Computer Science and Application 计算机科学与应用, 2018, 8(11), 1734-1743
Published Online November 2018 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2018.811192

Hans汉斯

SLDroid: Detection of the Usage and Risk of DCL in the Third-Party Market

Tianyang Li, Haoliang Cui, Shaozhang Niu

Beijing Key Laboratory of Intelligent Telecommunication Software and Multimedia, Beijing University of Posts and Telecommunications, Beijing
Email: 741310940@qq.com

Received: Nov. 4th, 2018; accepted: Nov. 13th, 2018; published: Nov. 20th, 2018

Abstract

Due to the long review cycle for mobile applications in the application market and the consideration of reducing the size of Android Package (APK), dynamic code loading (DCL) technology has been introduced into mainstream Android applications. DCL technology allows applications to load or execute external binaries, which can be loaded locally or downloaded from the network. It has solved the problem of long examination cycle in the application market. However, DCL technology also brings some security problems: only when the application runs or executes key points can trigger DCL. This increases the difficulty of market review application, and also gives malicious attackers an opportunity to attack applications. So we designed a tool SLDroid to automatically detect and analyze DCL. We used SLDroid to analyze the application of App Treasure Market. Our results were based on the use of DCL in 1934 apps of 20 different types in App Treasure Market. We investigated the possible risks of DCL usage in the application.

Keywords

Dynamic Code Loading, Third-Party Market, Automatic Detection

SLDroid：检测第三方市场DCL使用情况及风险

李天阳，崔浩亮，牛少彰

北京邮电大学智能通信软件与多媒体北京市重点实验室，北京
Email: 741310940@qq.com

收稿日期：2018年11月4日；录用日期：2018年11月13日；发布日期：2018年11月20日

摘　要

由于应用市场对于移动应用的审查周期较长，并且出于减少APK包大小的考虑，现在主流的Android应

文章引用：李天阳，崔浩亮，牛少彰. SLDroid: 检测第三方市场 DCL 使用情况及风险[J]. 计算机科学与应用，2018, 8(11): 1734-1743. DOI: 10.12677/csa.2018.811192

用中都引入了动态代码加载(DCL)技术。DCL技术允许应用加载或者执行外部的二进制文件，二进制文件可以从本地加载，也可以从网络下载。很好的解决了应用市场上架慢审查周期长的问题。然而DCL技术也带来了一些安全问题：只有在应用运行时或者关键点才会触发DCL功能。这增加了市场审查应用的难度，同时也给了恶意攻击者攻击应用的机会。所以我们设计了一个自动检测和分析DCL使用情况的工具SLDroid，我们使用SLDroid分析了应用宝市场应用，我们的调查结果基于应用宝市场20种不同类型的1934款应用使用DCL的情况，调查应用可能存在的DCL使用风险。本文介绍了我们设计实现的工具SLDroid，以及针对应用宝市场中应用使用DCL情况的初步结果。

关键词

动态代码加载，第三方市场，自动检测

1. 引言

随着移动市场的快速增长和 Android 操作系统的不断完善，Android 操作系统的活跃设备数达到了 20 亿[1]，已经成为全球最受欢迎的移动操作系统。Android 应用的数量也迅速增长。根据 AppBrain [2]统计：到 2018 年 3 月 21 日为止，GooglePlay 上的应用程序数量已经达到了 3,624,700，同比 2017 年 3 月，GooglePlay 上应用程序数量增加了 21%。

针对用户快速更新的需要以及同类应用竞争的压力。现在主流的 Android 应用广泛使用了动态代码加载(DCL)技术。DCL 技术允许应用加载或者执行外部的二进制文件，可以从本地加载，也可以从网络下载，其主要目的是为了达到让用户不重新安装应用就能升级应用。根据 Zhengyang Qu [3]的调查，在他收集的 Google 市场了 58,739 种应用程序中，69.60%的应用使用了 DCL。DCL 技术确实为应用的更新和维护带来了方便。

