抗 ProGuard 的 Android 第三方库检测方法

- 技术报告 -

Janus 研究中心
上海🄵众信息技术有限公司

英成果
<anonymous@pwnzen.com>

唐祝寿
<anonymous@pwnzen.com>

placeholder
<anonymous@pwnzen.com>

本项目无人资助。
文档版本：V1.0, 文档编译时间：December 20, 2019，

目录

序言

实验表明，*23.22%* 的 *Android* 移动应用被 *ProGuard* 处理，*16.25%* 使用了压缩和优化选项，传统的方法无法检测这类移动应用中的第三方库。为检测 *Android* 应用第三方库，本节首先分析现有检测方法的缺陷，为弥补此缺陷，本文提出了一种通用的检测方法。。

本报告是 *Janus* 研究中心在 *Android* 应用第三方库检测工作的一部分内容，报告标题、内容都将随着研究的不断深入，内容的不断扩展而变化。

Chapter 1

简介

为检测 Android 应用第三方库，本节首先分析现有检测方法的缺陷，为弥补此缺陷，本文提出了一种通用的检测方法。具体来讲，(i) 在第三方库语料库建立阶段，研究了在代码混淆、压缩和优化的情况下，第三方库中的不变特征，在此基础上，使用 Android 应用包结构和包的内容来做为第三方库的特征，并建立了第三方库语料库。来自数百万个 Android 应用的检测结果表明，本文提出的检测方法能成功识别出平均 12 种版本的 800 多个第三方库。通过研究这些第三方库版本之间的差异，我们发现了第三方库中的 10 多个安全问题。(ii) 在检测阶段，研究了第三方库分离方法，在此基础上，通过语料库和分离出的模块之间的对比，识别第三方库及其版本。(iii) 为提升检测的速度，采用集合分析方法，基于集合分析的检测方法在保证检测性能的同时，也解决了程序优化所导致的第三方库检测的难题。

本节内容组织如下：第2章对 ProGuard 处理过的 Android 第三方库进行定量分析，指出目前分析方法无法检测经过 ProGuard 处理过的第三方库；第3章提出第三方库语料库建立方法，该语料库对 ProGuard 混淆过的第三方库不敏感；第4章基于建立的语料库，检测经过 ProGuard 压缩、优化的第三方库；第5章对 Android 第三方库进行大规模测量；第6章研究 Android 第三方库的安全问题；最后对 Android 应用第三方库检测方法进行了总结。

Chapter 2

检测背景

Duet 等人 [11] 在对 100,000 个来自 Google Play 的应用统计表明，约有 20% 的第三方库被 ProGuard 混淆了。本节中，我们深入分析 ProGuard 对第三方库所造成的影响，通过研究说明现有大部分检测方法无法检测被 ProGuard 混淆的第三方库，全部检测算法无法检测被 ProGuard 压缩和优化的第三方库。

2.1 ProGuard 的功能分析

ProGuard 是谷歌推荐的 Android 应用字节码级的优化工具 [7]。除对代码进行混淆外，ProGuard 还提供其他功能，如对 Android 应用进行压缩和优化。ProGuard 的普遍使用给第三方库检测带来了一些挑战。

ProGuard 的混淆：通过使用短而无意义的名称来重命名应用中的类、域和方法。这种重命名对人工审核产生一定的阻碍，但不影响程序的自动分析。

ProGuard 的压缩：通过删除应用中未被使用的类、域和方法来实现压缩。由于第三方库提供了大量的功能接口，而第三方库客户可能只使用功能的一个子集，其他未使用的接口所对应的类、域和方法将被 ProGuard 删除。因此 ProGuard 的缩减将极大地减少应用的存储空间，部分测试用例的压缩结果如表2.1所示。在压缩场景下，针对原始或混淆的第三方库的检测方法将无法正确执行检测，使用子图同构（subgraph isomorphism）[6] 或默克尔树（Merkle tree）[1] 技术的研究方法在部分情况下可能对压缩的第三方库检测奏效。

ProGuard 的优化： ProGuard 的发行版本从 1.0 到 5.3.3，迄今为止共有 47 个版本。从 3.0.7 版开始，ProGuard 提供了优化功能，可在应用中进行过程内和过程间优化。具体来说，ProGuard 使用诸如控制流分析、数据流分析、partial evaluation、

Table 2.1: 经过 ProGuard 压缩过的第三方库

Table 2.1: Third-party library shrunk by ProGuard

第三方库	版本	类数量	剩余类数量	域数量	剩余域数量	方法数量	剩余方法数量
Amazon s3	2.4.3	6945	538 (7.75%)	15525	1426 (9.19%)	50585	2895 (5.72%)
Amazon s3*	2.4.3	1244	499 (40.11%)	3384	1270 (37.53%)	8703	2572 (29.55%)
Apache Email	1.5	349	232 (66.48%)	1253	711 (56.74%)	2817	1801 (63.93%)
Apache Ftp	3.6	195	57 (29.23%)	1270	207 (16.30%)	1815	313 (17.25%)
Gson	2.8.1	186	160 (86.02%)	421	290 (68.88%)	1027	771 (75.07%)
Flurry Analytics	7.2.3	338	328 (97.04%)	940	927 (98.62%)	1445	1435 (99.31%)
OkHttp	2.7.5	234	182 (77.78%)	992	724 (72.98%)	2187	1416 (64.75%)
Nine Old Androids	2.4.0	145	133 (91.72%)	474	339 (71.52%)	930	688 (73.98%)
Google Cloud Messaging	5	4132	120 (2.90%)	10398	424 (4.08%)	26118	673 (2.58%)
Picasso	2.5.2	92	72 (78.26%)	352	247 (70.17%)	522	362 (69.35%)
Http Client	4.5.3	732	382 (52.19%)	1909	698 (36.56%)	4707	2501 (43.57%)
InApplication Billing	5	320	28 (8.75%)	853	78 (9.14%)	2449	118 (4.82%)
Means			53.19%		45.98%		45.82%
Standard Deviation			0.33		0.30		0.32

[1] 通过对 ProGuard 5.3.3 进行了一些修改得出这些数据。被测试对象使用第三方库所推荐的 ProGuard 设置进行处理。
[2] 每个测试用例都尽量调用第三方库全部功能，比如，在使用 “Apache Ftp” 这个第三方库时，同时调用了其上传和下载功能。
[3] 当使用 ProGuard 时，只使用了代码压缩选项，从 ProGuard 的实现看，在该选项下，只会进行一次压缩迭代。
[4] 观察发现，部分应用由于压缩程度较高，在这种情况下，利用默克尔树 [1] 的技术是无法检测被 ProGuard 压缩的第三方库的。

静态单赋值、global value numbering 和活跃分析之类的优化技术来对应用进行优化。最新版本的 ProGuard 5.3.3 提供了 17 种类型的优化，包括 modifier changing、定值传播、内联、窥孔优化等。其中，最重要的窥孔优化包含了 200 多个优化功能，比如模拟执行两个字符串的合并等。鉴于之前的检测工作均未描述如何处理被优化的第三方库，这里对 ProGuard 的优化重点进行分析以说明由此所导致的检测问题。

优化使第三方库识别变得复杂。例如，第三方库的客户数据可能通过常量传播从调用方 → 被调用方的途径来侵入第三方库。第三方库的代码也可以通过内联从被叫方 → 调用方来侵入客户代码。代码2.1和代码2.2是内联优化的前后对比，如代码所示，第三方库中的代码经过 ProGuard 优化后，被嵌入到客户代码中，因此优化使得客户代码和第三方库之间的界限变得模糊。

