Beijing Forest Studio

北京理工大学信息系统及安全对抗实验中心

准确高效地检测安卓APP中的第三方库

硕士研究生 陆永鑫

2023年07月23日

问题回溯

- 总结反思
  - 讲解要有侧重点，控制好演讲节奏和总时长
  - 增加演讲过程中的互动和趣味性，减少听众的枯燥感

- 相关内容
  - 陆永鑫《Android第三方库检测》
  - 邢继媛《二进制代码开源成分分析》

内容提要

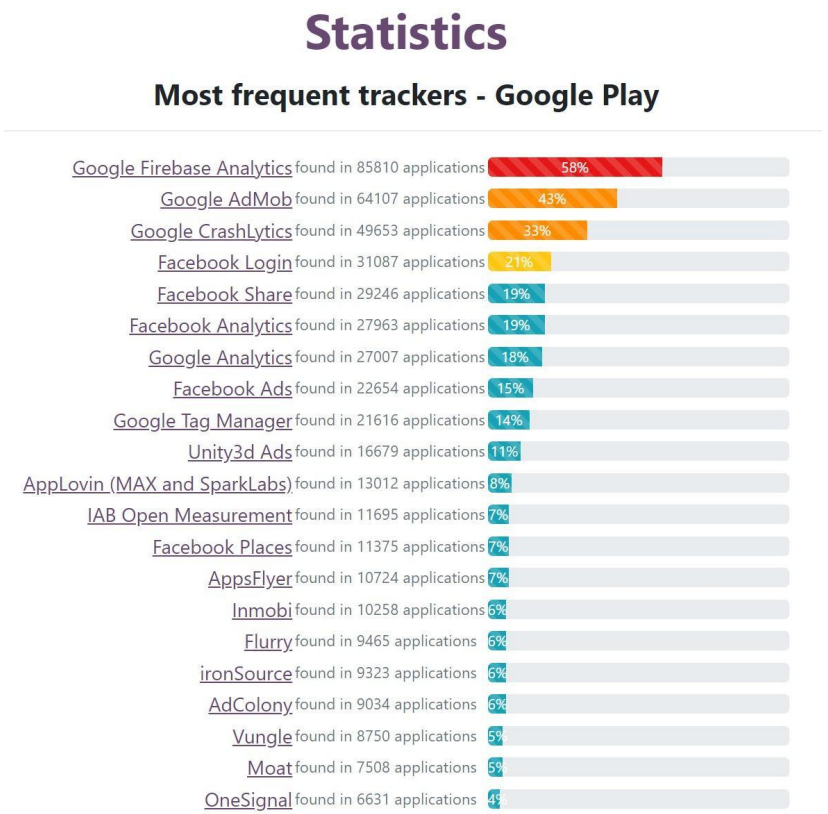
- 背景简介
- 基础概念
  - 主流混淆器与非结构保留混淆
  - 布隆过滤器
  - Kuhn-Munkres算法
- 算法原理
  - LIBLOOM
- 应用总结
- 参考文献

背景简介

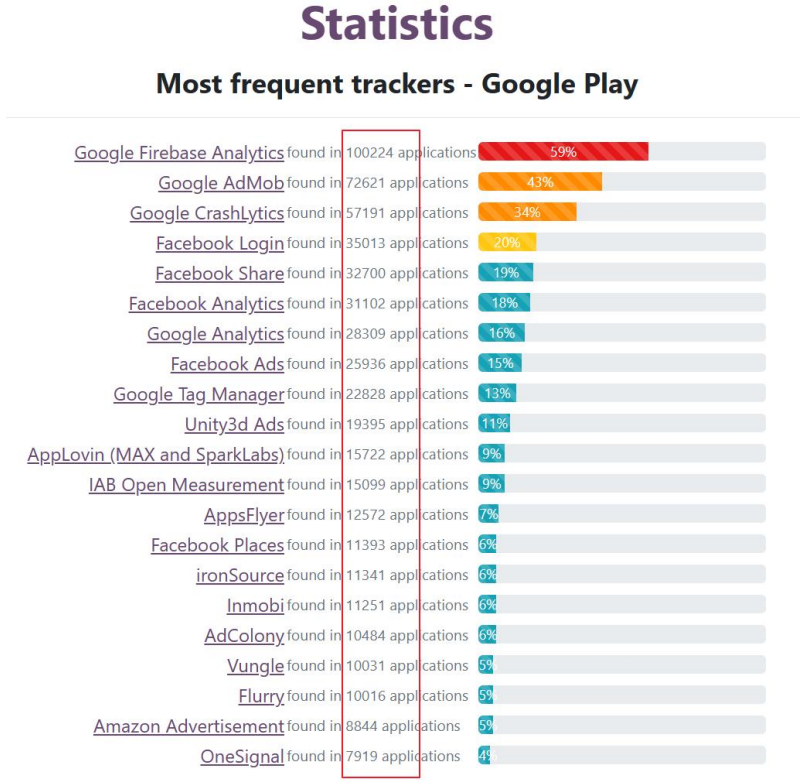
- 预期收获
  - 1. 了解Android第三方库检测的主要困难
  - 2. 理解准确高效检测APP内第三方库的意义
  - 3. 理解布隆过滤器在提升检测效率方面的基本原理
  - 4. 理解基于熵的代码混淆分析的具体方法

背景简介 第三方库

- **Android APP普遍地使用第三方库（Third-Party Library，TPL）**
  - **Exodus Privacy[2]统计Google Play中大约 59% 的APP包含Google Firebase**
  - **研究表明[2]，APP中超过 60% 的代码属于TPL**