然而 DCL 技术也产生了一些安全问题。通过使用 DCL，应用程序开发者可以在运行时加载没有经过审查的代码片段。Zhengyang Qu [3]等人实现了一个示例程序，通过 DCL 技术绕过 Google 市场的安全检查。除了绕过审查，不当使用 DCL 也会产生安全问题。Sebastian Poeplau [4]等人分析了 1632 个流行应用，发现 151 个应用易受 DCL 相关的代码注入攻击。越来越多的研究发现 DCL 易被恶意利用且不当使用会带来一些安全隐患。

所以本文旨在进一步分析 DCL 相关的安全问题。我们的目标是第三方市场，我们想要研究：1) 与审查严格的 Google 市场相比，第三方市场使用 DCL 情况是不是更为频繁？是不是更容易产生安全问题？2) 哪一类型的应用更容易产生安全问题，需要引起监管者重视？为了调查 DCL 的使用情况以及安全问题，我们分析设计并实现了一个 DCL 分析工具，我们选择应用宝市场，分析应用宝市场中应用的 DCL 使用信息。我们的贡献主要如下：

a) 设计并实现 SLDroid 工具：自动检测和分析应用使用 DCL 技术的情况。

b) 分析应用宝市场使用 DCL 技术的情况，与前人分析 Google 市场 DCL 使用情况进行对比研究。

2. 相关工作

作为全球最受欢迎的移动操作系统，Android 引入了动态代码加载机制(DCL)技术。用于代码更新和

功能扩展。但是同时也带来了新的安全风险。前人针对 DCL 技术做了细致的研究，如 Darell JJ Tan [6]经过研究认为使用了 DCL 的应用有可能存在安全问题。Yury Zhauniarovich [5]调查了应用中 DCL 使用情况，发现动态加载的代码很多没有进行完整性验证。

针对恶意行为的检测，Martina Lindorfer* [8]，Igor S [10]通过提取特征的方式检测恶意软件，T Book [9] Michael Grace [7]通过建立白名单库进行恶意软件分析。J Sahs [11], K Allix [12]通过机器学习检测和评估恶意软件。而由于 DCL 只有在特定条件下才会触发加载，DCL 相关检测与传统的恶意行为检测不同，需要应用运行时才能捕捉到相关行为，前人的通用性检测不适合。

遗憾的是针对 DCL 的研究相对比较少。Zhengyang Qu [3]等人通过重打包的方式修复 DCL 可能存在的问题。Sebastian Poeplau! [4]希望通过修改系统框架的方式进行检测和修复 DCL 相关问题。Luca Falsina [13]则提出了一套 DCL 相关规范 API，指导安全使用 DCL。我们认为修改系统不具有普遍性，反编译应用也改变了应用原本的签名，安全性难以保证。DCL 使用规范很难得到推广，都存在着一定的局限性。

所以在本文中，我们在 Zhengyang Qu [3]等人研究的基础上，研究第三方市场 DCL 相关的安全问题，回答 1 与审查严格的 Google 市场相比，第三方市场使用 DCL 情况是不是更为频繁？是不是更容易产生安全问题？2 哪一类型的应用更容易产生安全问题，需要引起监管者重视？为了回答这两个问题，我们设计并实现了 SLDdroid 工具，分析应用宝市场应用使用 DCL 的相关问题。。

3. 背景知识

动态代码加载：通过在运行时加载外部代码，改变应用程序行为的技术。由于 Java 的类加载机制，通过 Classloader 可以延迟加载 java 字节码文件(动态加载)。如图 1 所示，Android 继承于 java 的特性，通过虚拟机 Dalvik/ART 动态加载各类资源文件。同时 Android 为我们提供了用于动态加载的类加载器 DexClassloader：可以在应用运行时加载 jar、apk 和 dex 等 Android 相关的资源文件。

