摘要

网络流量分析是移动应用程序安全审计方面的一种吸引人的方法。之前的研究采用了各种技术（例如中间人攻击、TCPDUMP）来捕获应用程序的网络流量，并进一步识别其中的安全/隐私风险。然而，这些技术存在一些限制，例如流量混合、代理逃避和SSL钉住等。可能的解决方案是修改和定制Android系统。然而，现有的研究主要基于Android OS 6/7。当代应用程序通常无法正常运行在这些过时的Android OS上，这已经成为进一步流量分析研究的障碍。为了解决上述问题，我们提出了一种新的网络流量分析框架-TraceDroid。我们首先利用动态挂钩技术来挂钩用于发送网络请求的关键功能，然后保存请求数据以及代码执行跟踪。此外，TraceDroid提出了一种无监督的方式来识别应用程序中的第三方库（TPLs），以便于应用程序和TPL之间的责任分析。利用TraceDroid，我们对9,771个真实应用程序进行了大规模实验，以对隐私泄露现状进行实证研究。我们的研究发现，TPLs占当代应用程序隐私泄露的44.45%，而从用户设备传输的文件包含比网络请求更详细的隐私数据。我们揭示了应用程序中的超量数据收集和跨库数据收集问题。此外，我们揭示了TPL与其访问域之间的关系，这是以前的研究从未讨论过的。

1 介绍

作为最广泛使用的移动平台，Android操作系统为我们的社会带来了巨大的便利，但也引入了许多安全问题，如隐私泄露[12,38]，广告欺诈[8,9,16]和恶意软件[13,34]等。网络流量分析已被证明是缓解安全问题的最重要方法之一。一般来说，现有的流量分析研究[6,17,23–25,32,40]通常使用TCPDUMP或中间人攻击（MITM）技术来捕获网络流量。然而，TCPDUMP无法处理流量加密问题，因此MITM工具已成为捕获和检查加密流量的最常用方法。MITM技术运行在代理机制上，这意味着用户需要安装MITM工具的自定义证书，信任该证书，并在其设备上配置代理（MITM服务器）。这样做可以使设备的所有流量在到达目标服务器之前通过代理服务器进行路由。使用自定义证书，代理服务器可以通过在通信路径中充当中间人来解密网络流量。然而，这种机制无法处理流量混合问题-后台流量（Android操作系统流量，后台服务流量），应用程序流量和TPL流量混合在一起，这阻碍了细粒度流量分析。此外，由MITM工具引起的一些其他限制问题也没有得到很好的解决（第2.1节）。

为了缓解流量混合问题，之前的研究[19,30,37]使用HTTP请求头中的“User-Agent”或“Host”白名单来识别广告库流量。然而，这种方法需要从业人员维护一个包含已知域和TPL的列表，并且该列表需要经常更新。显然，随着新的TPL不断出现，这种方法可能导致低准确性，而过时的列表无法满足当代应用程序的需求[39]。

为此，我们提出了一个新的网络流量分析框架TraceDroid，可以同时解决上述问题，而无需修改Android系统。具体来说，我们利用动态hooking技术在负责执行HTTP（S）请求的函数中添加额外的代码，并保存未加密的数据和相应的代码执行跟踪。为了提供关于主机应用程序和TPL之间隐私泄露的细粒度责任研究，我们提出了一种无监督方法来通过相关请求和代码执行跟踪来识别TPL，并使用该方法区分主机应用程序和TPL之间的流量。借助TraceDroid和TPL识别的帮助，我们对9771个真实应用程序进行了大规模实验，并对收集的数据进行了全面分析，以确定现代应用程序中隐私泄露的现状。我们的新发现如下：1）44.45%的隐私泄

露请求是由TPL发起的，这表明TPL已成为一个不可忽视的隐私泄露通道。2）设备ID（例如IMEI，IMSI，SN）是最具吸引力的隐私数据。3）当代应用程序和TPL普遍存在超额收集隐私信息的情况。用户的私人数据被发送到多个后端服务器，而不引起他们的注意。4）从用户设备传输的文件往往包含比HTTP（S）请求中更详细的隐私信息，这长期以来一直被研究社区忽略。5）TPL和其访问域之间的关系是多对多的相关性，并且可以将域分类为自有域，授权域和主机应用程序域（第4.5节）。

我们的主要贡献总结如下：

- 我们提出了一个新的Android网络流量分析框架。据我们所知，TraceDroid是第一个能够同时解决流量混合、代理逃避、流量加密和SSL pinning等问题而无需任何操作系统修改的工作。此外，TraceDroid可以解决在现代Android系统上运行当代应用程序面临的新挑战（第3节）。此外，TraceDroid是轻量级的，易于扩展或修改。为了促进进一步的研究，我们在https://github.com/TraceDroid/TraceDroid-SciSec2022 上向研究社区发布了TraceDroid。

- 我们设计了一种新的方法来识别应用程序中的TPL。提出了一种无监督方法，通过相关HTTP（S）请求和代码执行跟踪来识别TPL。与之前的工作相比，我们的方法不需要任何TPL或白名单的先前知识。我们在我们的应用程序语料库中识别了300多个TPL，并在我们的Github存储库上发布了TPL及其访问域名。

- 在现代应用程序中进行了隐私泄露的大规模分析。我们对9,771个应用程序进行了实证研究，分析了泄漏隐私数据的传输方式和责任方（第4.2和4.3节），过度数据收集和跨库数据收集（第4.4节），以及TPL和其访问域之间的关系（第4.5节）。

2 背景

2.1 基于中间人攻击MITM的流量捕获

如第1节所述，各种MITM工具[7,10,18,20]使用代理服务器和自定义证书来捕获和检查HTTP（S）流量。不幸的是，我们通过调查我们的应用程序语料库发现了一些此方法的局限性，并总结如下：

- 流量混合。移动系统和应用程序正在运行许多守护进程，例如Google框架服务和推送服务。它们的流量作为背景噪声与应用程序流量混合在一起，并被MITM服务器捕获。因此，从背景流量、应用程序流量和TPL流量中区分网络流量并不容易。
- 反调试。应用程序可以检测到代理是否已配置，拒绝通信或更改网络行为。这在很大程度上是为了防止应用程序被调试。例如，如果存在代理，应用程序“com.outfit7.mytalkingtom.qihoo”将不会显示任何广告。
- 代理逃避。即使在Android上配置了代理服务器，应用程序也可以直接连接到目标服务器。例如，java.net.URLConnection中的公共方法“openConnection（Proxy proxy）”

可以使用参数“Proxy.NO PROXY”来忽略代理服务器，并创建一个直接的套接字连接，绕过代理服务器。

- SSL Pinning。它是信任预安装在应用程序中的特定证书的能力。通过SSL pinning，应用程序可以验证服务器的证书，以确保应用程序和服务器之间的通信的唯一性和安全性。因此，MITM工具生成的自定义证书将无法使用。

2.2 基于hook的流量捕获

Hooking（或Instrumentation）是指将额外的代码注入到程序中以收集运行时信息。一旦代码被注入到目标进程中，它就可以完全访问进程内存并修改其组件。开发人员可以利用这种能力来修改目标进程内存组件，从而替换或修改API函数。

为了捕获网络流量，我们可以hook相应的API并注入额外的代码以获取请求数据。Android应用程序开发人员通常使用各种网络库[1,3,11,21,22,26,27,31]执行网络请求。以Okhttp [21]为例，它是Android开发中广泛使用的网络库。开发人员通常创建一个“httpClient”对象，构建一个“request”，并使用它的方法“request.set()”设置请求头中的键值参数，如“content-type:application/json”。最后调用“perform(request)”函数发送此HTTP请求。在这种情况下，我们可以钩住函数“perform(request)”并保存其参数“request”以获取请求数据。我们将展示如何获取未加密的请求数据，详见第3.1节。