» 2022.11 的统计数据

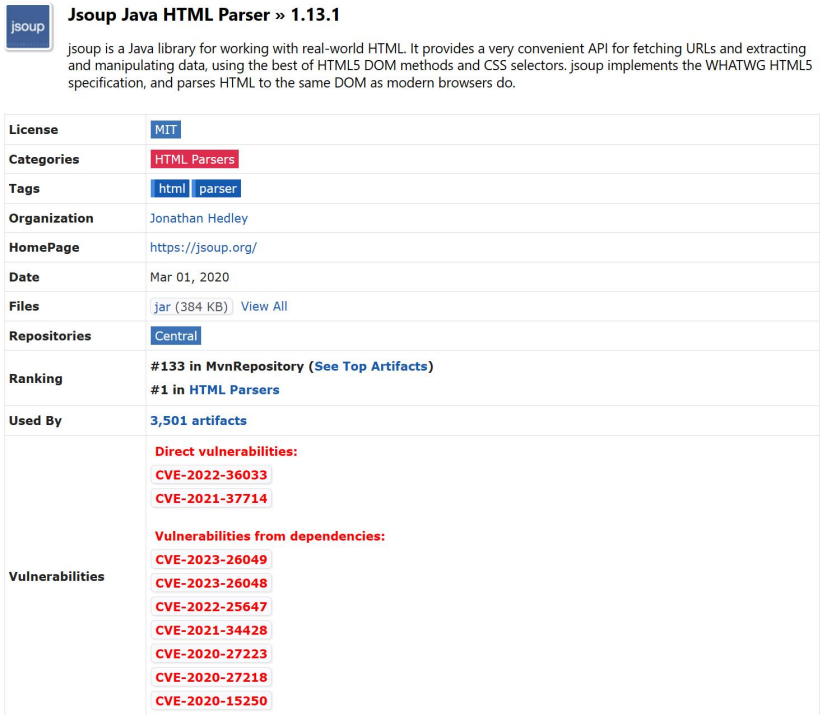


» 2023.07 的统计数据

背景简介 安全隐患

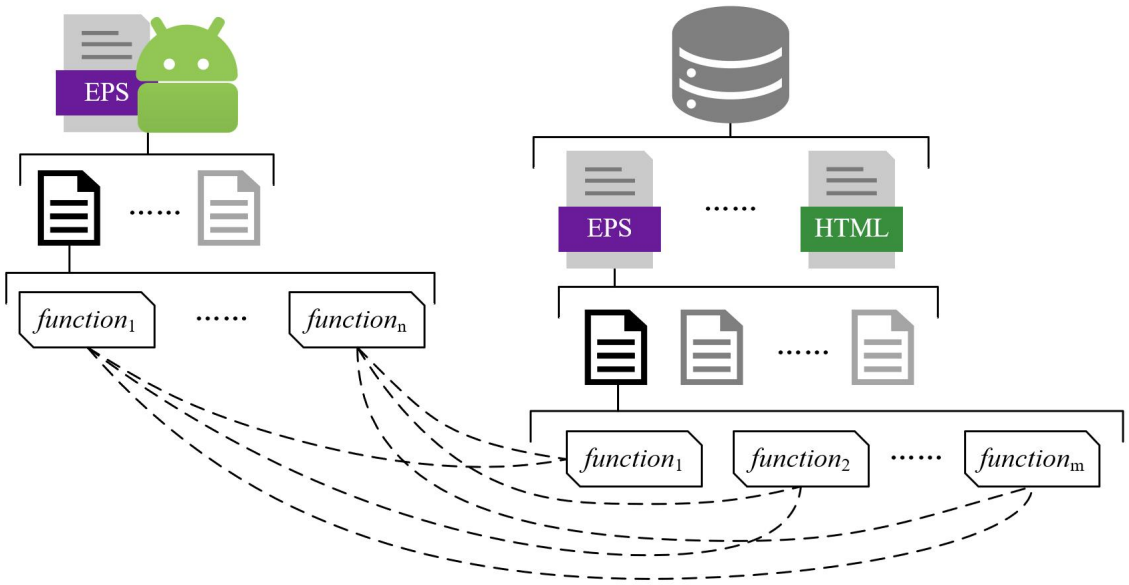
- TPL的广泛使用带来安全问题
  - APP在集成易受攻击的TPL时可能会被动引入安全漏洞
    - 例如，org.jsoup.jsoup（1.15.2 及更早版本）
  - TPL之间复杂的依赖关系会进一步加剧此类安全风险
    - 例如，jsoup-1.14.1 所依赖的 gson-2.7 同样存在漏洞

	Version	Vulnerabilities	Repository	Usages	Date
1.16.x	1.16.1		Central	119	Apr 29, 2023
1.15.x	1.15.4		Central	173	Feb 18, 2023
	1.15.3		Central	417	Aug 24, 2022
	1.15.2	1 vulnerability	Central	97	Jul 04, 2022
	1.15.1	1 vulnerability	Central	102	May 15, 2022
1.14.x	1.14.3	1 vulnerability	Central	392	Sep 30, 2021
	1.14.2	1 vulnerability	Central	245	Aug 15, 2021
	1.14.1	2 vulnerabilities	Central	51	Jul 10, 2021
1.13.x	1.13.1	2 vulnerabilities	Central	575	Mar 01, 2020



背景简介 检测困难

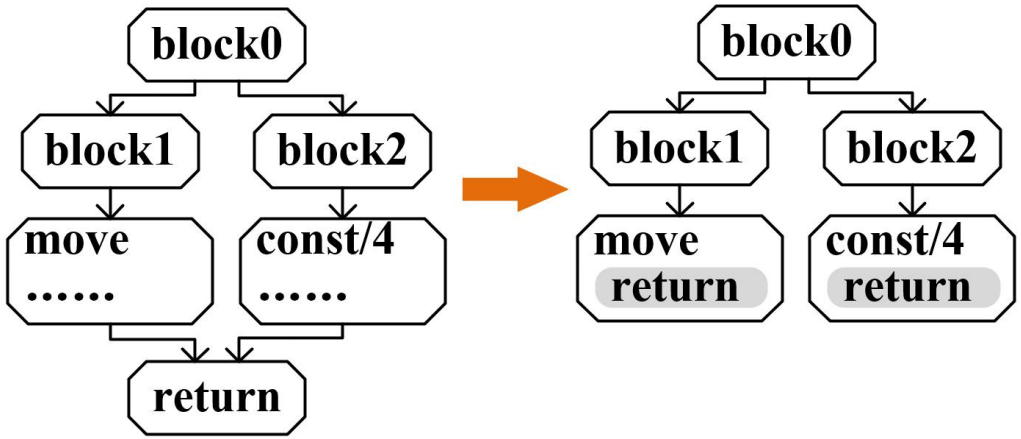
- TPL检测基本任务
  - 输入：APP
  - 输出：APP使用的TPL列表
- 为何需要高效检测?
  - 安全分析人员常面对一大批APPs
  - 数据库中的TPLs通常是数以万计的
- 为何难以高效检测?
  - 为了准确识别具体版本，特征粒度不能太粗
  - 细化特征粒度，导致成对比较数量骤增
  - 假设数据库中有1000个TPLs，每个TPL平均有10个类，每个类平均有5个函数。基于类级别检测的成对比较数量为：(1000×10)2 = 1亿，基于函数级别的成对比较数量为：(1000×10×5)2 =25亿



» 函数粒度下的成对比较

背景简介 检测困难

- 代码混淆技术进一步阻碍了TPL的准确检测
  - 标识符重命名
    - 修改TPL的包名称、类名称、函数名称
  - 控制流随机化
    - 改变函数控制流图
  - 包结构修改
    - 类重打包、包扁平化
  - 死代码删除
    - 删除未使用的方法和类，甚至字段