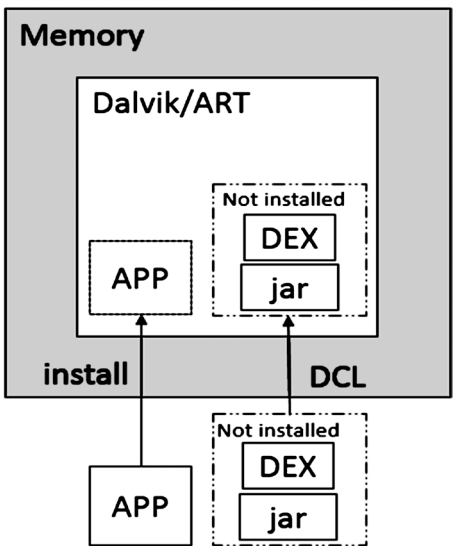

Figure 1. Dynamic code loading model
图 1. 动态代码加载模型

越来越多的应用开发者选择使用 DCL 技术，DCL 技术有以下好处：

1) 减少应用安装包的体积。当应用程序包较为庞大时，而部份资源只有满足某些触发条件才会加载进内存，可以选择使用 DCL 技术，当满足触发条件时，从远程下载，动态加载可执行模块。

2) 便于升级和维护，将主要功能模块存储在外部可执行代码中，在功能更新时，替换或者修改外部可执行代码模块。在用户不需要重新安装应用甚至无感知的情况下，完成对于应用功能模块的升级和维护工作。

3) 功能独立。使用了 DCL 技术的应用。只需要针对外部功能模块进行维护和更新。可以方便的添加和减少功能模块，这对于功能越来越臃肿的应用带来了巨大的收益。

然而使用 DCL 技术产生了安全问题：

风险 1：根据 Google Play 的开发者政策[16]：明确禁止从 Google Play 以外的其他来源下载可执行代码(例如 dex 文件或本机代码)的应用或 SDK。有研究证明[4]从网络不可靠来源下载可执行代码确实存在一定的风险。遗憾的是第三方应用市场如应用宝并没有这样的规定。

风险 2：动态加载的资源存储在可以被其他应用读写的目录中，如 Sdcard 中，并且应用本身也没有对加载的资源进行完整性验证，恶意攻击者可以轻易通过替换加载资源对应用本身进行恶意攻击。

我们设计 SLDroid 就是针对 DCL 存在的相关风险做对应的分析。

4. 设计

为了调查 DCL，我们设计了 SLDroid 工具(见图 2)。首先我们获取到应用市场的网页地址，分析应用市场网页特征，这一步由人工分析。应用市场网页的特征相对明显，一段时间固定不会改变，我们抓取我们所需的一个 app 的初步信息，包括 ApkName，二进制文件和其他基本信息。其中二进制文件用于静态分析和动态分析。静态分析主要作为是否使用了 DCL 技术的依据。同时分析应用是否在使用 DCL 中违反了内容策略。动态分析主要通过代码注入以及日志分析检查 DCL 实际的使用情况，同时辅助环境伪装进行测试可能的 DCL 远程下载。根据系统前面分析的结果，生成详细的 DCL 相关报告，包含应用的基本信息，触发条件，分析结果以及潜在风险或者建议。