Listing 2.1: 通过 “Apache Commons Email” 这个第三方库来发送一封邮件

```
private String hostName = "smtp.xxx.com";
private String userName = "user@xxx.com";
private String password = "password";
private String fromAddress = "from@xx.com";
private String fromName = "from";
private String toAddress = "to@xx.com";
private String toName = "to";

public void sendSimpleEmail() throws Exception {
   SimpleEmail email = new SimpleEmail();
   email.setHostName(hostName);
```

```
    email.setAuthentication(userName, password);
    email.setCharset("utf-8");
   email.setFrom(fromAddress, fromName, "utf-8");
   email.addTo(toAddress, toName, "utf-8");
   email.setSubject("Subject");
   email.setMsg("Simple mail");
   email.send();
}
```

Listing 2.2: ProGuard 将 “Apache Commons Email” 中的代码内联至客户代码

```
ApacheMailTest v0 = new ApacheMailTest();
try {
   SimpleEmail v1_1 = new SimpleEmail();
   v1_1.setHostName(v0.hostName);
   ((Email)v1_1).setAuthenticator(new DefaultAuthenticator(v0.userName, v0.password)); // Interface defined in
         ``org.apache.commons.mail'' are inlined to user code.
   v1_1.setCharset("utf-8");
   v1_1.setFrom(v0.fromAddress, v0.fromName, "utf-8");
   v1_1.addTo(v0.toAddress, v0.toName, "utf-8");
   v1_1.setSubject("Subject");
   String v2 = "Simple mail"; // Inlined ``setMsg'' method.
   if(EmailUtils.isEmpty(v2)) {
      throw new EmailException("Invalid message supplied");
   }

   v1_1.setContent(v2, "text/plain");
   ((Email)v1_1).buildMimeMessage(); // Inlined ``send'' method.
   ((Email)v1_1).sendMimeMessage();
}
catch(Exception v1) {
   v1.printStackTrace();
}
```

不同的优化技术的优化策略是不同的，如有的优化使用从调用方 → 被调用方优化策略，而其他优化使用从被调用方 → 调用方的优化策略，当这些优化随机且乱序进行时，即使使用同一个版本的 ProGuard 来对第三方库进行优化，也会生成不同的结果，优化结果是不可预估的。使用不同版本的 ProGuard 来优化应用会导致情况更加复杂。如同样对于内联优化，ProGuard 3.0.7 只简单内联了所有名字为 getter 和 setter 方法，而 ProGuard 5.3.3 支持常量，方法参数，返回值和小型的或仅被调用一次的函数的内联。因此，不同版本的 ProGuard 对第三方库的优化加剧了优化结果的不确定性。

为明确优化对第三方库所造成的影响，我们系统的观察了 ProGuard 的发行版本（从 3.0.7 到 5.3.3）对当前流行的第三方库进行优化时的效果。表2.2展示了优化通常会破坏用做第三方库检测的特征。

由于优化通常会破坏检测用的特征，且优化的应用占比不小，因此需要一种更细粒度的第三方库检测方法。传统的 Android 第三方库检测方法可分为四类，分别为

Table 2.2: Proguard 优化对第三方库所造成的影响

Table 2.2: Third-Party library optimized by ProGuard

第三方库	Amazon s3	Amazon s3*	Apache Email	Apache Ftp	Gson	Flurry Analytics	OkHttp	Nine
第三方库版本	2.4.3	2.4.3	1.5	3.6	2.8.1	7.2.3	2.7.5	
# optimize iterations	7	7	3	5	5	2	6	
# finalized classes	27	27	13	32	32	1	48	
# unboxed enum classes	0	0	0	0	0	0	1	
# privatized methods	313	206	17	26	109	0	55	
# staticized methods	111	106	7	9	71	0	23	
# finalized methods	1410	1234	137	160	399	1	646	
# removed method parameters	138	132	0	2	2	0	30	
# inlined constant parameters	26	26	4	5	7	0	22	
# inlined constant return values	10	10	0	1	0	0	6	
# inlined short method calls	2697	1364	26	168	142	0	553	
# inlined unique method calls	639	632	32	83	122	0	342	
# inlined tail recursion calls	0	0	0	0	5	0	2	
# merged code blocks	6	6	4	1	2	0	14	
# variable peephole optimizations	4165	2681	678	376	473	2	1293	
# field peephole optimizations	6	5	3	2	1	0	1	
# branch peephole optimizations	657	644	223	64	180	0	294	
# string peephole optimizations	480	318	5	28	82	0	179	
# simplified instructions	202	136	30	26	21	2	210	
# removed instructions	868	649	220	96	132	4	1016	
# removed local variables	158	118	13	23	40	0	53	
# removed exception blocks	31	31	9	6	1	0	12	
# optimized local variable frames	645	549	288	87	197	1	331	

[1] 通过修改 ProGuard 5.3.3 来获取这些数据。

[2] ProGuard 的默认配置（proguard-android-optimize.txt）关闭了 “vertically merged classes”，“horizontally merged classes”，“removed write-only field peephole optimizations” 选项，因此这些优化的结果未在此处展示。

[3] 表中 “optimize iterations” 代表优化的轮数，优化会持续进行，直至不再有可优化的代码。其他数据条目则是一轮优化的结果。

[4] ProGuard 中，压缩和优化是同时进行的，因此这里的数据是在表2.1的基础上进行的。压缩和优化交替使用，将极大改变应用的布局。

基于字符串的（string-based），基于令牌的（token-based），基于树的（tree-based）和基于语义的（semantics-based）方法 [20, 19, 12]。这些方法在检测 ProGuard 处理过的应用时大多都是无效的。表2.3总结了基于传统方法的检测工作及其能力。

目前还存在基于图的检测算法，其使用参数类型、返回值类型、常量字符串、访问修饰符和指令作为节点信息来构建图 [25, 1, 2]。但 ProGuard 的压缩和优化极大地改变了这些信息（见表2.2），这导致基于图的检测算法无法运用到压缩或优化的第三方库检测。例如，在对 “Amazon s3” 这个第三方库上使用 ProGuard 进行了一次迭代压缩、优化之后，共剩余 465 个类。依据类内的调用关系为这 465 个类建立依赖图后，使用 NetworkX [9] 发现，其中 34 个类生成的依赖图不再是原来类的子图，例如从图2.1所示的例子可以看出，节点 “m7471” 因为没有被调用而被压缩移除；“m7465” 只被调用了一次，因此 ProGuard 将其内联到方法 “m7469” 中，这都导致基于子图同构算法的方法失效。且在测试这些类的同构关系计算中，有 12 个类由于调用关系过于复杂，导致子图同构检测超时（20 秒）。虽然有 419 个类通过了子图同

Table 2.3: 第三方库检测相关工作

Table 2.3: Related work on third-party library detection

现有工作	方法	可检测原第三方库	可检测混淆	可检测压缩	可检测优化
AdDetect [18]	Machine learning	√	√	×	×
LibRadar [16]	Use system API which is resilient to obfuscation	√	√	×	×
LibScout [1]	Similarity analysis based on features which is resilient to obfuscation	√	√	×	×
LibD [14]	Use opcode which is resilient to obfuscation	√	√	×	×

[1] LibScout[1] 提出的基于集合的检测方法本身可用于压缩代码检测，但如表2.1所示，经过压缩过后，大约只剩下 53.19% 的类，且在剩下的类中，大部分方法也被移除，因此特征完全被破坏。所有这些都导致该方法无法应用在被压缩的第三方库的检测。

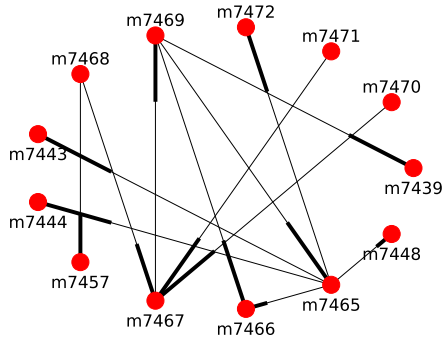
构测试，但因为有 150 个类中的节点小于 3（这些图的信息较少），这使子图同构检测不可信。除了这里展示的调用关系，数据依赖关系和整个应用的依赖关系也会受压缩和优化的影响。

