫进行在线检测
    - 基于聚类进行TPL发现





LibRoad 根包分析

- 根包分析算法
  - line1：逆向功能提取文件结构，获取所有AP
  - line3：构建包层次树并获取所有树节点
  - line4~17：遍历每棵树，找到最长相同路径
  - line18~26：查找每个根包的子包
- 例如下图APP
  - com.daimajia.swipe是一个ARP
  - com.daimajia.swipe.adapters/util是其ASP



Algorithm 1. Root Package Resolution Algorithm

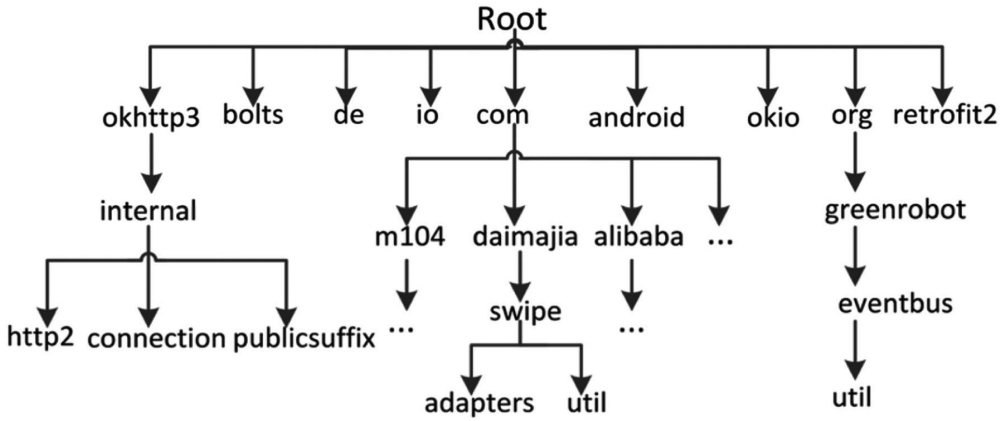
Input: apk: an installation file of an Android application;
Output: *ARPs*: a set of root packages; <*arp*, *ASPs*>:a map of sub packages of each *arp* ∈ *ARPs*

```
1: APs = getAllPackagesofAPK(apk);
2: ARPs = new HashSet<>();
3: PackageTree nodes = getPackageTreeNodes(APs);
4: for (PackageTreeNode root: nodes) do
5:    childs = getChildNodes(root);
6:    if (isEmpty (childs)) then
7:       addRootPackage(root, ARPs);
8:    else
9:       for (PackageTreeNode child: childs) do
10:         getCurrentARPName(root,child);
11:         offsprings = getChildNodes(child);
12:         if (getFirst(offsprings))
13:            getCurrentARPName (“.”+ offsprings [1]);
14:         end if
15:      end for
16:   end if
17: end for
18: for each arp ∈ ARPs do
19:    ASPs = new HashSet<>();
20:    for each ap ∈ APs do
21:       if ap.isChildof(arp) then
22:          addSubPackage(ap, ASPs);
23:       end if
24:    end for
25:    contructPackageMap(arp, ASPs);
26: end for
27: return ARPs & {<arp, ASPs>};
```

LibRoad 主模块&混淆识别

- 基于AndroidManifest.xml识别主包
  - 一个APP由多个AP组成
    - 主AP（不包含）和非主AP
      - 消除主AP可减少成对比较数量
  - action.MAIN：入口Activity
    - “com.m104”为主ARP
- 基于单字符检测识别混淆包
  - 流行混淆器的默认设置
  - 例如，“com.a”被认为是混淆包

```
<activity android:label="@string/AppName"
        android:name="com.m104.SplashScreenActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```



LibRoad 签名&权重生成

- 签名
  - 模糊函数签名 + 字符串哈希 + 层次签名生成
- 类权重

$$weight(c) = count(c.m) + deg^{+}(c) + deg^{-}(c)$$

  - 包含更多成员的类应该要有更大的权重