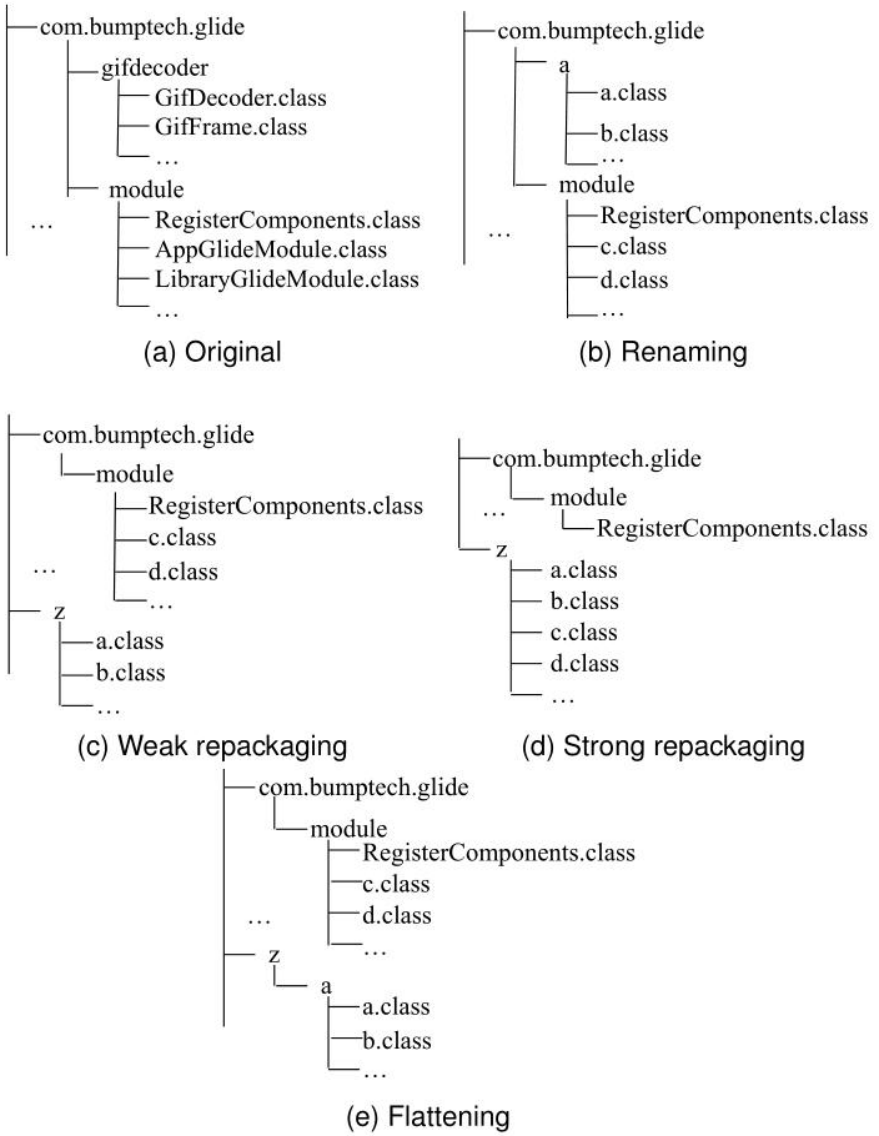


» 简单的控制流随机化示例

基础概念

基础概念 非结构保留混淆

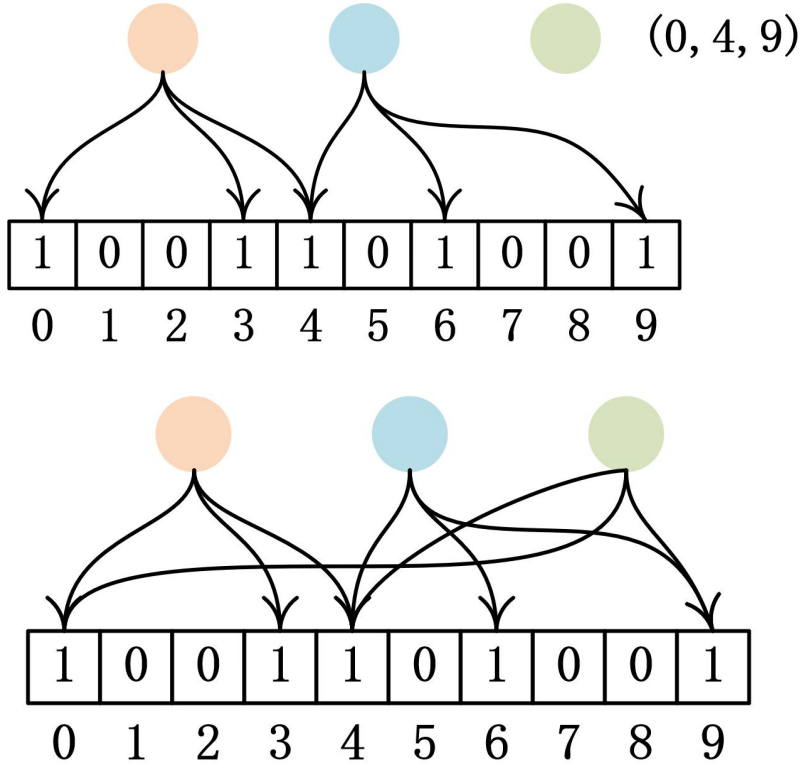
- 现存的 Android APP 主流混淆器
  - ProGuard、DashO
    - 支持类重打包、包扁平化、死代码删除
      - 重打包：将重命名类移动到单个包内
      - 包扁平化：将重命名包移动到单个包内
  - Allatori
    - 支持两种类重打包混淆
      - 弱重打包：仅邻居类均重命名的类可被移动
      - 强重打包：所有重命名类都可被移动
    - 但死代码删除程度较低



» 重命名与包结构混淆示例

基础概念 布隆过滤器

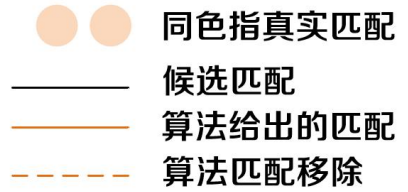
- 布隆过滤器（Bloom Filter）本质上是一个很长的二进制向量
  - 一个布隆过滤器可以相对紧凑地表示一个集合
  - 集合中的每个元素对应向量的 k 个随机位（右下图示例中设为3）
  - 布隆过滤器被广泛用于子集查询
    - 假定右下图绿色元素可能的两组随机位，
      可以看出布隆过滤器有何潜在问题?
    - 重要特性：可能存在误报，但没有漏报
    - 通过一系列推导近似[3]，得到以下参数关系
      - $k = -\frac{lnfpp}{ln2}$、$M = \frac{k \cdot n}{ln2}$
      - k为随机位数量，M为布隆过滤器长度，
        fpp为误报率，n为集合元素数量
      - 降低误报率则需要加大随机位数量

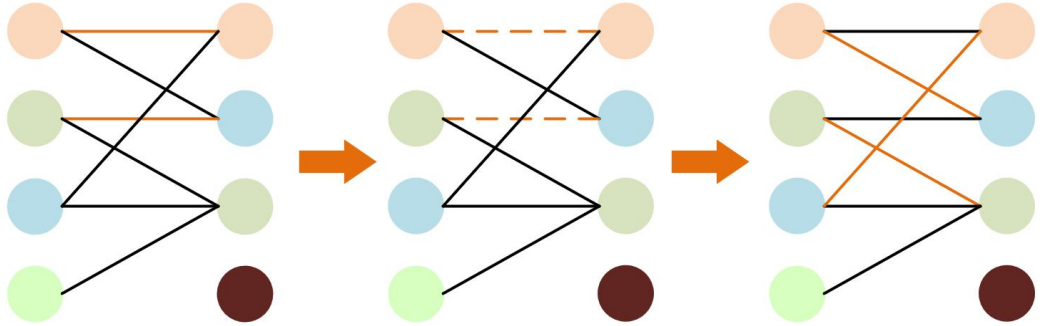


» 布隆过滤器示例

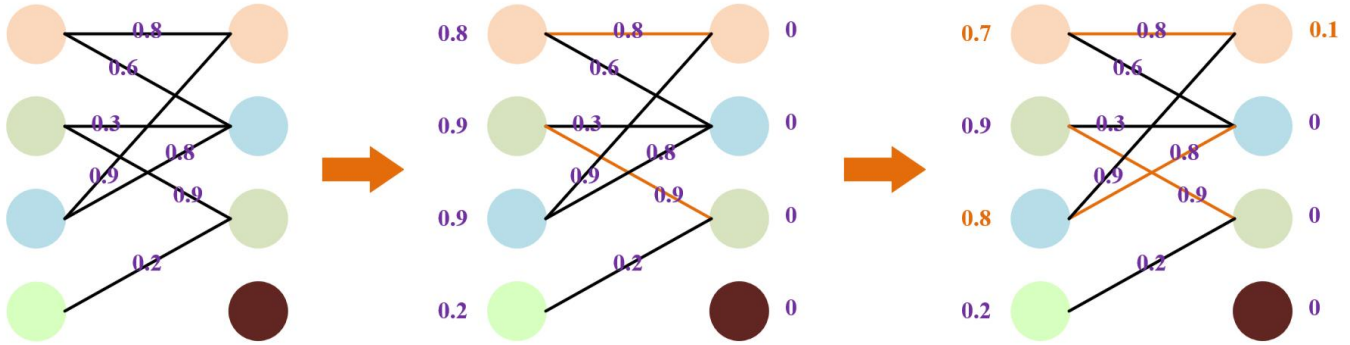
基础概念 Kuhn-Munkres算法

- 匈牙利算法和Kuhn-Munkres算法均被用来解决数据关联问题
  - 多目标跟踪、二分图最大匹配等
- 匈牙利算法
  - 基于递归的匹配过程
- Kuhn-Munkres算法
  - 基于最大边权重给左列顶点赋值
  - 右列顶点值初始化为0
  - 对分居两列的待匹配顶点，仅边权重大于等于两列顶点值之和时可成功匹配
    - 若匹配过程均无法换边，相关顶点值减小（设为减小0.1）