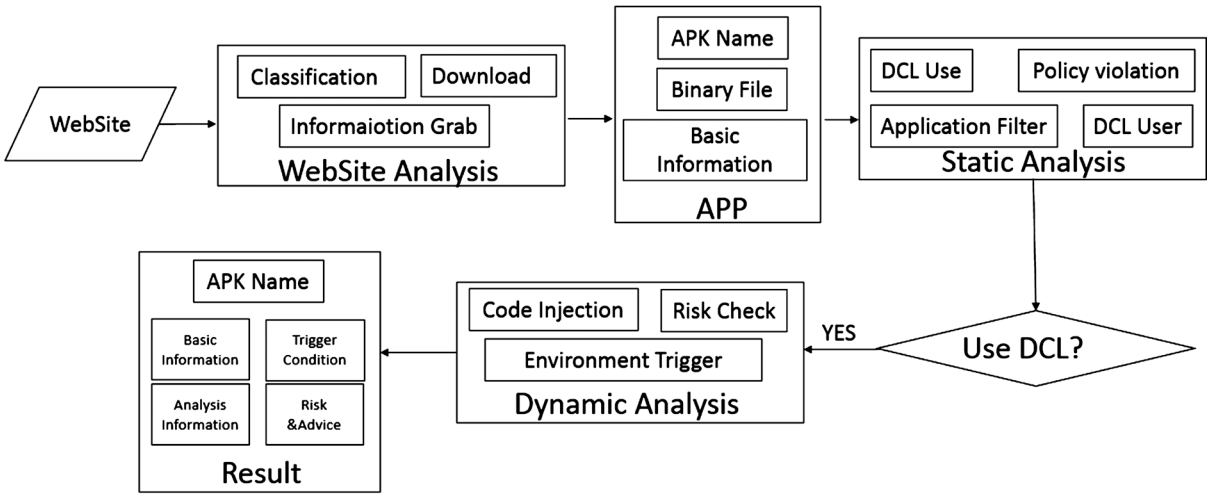

Figure 2. The detection process of SLDroid
图 2. SLDroid 检测过程

4.1. WebSite Analysis

为了获取应用信息，包括 ApkName：用于唯一标志一个应用，二进制文件：用于静态分析和动态分析的安装包。基本信息：包括下载量，分类，评价，用于结果的分析。我们需要对应用市场网页进行分析。

Figure 3. Features of URL
图 3. 网址特征

Classification：应用市场把应用分为 20 项，categoryId 范围从 101 到 120，代表一个应类(见图 3)。我们也以 categoryId 作为下载应用的分类标志，同时应用下载量和评价信息也存在于应用详情网页的固定位置，我们可以方便获取到。

Information Grab：应用市场都会采取措施针对非人工抓取信息，因为我们抓取的应用数量不大，所以我们选择通过自动打开市场页面的方式，模拟人工手动下载应用。

Download：在获取到包括下载链接等信息以后，我们下载应用的二进制文件，我们控制文件下载速度，检查文件大小和格式是否正确。

通过 Website Analysis，我们通过 categoryId 对应用进行分类，获取应用下载量和评价，以应用包名标志应用，下载二进制文件。

4.2. Static Analysis

DCL Use：应用是否使用了 DCL 是我们静态分析首要解决的问题。DCL 在 java 层和 native 层动态加载代码是不同的，在 java 层加载代码需要通过类 DexClassLoader [15]：加载未安装代码的 jar 或者 apk 文件。DexClassLoader 只有一种初始化方式，所以我们只需要检测代码中是否存在 DexClassLoader 初始化。Native 层加载代码需要在初始化阶段通过 System.load()加载，我们只需要检测是否有加载就可以确定是否在 native 层存在 DCL 的操作。

Application Filter：使用了 DCL 并不一定存在风险，我们想要进一步过滤应用，减少 Dynamic Analysis 成本。根据风险触发条件：如果可执行代码存储在其他应用的读写目录，如外部存储区。如果想要存储和加载外部存储区的可执行代码，在 4.4 以上的版本需要在配置文件中申明<android.permission.WRITE_EXTERNAL_STORAGE>以及<android.permission.READ_EXTERNAL_STORAGE>。当然在 4.4 及 4.4 以下的版本不需要申明权限。在 6.0 以上的版本需要动态申请权限。从版本兼容的角度考虑，我们认为如果使用的 DCL 存在风险，则必然申请了<android.permission.WRITE_EXTERNAL_STORAGE>以及<android.permission.READ_EXTERNAL_STORAGE>。

$$\text{Risk}_{\text{External}} \in \text{Permission}_{\text{R\&W}} \cap \text{Feature}_{\text{DCL}}$$

