基于软件分析的 Android 应用重打包检测与防御研究

刘天一 周延森* 崔见泉
(国际关系学院信息科技学院 北京 100091)

摘 要 基于 Android 操作系统智能手机应用程序的重打包由于其低成本、高收益等特性而成为了攻击者的首选。从软件分析的角度构造出特定的重打包行为作为数据样本,对目前流行的基于相似度和代码特性等开发的重打包检测系统进行分析与性能测试,并基于测试结果结合自解密代码和软件水印等技术设计一个改进的重打包防御框架。该框架不需要额外的数据集或原应用作比对,兼顾了针对重打包行为的自动防御以及软件开发者的版权所有权声明,增加了重打包攻击的难度,从而减少潜在重打包行为的发生。

关键词 Android 应用程序 重打包检测 软件分析 混淆

中图分类号 TP301 **文献标志码** A DOI:10.3969/j.issn.1000-386x.2023.07.002

DETECTION AND DEFENSE OF ANDROID APPLICATION REPACKAGING BASED ON SOFTWARE ANALYSIS

Liu Tianyi Zhou Yansen* Cui Jianquan
(*School of Information Science and Technology*, *University of International Relations*, *Beijing 100091*, *China*)

Abstract The repackaging of smart phone applications based on Android operating system has become the first choice for attackers because of its low cost and high profit. This paper constructed specific repackaging behaviors as data samples from the perspective of software analysis, analyzed and tested the performance of the current popular repackaging detection system based on similarity and code characteristics. An improved repackaging defense framework was designed and implemented based on test results combined with the technology of self-decrypting code and software watermarking. The framework did not require additional data sets or original applications to compare, and both the automatic defense against repackaging behavior and the copyright ownership statement of software developers were taken into account, and increased the difficulty of repackaging attack, thus reducing the occurrence of potential repackaging behavior.

Keywords Android applications Repackaging detection Software analysis Obfuscation

0 引 言

Android 作为智能手机主流操作系统,其恶意软件数量在移动恶意软件中所占比重超过 50% 且快速上升。攻击者将重新打包的应用程序作为其恶意行为的传播媒介,降低了其本应自行开发的成本与难度,同时也利用了原合法软件的受欢迎度和可信度。文献[1-2]表明 Android 应用市场中 5% ~15% 的应用是被重新打包过的。

国内外针对 Android 应用重打包的研究目前多集中在相似度比对上。Kim 等[3]提出的 RomaDroid 工具使用最长公共子序列(LCS)算法来测量两个应用程序之间的相似性;Hu 等[4]提出了一种基于 UI 结构相似性的克隆程序检测方法;Wu 等[5]提出的 MSimDroid 方法基于多维相似性,包括整个应用程序相似度、资源相似度、代码相似度、联合策略进行相似度计算;汪润等[6]提出了一种基于深度学习的 Android 重打包应用检测方法,并利用 Siamese LSTM 网络学习程序的语义特征表示,实现重打包应用的检测;沈月东[7]提出的 Android 重打包行为分析技术,通过进行非第三方库代码部分的 API 调用的统计和聚类,以应用对的形式检

收稿日期:2020-08-31。国际关系学院国家安全高精尖学科建设科研专项(2019GA38)。**刘天一**,硕士生,主研领域:网络安全,模式识别。**周延森**,副教授。**崔见泉**,硕士生。

测出重打包和被重打包的应用;熊鹰[8]提出的基于用户角度的重打包检查方法,通过用户端特征提取以及服务端相似应用匹配实现重打包检测。

基于相似度的重打包检测方式面临的最大问题是需要有足够的数据集来对比分析,如未能与原合法应用程序匹配则难以判定,且基于相似度的软件水印、软件胎记多数只能作为知识产权侵权行为被发现后的追责依据,并不能从根本上解决剽窃、重打包攻击等问题。同时,随着软件分析技术的发展,仅依靠相似度的重打包检测为攻击者绕过或欺骗检测所设定的门槛与难度也在随之降低。

1 相关研究工作

1.1 Android 重打包原理

1) Android 逆向。Android 应用程序通常由 Java 语言开发编写,编译成. class 文件,转换为一个可以在 Android 平台的 Dalvik 虚拟机上运行的. dex 文件,最后将字节码. dex 文件、描述权限信息等内容的 AndroidManifest. xml 文件、资源文件打包在一个. apk 文件中。作为解释型语言,Java 高度抽象的特性也意味着其易被反编译,随之衍生出的逆向工具也在很大程度上降低了 Android 应用重打包行为的工作量。