    - 库的功能是在函数中实现的
  - 具有更多依赖关系的类也应该有更大的权重
    - 反映了库的内部连接性
    - 类名称被混淆，但仍保留了依赖关系
- 包权重

$$weight(p) = count(p.c) + \sum_{i=1}^{n=|p|} weight(c_i)$$

  - 类数量 + 类权重
- 归一化
  - 同一个包中所有类 / 同一个TPL或ARP中所有包

LibRoad 包确定

- 设一个TPL $L = \{lp_1, \ldots, lp_n\}$，共$q$个TPL
- 对一个APP为A
  - $A = ASP^o \cup ASP^{\bar{o}}$
  - $ASP^o = \{asp_1^o, \ldots, asp_{m_1}^o\}$
  - $ASP^{\bar{o}} = \{asp_1^{\bar{o}}, \ldots, asp_{m_2}^{\bar{o}}\}$
- 基于包名称的匹配策略
  - 当ASP名称未被混淆时使用
  - 时间复杂度为$O(n \times q \times m_2)$
- 基于签名的匹配策略
  - 当ASP名称被混淆时使用
  - 时间复杂度为$O(n \times q \times z_1 \times z_2 \times m_1)$
  - $max\{\}$可以在TPL导入被删除部分类的情况下，减少漏报

$$sim_{name}(asp_j^{\bar{o}}, lp_i) = \begin{cases} 1, & asp_j^{\bar{o}}.name = lp_i.name \\ 0, & otherwise \end{cases}$$

$$sim_{c\_to\_c}(c_j^{asp}, c_i^{lp}) = \frac{|\{c_i^{lp}.member\_signatures\} \cap \{c_j^{asp}.member\_signatures\}|}{|\{c_i^{lp}.member\_signatures\} \cup \{c_j^{asp}.member\_signatures\}|}$$

$$sim_{c\_to\_p}(c_j^{asp}, lp) = max\left\{ sim_{c\_to\_c}(c_j^{asp}, c_i^{lp}) \middle|_{i=1}^{|z_1|} \right\}$$

$$sim_{p\_to\_p}(asp^o, lp) = \frac{|\sum_{j=1}^{|asp^o|} sim_{c\_to\_p}(c_j^{asp}, lp) \times \overline{weight}(c_j^{asp})|}{|asp^o|}$$

LibRoad 包确定

- 2条包过滤规则
  - asp_i^o包层级数与lp_j不同时
  - asp_i^o的顶级包名是非混淆的，并且与lp_j的顶级包名不同时

ASP	matched LPs (Library\LP)	similarity score	Really Matched
io.a.a.a.a.d	fabric-1.4.4 \io.fabric.sdk.android.services.events	0.4842	√
	androidasync-1.3.7 \com.koushikdutta.async.http.socketio.transport	0.4842	×

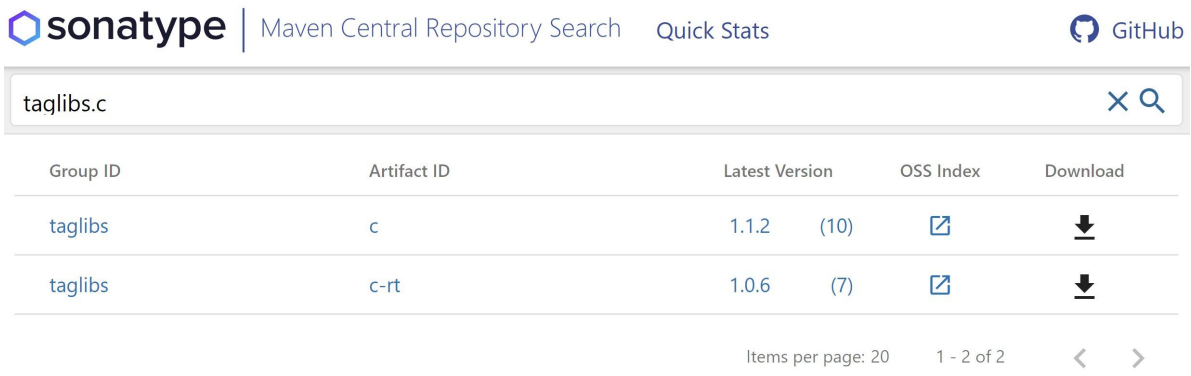
ASP		matched LPs		similarity score	Really Matched
name	level	Library\LP	level		
com.b.a.d.b	5	glide-3.8.0 \com.bumptech.glide.load.engine	5	0.9545	√
		dexter-5.0.0 \com.karumi.dexter.listener	4	0.9545	×

LibRoad TPL确定

- 基于本地存储库确定
  - 对于一个ARP，能找到≥1个LP时
  - 但本地存储库总是有限的，
    且新的TPL不断被开发
- 在线确定