» 匈牙利算法匹配过程示例



» Kuhn-Munkres算法匹配过程示例

LIBLOOM

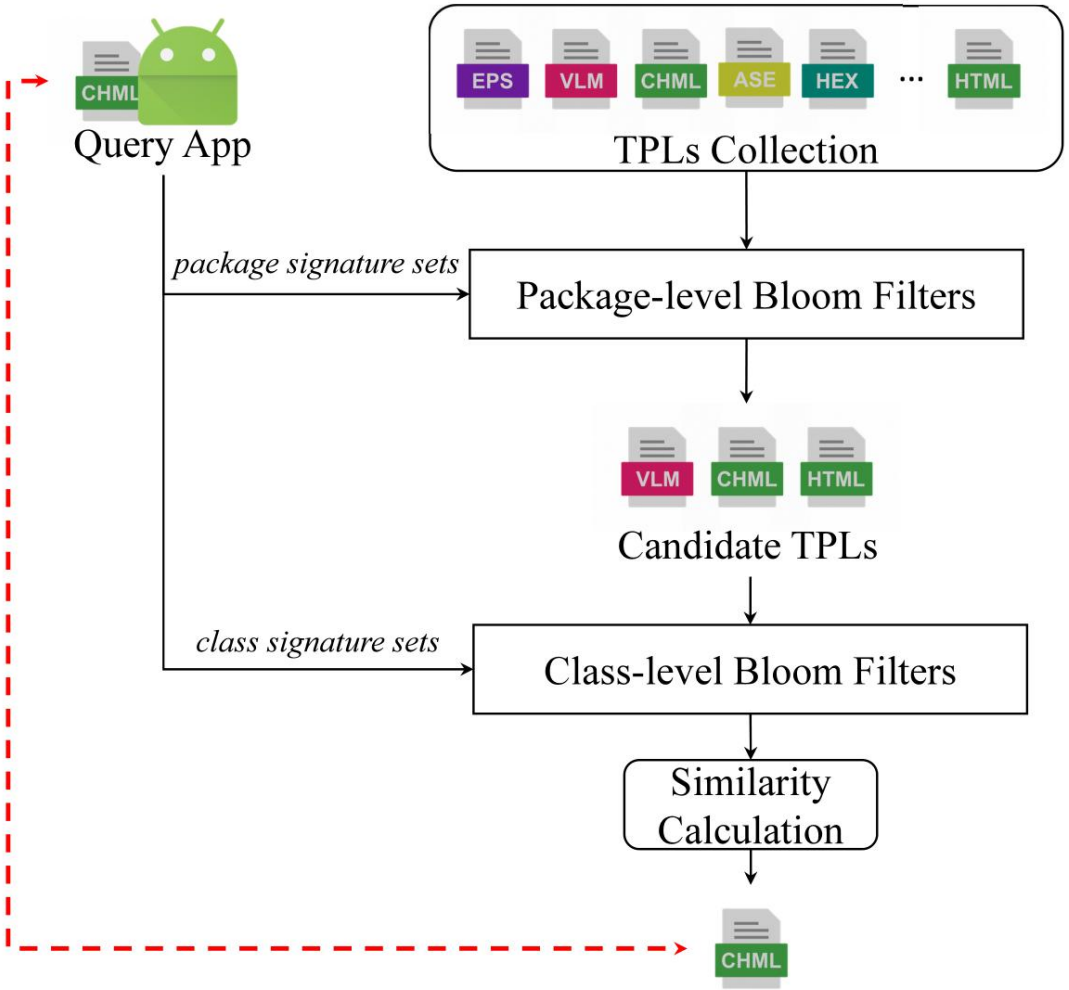
LIBLOOM TIPO PCDL

T	高效准确地检测Android APP中所使用的TPL
I	Android APP、原始TPL文件
P	1. 签名集生成 2. 布隆过滤器构造 3. 包级重叠测量 4. 类级子集查询 5. 相似度计算 & TPL 检测
O	APP中所使用的TPL列表

P	提高检测效率、提高对非结构保留混淆的抵抗能力
C	原始TPL文件可获得
D	如何避免大量无效的成对比较、如何处理复杂混淆
L	SCI 1区期刊（TSE 2022）

LIBLOOM 方法框架

- **Framework**
  - 签名集生成
    - 函数：类功能的实现
    - 字段：类处理的数据
    - 类依赖层次：类在依赖结构中的层次
  - 布隆过滤器构造
    - 包级别签名、类级别签名
  - 包级重叠测量
  - 类级子集查询
  - 相似度计算 & TPL 检测



» LIBLOOM 的检测过程

LIBLOOM 签名集生成

- 3种签名：生成多种类级别签名
  - 函数：使用模糊函数签名
  - 字段：移除变量名，仅保留变量类型
  - 类依赖层次：区分继承*extends*与接口实现*implements*
- 为应对标识符重命名混淆，仅保留系统类型名称、构造函数名称
  - 存在相同签名，映射到向量中也会完全相同，怎么办?

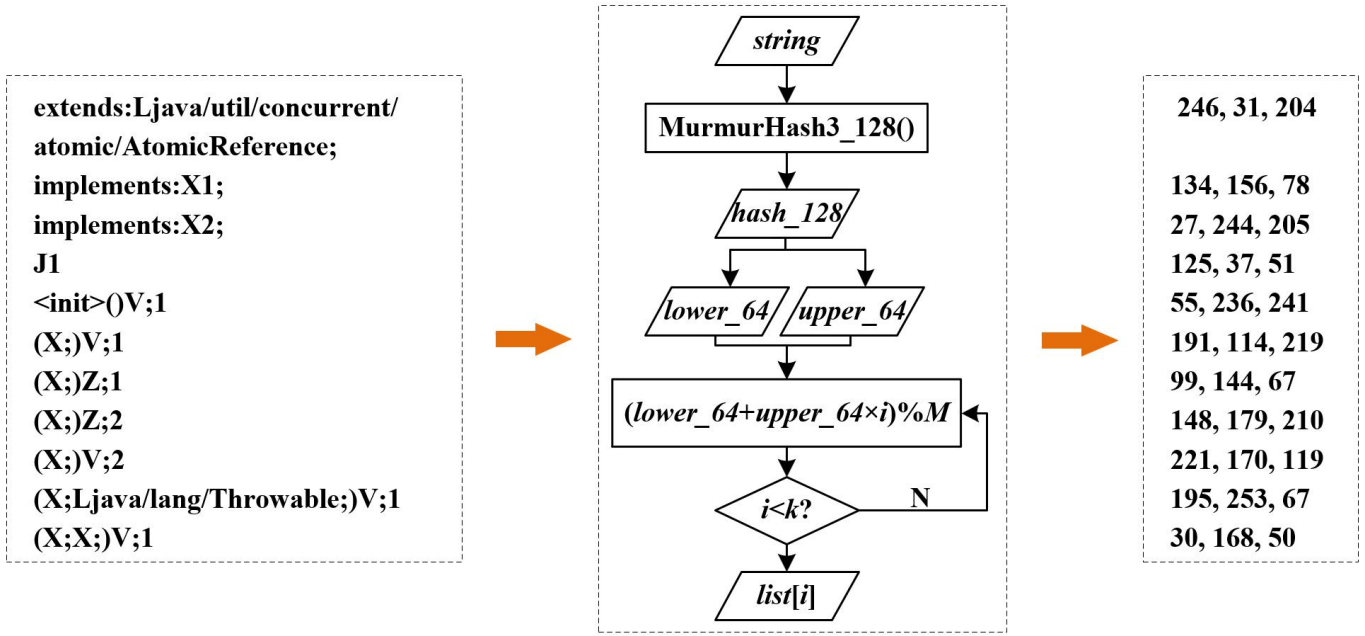
```
extends AtomicReference

implements CompletableObserver
           Disposable
long serialVersionID
EmptyCompletableObserver()
void dispose()
boolean isDisposed()
boolean hasCustomOnError()
void onComplete()
void onError(Throwable)
void onSubscribe(Disposable)
```

```
extends:Ljava/util/concurrent/
atomic/AtomicReference;
implements:X;
implements:X;
J
<init>()V;
(X;)V;
(X;)Z;
(X;)Z;
(X;)V;
(X;Ljava/lang/Throwable;)V;
(X;X;)V;
```

```
extends:Ljava/util/concurrent/
atomic/AtomicReference;
implements:X1;
implements:X2;
J1
<init>()V;1
(X;)V;1
(X;)Z;1
(X;)Z;2
(X;)V;2
(X;Ljava/lang/Throwable;)V;1
(X;X;)V;1
```

LIBLOOM 布隆过滤器构造

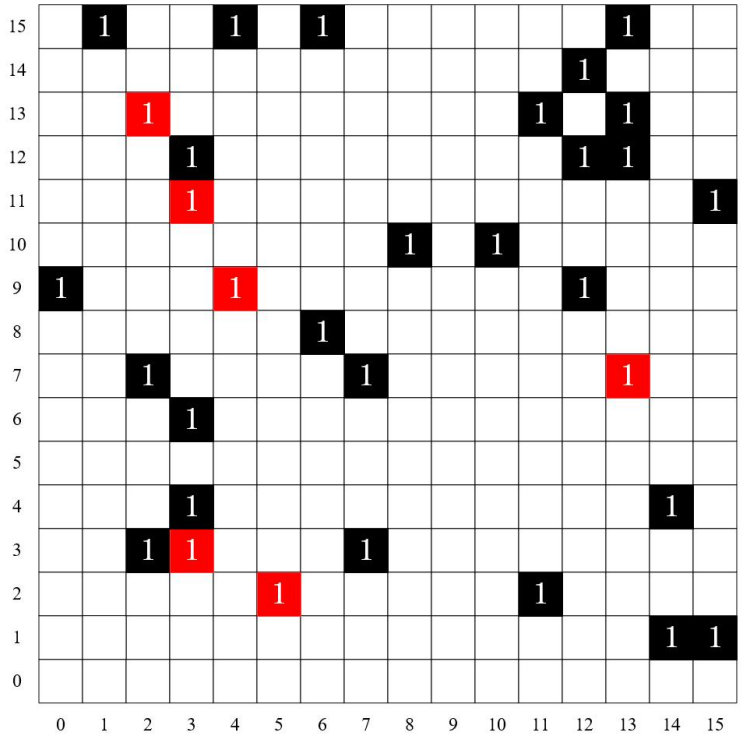
- 2级布隆过滤器
  - 为类 c（包 p）构造一个大小为 M_c（M_p）的类（包）级布隆滤波器 BF_c
    - 包签名集包含了 M_p 中所有类 $\{c_i | c_i \in p\}$ 的类签名
- 无偏哈希：使用无偏哈希进行签名映射，减少哈希冲突，进一步区分不同类
  - MurmurHash3 哈希算法，通过雪崩测试和卡方校验
    - 输入的微小变化会导致输出发生显著变化、哈希值均匀落入哈希空间