其逆向流程如图 1 所示,其中,Apktool 是被最广泛使用的开源 APK 逆向工程软件,可以将 Dalvik 字节码反编译成. smali 代码,然后再生成 App 的重打包版本;Jadx 或商用软件 JEB 可以对. smali 代码进行直接处理与调试,并能展示出与源代码几乎相同的 Java 代码;dex2jar 可以借助 baksmali 等反汇编方式生成. jar 文件进而解压得到 Java 代码。

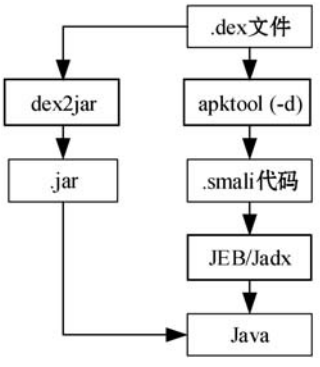

图 1 dex 文件逆向

2) Android 重打包。Android 应用程序的开发周期普遍较长,原软件开发者付出了大量成本。然而,重打包 App 的成本普遍较低,难度也较小,图 2 展示了 Android 应用程序重打包的基本流程。

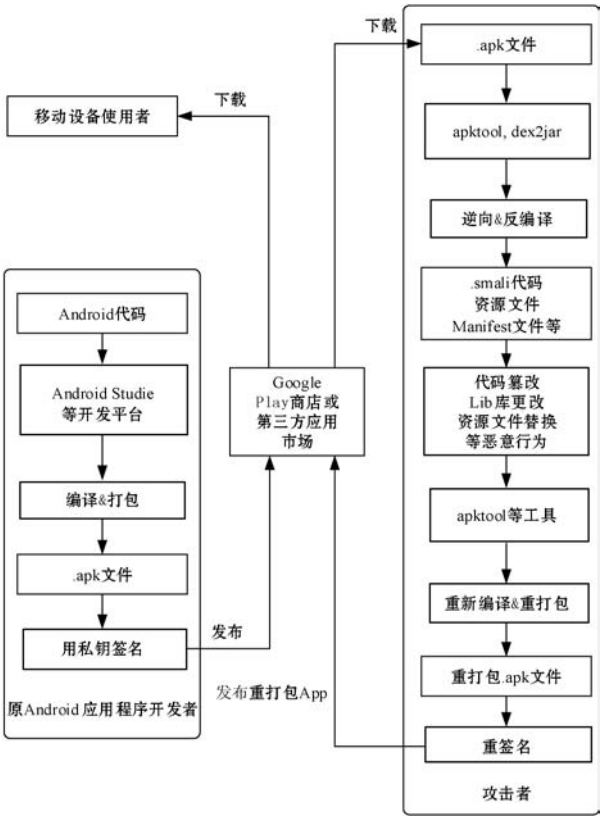

图 2 Android 应用程序重打包流程

应用程序开发者发布 App,用户通过 Google Play 或第三方应用市场下载软件到移动设备,由于“下载”行为不存在身份鉴别,攻击者同样能够以合法用户身份从应用市场下载. apk 文件到本地。通过自动化反编译或人工分析,从. apk 文件中能够获取到可读性较强的. smali 代码、Manifest 文件与资源文件等,并经由逆向工具进一步获得接近于原程序的 Java 代码。基于以上文件,攻击者通过修改. dex 代码、lib 库或向其他资源文件添加恶意代码片段或针对广告进行增添或替换,将修改后的文件再通过 Apktool 等工具重新打包成. apk 文件,使用官方的 Android 开源项目私钥将重打包软件签名合法化,上传到应用市场,诱导用户下载使用。

重打包后的应用会保留一些特性,如与原 App 有一定程度上的代码相似性、与原 App 有着不同的开发者签名。应用市场多以此为依据,结合恶意行为分析和短时间段内的模拟运行,判别一个 App 是否安全和能否被发布。目前重打包的防御措施多集中于提升攻击者的攻击难度以及重打包行为发生后的判定上。

