基于虚拟机字节码注入的 Android 应用程序隐私保护机制

宋宇波[1,2,3]，陈琪[1,2,3]，宋睿[1,2,3]，胡爱群[3,4]

（1. 东南大学网络空间安全学院，江苏 南京 211189；2. 东南大学江苏省计算机网络技术重点实验室，江苏 南京 211189；
3. 网络通信与安全紫金山实验室，江苏 南京 211189；4. 东南大学信息科学与工程学院，江苏 南京 211189）

摘 要：为了解决 Android 应用权限机制的滥用，提出了一种基于虚拟机字节码注入技术的 Android 应用程序权限访问控制方法。所提方法能够根据用户的安全需求和使用场景，生成虚拟机字节码形式的安全策略，并将其注入 Android 应用的涉及危险权限请求和敏感数据访问的代码单元中，从而实现动态应用行为控制。对国内 4 家主流应用商店爬取的应用程序进行测试，结果表明，所提方法可以对合法 App 的敏感 API 调用和危险权限请求进行有效拦截，并根据预定的安全策略实施控制，注入虚拟机字节码后的大部分 App 运行不受注入代码影响，稳健性得到保证，且具有较好的普适性。

关键词：安卓安全；隐私保护；安全策略；虚拟机字节码

中图分类号：TN92

文献标识码：A

DOI: 10.11959/j.issn.1000−436x.2021115

Android application privacy protection mechanism based on virtual machine bytecode injection

SONG Yubo[1,2,3], CHEN Qi[1,2,3], SONG Rui[1,2,3], HU Aiqun[3,4]

1. School of Cyber Science and Engineering, Southeast University, Nanjing 211189, China
2. Key Laboratory of Computer Network Technology of Jiangsu Province, Southeast University, Nanjing 211189, China
3. Purple Mountain Laboratories, Nanjing 211189, China
4. School of Information Science and Engineering, Southeast University, Nanjing 211189, China

Abstract: To solve the abuse of the Android application permission mechanism, a method of Android application access control based on virtual machine bytecode injection technology was proposed. The security policy in the form of virtual machine bytecode was generated according to the user's security requirement and usage scenario, and injected into the coding unit of Android application that involves dangerous permission request and sensitive data access, to realize dynamic application behavior control. Tests on applications crawled from four mainstream domestic App stores show that the method can effectively intercept sensitive API calls and dangerous permission requests of legitimate App programs and implement control according to pre-specified security policies. Also, after injecting virtual machine bytecode, most of the App program operation is not affected by the injected code, and the robustness is guaranteed. The proposed method has a good universality.

Keywords: Android security, privacy protection, security strategy, virtual machine bytecode

1 引言

目前，手机已成为人们生活不可分割的一部分，人们通过手机进行视频通信、电子商务和网络教学等活动。由于手机和个人信息的高度捆绑，恶意攻击者开始攻击智能设备，导致用户的敏感信息

收稿日期：2020−12−16；**修回日期**：2021−03−10

通信作者：宋宇波, songyubo@seu.edu.cn

基金项目：国家重点研发计划基金资助项目（No.2020YFE0200600）；江苏省网络与信息安全重点实验室基金资助项目（No.BM2003201）

Foundation Items: The National Key Research and Development Program of China (No.2020YFE0200600), Jiangsu Province Key Laboratory of Network and Information Security (No.BM2003201)

和个人隐私数据泄露。在众多智能终端中，安装了 Android 系统的终端占有最大比例，这意味着 Android 系统所受到的恶意攻击和安全威胁也最多。Android 系统的安全机制存在几个关键的薄弱环节。

1) Android 系统的碎片化问题给该系统的安全防护带来了巨大的障碍。Google 官方发布的系统补丁只优先提供给 Android 系统的最新发行版本，通常并不承诺甚至拒绝前向兼容旧版本的系统[1]。因此，大量安装了旧版本 Android 系统的设备只能被动地等待各级系统提供商发布更新补丁。

2) Android 系统采用基于权限的安全模型来保护用户的敏感资源和个人隐私，然而这种通用的模型并不能满足某些特定的安全需求。一些个人和组织出于使用环境或场景的考虑，有强烈动机需要对特定应用程序执行定制的安全防护策略[2-3]，仅依靠 Android 系统的安全架构完全无法满足这一需求。