  - 在线搜索（search.maven.org）
  - 爬取所有版本
  - 选择签名相似度最高的TPL版本
- 基于聚类方法发现
  - 挖掘大量APP中的共享代码
  - 匹配共享代码簇发现类似组件集
  - 比较类似组件集签名来发现潜在TPL

$$sim_{asp\_to\_l}(asp, l) = max\left\{sim_{p\_to\_p}(asp, lp_j)\Big|_{j=1}^{m}\right\}$$

$$sim_{arp\_to\_l}(ARP, L) = \frac{\left|\sum_{i=1}^{n} sim_{asp\_to\_l}(asp_i, L) \times \overline{weight}(asp_i)\right|}{n}$$



LibRoad 消融实验

- 数据集
  - 基于构建的APP、基于真实世界APP
- 实验结论
  - 主模块识别、基于包名称匹配极大提升检测速度
  - 包过滤降低假阳性率
  - 在线检测提高召回率

$$\text{recall} = \frac{\#\ of\ true\ positives}{\#\ of\ pairs\ in\ the\ ground\ truth}$$

$$\text{precision} = \frac{\#\ of\ true\ positives}{\#\ of\ detected\ airs}$$

$$\text{FPR} = \frac{\#\ of\ detected\ pairs - \#\ of\ true\ positives}{\#\ of\ detected\ pairs}$$

$$\text{FNR} = \frac{\#\ of\ pairs\ in\ the\ ground\ truth - \#\ of\ true\ positives}{\#\ of\ pairs\ in\ the\ ground\ truth}$$

Ground Truth	Component	Detected Pairs	True Positive	Recall	False positive ratio	Precision	Avg.Time per APK
2800 pairs (GTB-1) Customized	Primary module identification				-		
	Package name based comparison	2964	2796	99.86%	5.67%	94.33%	243.79
	Package Filters	4127	2796	99.86%	32.25%	67.75%	28.14
	Online Detection				-		
	The whole approach	2964	2796	99.86%	-	94.33%	30.63
14233 pairs (GTB-2) Real-world	Primary module identification	16063	14233	100%	11.39%	-	140.05
	Package name based comparison	16060	14233	100%	11.38%		791.42
	Package Filters	39969	14233	100%	64.39%		84.57
	Online Detection	7954	6134	43.08%	22.88%		56.66
	The whole approach	16060	14233	100%	11.38%		91.38

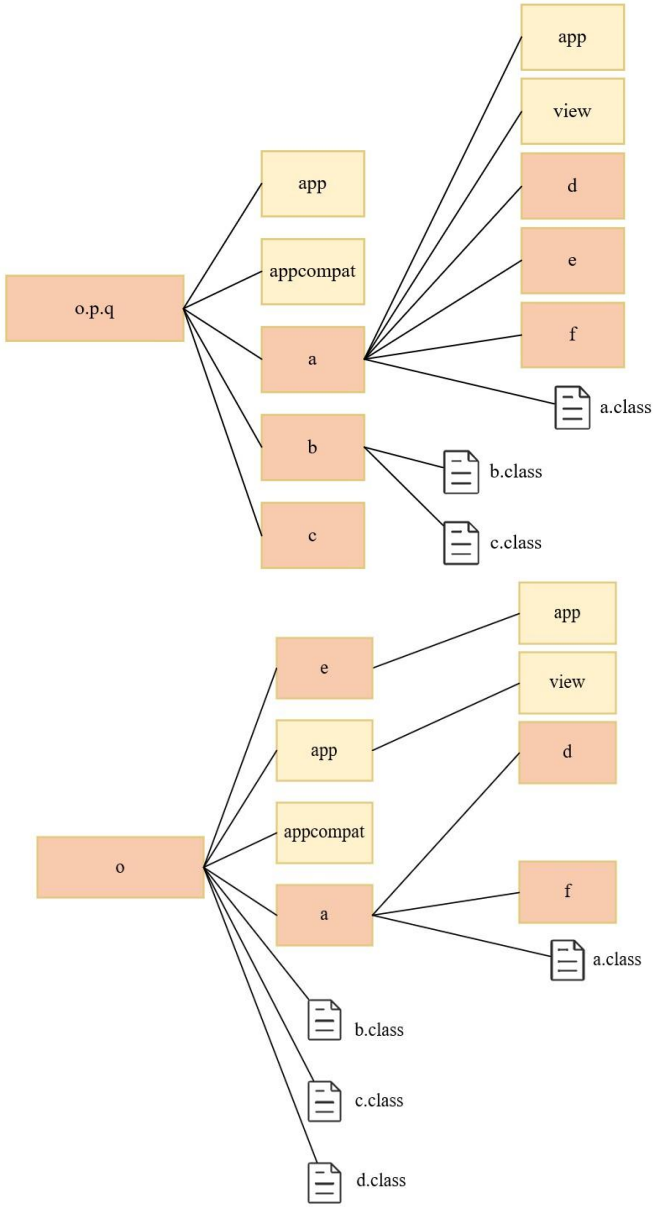
LibRoad 对比实验

- 对比方法
  - 聚类方法：LibRadar、LibD
  - 相似性对比方法：LibPecker
- 实验结论
  - 论文方法在基本事实集中无漏报
    - 得益于在线检测
    - 聚类方法特征粒度更粗、遗漏不流行TPL
  - 论文方法误报率最低
    - 得益于根包解析和包过滤
    - LibPecker采用类似的权重生成方式和匹配策略
  - 论文方法检测速度优于同类型方法，但低于聚类方法
    - 得益于主模块消除、混淆识别
    - 聚类方法无大量成对比较