1.2 自解密代码

自解密代码 SDC(Self Decryption Code)是在程序特定分支中依赖于分支内常量通过加密或 Hash 等方式对代码块进行处理,保障语义等价的前提下重写原分支,替换后仅可以通过动态运行实现自动解密恢复,

文献[9]论证了其可靠性的基础理论依据。

在借鉴文献[10 - 11]的基础上,本文所设计的 SDC 结构如图 3 所示。首先对条件分支中表达式的常量(记为 w)使用不可逆的处理(如单向 Hash)得到一个新常量 e_s,以此实现对原常量 w 的隐藏。之后将常量 w 与实时计算生成的校验码 c 一同作为密钥对分支中全部代码和划分后的部分水印信息进行加密操作,并用加密得到的乱码形式代码替换源代码。

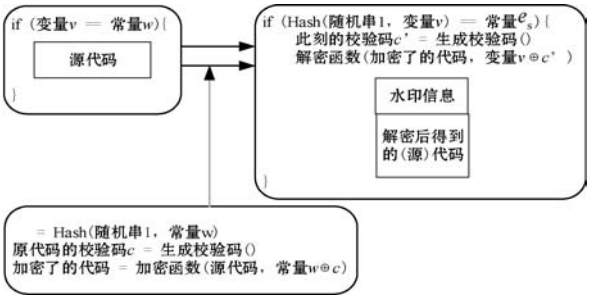

图 3　SDC 结构

当运行到该分支时,程序会实时计算校验码并执行解密操作从而完成自动解密,还原源代码和这部分的水印信息。其中变量 v 是程序运行到这一分支时变量中所存储的值,理论上应与 w 相等,因而用同样随机串对变量 v 的值进行同样处理,会得到一个与 e_s 相等的值,以上做到了对含有常量 w 的原表达式的等价转换。

由于原常量 w 已经通过 e_s 的替换实现了不可逆的隐藏,且由原常量 w 生成 e_s 的过程存在着一个无法预测的随机值,再加上 Hash 算法的单向性,从而提高了从 e_s 到 w 的还原难度,作为密钥组成部分的 w 就成为了只有原开发者知道的信息。因此,预测常量 w 的方法一种是根据程序上下文语义结合数据流分析等方式合理推断,但由于通过 SDC 算法转换程序最后的呈现大部分是无实际意义的乱码,难以完成单纯的静态分析;另一种是在动态运行过程中从变量 v 进行突破。

1.3　软件水印

软件水印是将特定数据 w 作为水印嵌入程序 P 中,得到一个其水印难以被轻易检测和移除的新程序 P_w。水印的目的不是阻止应用程序被非法使用,而是证明其所使用的软件和算法的所有权,从而辅助知识产权权益保护、打击盗版。水印分为静态和动态、明显可见和隐藏不可见,文献[7]利用 SDC 构造的水印即属于无须加以隐藏的动态软件水印。

2　基于软件分析的 Android 重打包检测研究

本文针对 Android 应用重打包检测在软件分析领域所面临的挑战,分析其检测方式在目前可预计的规避策略,从而设计开发出更为可靠的重打包防御方案。

2.1　实验设计

由于多数 Android 重打包检测研究并未公开代码和数据集,本文首先使用合法应用程序集借助混淆、自解密代码等技术构造特定数据样本,并简要实现了基于代码相似度的检测算法,复现了 WuFan、AndroidSOO 等开源重打包检测工具的检测过程。