Figure 2.1: 类 “com/amazonaws/regions/RegionMetadataParser” 的调用图

Figure 2.1: The call graph of “com/amazonaws/regions/RegionMetadataParser” class

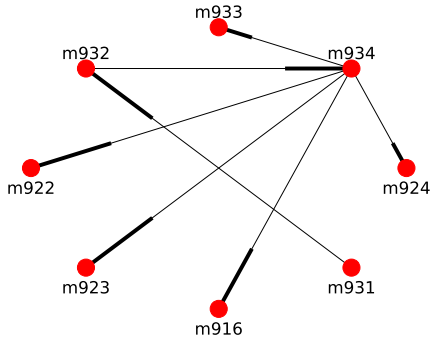
(a) 原始调用图

(a) Original call graph



(b) 经过一轮压缩和优化的调用图（图2.2中 n=1）

(b) The result of shrinkage and optimization with one iteration (n = 1 in Figure 2.2)



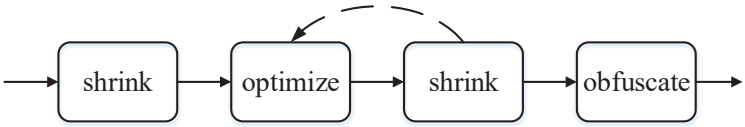
2.2 ProGuard 的运行策略

ProGuard 实际运行策略如图2.2所示。本小节中描述 ProGuard 灵活可配置的运行策略会使第三方库形态更加复杂。目前 ProGuard 的配置方式有三种，分别为原始配置、开发工具默认配置和开发者的定制配置。

对 ProGuard 来讲，默认情况下是开启了混淆、压缩和优化 [8]。ProGuard 实

Figure 2.2: ProGuard 工作流程

Figure 2.2: Workflow of ProGuard



际运行时压缩和优化的迭代次数由 “-optimizationpasss” 选项控制。如果指定了此参数，ProGuard 会持续对目标应用进行压缩和优化，直到达到此参数指定的上界或在此之前已经无法对应用进行任何优化。

应用开发环境目前也集成了 ProGuard，且 Android 开发工具可以生成使用 ProGuard 的配置文件（project.properties）。默认情况下，该配置文件中关闭了优化选项（选项 “ -dontoptimize”）。

开发者的配置也导致 ProGuard 运行策略的多样性。比如默认的内联策略是对不大于 8 的代码块进行内联，但开发者可以通过指定 “maximum.inlined.code.length” 参数来设计自己的策略。

总而言之，不同版本的 ProGuard 和 ProGuard 的不同用法最终导致应用中第三方库的多样性，尤其是优化使得 ProGuard 处理的第三方库的最终结果很难预测。鉴于 ProGuard 广泛应用于应用开发，因此有必要从头开始建立第三方库并研究新的第三方库检测方法。

Chapter 3

建立第三方库语料库

如前所述，必须寻找能代表 Android 应用第三方库的稳定特征，该特征必须可以对抗针对 Android 应用的混淆、压缩和优化，且运用该特征的检测方法必须能适应大规模移动应用的检测需求。为方便后续描述，本文首先对使用的名词进行定义。

原始第三方库：第三方库客户未使用 ProGuard 对应用进行处理，在这种情况下，嵌入的第三方库与原始第三方库一致。

混淆的第三方库：第三方库客户使用 ProGuard 对应用进行混淆处理，在这种情况下，嵌入的第三方库也被混淆。

压缩的第三方库：第三方库客户使用 ProGuard 对应用进行压缩处理，在这种情况下，嵌入的第三方库也被压缩。

优化的第三方库：第三方库客户使用 ProGuard 对应用进行优化处理，在这种情况下，嵌入的第三方库也被优化。

骨干包（Package Stem）：骨干包是全限定名 [22] 的一部分，也就是通常所说的命名空间，定义骨干包主要便于第三方库版本的识别。骨干包作为第三方库的内容容器应该足够短，以便覆盖某个第三方库的全部类。与此同时，骨干包的长度应该足够长，在这种情况下，骨干包本身的特征才足够明显。由于骨干包的长度未知，因此需要特定的策略对骨干包进行识别。

包依赖图：代表程序中包的依赖关系，主要是包中的域和方法的相互依赖。

本节中描述如何建立第三方库语料库，以用来检测原始或经过处理的（混淆，压缩和优化）第三方库。语料库本身对 ProGuard 的混淆不敏感，可直接用来检测原始的和混淆的第三方库。与此同时，还可以作为基准库来检测经过压缩和优化的第三方库。语料库的建立流程如图3.1所示。

第三方库语料库的建库过程分构建包的哈希树，标注和通过骨干包来发现第三

Figure 3.1: 建立第三方库的语料库

Figure 3.1: Establishing thrid-party library corpus

方库更多版本三个步骤。与传统基于全限定名为特征的检测方法不同，在构建哈希树时使用了 Android SDK API 和字符串的哈希做为特征。在发现第三方库的不同版本时，先尝试探测最有代表性的骨干包，然后通过骨干包识别第三方库的不同版本且采集代表不同版本的哈希值。整个建库过程是基于第三方库被很多应用集成这个事实，利用 "大代码" 的优势进行的。

构建包的哈希树： Android 应用的包结构为分层的树状结构，本文依据包结构建立了哈希树。在树中，非叶子节点代表文件夹，叶子节点是类文件。为构建哈希树，采用深度优先遍历（DFS）算法首先为叶子节点生成哈希值，然后基于子节点的哈希值生成当前节点的哈希值。在这种情况下，哈希值保存了第三方库的结构信息。与 Libradar [16] 类似，在计算叶子节点的哈希值时，使用了类中对 Android SDK API 的调用信息。另外考虑到即使 ProGuard 对应用进行了混淆，类中使用的字符串也会保持稳定，所以为了使特征足够丰富，还引入叶子结点中包含的字符串信息。具体来说，我们通过如下方法计算哈希树：

(i) 首先对 Android SDK API（从 android.jar 文件中提取）进行编号，然后对任何代表类的叶子结点 i，为其生成一个排序的稀疏向量 $\mathbf{v_i}$，向量中任何一个元素都表示类调用的 Android SDK API 的编号。向量 $\mathbf{v_i}$ 的长度用 $l_i = \|\mathbf{v_i}\|$ 表示。接下来计算 $\mathbf{v_i}$ 的哈希值，$s_i^1 = \text{SHA-256}(\mathbf{v_i})$；同时，对类中出现的字符串排序后所得到的字符串 str_i 计算哈希值，$s_i^2 = \text{SHA-256}(str_i)$。最后对排序的 s_i^1 和 s_i^2 再次计算哈希值，最终生成 $s_i = \text{SHA-256}(s_i^1, s_i^2)$。

(ii) 对非叶子节点 i，简单的将所有子节点的 s_{ic} 排序后进行哈希计算，最终生成 s_i。非叶子结点的向量 $\mathbf{v_i}$ 的为子节点的向量 $\mathbf{v_{ic}}$ 的并集，即 $\mathbf{v_i} = \bigcup_{\mathbf{1}}^{\mathbf{n}} \mathbf{v_{ic}}$，其 $l_i = \|\mathbf{v_i}\|$。这里的哈希 s_i 是该文件夹的特征，l_i 代表这个文件夹下的类使用了多少个 Android SDK API。最终，为每个非叶子节点 i 生成了一个四元组 $\langle s_i, l_i, p_i, v_i \rangle$。其中 p_i 是与节点 i 对应的限定名。