	# of detected pairs	# of true positives	FPR (%)	FNR (%)	Time (sec.)
LibRadar	9990	5481	45.13	61.67	17.8
LibD	12107	9592	20.77	32.61	20.5
LibPecker	19392	13872	28.47	2.54	602.2
LibRoad	16060	**14233**	**11.38**	0	91.4

LibRoad 优劣分析

- 优势
  - 充分利用APP中包含的未被改变的原始TPL信息
  - 采用组合策略（融合白名单方法、聚类方法）
- 劣势
  - 基于聚类的TPL发现作用较小
    - 实际上，仅发现14个非公开TPL
  - 约束条件
    - 要求原始TPL包结构未改变
      - 根包分析、签名&权重生成、包确定
    - 无法应对包扁平化混淆（混淆II）



ATVHunter

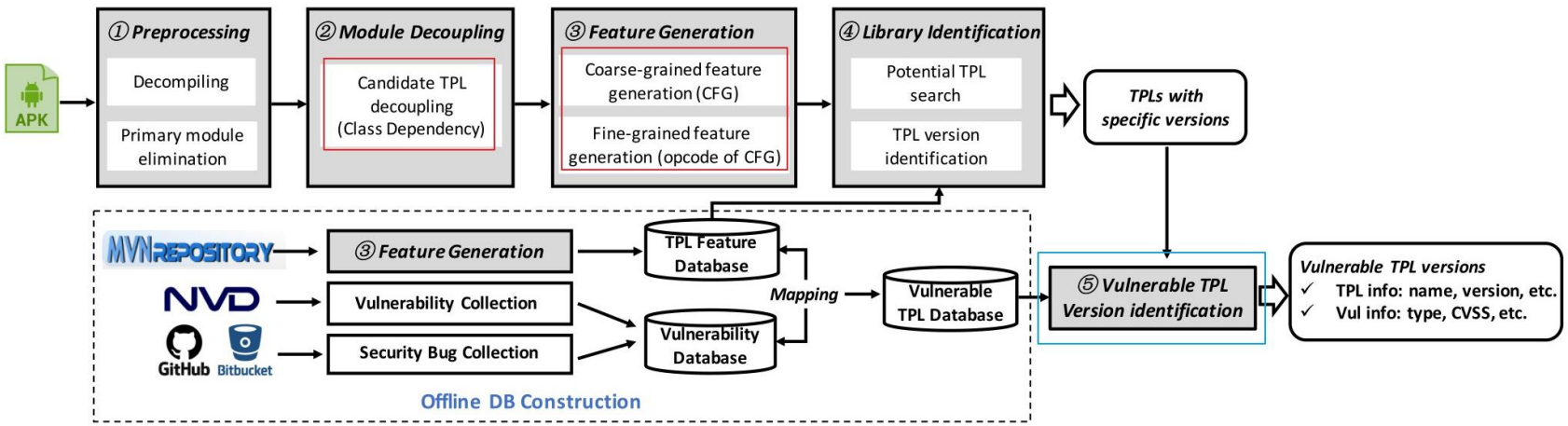
ATVHunter TIPO PCDL

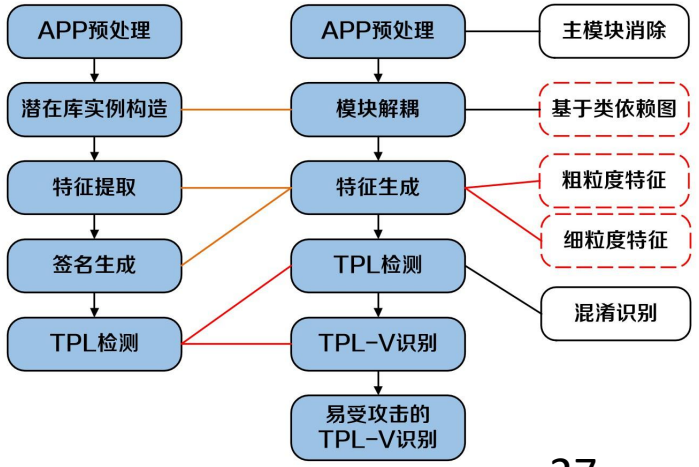
T	检测Android APP中所使用的TPL及其版本
I	Android APP、原始TPL文件
P	1. APP 预处理 2. 模块解耦 3. 特征生成 4. TPL检测 5. TPL版本识别
O	TPL名称和版本

P	精确识别APP中所使用的TPL版本
C	原始TPL文件可获得
D	如何提升方法对4类代码混淆技术的弹性
L	CCF A类会议（ICSE 2021）

ATVHunter 方法框架

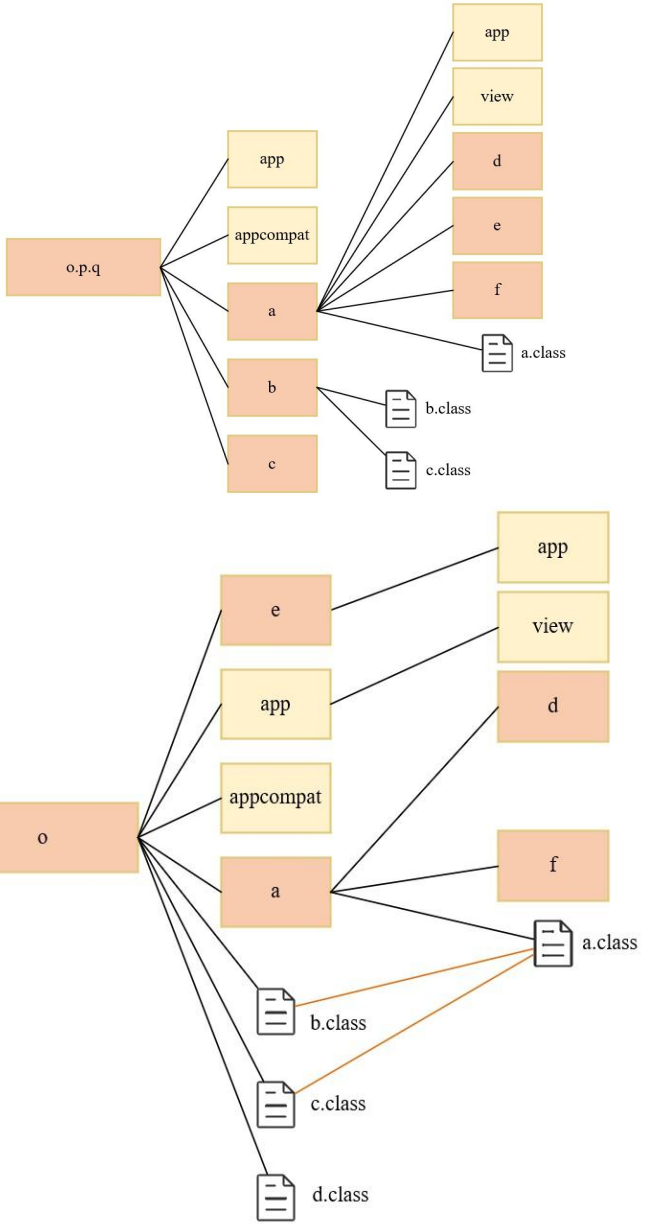
- Framework
  - APP预处理：反编译、主模块消除
  - 模块解耦：将非主模块拆分为多个潜在库实例
  - 特征生成：提取函数级别粗粒度、细粒度特征；生成类特征、TPL特征
  - TPL检测&TPL-V识别：基于包名称和3种特征检测TPL(-V)
  - 易受攻击的TPL-V 识别（非重点）：构建Vul TPL-V数据库，输出漏洞和bug报告





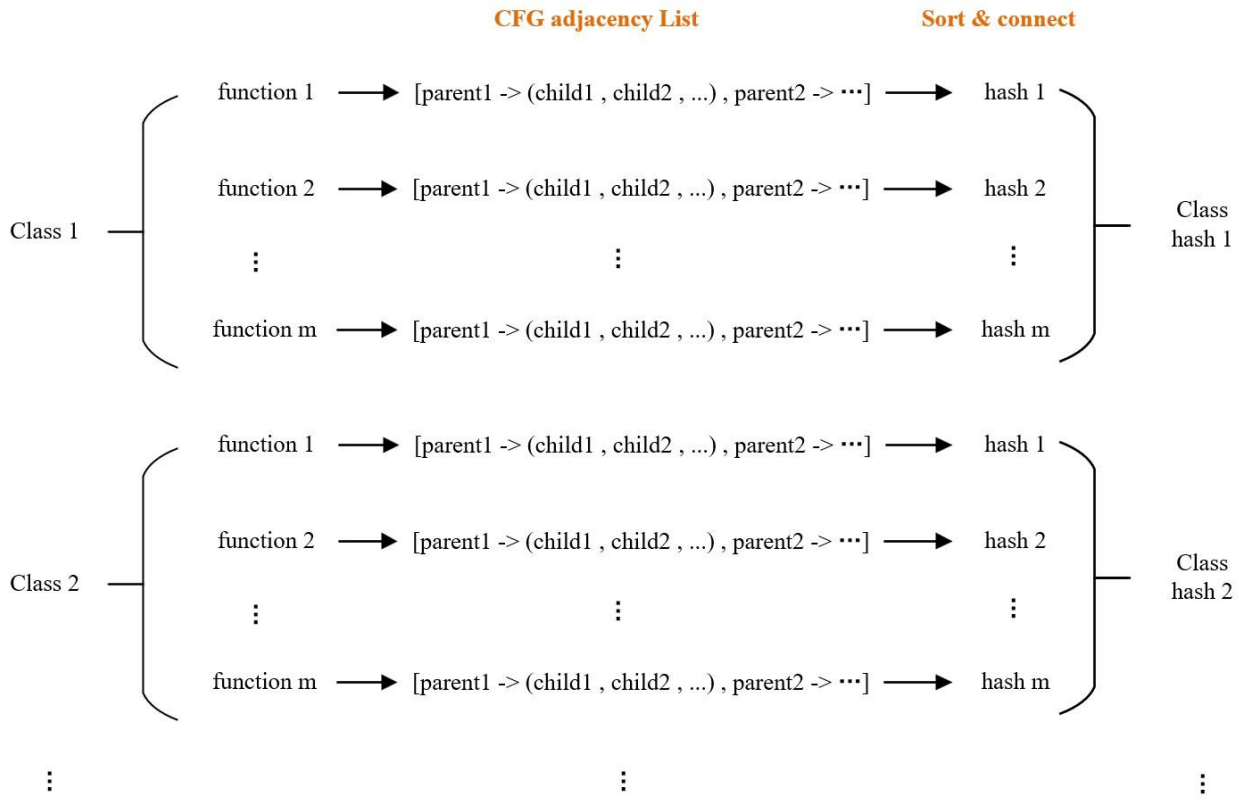
ATVHunter 模块解耦

- 包扁平化混淆（混淆Ⅱ）