与MITM工具相比，基于hook的流量捕获具有以下优点：1）Hooking仅针对特定应用程序中的特定API（Android操作系统中的特定应用程序进程ID），因此捕获的流量不包含应用程序之外的流量。2）Hooking不改变原始程序的逻辑完整性，对应用程序透明，因此不受反调试和代理逃避的影响。此外，它不需要自定义证书，因此无法通过SSL pinning阻止我们获取请求数据。

3 方法

如图1所示，TraceDroid分为四个阶段——网络hooking，用于在Android设备中进行流量捕获的仪器化; 流量触发，使应用程序产生更多的网络流量并解析流量以恢复网络请求、文件和调用堆栈; TPL识别，根据捕获的请求和调用堆栈识别应用程序中的TPL，这可以在第4.2节中受益于我们的责任分析; 隐私泄露分析，对我们的应用程序语料库进行大规模研究，以确定现代应用程序中隐私泄露的现状。

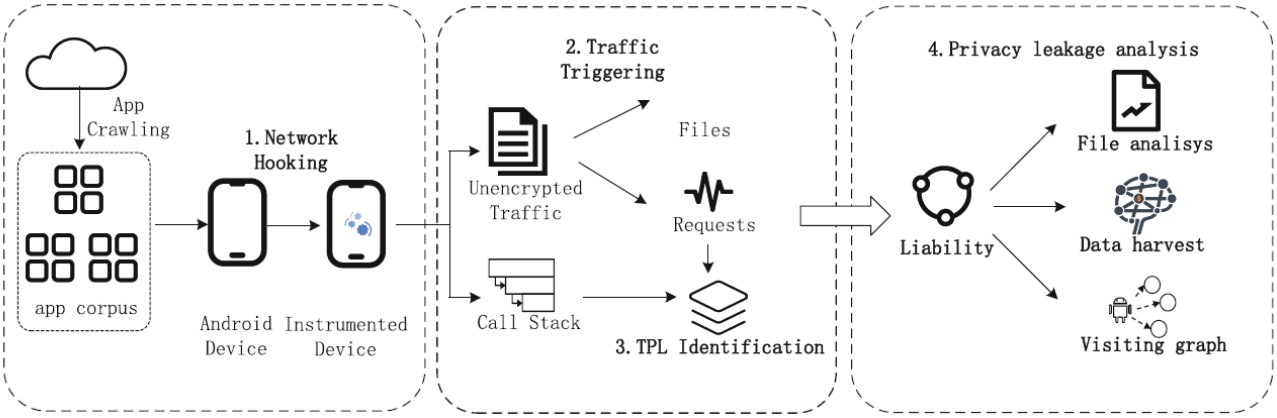


图1. TraceDroid的系统概述

3.1 网络Hooking

如第2.2节所述，我们使用基于hook的流量捕获方法来克服MITM工具的限制。我们的想法是钩住所有负责执行HTTP（S）请求的函数。尽管存在数十个网络库[1,3,11,21,22,26,27,31]，但挂钩所有这些库将是耗时且费力的，因为这需要进行逐个案例的研究和仪器化数十个API。为此，我们转向了网络库所依赖的底层API。

我们手动分析了网络库并发现它们依赖OpenSSL库进行SSL连接。OpenSSL（在Android 6之后被命名为BoringSSL）是Android系统中的默认SSL/TLS库[5]。它包含两个部分 - libssl和libcrypto。其中，libssl是SSL协议的实现，Android系统使用其内置函数（即“SSL读取”和“SSL写入”）来处理HTTPS请求。 "SSL读取"用于从已建立的SSL会话中读取数据并将其放入缓冲区。相反，“SSL写入”将数据写入缓冲区并将其发送到远程服务器。请注意，在调用“SSL读取”之前或在调用“SSL写入”之后，将调用libcrypto中的函数来解密和加密数据，因此在这里读取和写入的数据是明文（未加密数据）。

为了说明我们的网络Hooking方法如何工作，我们以“SSL写入”为例。它的函数原型是“int SSL Write(SSL *ssl*, *const void* buf, int num)”，API参数“ssl”是指定的SSL连接，“buf”是此函数写入数据的缓冲区，“num”是将写入“buf”的数据量（以字节数为单位）。如上所述，在此函数之后会自动调用libcrypto来加密数据，最终将数据发送到远程服务器。因此，我们挂钩“SSL写入”函数，在其中注入额外的代码，并从“buf”中保存“num”字节的数据 - 因此我们获取到未加密的请求数据（即在加密之前获得数据）。

同样地，我们挂钩了两个API - “java.net.SocketOutputStream.socketWrite0”和“java.net.SocketInputStream.socketRead0”，因为它们是Android系统中默认的HTTP API。表格1展示了TraceDroid中的Hooking函数。

评估。理论上，开发人员可以定制自己的TLS库，而不是使用Android系统提供的默认库，而我们的网络Hooking方法在这种情况下将无法工作。但考虑到开发成本和安全问题，我们认为很少有应用这样做。为了评估我们的方法的效果，我们随机调查了80个应用程序，只发现一个应用程序（包名：“com.ss.android.ugc.aweme”，版本17.3）具有自定义的TLS库。

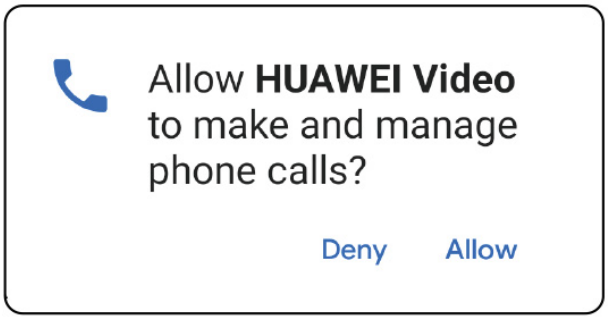
表1 网络捕获的Hooking函数

Traffic hooking	HTTPS	libssl SSL Write libssl SSL Read
	HTTP	java.net.SocketOutputStream.socketWrite0 java.net.SocketInputStream.socketRead0
Call stack hooking	HTTPS	ConscryptFileDescriptorSocket$SSLOutputStream < br/ > ConscryptFil$ SSLInputStream
	HTTP	java.net.SocketOutputStream.socketWrite0 java.net.SocketInputStream.socketRead0

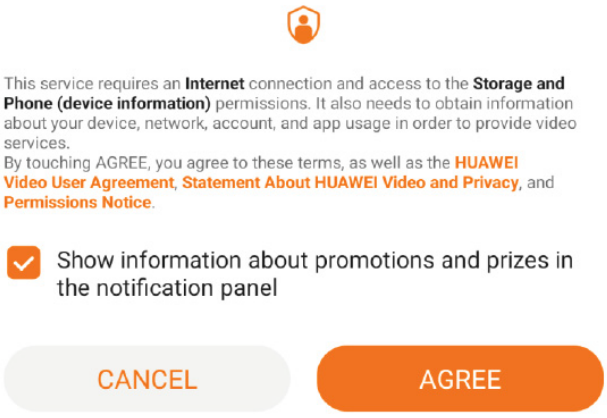
3.2 流量触发

在应用程序执行之前的障碍。为了捕获应用程序的流量，我们需要安装和执行应用程序以产生网络流量。但是，随着Android系统的演进，我们发现在现代Android系统中运行应用程序之前存在新的障碍。首先，应用程序可能会在用户进入应用程序之前请求权限。Google在Android 6.0（API 23）之后引入了一个运行时权限授权机制[2]，而像“android.permission.CALL_PHONE”这样的危险权限必须在运行时授予。如图2 a)所示，可能会出现一个请求权限的对话框，阻止应用程序执行，除非用户确认权限请求。其次，许多应用程序在启动时会出现各种弹出提示或闪屏。如图2 b)所示，提示对话框可能会呈现用户协议或隐私政策，定义各方的责任，提示对话框还需要用户确认。如图2 c)所示，闪屏通常是应用程序加载时的介绍，用户必须在这些屏幕上向左/右滑动以完成过程。