实验具体流程如图 4 所示。针对不同类型的重打包防御方式,论文相应地设计了不同的绕过检测或妨碍防御的策略,以此来对重打包检测算法进行有针对性的测试及分析。

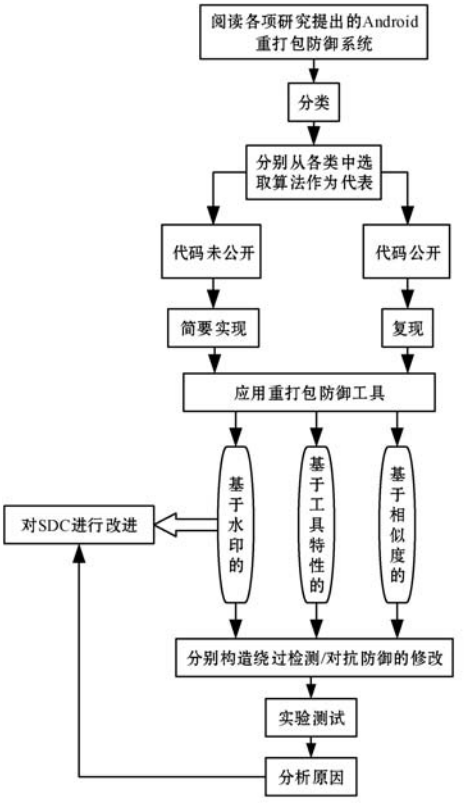

图 4　实验设计

2.2　自解密代码的实现及数据集构造

本实验数据集分为合法 App、简单处理的重打包 App、利用混淆等处理意图绕过检测的重打包 App 三部分。其中,本文默认从 Google 官方应用市场下载得到的应用均为合法应用(实际上可能会存在约 1.2% 的重打包应用),对其进行简单的逆向、修改和重打包后即构成了第二部分数据集,而构造第三部分数据样本时的处理如下:

1) 混淆相关处理。本文首先对重打包程序进行基础混淆操作,包括:① 名字改编,即将域名、方法名、类名、包名等有实际意义的标志符替换成无意义的相

对较短的字符串。② 在不影响程序语义的前提下修改修饰符。③ 加入无效代码对方法的实现结构进行调整。④ 方法参数转换。⑤ 常量计算替换。

2）自解密代码构造。本文通过自解密代码技术为意图绕过重打包检测的数据集构造提供进一步的辅助。SDC[10-11]作为一种有效的重打包防御方式，但本文认为其主要思想同样也可以成为攻击者对抗重打包检测的技术之一。

本文在 Java 代码层面，借助第二节中 SDC 的自加密-自解密框架和信息不对称理论，实现了一个简易版本的自解密代码转换器，其中，利用的单向函数为 MD5，对称加解密算法为 DES，随机串的添加在本文所编写的 md5enc 方法中实现。

以程序 1 为例，其在 Java 代码层面经过混淆、自解密代码转换等处理后得到的等价代码如程序 2 所示，可以看出二者直观结构呈现不同，但二者功能上并不存在差异，编译运行后得到同样的运行结果。

程序 1 原始代码

```
while(k < w)
{
  if(x[k] = =1) R = (s * y)%n
  else      R = s;
  s = R * R% n; L = R; k ++;
}
return L;
```

程序 2 经过混淆、自解密代码转换等处理后的代码

```
int next = 0;
for ( ; ; )
{
switch(md5enc(next))
{
case "4548cce2e2d7fbdea1afc51c7c6ad26":
k = Integer. parseInt ( desdec ( String. format ( "% 08d", next),"nISR6pPU35Y = ") );
s = Integer. parseInt ( desdec ( String. format ( "% 08d", next),"B/4RM7/980M = ") );
next = Integer. parseInt ( desdec ( String. format ( "% 08d", next),"B/4RM7/980M = ") ); break;
case "aab3238922bcc25a6f606eb525ffdc56":
s = R * R% n; L = R; k ++;
next = Integer. parseInt ( desdec ( String. format ( "% 08d", next),"4km57CcZvD8 = ") ); break;
……
case "9bf31c7ff062936a96d3c8bd1f8f2ff3": return L;
```

另一方面，对于较为复杂的程序，本实验利用开源的混淆工具，从编译器层面而非直观的代码层面进行混淆等处理。

2.3 实验及结果分析

检测算法的主要思想是在开发者签名不同的前提下对于一个从非官方渠道获取的未知来源的 App，通过在已知是合法、非重打包的众多数据集中逐一两两进行相似性比对，计算其相似度。相似度越高（即越接近 1.0）则说明这个未知来源的 App 是重打包的可能性更大，相似度最高值所对应的合法 App 被认定为其重打包的对象。