(iii) 以 s_i 为依据，在 "大代码" 环境下进行统计，得到另外一个五元组 $\langle s_i, l_i, p_i, v_i, a_i \rangle$。这里 a_i 表示包含相同 s_i 特征的应用数量。

至此，哈希树构建完毕，树中每个非叶子节点都对应一个五元组。构建的哈希树

的例子如图3.2所示。其中 path 表示 p_i，hash 表示 s_i，num 表示 a_i，weight 表示 l_i，v_i 是内部存储，未在图中展示。

Figure 3.2: 构建的哈希树

Figure 3.2: Hash tree building

path	hash	num	weight	marked
Loauth	bf835200f5bd16008a5594f47dadd8d354fe2382	6054	152	mark
Ljavax	618c3b2ae6524c07f9208e9e2f0f9b39aa00be77	6052	11	mark
Lnet	ca6520077549c8d19a39ef0b3b8e7b9ad19cd393	5975	56	mark
Lfr	b2d6d06fff71a76539636a71eac5effe308d6716	5936	45	mark
Lcom/google/android/gms/maps/model/a	7ece945164c4849509af996b12758f4a462006ed	5394	30	mark
Lio	4a0137447fa15573838296cd80caca23ba5fa35b	5358	426	mark
Landroid	42d88238c86a9d246c18a363373056d5a9b7abde	5281	1574	mark

对第三方库进行标注：为了花费尽量少的人力对第三方库进行标注，使用了如下的标记方法：

(i) 使用广度优先（BFS）方法来遍历系统中的全部文件夹。基于 a_i 将遍历结果按降序排序从而确保被广泛使用的第三方库首先得到标注。

(ii) 鉴于 a_i 是通过统计包含相同 s_i 的应用得出，为了后续能通过其对应的 p_i 来查询同一第三方库的不同版本，需要 p_i 具备足够的特征以区分其他第三方库。因此将排名靠前的 a_i 所对应的节点沿着 p_i 继续前移，直到 l_i 发生了改变或者当前节点 i 有多个子节点，至此，当前节点 i 对应的 p_i 可作为该第三方库的一个临时骨干包。通过这种探测的方式确保骨干包足够长，保证 p_i 具备足够的特征来检测同一第三方库的不同版本。

(iii) 为临时骨干包建立依赖图，然后使用 NetworkX [9] 来发现包依赖图中的环路，如图2.1所示。如果临时骨干包位于一个环路中，则临时骨干包将沿着 p_i 回退，然后为新的临时骨干包建立包依赖图，继续使用 NetworkX 发现环路，该过程持续进行，直到不再有新的环路产生为止，最终的 p_i 是代表该第三方库的骨干包。通过这种回退的方式确保骨干包不至于太长，而使得其对应的特征 s_i 不够丰富而不足以代表一个第三方库。

(iv) 以骨干包为线索，通过搜索引擎对第三方库的信息进行查询，然后使用手工方式对该骨干包进行标注。最终生成了一个代表第三方库的四元组 $\langle s_i, p_i, v_i, d_i \rangle$，其中 d_i 是第三方库的描述，p_i 是第三方库对应的骨干包。标注结果如图3.3所示。

发现第三方库的其他版本：系统在 $\langle s_i, p_i, v_i, d_i \rangle$ 的基础上自动进行了扩展，识别同一

Figure 3.3: 对第三方库进行标注

Figure 3.3: Tagging the third-party library

Tag	hash	path	info
kSOAP2	bf5eb2d6c2576880700c09443a645b58ba69523a	Lorg/ksoap2	https://code.google.com/p/ksoap2-android
SLF4J	27f7522194fa612a2005a0cf99b52347731f9e27	Lorg/slf4j	https://www.slf4j.org/index.html
XMLPull	8d71434d7c5ce80ef4c57aeb87981b1183425f7e	Lorg/xmlpull	http://www.xmlpull.org/
Cocos2d-x	65d62967d67c1c0e4ab1e8941c1cf3df38b3a8a8	Lorg/cocos2dx/lib	http://www.cocos2d-x.org/
Butter Knife	960a118a3cb51f21e7bd4d69f360bf5a2eede3c8	Lbutterknife	http://jakewharton.github.io/butterknife/
Barcode Scanner Libraries	9feb792cfed7ae655ab75bce8c7ab948c4e8c927	Lme/dm7/barcodescanner	https://github.com/dm77/barcodescanner
ini4j	c31f21b6b8b93571ccd38afb468917e599bd31e9	Lorg/ini4j	https://github.com/JetBrains/ini4j
RoboGuice	8aed9b53598a67abd8c282fec189217bcf2b6e0b	Lroboguice	https://github.com/roboguice/roboguice
date4j	51ded957b6cf82fb40da89be9583bd2b6182b3ef	Lhirondelle/date4j	http://www.date4j.net/
Image Cache	ac4cd5a7d4eae2b163ad7780a44177d9b5a15e39	Ledu/mit/mobile/android/imagecache	https://github.com/mitmel/Android-Image-Cache
Grassy	6aa51057274802212b0d4c90fcdcfb28706e1747	Lair/com/deschensdma/grassyhd	https://github.com/susisu/Grassy
PhotoView	8cdcbfb8591913682452e3e0400a2d45cc1f5b97	Luk/co/senab/photoview	https://github.com/chrisbanes/PhotoView

第三方库的不同版本。该过程为相同的 p_i 传播 d_i，最终为 p_i 相同的节点生成了一个三元组 $\langle s_i, d_i, v_i \rangle$，其中 s_i 是预先生成的与第三方库的特定版本相关的特征。随后对这些三元组进行了清洗，所有被污染的第三方库对应的三元组都通过环路发现来清除，所有的压缩和优化的第三方库都通过 v_i 的包含关系而合并，具体的清洗过程在下文描述。最终剩余的 $\langle s_i, d_i, v_i \rangle$ 可作为具体版本的第三方库的检测基准。

Chapter 4

检测应用中压缩和优化的第三方库

借助已建立的第三方库语料库，研究如何将其应用于第三方库检测。鉴于之前已经有研究使用 p_i 和 s_i 来对原始和混淆的第三方库进行检测，这里不再对这类检测方法进行描述。考虑到目前没有任何研究对经过 ProGuard 压缩或优化的第三方库进行检测，本文首先对应用进行解耦，然后使用第三方库语料库来识别这类第三方库。

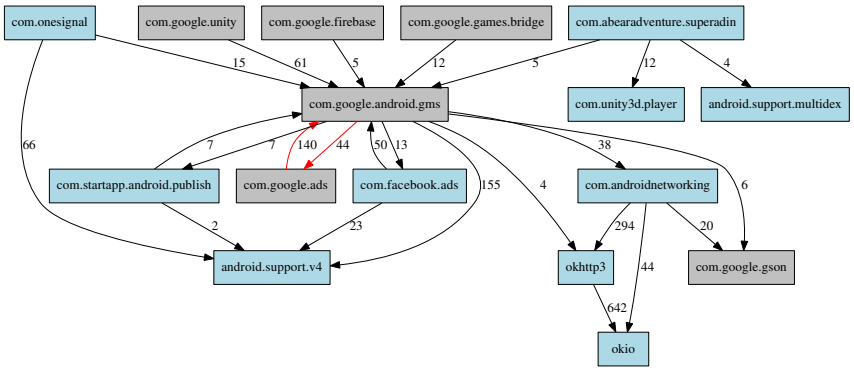
应用解偶：为识别应用中被压缩或优化的第三方库，使用模块解耦技术将应用划分为不同的模块。受 PiggyApp [24] 和后续工作 LibSift [21] 的启发，将骨干包探测技术运用于程序解耦。与对第三方库进行标注时所使用的骨干包探测技术不同，这里从第二层目录探测骨干包。图4.1是对应用进行解耦的结果示例。图中的节点代表模块；有向边表示两个模块之间的依赖关系，依赖关系包括模块间的域和方法的依赖，分配给每个边的权重是模块之间的依赖程度。