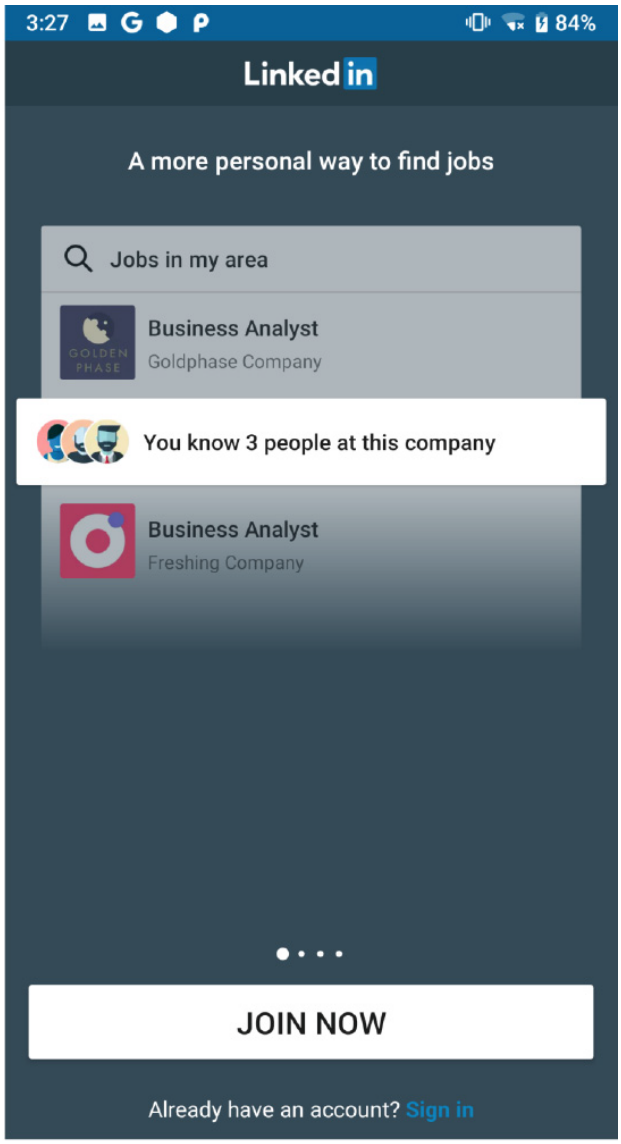
为了缓解这些障碍，我们设计并实现了一个自动工具obsCleaner来满足它们。其思想是识别屏幕上与障碍相关的Android小部件，并进行某些UI操作来完成它们。具体而言，对于权限请求和弹出提示，我们随机测试了200个应用程序，并提炼了与这些障碍相关的关键词列表（例如，“ok”，“agree”，“continue”和“start”）。根据关键词列表，在屏幕上定位障碍物的坐标，并模拟人类操作来生成相关事件以满足它们。我们认为这项工作是一次性的努力，因为这些单词不经常更新，而且列表可以很容易地扩展。对于闪屏，我们滑动屏幕一定的次数K，我们将K设置为5，因为我们的手动测试中所有的200个应用程序都可以在5次内满足。我们已经在我们的Github仓库上发布了obsCleaner，以促进未来的研究。



a) Dynamic permission authorization



b) User agreement



c) Splash screens

图2. 应用程序执行前的障碍

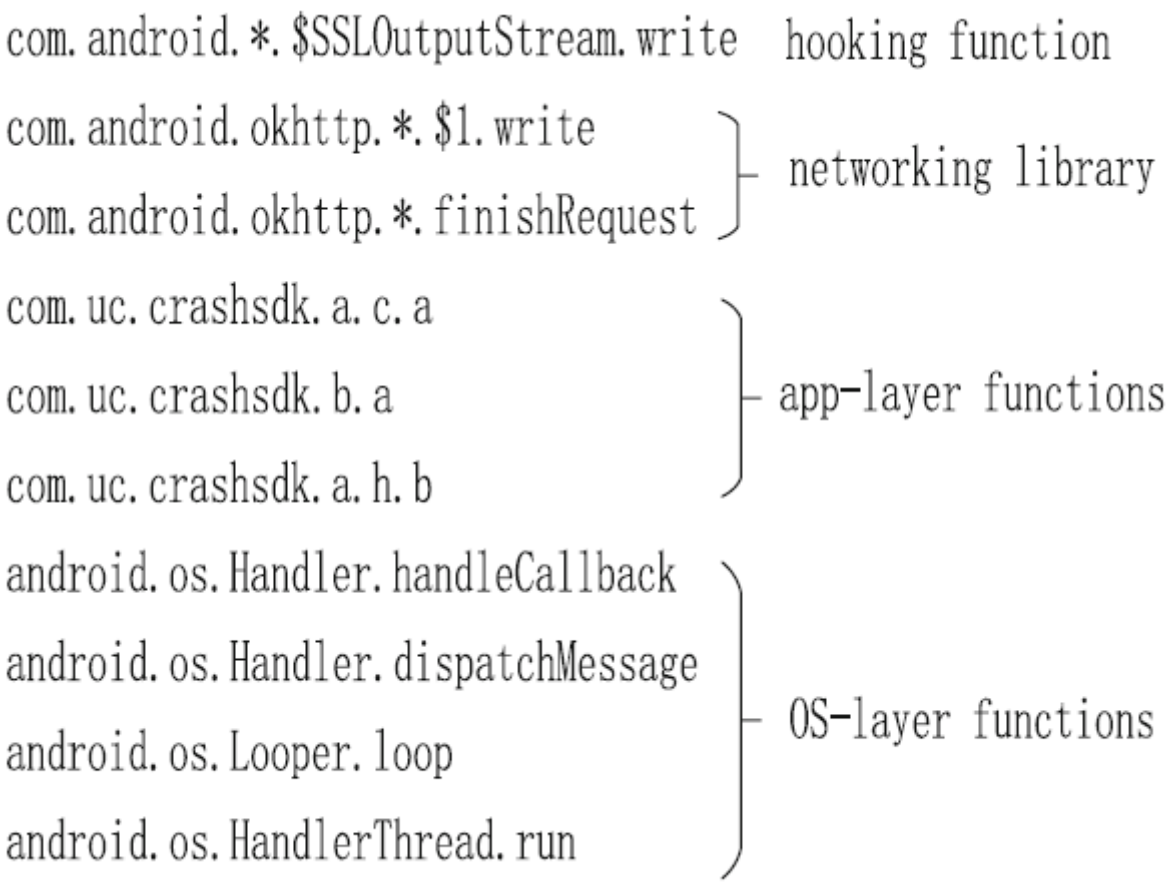


图3. 调用栈示例

触发更多的流量。网络流量分析中的一个重要任务是尽可能生成更多的流量。以前的研究通常使用Android UI模糊测试工具Monkey来触发应用程序的流量。然而，Monkey的随机事件在触发流量方面并不高效。在本文中，我们使用我们之前的工具AutoClick[4]来完成这个任务。AutoClick是一个轻量级、高效的工具，用于触发网络流量。它基于“先点击”的原则，自动点击Android活动中的所有“可点击”布局元素。我们之前的工作表明，在有限的时间内，它在生成不同的流量方面比Monkey高效了约85%!。