LIBLOOM 布隆过滤器构造

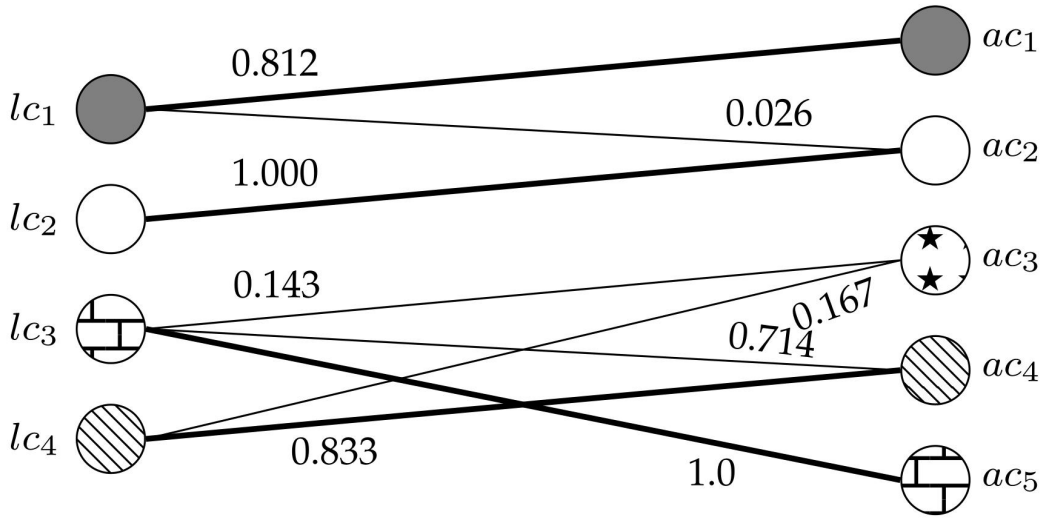
- 参数选取：$k = -\frac{lnfpp}{ln2}$、$M = \frac{k \cdot n}{ln2}$
- 类是LIBLOOM的基本分析单元，对类签名集影响最大的是死代码删除混淆
  - 类中的函数或字段可能被删除，混淆后的类签名集是原始签名集的子集
  - 通过严格的类级子集查询，可以在混淆情况下准确进行类匹配

extends:Ljava/util/concurrent/	246, 31, 204
atomic/AtomicReference;	
implements:X1;	134, 156, 78
implements:X2;	27, 244, 205
J1	125, 37, 51
<init>()V;1	55, 236, 241
(X;)V;1	191, 114, 219
(X;)Z;1	99, 144, 67
(X;)Z;2	148, 179, 210
(X;)V;2	221, 170, 119
(X;Ljava/lang/Throwable;)V;1	195, 253, 67
(X;X;)V;1	30, 168, 50



LIBLOOM 包级重叠测量&类级子集查询

- 包级重叠测量：当且仅当包重叠率高于重叠阈值（0.85），进行类级子集查询
  - $overlap\_ratio(lp, ap) = \frac{BF_{lp} \cdot BF_{ap}}{\min(|BF_{lp}|, |BF_{ap}|)} > 0.85$
  - 阈值过小很难消除不匹配的包对，效率降低；过大可能遗漏匹配包导致召回率降低
- 类级子集查询
  - 满足$BF_{ac} \& BF_{lc} = BF_{ac}$时计算$class_inclusion_score(lc, ac) = \frac{|BF_{ap}|}{|BF_{lp}|}$
  - 类间可信映射生成
    - ap（lp）中存在多个$ac(lc)$
    - 存在同一ac匹配多个lc的情况
    - 采用 Kuhn-Munkres 算法查找类之间的可信映射，进一步生成候选包匹配A

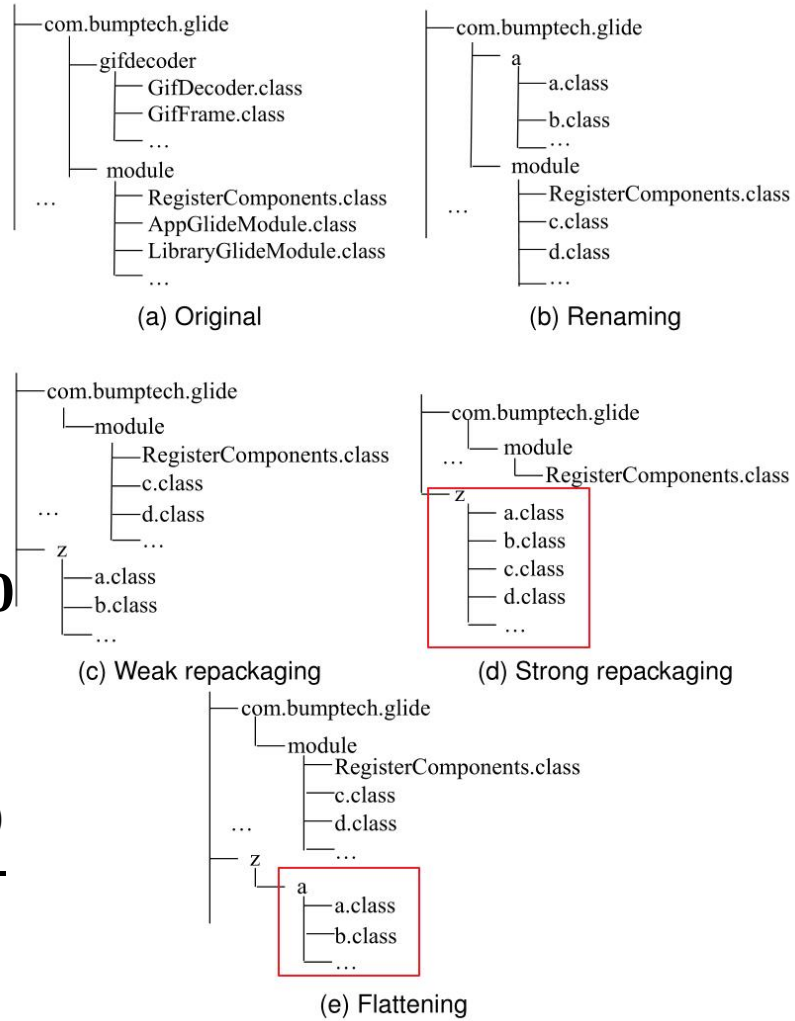


LIBLOOM 相似度计算

- 相似度计算：当且仅当相似度大于等于阈值，认为APP中存在对应的TPL
  - $sim\_score(lib, app) = \frac{\sum_{(lp,ap)\in A} |\text{max\_match}(lp,ap)|}{\#classes\ in\ lib} \geq 0.6$
- 基于熵的启发式方法：若相似度低于阈值，判断是否存在混淆包
  - 面向重打包的熵：会产生一个大的单一目标包，包含许多来自不同包的直接类
    - $H_f \sum_{t\in RS^f} p(t) log_2 \frac{1}{p(t)}, \text{where } p(t) = \frac{\sum_{c_i} cout(t\in c_i)}{\sum_{c_i} |c_i|}$
  - 面向包扁平化的熵：目标包涉及许多最初驻留在不同父包中的子包
    - $H_f \sum_{t\in RS^f} p(t) log_2 \frac{1}{p(t)}, \text{where } p(t) = \frac{\sum_{sp_i}\sum_{c_{ij}} cout(t\in c_{ij})}{\sum_{sp_i}\sum_{c_{ij}} |c_{ij}|}$
  - 修正后的相似度计算：若C为重打包的候选包匹配，则ap允许重复匹配
    - $sim\_score(lib, app, C) = \frac{\sum \max\left(|\text{max\_match}(lp,ap)|_{(lp,ap)\in C,\ name_{lp}=name_{ap}}, \frac{|\text{max\_match}(lp,ap)|}{name_{lp}=name_{ap}}\right)}{\#classes\ in\ lib} \geq 0.6$