  - 包内类文件不同，生成的包签名相似度降低
    - 例如，root.o多了一个d.class
- 类依赖图构建：一个CDG视为一个潜在库实例
  - 设APP类ac_A、ac_B分别为库类lc_A、lc_B的匹配类
  - 3种关系
    - 类继承关系
      - 如果lc_A是lc_B的超类，则ac_A也是ac_B的超类
    - 函数调用关系
      - 函数跨类调用，映射到类间的关系
    - 字段引用关系
      - lc_A中有lc_B定义的字段f
      - 若混淆后f保留在ac_A中，ac_A和ac_B有依赖



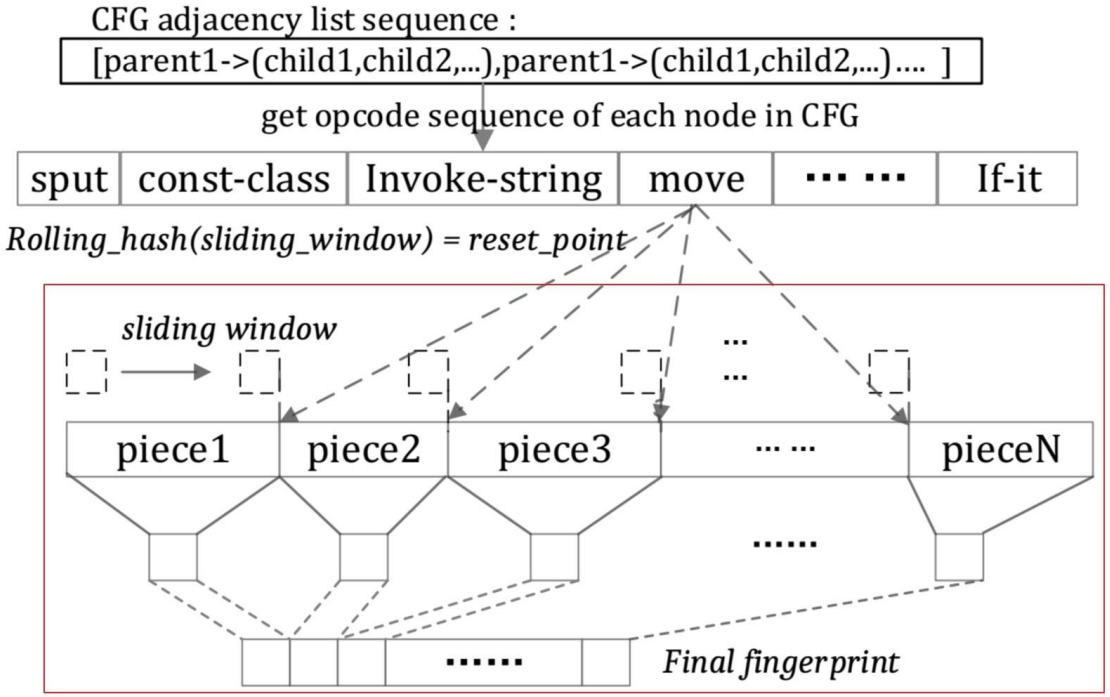
ATVHunter 粗粒度特征提取

- 控制流图（CFG）特征提取
  - 提取每个函数CFG
  - 遍历基本块（节点）
  - 按执行顺序给节点分配唯一序列号
  - 节点n出边最多的子节点序号为n+1
- 邻接表（adjacency list）
  - 一个adjacency list 对应一个函数
  - 函数哈希值进行排序、再次哈希，作为类哈希值
  - 类哈希值排序、再次哈希，作为TPL特征



ATVHunter 细粒度特征提取

- 粗粒度特征可能为不同版本TPL生成相同签名
  - 只在基本块中插入/删除/修改语句
- 基本块操作码（opcode）特征提取
  - 按执行顺序提取每个基本块中的操作码
- 签名生成改进
  - 传统哈希
    - 细微改变会导致哈希值巨大变化
  - 模糊哈希（fuzzy hash）
    - 使用滑动窗口将操作码序列切分成小块
      - 每一小块对最终签名都有一定的贡献
      - 但部分特征发生变化时，不会对签名造成太大影响
        - » 抵抗控制流图随机化混淆（混淆Ⅲ）、死代码消除混淆（混淆Ⅳ）



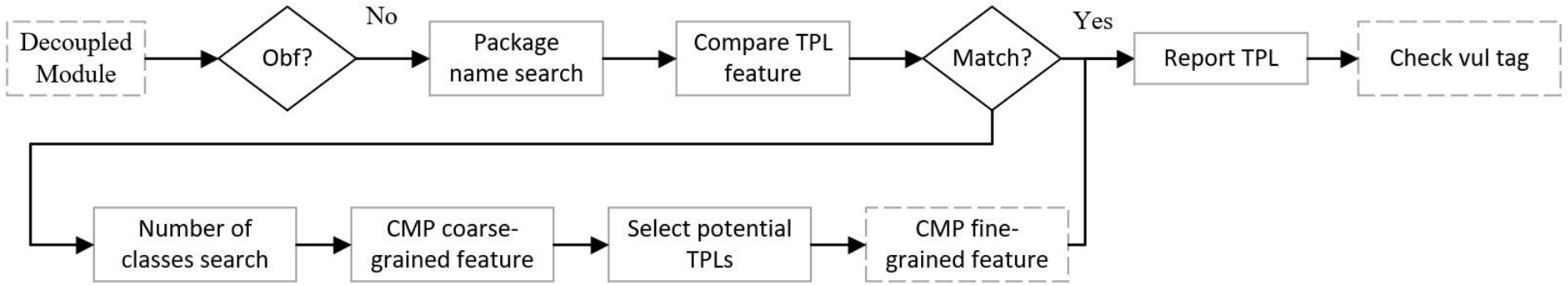
ATVHunter TPL检测

- 基于包名搜索
  - 若包名完全匹配，说明不存在包名称和结构的混淆
- 基于类数量搜索
  - 当一个TPL的类数量只有另一个TPL的40%，不做比较
- 基于TPL特征、粗粒度特征搜索，选择可能的TPL进入下一步

Database structure