3.3 第三方库识别

TPL识别是指在应用程序中检测TPL的存在，这是促进细粒度流量分析的重要任务。我们手动使用现有的TPL识别工具[14、15、28、35]对20个随机选择的应用程序进行了测试，以查看它们的准确性。不幸的是，结果表明，尽管它们在已知的TPL上表现良好（即它们已经训练过的TPL），但对于新出现的TPL（如“com.bytedance.*”）的准确性非常低，这与以前的研究结论一致[39]。然而，新出现的TPL通常会产生大量的流量，因为大多数应用程序都集成了它们，使它们成为我们研究中不可或缺的一部分。

幸运的是，以前的研究[29]表明，从相同TPL生成的网络流量在不同的应用程序中具有类似的网络行为。也就是说，它们的HTTP(S)请求在结构上相似。

代码1：TPL识别

Algorithm 1: TPL identification

Input: $G : request \rightarrow call\_stack$: a visited graph mapping HTTP(S) requests to the corresponding call stacks
Output: *libs*: TPLs in apps

```
libs ← ∅;
Identify domains visited by all requests in G;
for domain ∈ domains do
    Obtain reqs that visit domain, and reqs ∈ G⟨request⟩;
    clusters ← performClustering(reqs);
    for cluster ∈ clusters do
        for req ∈ cluster do
            cstack ← G(req);
            cstack′ ← filter(cstack) by removing hooking and networking
            libraries;
            for call ∈ cstack′ do
                lib ← truncate(call);
                libs.append(lib);
            libs.deduplicate() by removing redundant libs;
return libs;
```

``

- Input: G : request → call stack: a visited graph mapping HTTP(S) requests to the corresponding call stacks
- Output: libs: TPLs in apps
- 1 libs ← ø;
- 2 Identify domains visited by all requests in G;
- 3 for domain ∈ domains do
  - 4 Obtain reqs that visit domain, and reqs ∈ G ⟨request⟩ ;
  - 5 clusters ← perf ormClustering(reqs);
  - 6 for cluster ∈ clusters do
    - 7 for req ∈ cluster do
    - 8 cstack ← G(req);
    - 9 cstack′ ← f ilter(cstack) by removing hooking and networking libraries;
    - 10 for call ∈ cstack′ do
      - 11 lib ← truncate(call);
      - 12 libs.append(lib);
    - 13 libs.deduplicate() by removing redundant libs;
- 14 return libs;

``

以下是算法的完整描述： 输入：请求到调用堆栈的映射图G，其中键是HTTP(S)请求，值是调用堆栈。 输出：应用程序中的TPL列表。

1. 初始化一个空的TPL列表libs。
2. 对于每个请求r和相应的调用堆栈s，执行以下步骤： a. 从s中提取应用程序的包名pkg。 b. 检查pkg是否已知，即是否已经标识为使用某个TPL。 c. 如果pkg未知，则执行以下步骤： i. 从s中提取所有的库名。 ii. 对于每个库名lib，检查它是否已知，即是否已经标识为某个TPL。 iii. 如果lib已知，则将其对应的TPL添加到pkg的TPL列表中。 iv. 如果lib未知，则执行以下步骤： 1. 从库名中提取库的名称和版本号。 2. 使用TPL identification工具识别库。 3. 将库添加到libs中，并将其对应的TPL添加到pkg的TPL列表中。 d. 将pkg标识为使用pkg的TPL。
3. 返回libs。

因此，我们提出了一种无监督的方法，将网络请求与调用堆栈关联起来，以识别TPL。算法1展示了我们的TPL识别过程。首先，我们获取被超过M个应用程序访问的域名，其中M被经验性地设置为10（第2行）。对于每个域名，我们提取所有对应请求的签名作为reqs（第4行），其中一个请求是〈协议，方法，URL，主机，路径，键列表〉。执行聚类分析，将具有相同签名的请求分组在一起（第5行）。然后，我们获取每个请求对应的调用堆栈，以获得候选TPL（第6行）。一个调用堆栈是一个函数列表，其中包含从下到上的代码执行跟踪，如图3所示。我们通过删除第3.1节中的hooking函数和网络库（如Okhttp3）（第9行）来过滤堆栈，以获得候选TPL。接下来，我们截断候选TPL的前三个包名并删除冗余的包名，以获得最终的TPL（第10-13行）。

这里我们以图3中的一个调用堆栈为例。为了方便描述，我们省略了不相关的部分。代码执行跟踪包含四个部分：操作系统层函数、应用程序层函数、网络库和hooking函数。TraceDroid首先删除hooking函数、网络库“com.android.okhttp.”*和操作系统层函数（例如“android.os.”）*。这样，我们就得到了剩余的包名，前缀为“com.uc.crashsdk.*”。我们截断了前三个包名，并对这些包名进行去重，以获得TPL -“com.uc.crashsdk”。

参数设置。我们进行了一项统计，显示有多少个应用程序访问每个域。它展示了一个长尾分布，95.8%!的(MISSING)域被少于10个应用程序访问，因此我们将M设置为10，以获取候选域进行进一步分析。我们已经手动检查了我们的方法识别出的53个TPL，并且它们全部是真实的TPL（结果也发布在我们的Github存储库中）。

4 评价与测量

实现和实验设置。我们使用基于UIAutomator2和Frida的约6K行Python和JavaScript代码实现了TraceDroid。对于障碍物消除，我们使用UIAutomator2 API“XPath（）”根据关键词列表定位障碍物的坐标；对于网络挂钩，我们使用Frida API“Interceptor.attach（）”挂钩函数。对于每个应用程序，TraceDroid安装它，触发它运行10分钟，并在运行时收集网络流量。测试时间结束后，TraceDroid终止应用程序并卸载它。TraceDroid将收集的数据发送到我们的服务器，设备准备好进行下一个测试应用程序。我们的实验使用Android 9的两部Pixel3手机进行了约33天。

4.1 数据集构造

我们从多个应用商店爬取了9,771个应用程序，其中5,503个应用程序来自诸如小米[36]之类的替代应用商店，4,268个应用程序来自Google Play。对于每个应用商店，我们下载每个类别的TOP列表应用程序。如表2所示，TraceDroid收集了16.9 GB的.pcap文件和6.2 GB的调用堆栈文件。通过解析这些文件，我们获得了301,381个HTTP（S）请求及其对应的调用堆栈。我们从捕获的流量中还原了55,843个下载文件和2,914个上传文件。

表2. 实验数据和结果清单

Data	Count	Description
Android apps	9,771	5,503 from alternative markets and 4,268 from Google
.pcap files	16.9 GB	The volume of captured traffic
Call stack	6.2 GB	The volume of function call stacks
Requests	301,381	HTTPS: 174,486 (57.9%) HTTP: 126,895 (42.1%)
Files	58,757	55,843 downloaded files, 2914 uploaded files

数据	数量	描述
Android应用程序	9,771	其中5,503来自替代应用商店，4,268来自Google Play
.pcap文件	16.9 GB	捕获流量的容量
调用堆栈	6.2 GB	函数调用堆栈的容量
请求	301,381	HTTPS：174,486（57.9%）HTTP：126,895（42.1%）
文件	58,757	下载的文件55,843个，上传的文件2914个