LIBLOOM 基于熵的启发式方法

- $H_f \sum_{t \in RS^r} p(t) log_2 \frac{1}{p(t)}$, where $p(t) = \frac{\sum_{c_i} cout(t \in c_i)}{\sum_{c_i} |c_i|}$
  - RS^r为潜在重打包目标包P^r的t列表
  - t为类c_i的类签名中APP定义类出现的次数
  - 直接理解：设原始两个包均有且仅有5个类，每个类对应的t均为1
    - $5 \times \frac{1}{5} \times log_2 \frac{1}{\frac{1}{5}} = log_2 5 < 10 \times \frac{1}{10} \times log_2 \frac{1}{\frac{1}{10}} = log_2 10$
    - $\frac{4}{5} \times log_2 \frac{1}{\frac{4}{5}} + \frac{1}{5} \times log_2 \frac{1}{\frac{1}{5}} < 5 \times \frac{1}{5} \times log_2 \frac{1}{\frac{1}{5}}$
- $H_f \sum_{t \in RS^f} p(t) log_2 \frac{1}{p(t)}$, where $p(t) = \frac{\sum_{sp_i} \sum_{c_{ij}} cout(t \in c_{ij})}{\sum_{sp_i} \sum_{c_{ij}} |c_{ij}|}$
  - RS^f为潜在包扁平化目标包P^f的子包的t列表
  - 直接理解：与上类似，但扩展到两级包结构（包-子包）



LIBLOOM 实验数据集

- 开源数据集（I）
  - 100个来自F-Droid存储库的开源APPs
  - 使用3个主流混淆器共7种混淆策略
    - 现实世界中常用的默认混淆设置
- 闭源数据集（II）
  - 221个来自Google Player的闭源APPs
- 大规模数据集（III）
  - 2552个来自小米应用商城的APPs
- 实验设计
  - 参数实验（I部分）：确定布隆过滤器参数、包重叠阈值
  - 有效性实验（I、II）：比较对比方法、消融方法的Precision/Recall/F1
  - 效率实验（I、II、III）：比较对比方法、消融方法的检测用时

» 实验数据集汇总

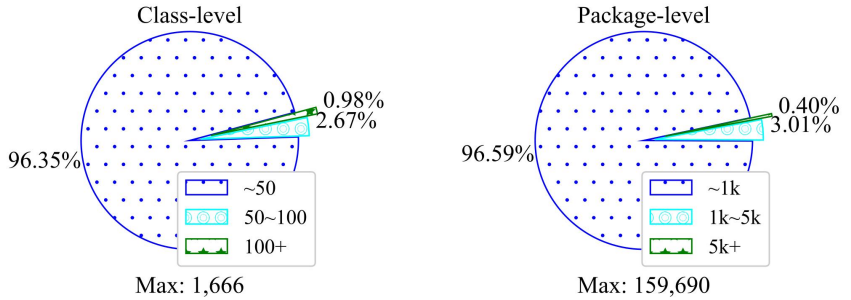
	open source	closed source	large scale
#APPs	100×7	221	2,552
#TPLs	349	59	515

» 闭源数据集混淆情况

混淆器	混淆类型	包结构混淆	死代码删除
ProGuard	PGD	无	√
	PGR	重打包	√
	PGF	包扁平化	√
Allatori	ALW	弱重打包	√
	ALS	强重打包	√
DashO	DOR	重打包	√
	DOF	包扁平化	√

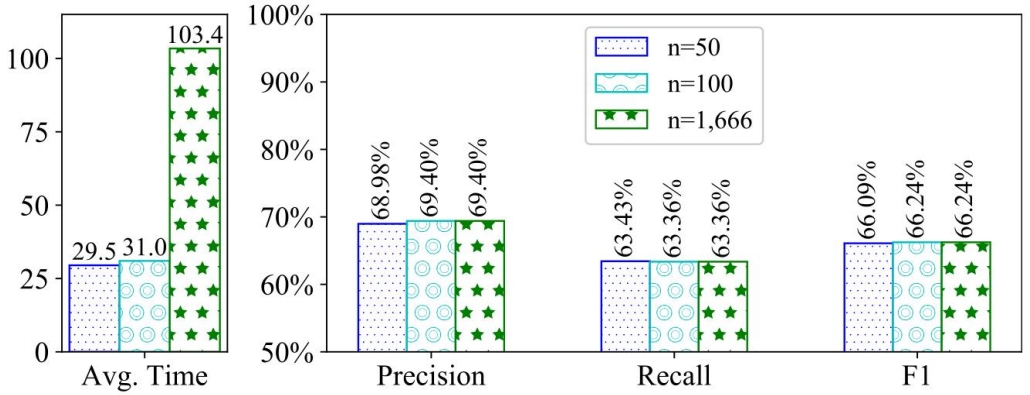
LIBLOOM 参数实验

- 类级布隆过滤器参数
  - 选择开源数据集中的10×7个APPs
  - 移除包级重叠测量模块（LIBLOOM-p）
  - 简单统计确定调参范围
  - 理论约束下调参
    - $k = -\frac{lnfpp}{ln2}$、$M_c = \frac{k \cdot n}{ln2}$
  - 结果
    - 不同fpp下Precision/Recall/F1相同，因为布隆过滤器本身的fpp小于0.001，固定$k = 3$
    - 权衡效率与Precision/Recall/F1，选择$n = 50$，故$M_c \approx 216$，固定为256



» 类（包）签名集中签名数量统计

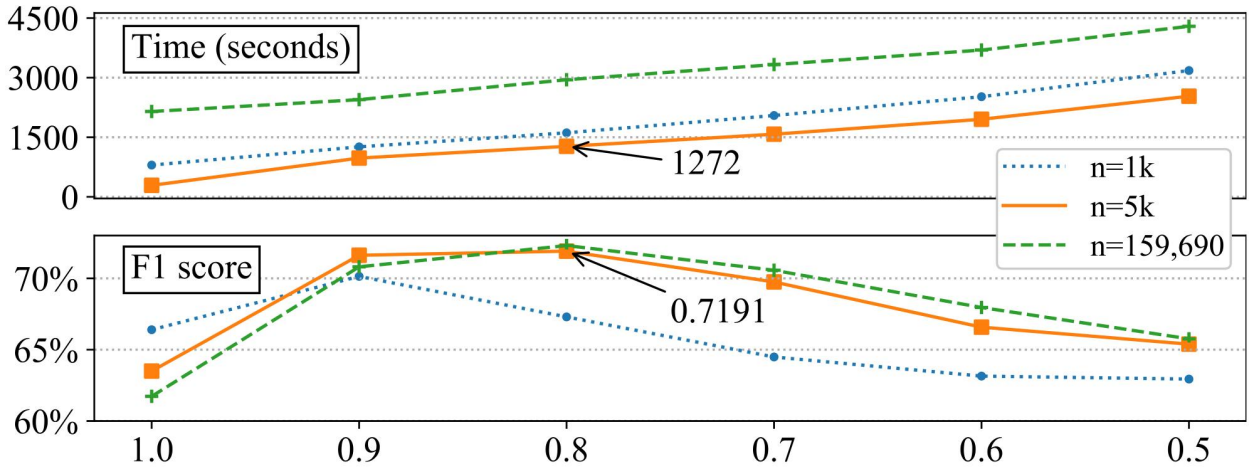
	fpp=0.1	*fpp*=0.01	*fpp*=0.001
k	3	7	10
Precision/Recall/F1	69.40% / 66.36% / 66.24%		
Avg.Time(s)	103.4	205.7	258.7



» 类级布隆过滤器调参选择与结果

LIBLOOM 参数实验

- 包级布隆过滤器参数&包重叠阈值
  - 固定$k = 3$和$M_c = 256$
  - 使用LIBLOOM进行参数选择
  - 包重叠阈值越小，耗时越长
  - $n = 5000$时检测速度最快
    - 而非$n = 1000$，潜在原因可能是包级别哈希冲突加剧导致包级重叠测量有效性降低
    - $M_p = \frac{k \cdot n}{ln2} \approx 21640$
  - 包重叠阈值固定为0.8时，F1分数达到峰值