Policy Check：根据 Google Play 的开发者政策[16]：明确禁止从 Google Play 以外的其他来源下载可执行代码(例如 dex 文件或本机代码)的应用或 SDK。我们了解应用宝市场并没有明确的政策规定不允许从不可靠来源下载可执行代码，但是我们认为如果从不可靠来源加载代码是非常危险的。关于策略的检查，首先应用必须申请了网络权限且存在 DCL 相关特征，其次我们认为如果从不可靠来源下载代码，会存在一条路径，从网络下载到代码加载的一条路径。

DCL User：为了了解是应用还是第三方库使用了 DCL，我们记录使用了 DCL 所在的类的全路径，我们过滤了诸如 com、android，org 等等会影响结果的关键字.，将过滤后的全路径与 L LI [18]收集的 1130 个功能库好 240 个广告库进行相似度对比。在无法找到对应第三方库时，与应用包名进行相似度对比，我们认为高于 50%的匹配度就是匹配成功，如果无法匹配，我们将进行手动分析。

4.3. Dynamic Analysis

考虑到通用性，我们实现了一个应用层的容器，不需要修改 Android 框架层，实现针对应用中 DCL

功能的 Dynamic Analysis(见图 4)。借助 SLDroid 实现了针对环境的伪装，通过代码实现环境改变，测试 DCL 触发点。SLDroid 主要由俩部分组成：Environment Trigger 和 DCLRisk Check。通过环境伪装触发 DCL，捕获 DCL，进行对应的 DCL 风险检查，保存检查结果，转发对应请求。

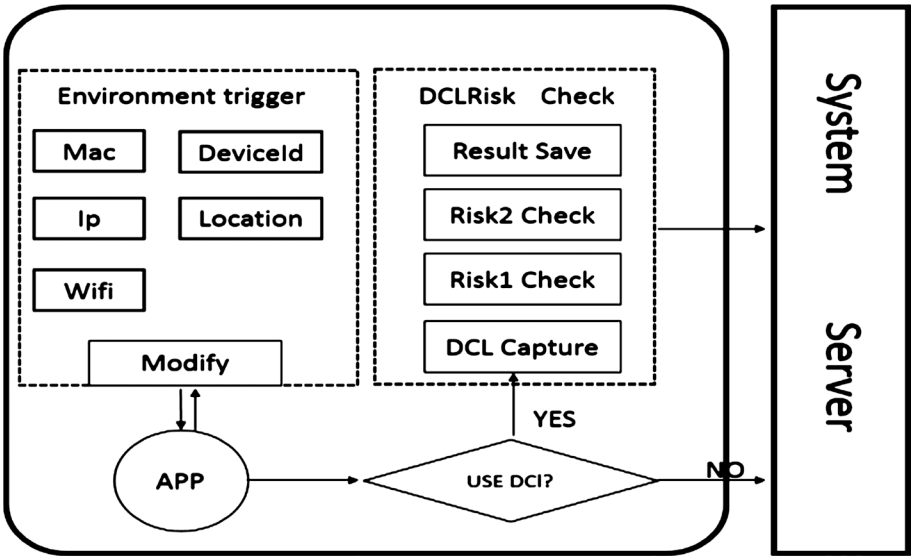

Figure 4. Frame structure of Dynamic Analysis
图 4. Dynamic Analysis 框架

Environment Trigger：为了逃避审查，DCL 的下载或者加载都是有一定的触发条件：或是检测到真机(存在 Mac 地址，DeviceId)，或是打开了 Wifi，存在真实的位置信息。为了触发 DCL，我们在容器中对应用这些信息的请求都做了处理，一旦应用请求这些信息，我们对数据进行合理的伪装，返回给应用需要的信息，通过伪装触发可能的 DCL 相关的操作。节省人工手动设置的时间和判断误差。