本文利用前文所构造的数据样本对基于相似度的重打包检测算法进行对抗测试，具体步骤为：

1）从可靠性较高的应用市场随机选取. apk 文件（记作 app-ori）下载到本地。

2）借助 Apktool 工具对. apk 文件进行解包与反编译，如图 5 所示。

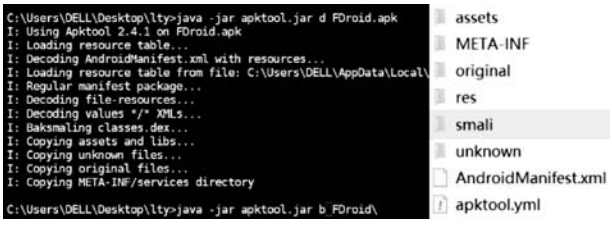

图 5 对. apk 文件进行反编译与重打包

3）使用逆向工具得到 Java 代码。

4）添加或修改代码片段得到文件 re1。

5）对 re1 采用前文所述混淆等方法进行转换，用于妨碍重打包检测，得到 re2。

6）使用 Apktool 分别对 re1、re2 重打包生成. apk 文件 app-re1、app-re2。

7）将生成的两个. apk 文件分别放入重打包检测系统中检测，分别记录各系统关于是否为重打包应用所做出的判定。

8）通过 Android Studio 平台模拟器验证应用程序本身功能完整性是否被破坏。

9）对检测算法的表现进行评估与分析。

上述相似度重打包检测步骤如图 6 所示。

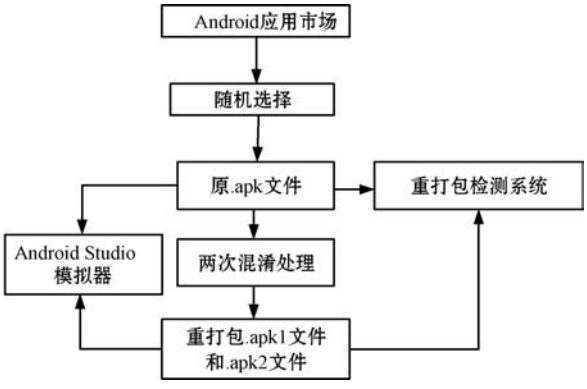

图 6 相似度重打包检测流程

实验主机为:CPU Intel Core i5 @2.30 GHz 和 8 GB 的 RAM。由于一个完整的 Android 应用规模较大,包含上万个文件,尽管本文复现算法时已尽可能简化了检测过程,基于相似度的检测算法仍需数小时的判定时间。基于此我们仅在小规模测试集中实验,针对实验的分析主要基于代码层面。

图 7 展示了重打包文件与原. apk 文件的相似度得分(即 Similarity),若作者签名不同而相似度很高则可以判定其是重打包应用。

图 7　计算两个. apk 文件的相似度

本实验默认 app-ori 为合法应用,将其作为基准分别计算 app-re1 和 app-re2 与该应用的相似度,从而对二者是否为重打包程序进行判断。通过对比针对同一 $app\text{-}ori_{[i]}$ 的两个不同重打包版本 $app\text{-}re1_{[i]}$ 与 $app\text{-}re2_{[i]}$ 在检测中表现出来的差异,以及进一步统计检测系统在不同情况下的漏报率,可以推测出本文在构造特定重打包行为时所采取的混淆、自解密代码转换等技术对重打包检测算法所造成的影响。

实验结果如表 1 所示,本文构造数据样本时混淆等处理降低了重打包应用与原应用的相似度,使其低于重打包攻击行为的判定阈值,从而在一定程度上欺骗了检测系统。本文所复现的检测算法没有针对代码以外的信息进行处理,将两个包括作者签名在内完全相同的. apk文件(其相似度高达 1.0)判定为重打包,而 Wu Fan 可以对. apk 文件的签名做出识别,将其正确判定为作者相同,即"非重打包",如图 7 所示。但现实中 Android 应用市场在正式发布应用前都至少会对其签名进行最基本的验证与校对,验证时即可获知作者信息,因此对持有相同作者信息的 App 重打包判定可暂不作考虑。

表 1　检测性能对比

比较对象	基于代码相似度	基于控制流相似度	Wu Fan
原 apk 与原 apk	100% 判定为重打包	100% 判定为重打包	100% 判定为作者相同
重打包 apk 与原 apk	100% 判定为重打包	100% 判定为重打包	相似度数值均极高
混淆等处理后的重打包 apk 与原 apk	13% 的程序被判定为重打包	6% 的程序被判定为重打包	相似度数值较之前均有所降低