解耦过程中，如果兄弟节点之间的依赖权重大于上界 U，则认为兄弟节点属于同一个模块；如果兄弟节点之间存在环路，则认为兄弟节点属于同一个模块。在这些情况下，都将停止解耦，算法1描述了模块解耦的过程。

压缩和优化的第三方库检测：应用解耦后，对每个模块进行检测。通过与语料库的对比，来判断当前模块是哪个第三方库。对比过程中未使用字符串，因为 ProGuard 的窥孔优化技术可能导致原始字符串消失，同时产生新的字符串，如优化会将 “a” + “b” 将替换为 “ab”。在使用 v_i 做为特征来检测第三方库时，我们注意到第三方库中调用某个 Android SDK API 的次数可能因为从第三方库到客户代码的内联而减少，可能因为死代码而被移除，与此同时，第三方库内部的内联操作也可能导致对某个 Android SDK API 调用次数的增加。因此，比对时只考虑 Android SDK API 的种类，而不考虑调用的次数，即使用基于 v_i 的包含关系（$\supseteq$）来判断是否为经过压缩或优化的第三方库。

Figure 4.1: 应用解耦、建立包的依赖图

Figure 4.1: Module decoupling and package dependency graph building



4.1 存储开销和计算复杂度分析

为建立包含 v_i 的基准集，本文从 “android.jar” 采集了 Android SDK API 并对其进行编号。以 android-22 版本的 “android.jar” 文件为例，其中包含的修饰符为 “public” 的方法总数为 32,203 个。v_i 生成中会以这些方法为基准，如果使用比特向量表示 v_i，需要为每个第三方库准备 4,026 字节。实际统计发现，一般第三方库只使用了 Android SDK API 的一个小的子集。在这种情况下，如果使用稀疏向量来表示 v_i，则可以压缩第三方库表示的空间。如不同版本的 “Android Support V4” 使用了 770 到 1,805 个 Android SDK API，平均为 1,107 个 API，在这种情况下，如果使用稀疏比特位向量表示第三方库 “Android Support V4”，则平均需要 2,214 字节。采用稀疏向量表示的方法在边缘计算环境下的传输和存储所需的空间是比特向量的一半。相比需要保存节点关系的基于树（图）的检测方法，本文的表示方法极大降低了数据传输和数据存储的压力。

在第三方库识别时，比较过程是比特位的逻辑操作，相比基于树（图）的检测方法的计算复杂度 $\exp((\log n)^{O(1)})$，本文提出的检测方法的计算复杂度是 O(1)。

Algorithm 1 模块解耦算法

Input: Android package structure and metadata;
Upper-threshold of PDG: U
Output: Modules: $M = \varnothing$

for node $i \in$ DFS **do**
  $i_c \leftarrow$ child's nodes of node i;
  $w_{ic} \leftarrow \sum \mathrm{PDG}(i_c)$;
  if $w_{ic} > U$ **then**
    $M \leftarrow M \cup i$;
  end if
  if cycle(i_{ci}, i_{cj}) = True **then**
    $M \leftarrow M \cup i$;
  end if
end for
for $m_i \in M$, $m_j \in M$ **do**
  if cycle(m_i, m_j) = True **then**
    $M \leftarrow M$ - m_i;
    $M \leftarrow M$ - m_j;
    $M \leftarrow M \cup m_{ij}$;
  end if
end for

Chapter 5

Android 应用第三方库大规模测量

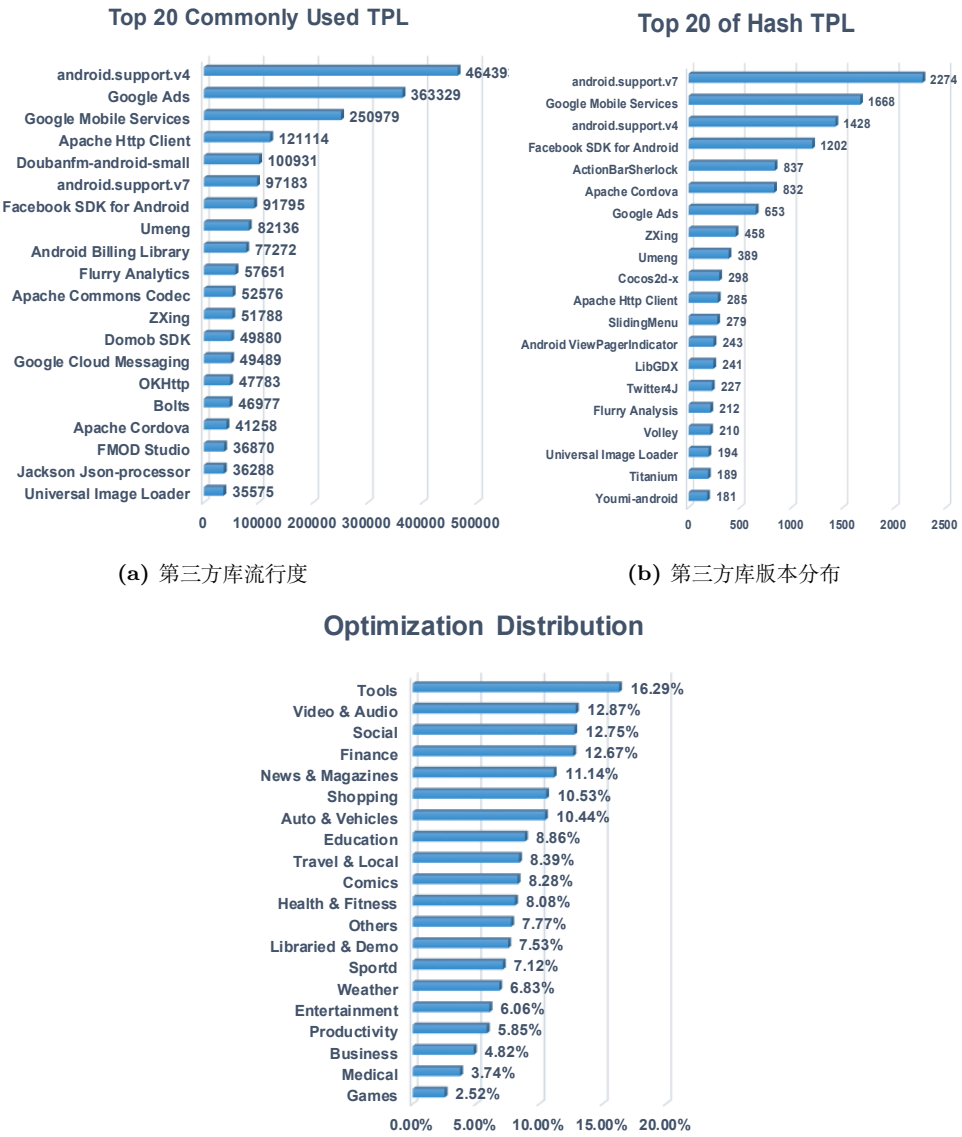
本节中，综合本文提出的检测方法和之前的工作，在“大代码”的基础上，对 Android 应用第三方库进行测量。