DCLRisk Check：我们对 DCL 相关操作进行拦截，获取 DCL 相关信息。检查可能存在的风险。风险 1：从不可靠来源下载可执行代码。在静态分析阶段我们已经分析了可能存在不可靠来源下载的代码，我们记录了代码的调用链，当应用加载外部代码时，我们将调用链与堆栈信息进行对比。如果吻合，则证明该应用确实存在风险 1。风险 2：动态加载的 DEX 文件存储在被其他应用读写的目录中，通过检查 DexPath 是否是外部存储路径或者其他应用可读写的目录。而在 native 层代码，我们检查加载的路径外部存储路径或者其他应用可读写的目录。我们对风险检查的结果进行保存，同时当检查完毕，执行正常操作，不影响应用的使用。

4.4. Analysis Result

我们对应用中使用 DCL 相关信息进行详细的记录。以包名作为唯一标识，记录应用的基本信息，DCL 使用信息，分析过程，风险以及建议等四项。作为我们的实验数据。

5. 实现

我们使用 Selenium [14]编写了一个自动抓取应用宝信息并且下载应用的脚本，结合 androguard [19]和 flowdroid [17]实现我们对应用的静态分析.我们对工具 Virtualapp [20]进行改进，对环境相关的方法进行拦截，如 getdeviceid()，getLatitude()，.getLongitude()等等方法，同时针对 DCL 相关方法进行拦截。我们将结果信息写入数据库，通过数据库进行查询。

6. 实验结果

我们使用 SLDroid 收集了 2018 年 6 月 16 日应用宝市场 20 分类的数据集。每类应用只收集下载次数前 100 的应用。以下是实验结果。

6.1. 检测结果

我们对应用宝市场 20 类应用进行分析，每类应用收集下载次数前 100 的应用。由于部分分类应用不足 100，所以总共对应用宝市场 1934 款应用进行了分析，我们将结果分为 Website Analysis，Static Analysis 以及 Dynamic Analysis 阶段分别展示，如表 1 所示。

Table 1. Analysis result
表 1. 分析结果

Type	Website Analysis	Static Analysis				Dynamic Analysis
		Decompile	Reflect	DCL	Analysis	
儿童	100	99	98	38	37	35
音乐	100	100	98	74	72	70
阅读	90	89	88	82	80	78
视频	100	100	99	91	88	86
摄影	100	99	99	74	72	70
娱乐	100	100	99	75	73	69
社交	100	100	100	82	81	79
生活	100	99	98	89	87	85
旅游	100	100	99	85	83	81
健康	100	97	94	86	84	82
新闻	100	100	99	95	92	91
教育	100	99	96	87	83	80
出行	100	99	95	66	58	56
办公	100	99	96	64	59	59
理财	100	99	95	90	89	89
工具	100	98	98	84	79	77
通讯	73	73	72	41	39	38
系统	100	99	97	53	52	51
安全	71	69	67	42	41	48
美化	100	100	98	56	56	55
总计	1934	1918	1885	1454	1405	1379

加固：在我们分析的 1934 款应用中，16 款应用我们不能进行反编译分析，我们手动检查了这些应用，发现这些应用使用了加固技术。

DCL 使用情况：1454 (75.18%)的应用在使用了 DCL 技术。这项数据远远高于 Zhengyang Qu [3]针对 Google 市场的 69.60%应用使用 DCL 技术的实验数据，因此第三方市场应用使用 DCL 技术更为频繁。同时我们也发现视频和新闻类的应用使用 DCL 非常常见，可能由于视频和新闻类应用体积较大，且功能模块比较多，需要使用 DCL 技术降低应用体积。市场审查者需要重视新闻类和视频类应用的审查。

DCL 使用者：我们总共分析了 1918 款应用(加固无法分析)，如图 5 所示：996 (51.93%)款应用自身使用了 DCL 技术，1152 (60.06%)款应用引入的第三方库使用了 DCL 技术，74 (3.86%)款应用引入的 Google 官方库中使用了 DCL 代码。Google 市场[3] DCL 使用的主体是第三方库，应用开发者很少使用 DCL 技术。而在第三方应用市场，应用开发者和第三方库都是使用 DCL 技术的主体，出现这种不一致性的原因可能是由于第三方市场对于 DCL 使用的审查不够严格有关系。