针对于仅完成了重打包的. apk 文件,基于相似度的检测算法均能以很高的准确率检测出其重打包行为(本实验在重打包过程中仅采取较为简单的人工修改或添加,且判定阈值设置较低,真实情况不一定会达到如表 1 所示百分之百正确率和零漏报率)。针对本实验处理后的重打包测试样本,其与原文件的相似度数值呈明显降低。实验结果显示混淆、自解密代码转换等方法使得多数重打包. apk 与原. apk 文件的相似度降至基于代码相似度算法的判定阈值之下,从而存在很大概率可成功绕过此类型的重打包检测。

3　Android 重打包攻击防御的改进

3.1　系统框架

上文分析了软件分析技术为目前被广泛应用的三种重打包检测方法所带来的影响,基于此,本文综合了以 SDC 等技术为依据的检测或防御算法,在此基础上提出一种改进的重打包防御框架,如图 8 所示。

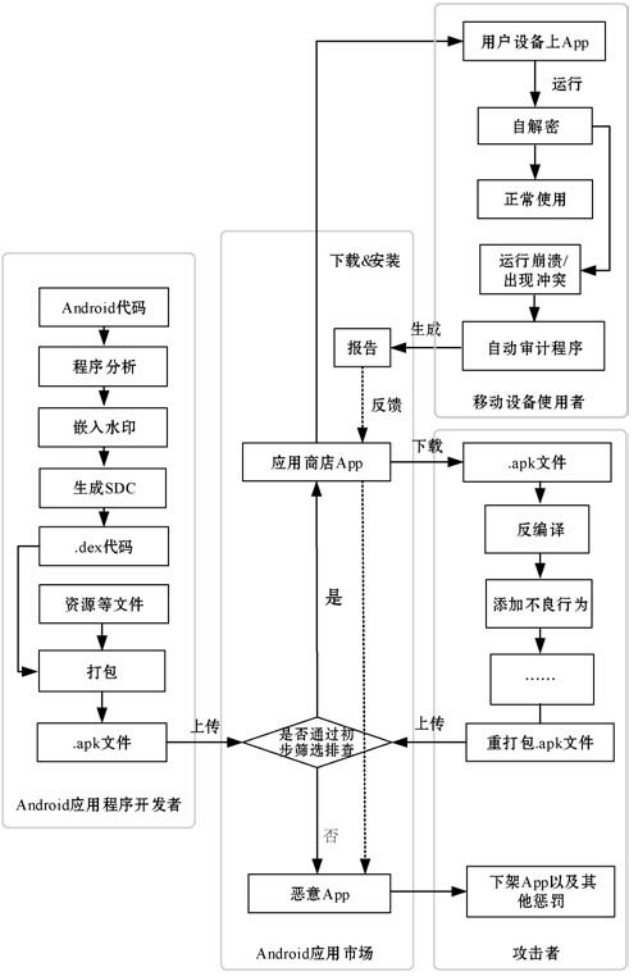

图 8　重打包攻击防御系统框架

Android 应用开发者在编写完程序代码后:

1)将源代码进行自动化程序分析,其环境与软件

测试保持一致即可,在这个过程中:① 对程序结构进行一定的转换与优化,使之便于后续的水印嵌入和 SDC 生成,同时也起到混淆的作用,从而降低程序的可读性;② 对可利用的分支语句进行标记,如程序规模较小则利用不透明谓词添加可利用的分支。

2) 构造水印信息并进行切分和压缩等处理。

3) 结合自解密与自防御算法,将水印嵌入程序,同时对代码进行加密处理,生成自解密代码。

4) 将代码与资源等文件打包、签名和上传到应用市场待审核。

随后,权威 Android 应用市场会对 App 进行初步检测,在模拟环境下进行运行测试。在测试过程中运行到 SDC 时,正常情况下程序会自动进行解密还原,若无法进行解密则显然会因为乱码从而导致程序运行崩溃,此时权威应用市场即可根据程序运行错误的提示判定该 App 有缺陷或有被重打包的可能,拒绝发布该应用。而通过了初步检测以及模拟运行测试的 App 会被发布到 Android 应用商店供用户下载。