5.1 第三方库分布统计

第三方库流行度分析：在大规模数据集上进行 v_i 的匹配不太现实，因为这种方法需要对待检测应用进行解耦。因此这里使用 s_i 对数据进行统计。应用中的第三方库的分布如图5.1a所示。相较其他方法 [13, 1, 16]，使用 s_i 能发现更多的第三方库。因为以 s_i 进行检测可以对抗混淆对 p_i 所造成的破坏。如果两个 s_i 相同，本文提出的方法不仅确保了两个第三方库中的调用的 Android SDK API 相同，也确保了其结构信息相同，因此减少了误报。以“Apache Http Client”这个第三方库为例，我们的检测结果表明仅仅使用 p_i 是不足以检测被重构的第三方库，检测结果如表5.1所示。

Figure 5.1: 第三方库分布统计

Figure 5.1: Distribution of third-party library



(a) 第三方库流行度 **(b)** 第三方库版本分布

(c) 优化比例统计

Table 5.1: “Apache Http Client” 检测结果

Table 5.1: Detection result for “Apache Http Client”

Path	# detection
Lorg/apache/http	87,544
Lorg/apache/commons/httpclient	25,344
La/a/a	3,467
Lcom/flurry/org/apache/http	896
Lorg/a/b	689
Lorg/a/a	518
Lorg/apache	495
Others	2,161

[1] 通过重构，“Lorg/apache/http” 中的类被转移到 “Lorg/apache/commons/httpclient” 中了。
[2] “Lorg/a/b” 之所以被检测到，是因为其与语料库中的 s_i 相匹配。

清洗前第三方库版本的分布： 未清洗前第三方库的版本的分布如图 5.1b所示，如图所示，未清洗前 “Android.support.v4” 对应 1,428 个不同的 s_i，然而官方只发布了 61 个版本。版本数量的巨大差异是由第三方库被污染或 ProGuard 的压缩和优化所导致的。

清洗后的第三方库版本分布： 接下来通过清洗过程来移除被污染的第三方库版本，合并由于压缩和优化所生成的新版本。具体来讲，如果应用解耦的模块所对应的路径与一个第三方库的骨干包匹配，那么我们搜索包含此模块的环路。如果该模块包含在环路中，则认定该第三方库被污染，进而将其对应的版本从语料库中移除；通过集合 v_i 间的包含关系（$\supseteq$），合并由于压缩和优化所生成的新版本。净化的过程结果如图5.2所示。例如，对于第三方库 “Android.support.v4” 来说，包含该库的应用中大约有 64% 能成功解耦。对于成功解耦的 80 个样本，我们发现 63% 的模块处于环路中，这表明这个第三方库被污染的很严重。继续以 “Android.support.v4” 为例进行审计发现，一些开发框架，如 “Titanium” 或 “ActionBarSherlock” 通过向这个库中注入代码来破坏了基础第三方库的完整性。

最后，本文尝试测量不同第三方库对应 v_i 的编辑距离，如表5.2所示，不同第三方库之间的编辑距离足够大，表明他们之间有明显的区别，其差异性也表明基于包含关系（$\supseteq$）算法的误报率比较低。

Figure 5.2: 第三方库净化过程

Figure 5.2: Third-party library sanitization process

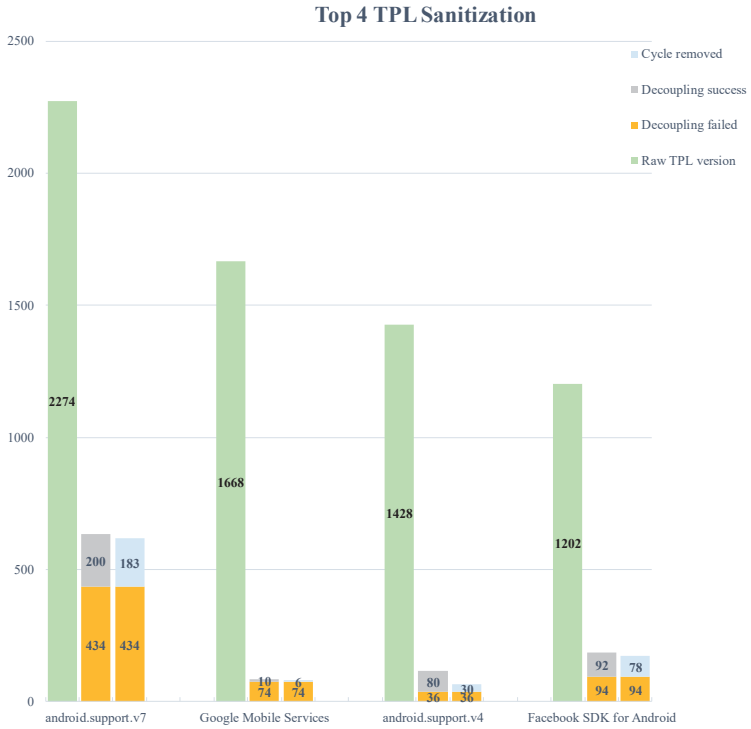


Table 5.2: 第三方库之间的编辑距离

Table 5.2: Third-Party library edit distance

Distance between TPL signature	Max Distance	Min Distance	Mean	Standard Deviation
Android.support.v7 VS. Google Mobile Service	1672	325	1009.99	206.70
Android.support.v7 VS. Android.support.v4	1853	340	1252.77	299.13
Android.support.v7 VS. Facebook SDK for Android	2278	318	1411.81	276.74
Google Mobile Service VS. Android.support.v4	2031	141	1148.52	393.34
Google Mobile Service VS. Facebook SDK for Android	1930	33	1030.26	273.94
Android.support.v4 VS. Facebook SDK for Android	2507	140	1475.85	371.92

被 ProGuard 处理的应用比例: 在 7,602,323 应用中，我们发现有 5,948,438（78.25%）个应用中包含如表5.3所示字符，该字符通常被用来做为检测应用是否被 ProGuard 处理的特征 [13]。

Table 5.3: ProGuard 特征

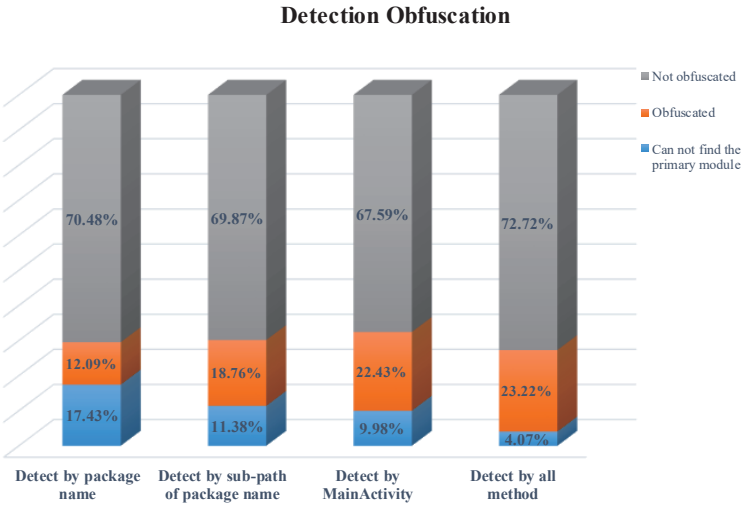
Table 5.3: ProGuard signature

“.*/a;->.*” and “.*/b;->.*”

然而这个特征是第三方库提供商和库的客户共同作用的结果。当库的客户集成使用 ProGuard 处理过的第三方库后，即使库的客户未对其应用进行处理，在使用该特征时，该应用也会被误报为使用 ProGuard 处理过。这里我们使用 Duet [11] 的术语将第三方库处理分为库客户的后处理和库供应商的后处理。为了确定库客户的后处理的比例，我们使用三种方法来发现客户代码。这个测量是在一个包含 4,795,627 个应用的数据集上进行，测量结果如图5.3所示。图中第一列表示 17.43% 的应用无法通过定义在 “AndroidManifest.xml” 文件中的 “package” 所指定的路径下找到任何代码；第二列表示我们采用 “package” 的子目录做为客户代码的统计结果；第三列使用 “MainActivity” 来检测库客户的后处理，其结果表示使用 “MainActivity” 可以找到更多的客户代码；最后一列是综合了前述方法的结果，结果表明，大约 23.22% 的应用被客户进行了后处理。