4.2 责任分析

为了对主机应用程序和第三方库之间的隐私泄露进行责任分析，我们遵循前一项工作[25]的隐私定义，将隐私数据分为三种类型：1）设备信息（例如IMEI，序列号）；2）位置信息（例如基站，GPS）；3）网络信息（例如WiFi状态，IP地址）。我们收集了我们的两个Pixel3手机的上述信息进行进一步分析。

在我们的实验中，TraceDroid使用第3.3节提出的方法识别了357个第三方库(TPLs)。根据这个结果，我们进行了细粒度的隐私泄露责任分析，即展示隐私数据是由TPLs还是主机应用程序泄露。我们将传输隐私数据的请求称为隐私请求，并使用5元组 - 〈appPackageName，source IP，sourcePort，destIP，destPort〉将HTTP（S）请求与调用堆栈相关联。TraceDroid搜索隐私请求的调用堆栈，如果调用堆栈中存在TPL包名，则认为TPL发起了隐私请求；否则，它是由主机应用程序发起的。我们发现，TPLs发起了44.45%的隐私请求。更详细地说，39%的设备信息，45.8%的位置信息和42.6%的网络信息泄露是由TPLs传输的。我们的结果表明，TPLs已成为一个不可忽略的隐私泄露渠道，应对TPLs进行更多的监管。

4.3 通过文件泄露隐私

以往的研究主要集中在网络请求分析上。然而，文件也是网络流量的重要组成部分，令人惊讶的是，没有研究分析文件内容，这使得它长期以来被研究界忽视。在本节中，我们首次尝试检查文件的内容。TraceDroid利用我们的数据包级挂钩能力，恢复了2,914个上传的文件（涉及329个应用程序）。使用我们的责任分析，我们发现1,793个文件是由主机应用程序传输的，1,121个文件是由TPLs传输的（涉及84个TPLs）。

首先，我们手动分析了由TPLs发送的文件，并发现所有这些文件都不可读，并且大多数文件没有后缀指示文件类型。为了进一步调查，我们使用二进制查看器检查这些文件以分析编码或加密方法，并发现：1）由不同TPLs发送的文件在文件名上具有相似之处。例如，810个文件名以“stm d”或“stm p”结尾，154个文件名以“*.*.pvuv.log”结尾。2）“stm_d”和“stm_p”文件中的前十个字节是相同的。

其次，TraceDroid检查所有由主机应用程序生成的文件，并在必要时尝试解压缩它们。然而，由于编码格式各不相同，很难自动处理它们，因此我们对这些文件进行了半自动分析。我们首先手动分析一些文件，看看它们是否可读，然后自动分析相似的文件（具有相同的文件名或由同一主机应用程序发送）。令人惊讶的是，我们发现这些文件往往包含比请求更详细的隐私信息。例如，一个文件包含网络状态（移动运营商APN名称，它指示运营商名称或WiFi名称）、应用程序名称、应用程序版本、分发渠道、客户端时间戳、设备型号、操作系统版本、Android ID，文件的其余部分包含用户操作，如“用户在上午11:00在主活动中单击按钮”。更详细的信息如表3所示。

表3. 通过文件传输的信息

Process info	Memory info	CPU info
process id, thread id	RAM total,free,available	abi,processor,manufacturer, utility
-		
Device info	Battery info	Disk Info
model, brand	voltage, health, temperature	disk total, available, block size
Root status	Kernel info	Network info
isRoot, su permission	Linux kernel info, Dalvik version	WiFi, mobile operator, SIM card
-		
Installed app list	Location info	Device IDs
apps installed in the device	gps, city	IMEI,IMSI,SN,UUID,UDID
-		
User action info	Permission info	Phone number

Process info	Memory info	CPU info
pvuv info	permissions app applied	user phone number