» 包级布隆过滤器调参选择与结果

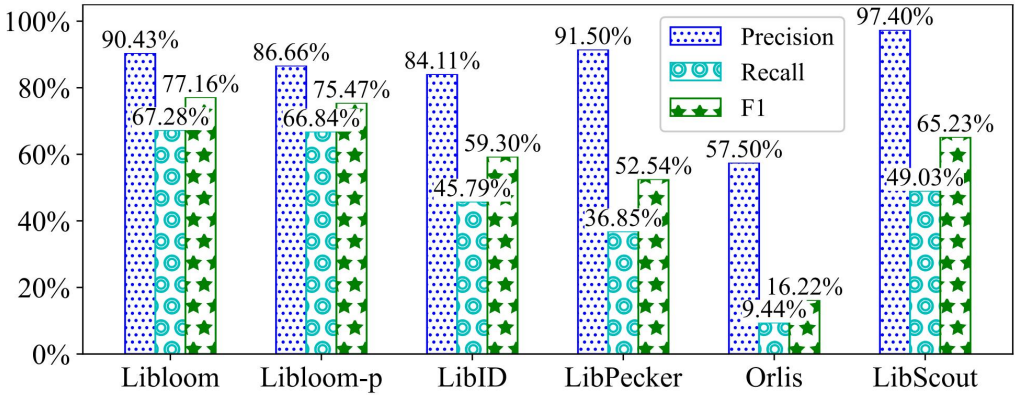
LIBLOOM 有效性实验

- 开源数据集下的有效性实验结果
  - LIBLOOM在所有类型非结构保留混淆APPs检测方面优于其他工具
  - 对于Allatori混淆器混淆的APPs取得显著效果
    - 没有大幅死代码删除，基于熵的启发式方法可以有效地识别重打包或包扁平化的目标包
  - 面对ProGuard和DashO的更多的死代码删除
    - 通过子集查询
  - LIBLOOM-p 比 LIBLOOM 的召回率更高
    - 说明包级重叠测量模块仍会过滤掉部分匹配包
    - 但包级重叠测量模块极大提升精度

(%)		LIBLOOM	LIBLOOM-p	LibID	LibPecker	Orlis	LibScout
PGD	P	97.30%	75.63%	35.20%	57.41%	61.75%	67.29%
	R	49.02%	49.75%	45.65%	28.19%	11.91%	15.06%
	F1	65.19%	60.02%	39.75%	37.81%	19.97%	24.61%
PGR	P	91.59%	59.81%	14.83%	45.96%	60.50%	89.08%
	R	36.29%	39.40%	39.39%	16.34%	10.41%	7.98%
	F1	51.98%	47.51%	21.55%	24.11%	17.76%	14.65%
PGF	P	88.42%	54.35%	29.33%	48.83%	60.53%	85.16%
	R	45.73%	45.96%	34.37%	18.03%	10.99%	9.12%
	F1	60.28%	49.80%	31.65%	26.34%	18.60%	16.48%
ALW	P	91.04%	79.50%	16.67%	49.79%	72.04%	48.79%
	R	89.75%	90.67%	83.39%	42.94%	13.22%	10.55%
	F1	90.39%	84.72%	27.79%	46.11%	22.34%	17.35%
ALS	P	73.06%	59.20%	11.33%	0.00%	-	0.00%
	R	90.73%	91.31%	27.57%	0.00%	0.00%	0.00%
	F1	80.94%	71.83%	16.06%	-	-	-
DOR	P	87.62%	58.00%	12.96%	42.09%	-	26.67%
	R	33.64%	35.86%	0.45%	11.96%	0.00%	0.52%
	F1	48.62%	44.32%	0.87%	18.63%	-	1.02%
DOF	P	90.37%	49.79%	14.71%	45.21%	100.0%	22.22%
	R	46.02%	46.35%	0.47%	12.46%	0.25%	0.65%
	F1	60.98%	48.01%	0.91%	19.54%	0.50%	1.26%
Avg.F1		65.48%	58.03%	19.80%	24.65%	11.31%	10.70%

LIBLOOM 有效性实验

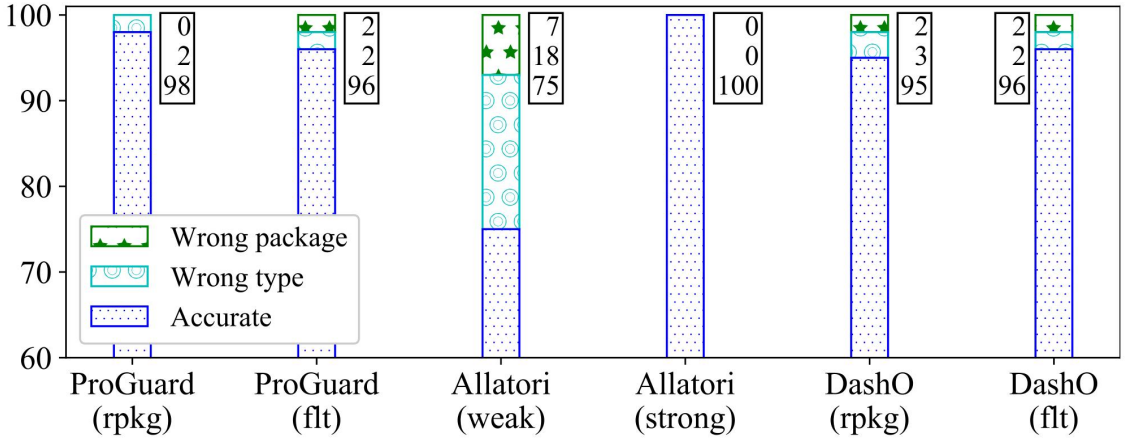
- 闭源数据集下的有效性实验结果
  - LIBLOOM 的精度略低于LibPecker和LibScout，但召回率和F1分数最高
  - 与 LIBLOOM-p相比，LIBLOOM在检测闭源APPs方面不会失去准确性
  - 由于闭源，混淆技术未知，通过手动识别30个闭源APPs的混淆技术
    - 多数是死代码删除混淆、重打包或包扁平化混淆及其的组合，证明开源数据集贴近真实世界APP
    - LibID和Orlis无法在这30个APPs上运行
    - LIBLOOM通过检测更多的TPL，获得了最高的F1分数



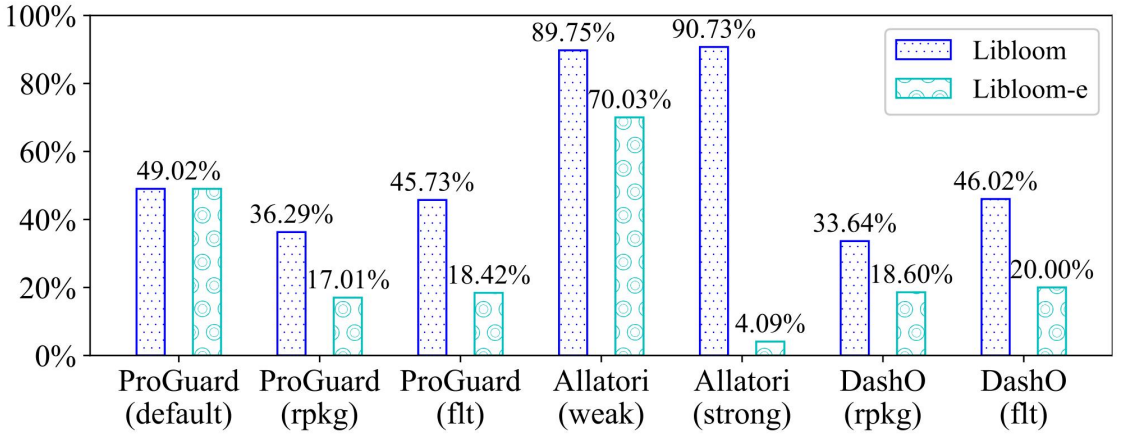
%	LIBLOOM	LibPecker	LibScout
Precision	87.69	90.38	100.00
Recall	34.20	19.09	23.48
F1	49.21	31.52	38.02

LIBLOOM 有效性实验（消融）

- 基于熵的启发式方法有效性验证
  - 基于开源数据集中的6种包结构混淆
  - 将目标包识别结果分为3类
    - 正确、错误类型、错误包
  - 错误包大多是由最初包含大量类或子包的复杂包引起的，因为其原始熵就较大
  - 尽管Allatori的弱重打包（ALW）上只有75 个被准确识别，但对检测效果的影响不大因为该混淆类型只打破了少量包-类结构
  - 移除基于熵的启发式方法（LIBLOOM-e）
    - 结果证明该方法可以帮助LIBLOOM在非结构保留混淆下检测到更多TPLs