Figure 5.3: 开发者对第三方库后处理的比例

Figure 5.3: Developer’s post-processing



应用优化的比例： 通过观察 ProGuard 的历史版本，我们发现将一个未被继承的方法的修饰符修改成 “final” 是 ProGuard 优化的最显著特征。因此，我们搜集 Android 支持库 (“Android Support Repository”) 中所有非 “final” 方法，并观察该方法在应用中是否被修改为 “final”，以此判断应用是否被优化。为避免误报，我们搜集了 Android 支持库所有历史版本，确保所采集的方法列表在所有的历史版本中都为非 “final” 类型。实验发现 5,353,359 个应用集成了 Android 支持库，其中 442,605 (8.26%) 个应用中的 Android 支持库的非 “final” 类型方法的修饰符被修改。也就是说，当开发者使用 ProGuard 来对应用进行处理时，36% 的开发者开启了优化选项。基于应用分

类的优化占比统计如图5.1c所示。由图中可见，工具类开发者更倾向优化他们的代码。另外一个针对 Janus 新出现的应用的测量表明，304,845 个应用中的 49,538（16.25%）个应用被优化，这表明对应用进行优化呈现一种增长的趋势。

Chapter 6

Android 第三方库安全检测

第三方库安全检测分漏洞检测和恶意代码检测。检测方法采用了本文中提出的第三方库版本检测、第三方库完整性检测等手段。

6.1 检测有漏洞的第三方库

AliPay SDK for Android：“AliPay SDK for Android” 由支付宝发布，安卓应用开发者集成这个第三方库以提供在线支付功能。2016 年 1 月通过 Janus 检索发现，1,004,498 个应用集成了这个库。

在该库语料库构建阶段，共发现了 1,031 个与这个库相关的 s_i。在语料库清洗阶段，发现有两个版本的库被污染，集成了被污染的库的应用分别是：SHA-1: `9e1dab145cc524d0ad5ea934510b1247f81be1dc` 和 `b9ce54e8e4e3a21dbe85c8dc8814000db419b84f`。对代码审查后发现，开发者将代码注入到该库中来将支付重定向到客户代码，该行为破坏了第三方库的完整性。在合并特征时，我们找到了包含子集最多的版本，这个版本是 SHA-1: `01dd3693451b4c3447013297bdda7005a2e6b32c`，这个版本下有 102 个子版本，比如 SHA-1: `0f64ca13e6851aaa776ffd351747c5a0b32772b1`。我们发现这个版本（目前没有证据表明这个版本是支付宝的官方版本）存在一个安全漏洞。与当前版本不同，即要求订单签名所使用的私钥必须保存在服务器上，该版本要求开发者必须将私钥保存在客户端，在这种情况下，私钥泄漏在客户端，攻击者利用该私钥，可进行 0 元支付。我们随机挑选了使用该版本的第三方库的一些样本发现，大部分应用在使用这个有漏洞的版本时，将私钥泄漏在应用中。最终分析发现共有 13,578 个应用集成了这个有漏洞的版本，其中有超过 20% 的应用使用 ProGuard 进行了压缩和优化处理。“AliPay SDK for Android” 的官方版本原本提供

预置的接口来查询版本信息，但在 ProGuard 压缩和优化的作用下，该接口被移除，导致查询失效。而用本文所述的方法可以对这些经过压缩和优化的特定版本进行检测。

阿里云 OSS：在安全审计时，我们发现许多客户在使用 “阿里云 OSS” 时泄漏了凭证。然后，我们关注 “阿里云 OSS” 凭证泄漏问题。在浏览 “阿里云 OSS” 的开发文档后，我们注意到 “阿里云 OSS” 第三方库提供了一个测试接口，该接口会误导开发人员泄漏其凭证。目前，“阿里云 OSS” 已删除了该接口的说明，但这种简单的缓解措施并不彻底，需要检测出所有使用该库的应用，并逐个审查以确保其并未使用该接口。该库的检测在新版本上比较简单，因为据我们所知，新版本 “阿里云 OSS” 的开发文档向开发人员提供了使用 ProGuard 建议，以减少对此版本的库的更改。但旧版本的 “MBAAS_OSS_Android_1.0.0_” 对如何使用 ProGuard 没有给出任何说明，因此如果客户集成了旧版本的 “阿里云 OSS”，且使用 ProGuard 对其应用进行处理，那最终的 “阿里云 OSS” 这个第三方库的形式将是多样的。这里采用与前面描述类似的语料库构建、版本匹配过程识别该库后，人工分析发现大约 25%的开发人员在集成 “阿里云 OSS” 时泄漏了凭证。

6.2 检测被污染的第三方库

基于第三方库语料库，可以检测被污染的第三方库。将应用解耦后的模块特征和第三方库语料库进行比较，如果给定的模块与语料库中 p_i 一致，但 s_i 不属于同一第三方库中的任何一个版本，则这个模块的完整性可能遭到破坏。以病毒 GhostClicker [17] (SHA-1:`0a6583a741debc90498fb693eb56509f603fc404`) 为例，比较后发现，在与其距离最近的一个版本的第三方库的基础上，该病毒在 “`com.google.android.gms`” 中插入了 289 个 Android SDK API 调用，在 “com.facebook” 中插入了 461 个 Android SDK API 调用。

GhostClicker 解耦后建立的依赖图如图6.1所示，依赖图中有 3 个环路，在病毒代码的注入下，“com.google” and “`com.facebook.ads`” 产生了依赖。更具体来讲，“Android Mobile Service” 中的方法产生了如下的调用，调用目标定义在 “`com.facebook.ads`” 模块中。

green Lcom/google/android/gms/logs/fb;-><init>(Landroid/content/Context;)V

green

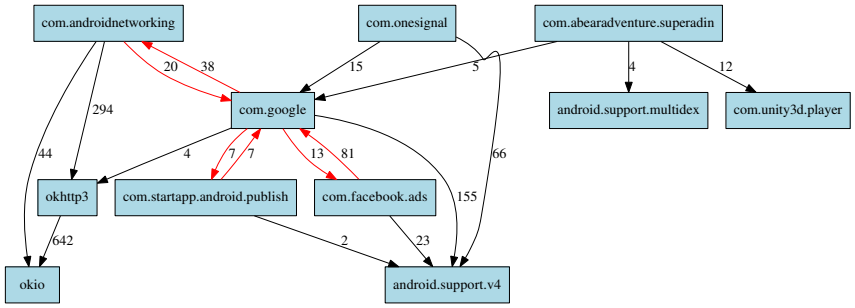
Lcom/facebook/ads/InterstitialAd;-><init>(Landroid/content/Context;Ljava/lang/String;)V

black

这些调用是不正常的，因为模块“com.google.gms”实际上并不依赖于“com.facebook.ads”，这些不合理的调用依赖导致两个模块间产生环路。因此这些环路可用于检测被注入代码的第三方库。