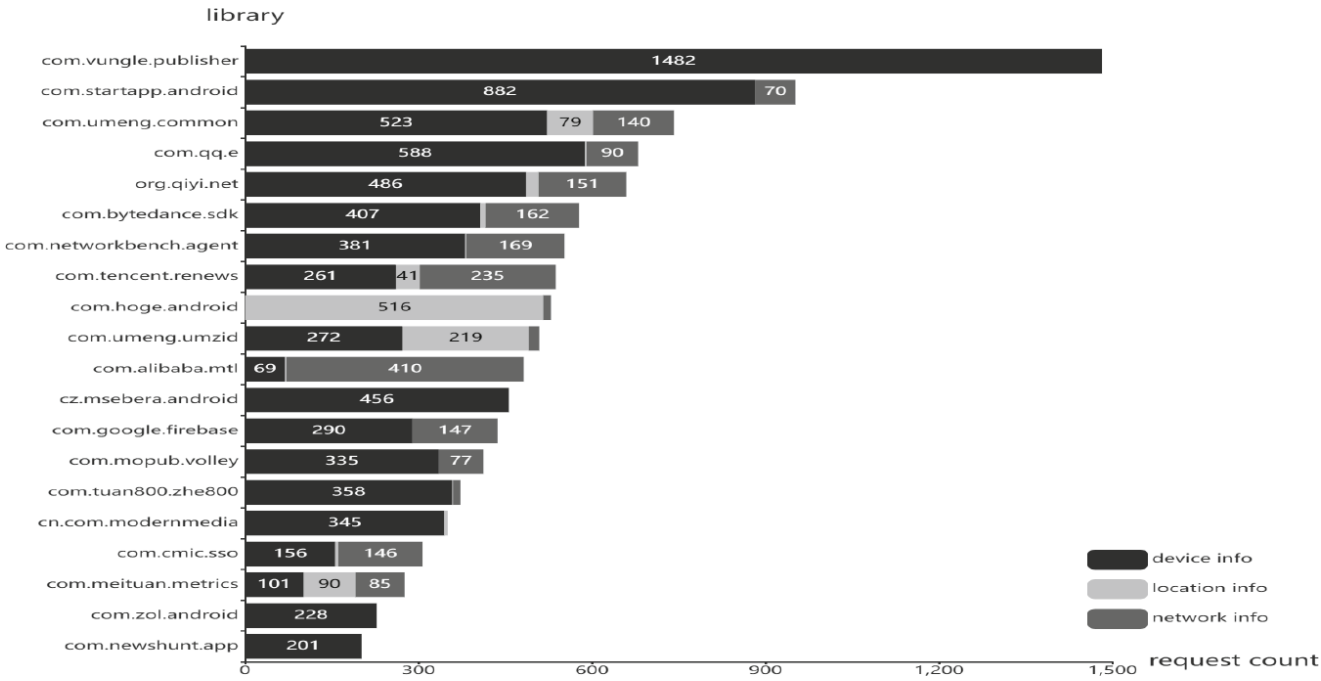


图4. TPLs泄露的隐私信息

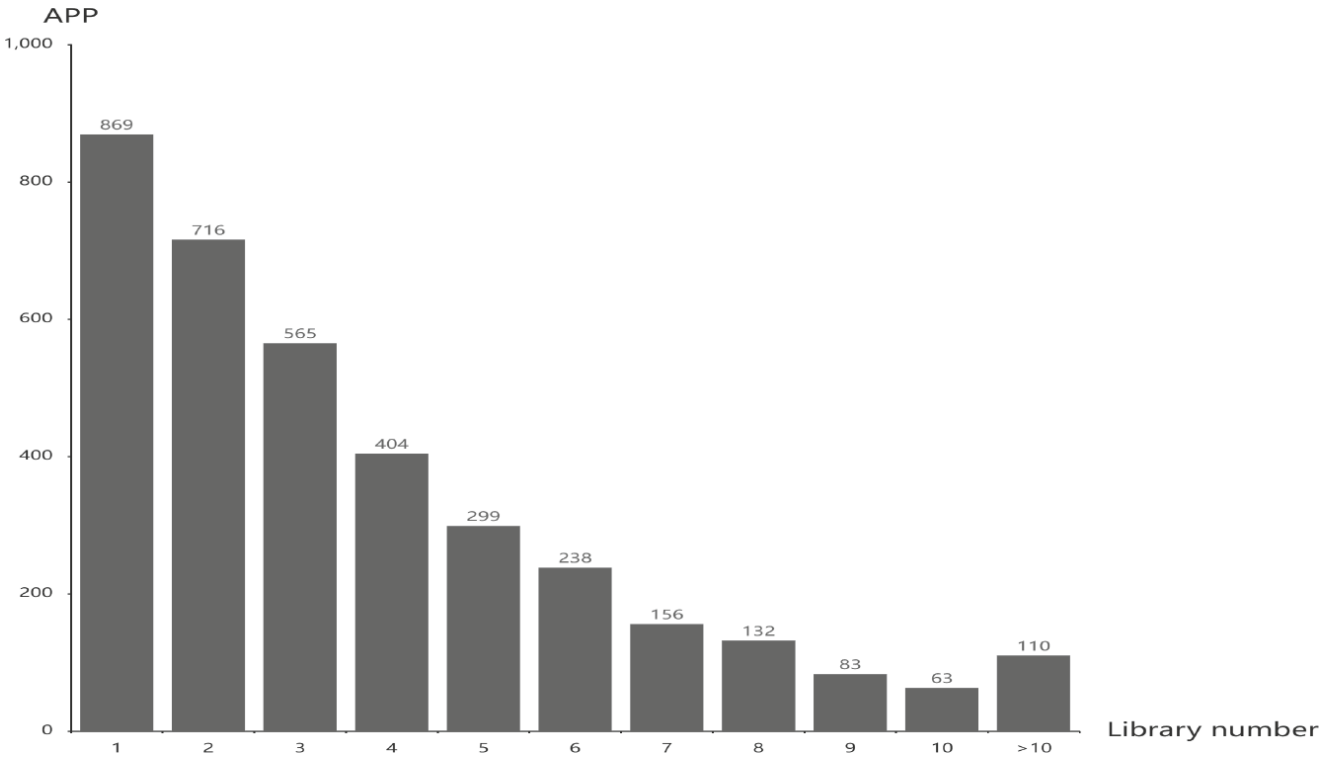


图5. 每个应用程序的TPL数量

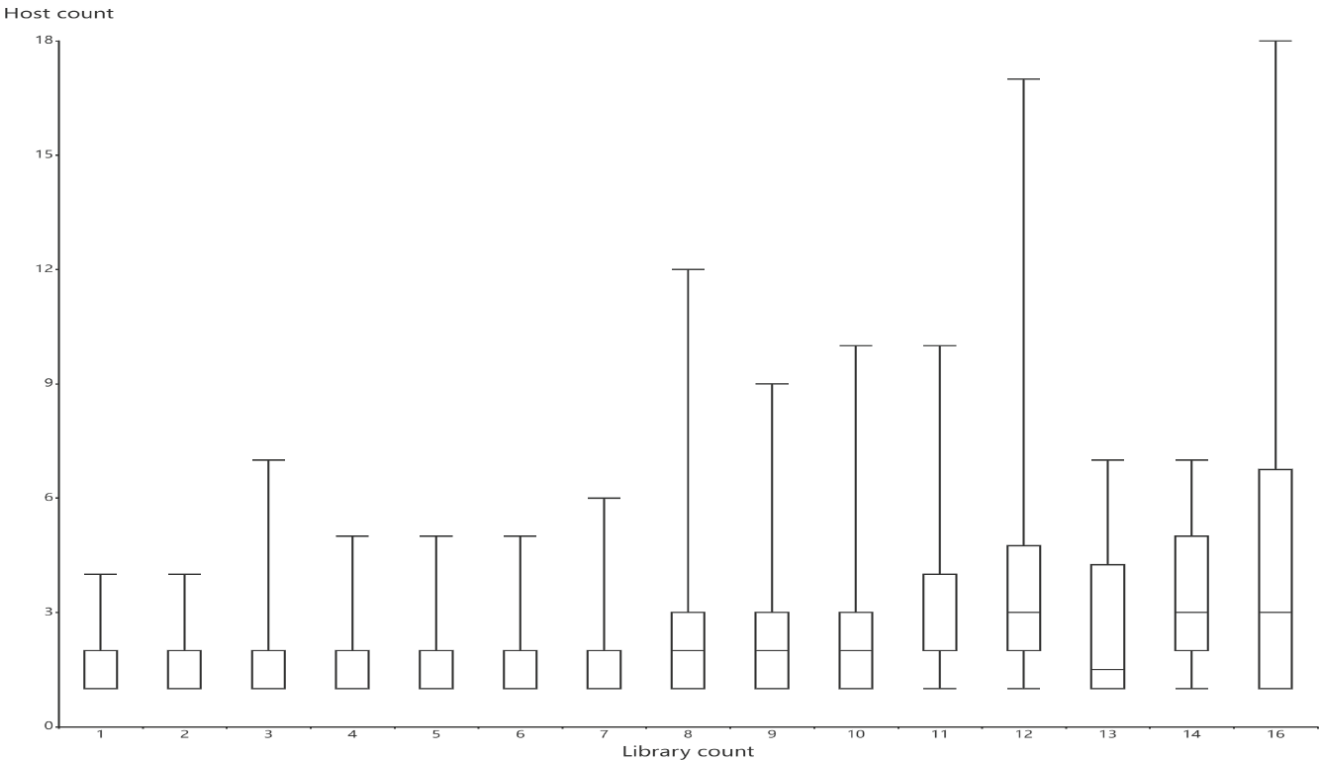


图6. TPL数量和隐私域

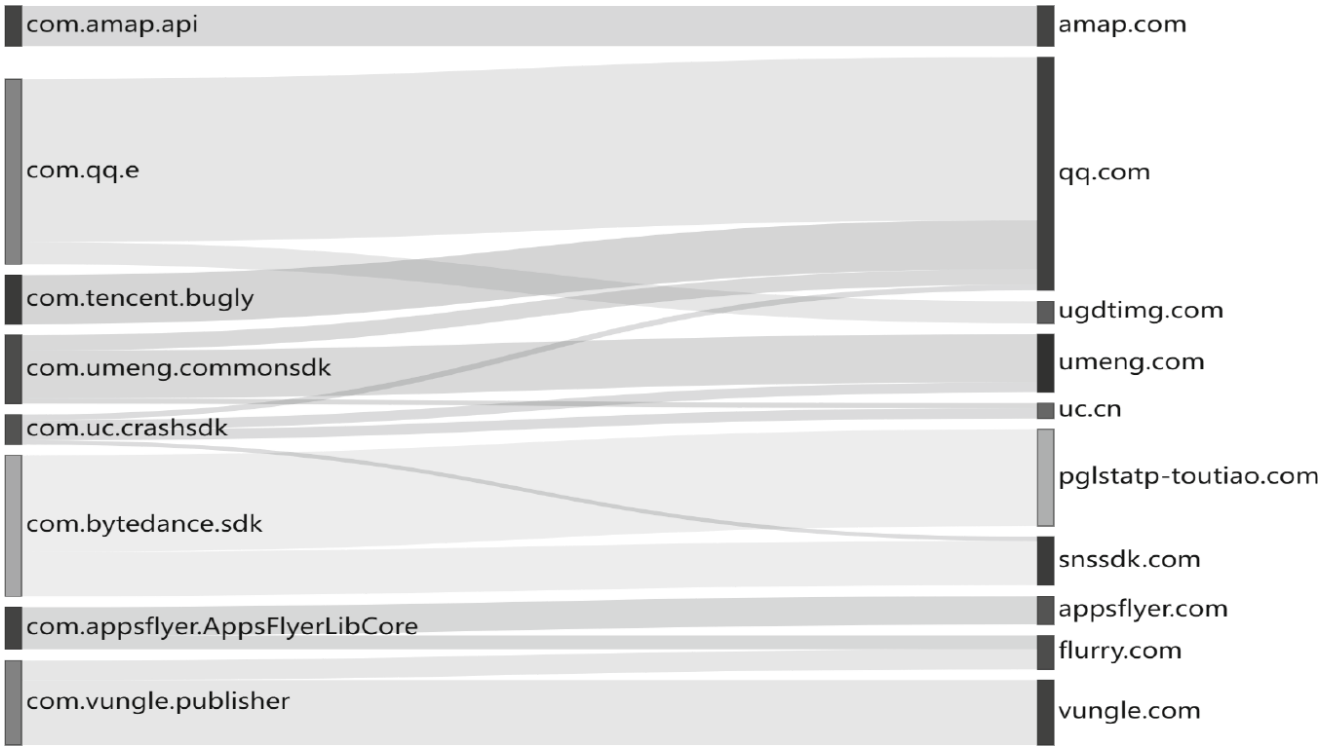


图7. TPL访问图

总之，在本节中，我们对从用户设备传输的文件进行了细粒度的隐私泄露分析。我们的发现如下：1）由TPL传输的文件具有良好的编码/加密，可以防止文件内容被提取。然而，一些TPL可能使用相同的编码/加密方法，如果有人反转了一个TPL，他可能有能力解码其他TPL的文件。2）由主机应用程序传输的文件保护不佳，并且它们包含比HTTP（S）请求更多的详细隐私数据，这一点长期以来一直被研究界忽视。

4.4 隐私数据收集

过度数据收集。图4显示了前20个TPL传输的隐私数据。它表明，4个TPL只传输一种类型的隐私数据，6个TPL传输两种类型的隐私数据，10个TPL传输三种类型的隐私数据。我们进一步的分析表明，33%的应用程序同时与TPL传输隐私数据，这意味着主机应用程序和TPL都收集隐私数据以提供其服务。然而，用户无法控制这些数据何时和多频率被收集，以及它们将如何被使用。

我们打算回答的另一个问题是，一个应用程序集成了多少个TPL，而且TPL之间是否存在过度数据收集？图5显示，78.5%的应用程序集成少于5个TPL，18.5%的应用程序集成6至10个TPL，仅约3%的应用程序集成超过10个TPL。图6显示TPL数量与接收隐私数据的域数量之间的关系。我们可以看到，一个应用程序中集成的TPL越多，接收隐私数据的域就越多，表明TPL之间存在过度数据收集的情况。

我们的研究结果表明，主机应用程序和TPL之间存在广泛的过度数据收集。考虑到TPL在应用程序中的广泛集成，我们指出应用程序开发人员应该更加关注TPL的收集行为，因为如果TPL违反规定收集隐私信息，将对应用程序的声誉产生破坏性影响。

跨库数据收集（Cross-Library Data Harvest）是近年来的一种新的攻击模型，与从用户设备或服务器收集隐私数据不同。它指的是恶意的第三方库或工具包从同一主机应用程序中收集其他第三方库或工具包的隐私数据。我们将这种攻击模型称为交叉库数据收集（XLDH）。Wang等人[33]首次指出了XLDH的威胁。在他们的研究中，恶意TPL检查其主机应用程序中是否存在受害TPL，并使用Java反射机制调用受害TPL的API，从而获取其隐私数据。我们将这种威胁模型称为基于反射的XLDH。Wang等人提出了一种识别异常Java反射机制以检测此威胁的方法。