用户从应用商店等多种渠道下载 App 到智能移动设备上,若该应用为合法应用,程序会逐步自动解密还原,用户可以在没有额外附加感受的前提下正常使用;若该应用是被重打包过的,程序存在很大的概率会在运行到某处时由于自解密失败而崩溃。此时该程序便无法继续运行,在一定程度上阻止了该重打包应用继续危害用户设备的可能,程序的崩溃也为移动设备使用者发出了该应用可能存在问题的警示,用户可以采取卸载程序以及举报该程序等行为。Android 应用市场收到用户的反馈后会对该可疑应用进行进一步筛查和更细致的审计。若该 App 仅为运行错误则提示更新修复,若该 App 确实存在恶意行为则下架该应用并对发布者进行一系列惩罚。如果该应用为恶意的且已经对移动设备使用者造成了损失,则需要进一步借助水印中所携带的信息来判定该应用是否为重打包以及责任归属问题。

3.2 改进分析

由于自防御代码[2]意图以程序崩溃作为重打包的代价从而阻止重打包攻击,而自解密代码则是借助 SDC 嵌入软件水印[7],我们希望可以通过优化水印的构造方式来降低嵌入的成本,从而将二者结合,既可以实现让重打包 App 自动暴露以达到防御效果,又可以在重打包攻击发生后通过水印所携带的作者信息、发布信息等进行版权保护或恶意行为的责任判定。同时在此基础上结合其他重打包检测与防御系统的优势设计出一个更为可靠的防御框架。

经过处理后的 SDC 既可以在水印与载体代码之间建立内在依赖,将二者加密为一个 SDC 段,从而更好地将软件水印融合到程序之中,并提升攻击者移除或篡改水印的成本;又可以利用程序完整性校验码等构造密钥以实现重打包应用的自动化防御;在此基础上引入的加密算法,作为一种重要的混淆技术,也在一定程度上提升了程序的复杂性,使得攻击者阅读代码和重构代码更加困难。

随着软件分析领域的发展,程序综合等技术使得条件分支语句也就是初始 SDC 的各个“入口”是可以以自动化分析的形式找到并基于概率求解的,这使得攻击者以低于重开发的成本重打包一个被 SDC 保护的 Android 应用成为了可能。因此,我们的框架在 SDC 生成前加入一个步骤,先对源程序进行程序分析以及类似于混淆技术的代码转换处理,以此来增加攻击者自动化求解的难度。

本框架的优势之一是让 Android 应用程序开发者、移动设备使用者、Android 应用市场三方全都参与到重打包的防御中来。开发者可以以自身的操作对代码加以保护从而从根本上阻碍他人的剽窃行为,而不是将版权保护寄托于检测能力参差不齐的第三方应用市场、在他人重打包攻击行为发生后才进行耗时耗力的追责;用户也可以通过程序的运行崩溃感知重打包行为,从而阻止恶意应用继续存在于移动设备之上。

4 结 语

软件分析技术的发展既使攻击者绕过检测、对抗防御、降低攻击成本成为了可能,同时也为重打包防御提供了提升的空间。本文分析了目前常见的重打包检测算法,构造了特定数据样本及绕过检测的策略进行测试,并基于此设计一个综合的重打包防御框架。本文的不足之处在于,没有对提出的方案进行完整的实现与测试评估,仅通过第二节中实验所得出的结论来辅助验证改进设计的合理性,且仅在代码层面而非编译器层面实现了 SDC。

参 考 文 献

[1] 田振洲,刘烃,郑庆华,等. 软件抄袭检测研究综述[J]. 信息安全学报,2016(3):52-76.

[2] Li L, Bissyandé T, Klein J. Rebooting research on detecting repackaged android apps: Literature review and benchmark [J]. IEEE Transactions on Software Engineering, 2018, 47(4):676-693.

(下转第 17 页)

关联条目,提示或通知用户。

(4) 管理变更。工具能够在不同角色之间定义流程和表单,实现完整的需求变更申请、审批和通知流程。某些工具还具有基线管理功能,能够对比基线差异。

(5) 状态统计。工具能够跟踪和统计需求的状态(例如:已提出、已接受、已批准、已实现、已验证、已删除、已否决等),生成统计图表,便于管理人员从需求实现的角度掌握项目进展。

在商业竞争和开源社区的推动下,需求管理工具推陈出新。一些需求管理工具是项目管理套件的一部分,另一些工具支持扩展,能够与其他管理软件协同实现需求到任务计划、需求到测试、需求到缺陷的跟踪管理功能。一些工具向云端应用和移动端应用延伸。一些工具是 DevOps 工具链的组成部分。很多工具采用 Web 架构从而支持异地多团队协作开发。企业和组织可以根据自身需要,选择合适的需求管理工具。

4 结　语

需求管理人员要有足够广阔的视角,面向软件研制全生命周期的工作产品和主要活动,实施基于用户需求的一致性管理。在用户需求规模不断增长的同时,对需求管理的要求也越来越精细和精准,恰当选择和有效使用工具,能够对需求管理工作有所助益。

参 考 文 献

[1] Wiegers K, Beatty J. 软件需求(第 3 版)[M]. 李忠利,李淳,霍金健,等译. 北京:清华大学出版社,2016.