package name	Group id	Artifact Id	version	vul tag	TPL feature	coarse-grained feature	fine-grained feature

Detection process



ATVHunter TPL-V识别

- 函数相似度比较
  - 基于编辑距离（edit distance）
    - 2个签名的编辑距离定义为最小编辑操作数
      - 插入、删除、替换
      - 从一个签名变成另一个签名
    - 当 MSS≥0.85（实验得到的阈值），认为两种方法匹配

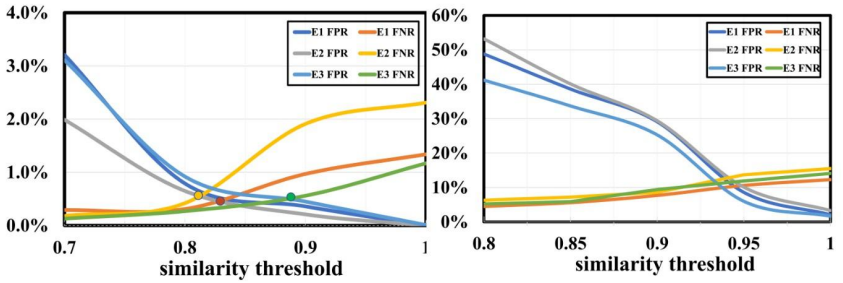
$$MSS(m_a, m_b) = 1 - \frac{d[m_a, m_b]}{max(m, n)}$$

- TPL相似度比较
  - 基于匹配方法的数量
  - 分子必须满足
    - MSS≥0.85
    - 存在MSS=1，即至少有一个完全相同的函数
  - TSS≥0.95（实验得到的阈值）

$$TSS(t_1, t_2) = \frac{M_{|t_1 \cap t_2|}}{M_{|t_1|}}$$

ATVHunter 阈值&有效性&效率实验

- 数据集：基于真实世界APP
- 对比方法：均为相似性对比方法
- 实验结论
  - 阈值实验：权衡误报和漏报选择0.85/0.95
  - 有效性实验
    - 论文方法精确率、召回率、F1值均最高
    - 其他方法
      - 特征粒度过粗导致版本识别精确率低
      - 依赖包结构构造潜在库实例导致召回率低
  - 效率实验
    - 论文方法检测速度最快
      - 基于包名搜索、基于类数量搜索
      - 2阶段识别版本，第一阶段过滤了很多不太可能匹配的TPL



(a) Method-level (b) TPL-level

Tools	Library-level			Version-level		
	Precision	Recall	F1	Precision	Recall	F1
ATVHunter	**98.58%**	**88.79%**	**93.43%**	**90.55%**	**87.16%**	**88.82%**
LibID	98.12%	68.45%	80.64%	68.70%	66.42%	67.54%