在我们的实验中，我们揭示了一种新的基于仪器的XLDH模型 - instrumentation-based XLDH。在这种模型中，TPL使用仪器技术从同一主机应用程序中的其他TPL中收集数据。具体而言，我们基于“调用堆栈链不一致”的想法进行了分析，即我们检查一个TPL在不同应用程序中如何被调用，以查看它们之间是否存在任何不一致之处。我们选择了数据集中使用最广泛的20个TPL，提取它们的调用堆栈，比较它们在不同应用程序中的调用堆栈链，并搜索“调用堆栈链不一致”的情况。例如，我们分析一个名为“com.uc.crashsdk”的TPL的所有调用堆栈，并发现在主机应用程序“com.wondertek.paper”中，它调用了一个名为“com.networkbench.agent”的TPL。然而，在其他应用程序中，“com.uc.crashsdk”并没有调用“com.networkbench.agent”。为了弄清楚为什么会出现这种现象，我们手动阅读了这两个TPL的开发人员指南。我们发现它们属于不同的供应商，并且根据它们的开发人员指南，“com.uc.crashsdk”不应该调用“com.networkbench.agent”，这意味着这种现象是可疑的。为了找出真相，我们对应用程序进行了反编译，手动分析了源代码、调用堆栈和网络请求，并发现“com.networkbench.agent”通过仪器技术对“com.uc.crashsdk”进行了隐私数据收集，并最终将隐私数据传输到“networkbench.com”域名，这与我们在数据集中观察到的请求相一致。基于“调用堆栈链不一致”的想法，我们在应用程序语料库中发现了两个具有基于仪器的XLDH行为的应用程序，两者都从应用商店下载了数万次。更糟糕的是，基于仪器的XLDH可以回避Wang等人[33]提出的检测方法，因为它不依赖于Java反射机制。我们认为这是一种新的XLDH模型，未来的研究应该研究大规模、自动的方法来应对这种威胁。

总之，在本节中，我们对现代应用程序中的数据收集进行了实证分析。我们的发现如下：1) 主机应用程序和TPL之间存在隐私过度收集的问题。开发人员应该更加关注TPL的行为，因为

如果它们违反规定规则，可能会损害应用程序的声誉。2）跨库数据收集是一种新的和隐蔽的收集隐私数据的方式，应该提出新的方法来自动检测这种威胁。

4.5 TPL和域之间的关系

在本节中，我们旨在明确TPL和其访问域之间的关系。Libspector [40]得出结论，TPL和域之间没有严格的一对一关系，但是它们之间的确切关系仍不清楚。

我们收集了53个手动检查的TPL的访问域。图7是一些示例TPL的访问图：左侧是TPL名称，右侧是它们的访问域，左右之间的流表示流量。图表显示，TPL通常访问多个域，一个域可能会被多个TPL访问，这与Libspector [40]的结论一致。然而，为什么会出现这种现象？它们之间的关系是什么？为了弄清楚这些问题，我们阅读了TPL的开发文档，并在应用程序源代码中搜索域来研究它们之间的关系。我们发现，对于特定的TPL，访问域可以分为三种类型：1）自有域：由TPL提供者拥有，域名将被TPL跨应用程序访问；2）授权域：为那些想要使用其服务的人提供授权机制的域。当一些TPL需要相互合作或提供公共服务时，这很有用。授权机制可以是一个“密钥”，任何想要使用该服务的人都必须注册或申请一个“密钥”来访问域提供的API；3）主机应用程序域：由主机应用程序提供商拥有。应用程序使用此域来提供其服务。