[2] 刘姝,程胜. 航天软件需求工程[M]. 北京:中国宇航出版社,2016.

[3] 任甲林. 术以载道——软件过程改进实践指南[M]. 北京:人民邮电出版社,2014.

[4] 中国人民解放军总装备部. 军用软件研制能力成熟度模型:GJB5000A—2008[S]. 北京:中国人民解放军总装备部,2008.

[5] Beatty J. Winning the hidden battle: Requirements tool selection and adoption[C]//21st IEEE International Requirements Engineering Conference (RE),2013:364 - 365.

[6] CMMI Product Team. CMMI for Development, version 1.3 [R]. CMU/SEI, 2010.

[7] Gotel O, Mader P. How to select a requirements management tool: Initial steps[C]//17th IEEE International Requirements Engineering Conference,2009:365 - 367.

[8] Keshta I, Niazi M, Alshayeb M. Towards implementation of requirements management specific practices (SP1.3 and SP1.4) for Saudi Arabian small and medium sized software development organizations[J]. IEEE Access,2017,5:24162 - 24183.

[9] Kirova V, Kirby N, Kothari D, et al. Effective requirements traceability: Models, tools, and practices[J]. Bell Labs Technical Journal,2008,12:143 - 157.

[10] Madsen J, Munck A. A systematic and practical method for selecting systems engineering tools[C]//Annual IEEE International Systems Conference (SysCon),2017:1 - 8.

[11] McGee S, Greer D. A software requirements change source taxonomy[C]//4th International Conference on Software Engineering Advances,2009:51 - 58.

[12] Nurmuliani N, Zowghi D, Powell S. Analysis of requirements volatility during software development life cycle[C]//Australian Software Engineering Conference,2004:28 - 37.

[13] Ramesh B, Jarke M. Toward reference models for requirements traceability[J]. IEEE Transactions on Software Engineering,2001,27(1):58 - 93.

[14] Shah A, Alasow M A, Sajjad F, et al. An evaluation of software requirements tools[C]//8th International Conference on Intelligent Computing and Information Systems (ICICIS),2017:278 - 283.

(上接第 12 页)

[3] Kim B, Lim K, Cho S, et al. RomaDroid: A robust and efficient technique for detecting android app clones using a tree structure and components of each app's manifest file[J]. IEEE Access,2019,7:72182 - 72196.

[4] Hu Y, Xu G, Zhang B, et al. Robust app clone detection based on similarity of UI structure[J]. IEEE Access,2020,8:77142 - 77155.

[5] Wu P, Liu D, Wang J, et al. Detection of fake IoT app based on multidimensional similarity[J]. IEEE Internet of Things Journal,2020,7(8):7021 - 7031.

[6] 汪润,唐奔宵,王丽娜. DeepRD:基于 Siamese LSTM 网络的 Android 重打包应用检测方法[J]. 通信学报,2018,39(8):69 - 82.

[7] 沈月东. Android 重打包应用行为分析系统设计与实现[D]. 北京:北京邮电大学,2019.

[8] 熊鹰. 基于用户角度的安卓重打包应用检测系统的研究与应用[D]. 北京:北京邮电大学,2019.

[9] 周旷. 基于代码水印的 Android 应用重打包自检测技术研究[D]. 成都:电子科技大学,2018.

[10] Chen K, Zhang Y, Liu P. Leveraging information asymmetry to transform android apps into self-defending code against repackaging attacks[J]. IEEE Transactions on Mobile Computing,2018,17(8):1879 - 1893.

[11] Ren C, Chen K, Liu P. Droidmarking: Resilient software watermarking for impeding android application repackaging [C]//29th ACM/IEEE International Conference on Automated Software Engineering,2014:635 - 646.