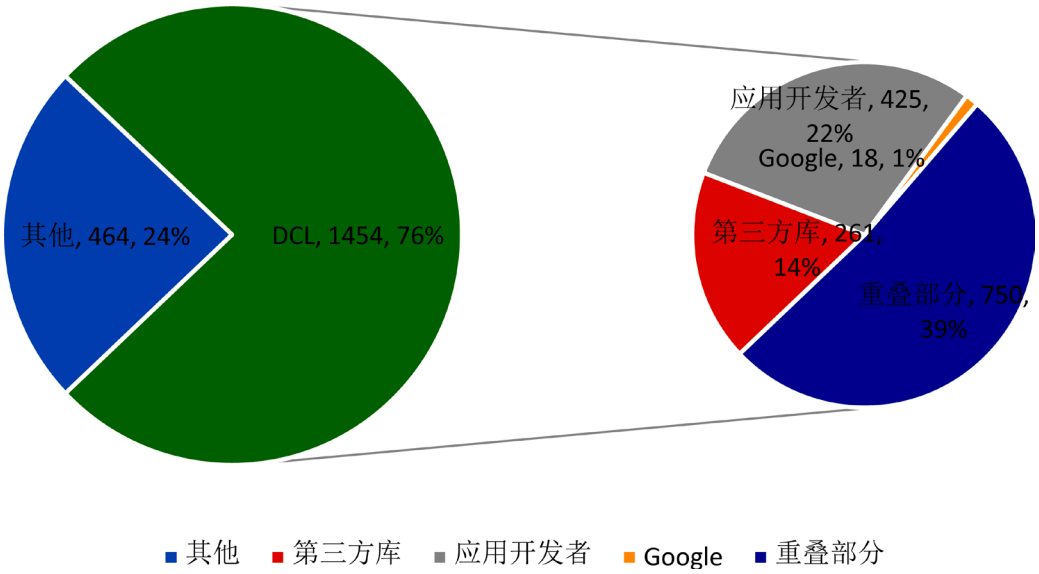

Figure 5. Result of DCL user
图 5. DCL 使用者情况

6.2. 风险结果

Risk 1：根据 Google Play 的开发者政策[16]：明确禁止从 Google Play 以外的其他来源下载可执行代码(例如 dex 文件或本机代码)的应用或 SDK。Sebastian Poeplau 等人[4]证实经过研究认为从网络不可靠来源下载可执行代码确实存在一定的风险。

Table 2. Result of risk1
表 2. 风险 1 的结果

来源	数量	占比
程序开发者	8	34.8%
第三方库	15	65.2%

如表 2 所示我们一共发现 23 款应用存在风险 1，其中 8 款应用由于应用开发者引起，15 款应用由于第三方库产生了安全风险。同时我们也希望引入百度 SDK 的应用进行检查，在第三方库产生风险的 15 款应用中，10 款应用由于引入了百度 SDK 产生了风险(见表 3)。

Table 3. Result of third-party library generating risk1
表 3. 第三方库产生风险 1 的情况

第三库包名	应用数量	占比
com.baidu.mobads	7	46.67%
com.baidu.location	3	20%
com.tencent.bugly.proguard	3	20%
com.zhangyue.iReader.	2	13.33%

Risk 2 Result：动态加载的资源存储在可以被其他应用读写的目录中，如 Sdcard 中，并且应用本身也没有对加载的资源进行完整性验证，恶意攻击者可以轻易通过替换加载资源对应用本身进行恶意攻击。