这里我们以一个名为“com.antutu.ABenchMark”的应用程序为例。它集成了一个名为“com.umeng.commonsdk”的库。对于这个库，“umeng.com”是一个自有域，“qq.com”是一个授权域。请注意，由“com.umeng.commonsdk”初始化的请求访问了一个名为“autovote.antutu.net”的域。图7没有显示该域，因为它们之间的流量非常小。但事实上，“autovote.antutu.net”是“com.umeng.commonsdk”的主机应用程序域。 “umeng”访问“antutu.net”的原因是TPL具有回调API，供调用者交付其服务，当开发人员调用它时，它们可以使用回调API访问其域。

总之，在本节中，我们努力寻求TPL和域之间的明确关系。基于大量的数据分析，我们得出结论，TPL和域之间存在多对多的关系。此外，我们将域分为三种类型，并阐述了TPL和域之间访问现象背后的原因。

5 总结

本文提出了一种新的框架TraceDroid，用于网络流量捕获和分析。与现有的工作相比，TraceDroid更加稳健和高效，因为它解决了流量混合、代理逃避和SSL固定的限制。借助TraceDroid的帮助，我们提出了一种无监督的方法来识别第三方库（TPL），并对9,771个真实世界应用程序进行了大规模综合分析，揭示了TPL和文件导致的隐私泄露，并评估了TPL和主机应用程序之间超量数据收集的现象。此外，我们揭示了一种新的和隐蔽的数据收集方式--基于instrumentation的跨库数据收集，这需要进一步研究。最后，我们首次尝试给出TPL和它们访问域之间的明确关系。为促进未来的研究，我们在我们的Github代码库上发布了所有的源代码和实验结果。

致谢。本工作得到中国国家重点研发计划的支持（编号：2019YFB1005205）。

参考文献

1. https://developer.android.com/reference/java/net/HttpURLConnection (2021)
2. https://developer.android.google.cn/about/versions/marshmallow/android-6.0-changes?skip cache=false (2021)
3. Async-http (2021). https://github.com/android-async-http/android-async-http
4. AutoClick (2021). https://github.com/BlcDle/AutoClick
5. BroingSSL (2021). https://boringssl.googlesource.com/boringssl/
6. Caputo, D., Pagano, F., Bottino, G., Verderame, L., Merlo, A.: You can't always get what you want: towards user-controlled privacy on android. arXiv preprint arXiv:2106.02483 (2021)
7. Charles (2021). https://www.charlesproxy.com/
8. Dong, F., et al.: Frauddroid: automated ad fraud detection for android apps. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 257–268 (2018)
9. Dong, F., Wang, H., Li, L., Guo, Y., Xu, G., Zhang, S.: How do mobile apps violate the behavioral policy of advertisement libraries? In: Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications, pp. 75–80 (2018)
10. Fiddler (2021). https://www.telerik.com/fiddler
11. HttpClient (2021). https://hc.apache.org/httpcomponents-client-5.1.x/
12. Li, L., et al.: ICCTA: detecting inter-component privacy leaks in android apps. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 280–291. IEEE (2015)
13. Li, L., Li, D., Bissyand´e, T.F., Klein, J., Le Traon, Y., Lo, D., Cavallaro, L.: Understanding android app piggybacking: a systematic study of malicious code grafting. IEEE Trans. Inf. Forensics Secur. 12(6), 1269–1284 (2017)
14. Li, M., et al.: Libd: scalable and precise third-party library detection in android markets. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 335–346. IEEE (2017)
15. LibRadar (2021). https://github.com/pkumza/LibRadar
16. Liu, T., Wang, H., Li, L., Bai, G., Guo, Y., Xu, G.: Dapanda: detecting aggressive push notifications in android apps. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 66–78. IEEE (2019)
17. Liu, T., et al.: Maddroid: characterizing and detecting devious ad contents for android apps. In: Proceedings of The Web Conference 2020, pp. 1715–1726 (2020)
18. Lumen (2021). https://www.haystack.mobi/
19. Ma, Z., Wang, H., Guo, Y., Chen, X.: Libradar: fast and accurate detection of third-party libraries in android apps. In: Proceedings of the 38th International Conference on Software Engineering Companion, pp. 653–656 (2016)
20. Meddle (2021). https://meddle.mobi/
21. Okhttp: https://square.github.io/okhttp/ (May 2021)

22. OpenFeign (2021). https://github.com/OpenFeign/feign
23. Razaghpanah, A., et al.: Haystack: In situ mobile traffic analysis in user space, pp. 1–13. arXiv preprint arXiv:1510.01419 (2015)
24. Reardon, J., Feal,´A., Wijesekera, P., On, A.E.B., Vallina-Rodriguez, N., Egelman, S.: 50 ways to leak your data: an exploration of apps' circumvention of the android permissions system. In: 28th USENIX security symposium (USENIX security 2019), pp. 603–620 (2019)
25. Ren, J., Rao, A., Lindorfer, M., Legout, A., Choffnes, D.: Recon: revealing and controlling pii leaks in mobile network traffic. In: Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, pp. 361–374 (2016)
26. RestTemplate (2021). https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html
27. Retrofit (2021). https://square.github.io/retrofit/
28. Soh, C., Tan, H.B.K., Arnatovich, Y.L., Narayanan, A., Wang, L.: Libsift: automated detection of third-party libraries in android applications. In: 2016 23rd AsiaPacific Software Engineering Conference (APSEC), pp. 41–48. IEEE (2016)
29. Taylor, V.F., Spolaor, R., Conti, M., Martinovic, I.: Robust smartphone app identification via encrypted network traffic analysis. IEEE Trans. Inf. Forensics Secur.13(1), 63–78 (2017)
30. Tongaonkar, A., Dai, S., Nucci, A., Song, D.: Understanding mobile app usage patterns using in-app advertisements. In: Roughan, M., Chang, R. (eds.) PAM 2013. LNCS, vol. 7799, pp. 63–72. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36516-4 7
31. Volley (2021). https://developer.android.com/training/volley/index.html/
32. Wang, H., et al.: Beyond google play: a large-scale comparative study of Chinese android app markets. In: Proceedings of the Internet Measurement Conference 2018, pp. 293–307 (2018)
33. Wang, J., et al.: Understanding malicious cross-library data harvesting on android. In: 30th USENIX Security Symposium (USENIX Security 2021), pp. 4133–4150 (2021)
34. Wang, W., et al.: Constructing features for detecting android malicious applications: issues, taxonomy and directions. IEEE Access 7, 67602–67631 (2019)
35. Wang, Y., Wu, H., Zhang, H., Rountev, A.: Orlis: obfuscation-resilient library detection for android. In: 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 13–23. IEEE (2018)
36. XiaoMi: Xiaomi app store (2021). https://app.mi.com/
37. Xu, Q., Erman, J., Gerber, A., Mao, Z., Pang, J., Venkataraman, S.: Identifying diverse usage behaviors of smartphone apps. In: Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference, pp. 329–344 (2011)
38. Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., Wang, X.S.: Appintent: analyzing sensitive data transmission in android for privacy leakage detection. In:

Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 1043–1054 (2013)
39. Zhan, X., et al.: Automated third-party library detection for android applications: are we there yet? In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 919–930. IEEE (2020)
40. Zungur, O., Stringhini, G., Egele, M.: Libspector: context-aware large-scale network traffic analysis of android applications. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 318–330. IEEE (2020)