LibScout	97.10%	46.65%	63.02%	44.82%	43.50%	44.15%
OSSPoLICE	97.91%	43.39%	60.13%	88.83%	42.25%	57.26%
LibPecker	93.16%	57.82%	71.35%	60.35%	57.67%	58.98%

Tool	ATVHunter	LibID	LibScout	OSSPoLICE	LibPecker
Q1	15.92s	51.43s	30s	33.48s	12168s
Mean	66.24s	59616s	83s	2052.34s	16396s
Median	47.78s	9286s	64s	80.42s	16632s
Q3	90.30s	38300s	100s	226.60s	23292s

ATVHunter 混淆弹性能力实验

- 数据集
  - 使用混淆工具Dasho进行4种类型的混淆
- 实验结论
  - 无混淆、混淆Ⅰ/Ⅱ/Ⅲ，论文方法精确率最高
    - LibID效果骤降的原因
      - dex2jar无法反编译这100个APP
    - 包扁平化混淆（混淆Ⅱ）
      - 论文方法使用类依赖图构造潜在库实例
    - 控制流图随机化混淆（混淆Ⅲ）
      - 论文方法采用CFG + 操作码、模糊哈希
  - 死代码删除混淆（混淆Ⅳ），论文方法次于LibPecker
    - LibPecker：基于类依赖关系生成签名

Tool	No Obfuscation	Obfuscation			
		Renaming	CFR	PKG FLT	Code RMV
ATVHunter	**99.26%**	**99.26%**	**90.13%**	**99.26%**	75.57%
LibID	12.93%	12.93%	0.03%	1.58%	2.49%
LibScout	88.75%	88.75%	18.24%	17.69%	17.69%
OSSPoLICE	85.62%	85.62%	23.04%	39.52%	48.86%
LibPecker	98.79%	98.79%	86.63%	73.56%	**79.28%**

ATVHunter 优劣分析

- 优势
  - 充分分析4种代码混淆技术（尤其是包扁平化混淆），并在方法上做出改进，进行较完备的对比实验