Figure 6.1: GhostClicker 调用依赖图中的环路

Figure 6.1: Cycles in GhostClicker's Dependency Graph



Chapter 7

Android 应用第三方库识别方法讨论

7.1 检测方法的局限性

目前并没有一站式解决方案来进行第三方库识别。实际上，我们的检测方法也在以下几个方面受到限制。

- 阈值U的设置很困难。在实验中，U的某个具体值可以成功解耦一个第三方库，但同一应用内的另一个第三方库却被过度解耦，反之亦然。
- 第三方库可能分布在完全不同的文件夹中，如最新版的 “AliPay SDK for Android”，在这种情况下，一个第三方库对应两个或多个 p_i，这增加了表示和检测的复杂性。
- 检测方法对第三方库的细微修改不敏感，如针对漏洞修复的第三方库版本对库的改动很小，这些修改不会改变第三方库的 Android SDK API 调用，因此无法对此种修改进行检测。
- 在第三方库检测的实践中，我们发现一些第三方库（例如 Twitter，PayPal）的代码在的根目录下，这为第三方库语料库的建立和检测时的解耦带来了一些困难。
- 目前最大的挑战是很难验证第三方库语料库的准确性。大部分情况下，甚至第三方库提供者也无法提供其第三方库的所有版本，因此很难验证基准集的准确性。

- 解耦过程是一个耗时的过程，而循环搜索是消耗内存的过程，这些方法都无法很好地扩展到大型数据集上。

7.2 未来的工作

- 本文根据经验设置了孩子节点相关性的上限U，未来将尝试搜索该变量的最佳值。
- 验证通过解耦生成的模块中的循环，以发现更多针对第三方库的安全威胁。

Chapter 8

本文小结

针对 Android 应用的安全性研究正在飞速发展 [3]，Android 第三方库检测目前受到越来越多的关注。

克隆检测： 第三方库的使用可以看成一个代码克隆问题，代码克隆检测技术已在 Android 应用安全检测中广泛使用。DroidMOSS [23] 计算指令序列的哈希值，通过组合这些哈希值得到整个应用的指纹。通过对指纹的比较以识别克隆应用。Juxtapp [10] 使用 k-grams 操作码序列的哈希做为应用的特征来识别应用商店中的克隆应用。DNADroid [5] 和 AnDarwin [4] 都创建了程序依赖图作为应用的签名，DNADroid 使用子图同构来检测克隆，AnDarwin 在此基础上，将图转化为向量以加快相似度计算。

相似度计算： Li 等 [13] 采用聚类算法来获得代表第三方库的包，然后使用这些包作为特征来检测第三方库，他们也提出了判断应用是否被混淆的特征。LibRadar [16] 仅考虑使用抗混淆的 Android SDK API 做特征，然后执行多级聚类以识别第三方库。与 LibRadar 不同，我们的方法还考虑了其 他突出的特征，适用第三方库中很少甚至根本不调用 Android SDK API 的情况。LibD [14] 提取内部代码依赖项作为特征，并使用这些特征的哈希值识别第三方库。LibScout [1] 利用类层次结构构建固定深度为 3 的 Merkle trees 作为每个库的特征，提出了一种匹配算法来计算与所收集的库的相似度。与该工作相比，本文所提出的方法不仅对被混淆的第三方库具有检测能力，而且还可以应用于被压缩和优化的第三方库，而这类第三方库在 Android 应用中占很大比例。

基于机器学习的检测： PEDAL [15] 提取了 Android 第三方库中的功能代码来训练分类器，用以识别第三方库。AdDetect [18] 和 LibSift [21] 提出使用分层包和包依赖信息作为特征，并建立分类器来对第三方库进行分类。此外，他们还提出了将客户代

码与第三方库分离的方法。

本文提出了检测 Android 应用中第三方库的方法，该方法使用应用结构信息和代码内容信息作为特征，并执行特征匹配算法来识别第三方库。具体来讲，通过研究确定了抗 ProGuard 处理的不变特征，解决了因混淆、压缩和代码优化所导致的检测失效问题。到目前为止，我们成功识别了第三方库中的 10 多个安全问题。

参考文献

[1] Michael Backes, Sven Bugiel, and Erik Derr. “Reliable third-party library detection in android and its security applications”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 356–367.

[2] Benjamin Bichsel et al. “Statistical deobfuscation of android applications”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 343–355.

[3] Yimin Chen et al. “EyeTell: Video-Assisted Touchscreen Keystroke Inference from Eye Movements”. In: *EyeTell: Video-Assisted Touchscreen Keystroke Inference from Eye Movements*. IEEE. 2018, p. 0.

[4] Jonathan Crussell, Clint Gibler, and Hao Chen. “Andarwin: Scalable detection of semantically similar android applications”. In: *European Symposium on Research in Computer Security*. Springer. 2013, pp. 182–199.

[5] Jonathan Crussell, Clint Gibler, and Hao Chen. “Attack of the clones: Detecting cloned applications on android markets”. In: *European Symposium on Research in Computer Security*. Springer. 2012, pp. 37–54.

[6] Ming Fan et al. “DAPASA: detecting android piggybacked apps through sensitive subgraph analysis”. In: *IEEE Transactions on Information Forensics and Security* 12.8 (2017), pp. 1772–1785.

[7] Google. *Shrink Your Code and Resources*. https://developer.android.com/studio/build/shrink-code.html. Accessed August 3, 2017. 2017.

[8] GuardSquare. *ProGuard manual*. https://www.guardsquare.com/en/proguard/manual/usage. Accessed August 27, 2017. 2017.

[9] Aric Hagberg et al. “Networkx. High productivity software for complex networks”. In: *Webová strá nka https://networkx. lanl. gov/wiki* (2013).

[10] Steve Hanna et al. “Juxtapp: A scalable system for detecting code reuse among android applications”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2012, pp. 62–81.

[11] Wenhui Hu et al. “Duet: library integrity verification for android applications”. In: *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM. 2014, pp. 141–152.

[12] Lingxiao Jiang et al. “Deckard: Scalable and accurate tree-based detection of code clones”. In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society. 2007, pp. 96–105.

[13] Li Li et al. “An investigation into the use of common libraries in android apps”. In: *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. Vol. 1. IEEE. 2016, pp. 403–414.

[14] Menghao Li et al. “Libd: Scalable and precise third-party library detection in Android markets”. In: *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press. 2017, pp. 335–346.

[15] Bin Liu et al. “Efficient privilege de-escalation for ad libraries in mobile apps”. In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM. 2015, pp. 89–103.

[16] Ziang Ma et al. “Libradar: Fast and accurate detection of third-party libraries in android apps”. In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM. 2016, pp. 653–656.

[17] Trend Micro. *GhostClicker Adware is a Phantomlike Android Click Fraud*. http://blog.trendmicro.com/trendlabs-security-intelligence/ghostclicker-adware-is-a-phantomlike-android-click-fraud/. Accessed August 22, 2017. 2017.

[18] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. “Addetect: Automated detection of android ad libraries using semantic analysis”. In: *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*. IEEE. 2014, pp. 1–6.

[19] Chanchal K Roy, James R Cordy, and Rainer Koschke. “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach”. In: *Science of computer programming* 74.7 (2009), pp. 470–495.

[20] Chanchal Kumar Roy and James R Cordy. “A survey on software clone detection research”. In: *Queen¡¯s School of Computing TR* 541.115 (2007), pp. 64–68.

[21] Charlie Soh et al. “LibSift: Automated Detection of Third-Party Libraries in Android Applications”. In: *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*. IEEE. 2016, pp. 41–48.

[22] Wikipedia. *Fully qualified name*. https://en.wikipedia.org/wiki/Fully_qualified_name. Accessed August 3, 2017. 2017.

[23] Wu Zhou et al. “Detecting repackaged smartphone applications in third-party android marketplaces”. In: *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM. 2012, pp. 317–326.

[24] Wu Zhou et al. “Fast, scalable detection of piggybacked mobile applications”. In: *Proceedings of the third ACM conference on Data and application security and privacy*. ACM. 2013, pp. 185–196.

[25] Yajin Zhou et al. “Harvesting developer credentials in android apps”. In: *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM. 2015, p. 23.