3) 与另一个主要的移动设备操作系统 iOS 不同，Android 系统并不强制要求统一的应用程序发布和审查机制。Android 系统的应用程序不需要经过充分有效的审核流程就可以被安装到设备上，这给 Android 系统用户带来了巨大的安全风险。

针对上述安全漏洞和恶意攻击方法，已经有很多研究者提出了各种分析和检测 Android 系统恶意应用的方案，并且开发了许多技术和工具[4-6]。研究者通常采用静态代码分析或动态运行时检测等手段对恶意应用软件进行检测与甄别[7]。静态检测通常通过分析 Android 应用程序的 AndroidManifest.xml 文件来考察应用程序的权限调用情况，或者逆向得到应用程序的源代码，并对代码的执行逻辑进行分析。然而，考虑到 Android 应用程序的软件规模与日俱增，同时代码混淆和加固技术被广泛应用于软件发布和打包过程，静态源代码分析的甄别能力极其有限。相对地，动态分析方法能够从程序执行逻辑层面考察应用程序的行为，通常构建独立检测环境并将应用程序置于该环境中，通过污点分析、黑盒测试等方式对应用程序进行分析和测试[8]。然而，现在的恶意攻击者能够检测应用程序运行时是否处于被监控环境或仿真器环境下，进而使其恶意逻辑失活从而逃避动态运行时检测[9]。

静态分析和动态分析等手段是对恶意应用程序进行甄别，并对甄别出的应用程序进行相应的封禁处理。这些已有的研究无法解决新的 Android 版本中增加的动态权限申请特性带来的识别困难问题，也无法解决利用反射特性等技术进行动态代码构造的恶意应用程序的识别问题。

基于此，本文设计并实现了一个 Android 应用程序的动态策略权限控制系统，可根据用户需求实施权限控制和生成安全策略。该系统基于 Android 的虚拟机字节码注入技术，在目标应用程序涉及敏感操作和危险权限调用的方法单元中添加安全策略执行代码，并通过 Java 反射特性动态地调用安全功能，在安全需求发生变更时动态切换权限控制级别，而不需要重新注入字节码或对应用程序进行重打包。通过这种权限策略控制，用户和组织能够根据需求动态地允许或禁止应用程序的权限授予和使用，从而更好地保护个人和组织隐私安全。

2 相关工作

Android 移动操作系统的安全性构建在以权限系统为基础的敏感资源访问模型上，这些敏感资源包括但不限于位置信息，以及相机传感器、麦克风、通信录和 SMS（short messaging service）的信息等。通常，若一个 Android 应用程序企图获取对某个敏感资源的访问权限，就必须在 Android Manifest.xml 清单中显式地声明所需的权限。而用户可以选择授予或者不授予该权限。但是，这样的权限系统正在被一些应用程序滥用，这些应用程序声明的权限远多于实现必要功能所必需的权限[10]。Felt 等[11]考察了应用市场上的相当一部分应用程序，并发现大量应用程序都存在上述问题。更糟糕的是，每当操作系统或者应用程序更新时，这些恶意的应用程序都可以通过更新累积的方式进行特权升级，从而在静默状态下转换成恶意软件[12]。

针对上述问题，近年来相关领域研究者主要采用静态分析和动态分析 2 种思路来进行恶意软件检测与防护。静态分析方法主要利用静态代码分析的手段来检查应用程序是否包含异常的权限申请和信息流[13]，具有较好的可扩展性，可以从较大数量级的应用程序中快速筛选出可疑的应用程序。作为一个面向普通消费者的商业级操作系统，Android 系统需要面对各种应用场景，因此必须使用事件驱动的设计思路，具体来说就是广泛利用生命周期钩子和 GUI（graphical user interface）回调处理来处理用户的实时请求。运行时的权限申请控制和数据流并非总可以通过静态代码分析检测出来，因为应用

程序的执行流程取决于具体的运行时环境和用户行为[14]。另外，静态分析方法检测通过动态代码构造的恶意行为，如通过 Java 反射机制调用敏感应用程序接口（API, application programming interface）的能力极其有限。恶意攻击者通常能够轻易攻击这一缺陷构造，采用代码混淆[15]或代码异变[16]的方法来误导静态检查机制。Suarez 等[17]提出了以资源为中心的新的静态分析方法以克服上述的限制，但是恶意程序仍然可以通过采用资源混淆的方法来规避以资源调用为中心的检测系统[18]。