- 劣势
  - 当多个版本TPL相似度过高，会产生误报
  - 如果TPL被导入APP主包中，预处理会将之删除，导致漏报
  - 采用静态分析方法，会遗漏一些动态方法加载的库

应用总结

应用总结

- 应用
  - APP成分分析 + 安全性检查
    - ATVHunter：https://scantist.com/
  - 众多下游任务
    - TPL权限隔离、TPL漏洞检测
    - 克隆检测、恶意软件检测
- 发展前沿
  - 同时检测Java库和Native库
    - 也有研究专注于Native库检测（另一小方向）
  - 考虑更加复杂的代码混淆技术
    - API 隐藏
    - 基于虚拟化的保护

参考文献

[1]https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/
[2]https://reports.exodus-privacy.eu.org/en/trackers/stats/
[3]Zhan X, Liu T, Fan L, et al. Research on Third-Party Libraries in Android Apps: A Taxonomy and Comprehensive Survey[J]. IEEE Transactions on Software Engineering, 2021: 1-32.
[4]Xu J, Yuan Q. Libroad: Rapid, online, and accurate detection of tpls on android[J]. IEEE Transactions on Mobile Computing, 2020, 21(1): 167-180.
[5]Zhan X, Fan L, Chen S, et al. ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. Proceedings of the 43rd International Conference on Software Engineering [C]. Madrid, ES: IEEE, 2021: 1695-1707.

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！