» 包识别结果（纵坐标为包数量）

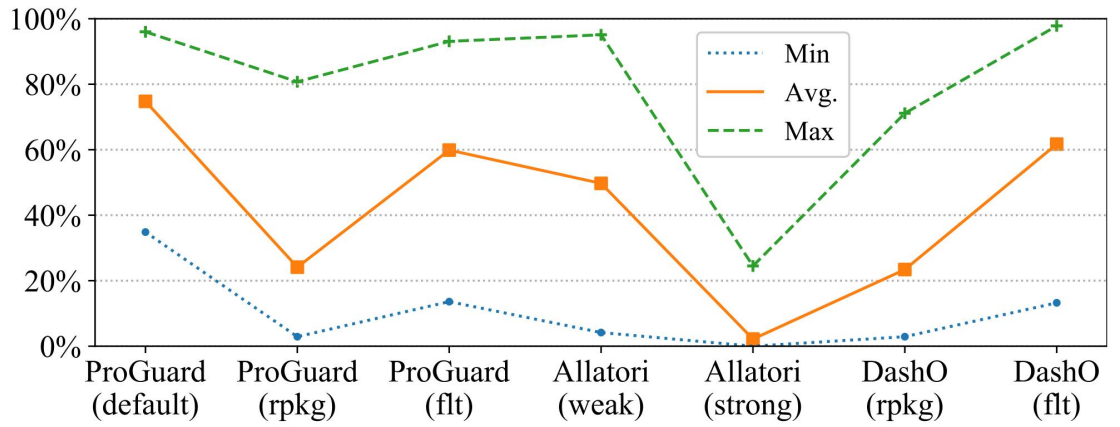


» LIBLOOM与LIBLOOM-e的召回率对比

LIBLOOM 效率实验

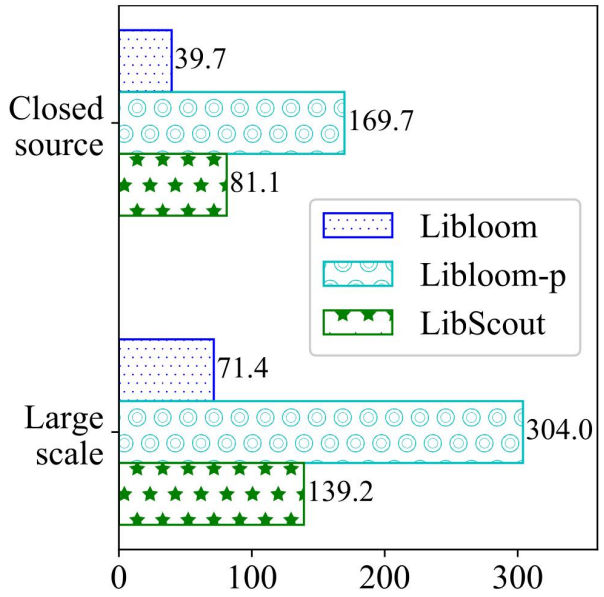
- 开源数据集下的效率实验结果
  - LIBLOOM和LibScout检测效率更高，但后者是因为过滤了更多的包，然而其包过滤算法效果不佳，导致召回率很低
  - 相比于LIBLOOM-p，LIBLOOM的检测用时减少了2~10倍（除了ALS）
  - 右下图显示了被包重叠测量模块排除的TPL数量
    - 在部分APPs甚至能过滤掉95%的TPLs
    - ALS情况下过滤掉最少的TPLs，原因在于“体积巨大”的目标包使得许多不相关包重叠，并将大量类留给类级子集查询

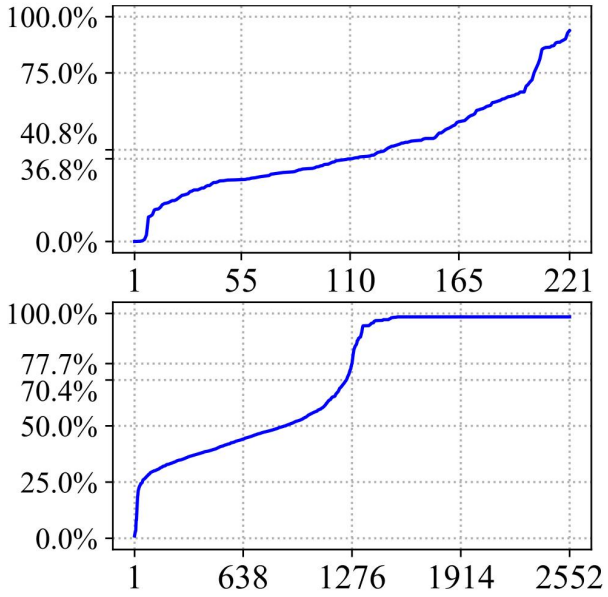
	LIBLOOM	LIBLOOM-p	LibID	LibPecker	Orlis	LibScout
PGD	99	676	217h	66h	42h	611
PGR	637	1591	24h	67h	16h	612
PGF	140	1508	233h	66h	25h	610
ALW	405	1080	185h	70h	32.7h Recall:0%	539
ALS	4124	5430	75h	69h Recall:0%	290 Recall:0%	548 Recall:0%
DOR	690	1745	2037	41h	1717	837
DOF	146	1644	132h	10h	1684	988



LIBLOOM 效率实验

- 闭源、大规模数据集下的效率实验结果
  - LibID、LibPecker、Orlis在上述实验中已被证明效率低下，此处不再比较
  - 下左图显示检测单个APPs需要的时间（单位：秒）
    - LIBLOOM比LIBLOOM-p、LibScout分别快3倍、1倍
  - 下右图显示了每个APP检测时LIBLOOM过滤的TPL数量
    - 在闭源数据集中，过滤的TPL数量平均值为40.8% (877/2114)
    - 在大规模数据集中，过滤的TPL数量平均值为70.4% (8292/11648)
    - 这是因为后者TPL数据库中的“无关”TPLs更多，这也是大规模检测场景的特点之一





LIBLOOM 易受攻击的TPL检测

- 在大规模数据集上评估LIBLOOM检测易受攻击TPL的有效性
  - 检查4个易受攻击TPL
    - retrofit（版本：2.0~2.4.0，CVE-2018-1000850）
    - okhttp（版本：3.1.2之前，CVE-2016-2402）
    - gson（版本：2.8.9之前，CVE-2022-25647）
    - fastjson（版本：1.2.83 之前，CVE2022-25845）
  - 对比LIBLOOM和LibScout的检测结果，并手动检查真实存在漏洞的APP数量

%	retrofit	okhttp	gson	fastjson
LIBLOOM	392	33	997	514
LibScout	359	33	899	481

LIBLOOM 优劣分析

- 优势
  - 从基础理论的基本概念出发，对应用问题进行数学建模
    - 将检测过程抽象为集合查询问题，引入布隆过滤器加快检测速度
    - 通过统计分析手段尝试从定量角度分析代码混淆带来的改变

- 劣势
  - 由于较粗的特征粒度，LIBLOOM版本级别检测精度受到限制
    - 简单地细化特征粒度会导致检测效率降低，且可能无法应对更细粒度的代码混淆
  - 采用严格的子集查询，基于膨胀的代码混淆可能会降低LIBLOOM的效果
    - 添加随机字段、创建重载函数等
  - 难以应对死代码删除混淆程度很高的情况

应用总结

应用总结

- 应用
  - TPL安全性检查、TPL权限隔离、TPL漏洞检测
  - APP克隆检测、恶意软件检测

- 发展前沿
  - 针对高程度死代码删除混淆，尝试引入多维度或多尺度特征，并基于代码簇检测
  - 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://reports.exodus-privacy.eu.org/en/trackers/stats/

[2]Wang H, Guo Y. Understanding third-party libraries in mobile app analysis. Proceedings of the 39th International Conference on Software Engineering Companion [C]. Buenos Aires, Argentina: IEEE, 2017: 515-516.

[3]Bloom B H. Space/time trade-offs in hash coding with allowable errors[J]. Communications of the ACM, 1970, 13(7): 422-426.

[4]Huang J, Xue B, Jiang J, et al. Scalably Detecting Third-Party Android Libraries With Two-Stage Bloom Filtering[J]. IEEE Transactions on Software Engineering, 2022, 49(4): 2272-2284.

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！