Table 4. Result of risk2
表 4. 风险 2 的结果

来源	数量	占比
程序开发者	2	34.8%
第三方库	5	65.2%

如表 4 所示我们一共发现 7 款应用存在风险 2，其中 2 款应用由于程序开发者引起，5 款应用由于第三方库产生了安全风险。

7. 结论

Android 引入了动态代码加载机制(DCL)技术。用于代码更新和功能扩展。越来越多的 Android 应用使用 DCL 技术。然而应用市场并没有严格审查 DCL 使用情况，特别是第三方市场。我们设计并实现了一个自动化工具 SLDroid：检测第三方市场 DCL 使用情况及风险。我们的实验结果表明，与 Google 市场相比，第三方市场应用使用 DCL 技术更频繁，新闻类和视频类应用使用 DCL 技术较多。同时由于没有严格限制 DCL 技术使用，相比 Google 市场，程序开发者也频繁使用 DCL 技术，更容易产生安全风险。

基金项目

国家自然科学基金项目(U1536121, 61370195)。

参考文献

[1] Richard Nieva, Google Is Doing Deep Surgery on Android. https://www.cnet.com/news/google-io-2017-android-o-project-treble-tv-go/2017

[2] http://www.appbrain.com/stats/number-of-android-apps

[3] Qu, Z.Y., Alam, S., Chen, Y., Zhou, X.Y., *et al.* (2017) DYDROID: Measuring Dynamic Code Loading and Its Security Implications in Android Applications. 47*th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (*DSN*), Denver, 26-29 June 2017, 415-426.

[4] Poeplau, S., Fratantonio, Y., Bianchi, A., *et al.* (2014) Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. *NDSS Symposium*, San Diego, 23-26 February 2014.

[5] Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., *et al.* (2015) StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. https://doi.org/10.1145/2699026.2699105

[6] Tan, D.J.J., Chua, T.-W., Thing, V.L.L., *et al.* (2015) Securing Android: A Survey, Taxonomy, and Challenges. *ACM*

Computing Surveys (*CSUR*), **47**, 58.

[7] Grace, M.C., Zhou, W., Jiang, X. and Sadeghi, A.-R. (2012) Unsafe Exposure Analysis of Mobile In-App Advertisements. *WiSec*, Tucson, 16-18 April 2012.

[8] ANDRUBIS-1,000,000 Apps Later: A View on Current Android Malware Behaviors.

[9] Book, T., Pridgen, A., DanLongitudinal, S.W., *et al.* (2013) Analysis of Android Ad Library Permissions. *Computer Science*.

[10] Igor, S., Felix, B., Xabier, U.P. and Pablo, G.B. (2013) Opcode Sequences as Representation of Executables for Data-Mining-Based Unknown Malware Detection. *Information Science*, **231**, 64-82. https://doi.org/10.1016/j.ins.2011.08.020

[11] Sahs, J. and Khan, L. (2012) A Machine Learning Approach to Android Malware Detection. *European Intelligence & Security Informatics Conference*, Odense, 22-24 August 2012, 141-147.

[12] Allix, K., Bissyandé, T.F., Jérome, Q., Klein, J., *et al*. (2016) Empirical Assessment of Machine Learning-Based Malware Detectors for Android. *Empirical Software Engineering*, **21**, 183-211.

[13] Falsina, L., Fratantonio, Y., Zanero, S., *et al*. (2015) Grab'n Run: Secure and Practical Dynamic Code Loading for Android Applications. *Computer Security Applications Conference*, Los Angeles, 7-11 December 2015, 201-210.

[14] https://www.seleniumhq.org/

[15] https://developer.android.com/reference/dalvik/system/DexClassLoader

[16] https://play.google.com/about/developer-content-policy-print/

[17] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., *et al*. (2014) FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Edinburgh, 9-11 June 2014, Vol. 49, 259-269.

[18] Li, L., Bissyande, T.F., Klein, J. and Traon, Y.L. (2016) An Investigation into the Use of Common Libraries in Android Apps. 23*rd International Conference on Software Analysis*, *Evolution*, *and Reengineering*, Suita, 14-18 March 2016, 403-414. https://doi.org/10.1109/SANER.2016.52

[19] Desnos, A. Androguard—Reverse Engineering, Malware and Goodwar Analysis of Android Applications and More (ninja!). http://code.google.com/p/androguard/

[20] https://github.com/asLody/VirtualApp