相比之下，动态分析提供了检测和防护恶意应用程序的补充和完善方法。常用的动态检测方法包括基于行为的恶意软件检测[19]、基于系统 API 调用追踪的程序行为建模[20-21]以及基于资源利用的已用程序行为分析[22]。在动态分析领域，机器学习技术越来越广泛地应用于区分恶意应用和良性应用。然而，当恶意应用对系统调用进行代码混淆时，基于系统调用分析的动态检测系统仍然有可能被规避[23]。

综上所述，无论是静态检测还是动态检测方法，都存在固有的弊端，并不适用于全部的 Android 应用程序。这些方法主要着眼于分类并甄别出恶意应用，而很少讨论应该如何对甄别出的恶意应用的攻击行为进行防护。

此外，也有一些研究在字节码级别上对需要提供安全性的方法单元添加安全功能。Lee 等[24]提出了一种使用应用程序包装技术的方案，在代码的方法上添加必要的安全功能。该方案在字节码级别添加了安全功能，而不依赖 Android 的 Java 或 Kotlin 源代码。具体地，该方案将所需的安全功能封装成字节码级别的 smali 代码，并在需要安全性的方法点上将这些代码注入应用程序中。这种方法的缺点很明显，它是在静态策略的基础上运行的，这意味着每当更改安全策略时，都需要重新提取并注入安全代码，并重新包装应用程序。

针对这些问题，本文提出了一种基于虚拟机字节码的动态安全策略控制方法，该方法能够检测 Android 应用程序中涉及危险权限请求和敏感数据访问的代码单元，并将用户指定的安全策略以虚拟机字节码的形式注入相应代码位置，依照用户的安全需求对应用程序的危险行为进行防护和控制，并根据用户所指定的安全策略的变更而动态调整控制级别。

3　基于字节码注入的应用隐私保护机制

本文提出了一种控制 Android 应用程序对用户敏感信息访问和危险权限调用的隐私保护系统（下文简称为本系统），其基于安全策略生成的虚拟机字节码注入技术，通过在目标应用程序中注入控制机制以限制危险权限的访问并强制规范应用程序的行为。本系统基于动态安全策略生成机制，这意味着系统所应用的安全策略可以根据用户需求动态调整，该机制通过将安全约束规范和具体代码行为解耦合来实现动态性和灵活性。动态策略生成机制保证即使用户或组织频繁地修改安全策略，本系统也不需要重新编译应用程序或对应用程序进行重打包。

图 1 是本系统的结构概览。本系统能够根据用户和组织的需求，通过数学和形式逻辑的相关理论抽象出特定的行为模型。该行为模型将数据流、事件时间、用户身份与角色因素列入考虑，并得到抽象的安全策略。同时，本系统提供了一个安全策略生成器（SSG, security strategy generator）。SSG 能够通过执行一组细粒度的安全规则将用户输入的抽象策略转换成具体的权限控制策略。实例化的安全策略将指导系统生成一个包含安全方法的字节码

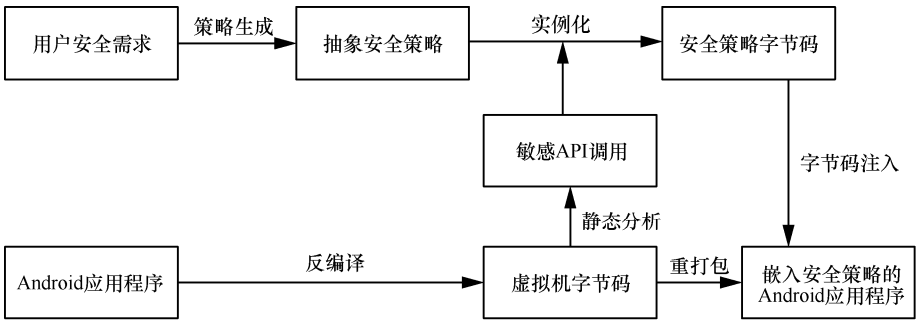

图 1　本系统的结构概览

安全运行库，其中包含了执行安全策略的代码。安全运行库被嵌入应用程序字节码级的危险权限请求和敏感 API 调用位置，并与原始应用字节码一起被重新打包生成一个包含安全决策中心（SDC, security decision center）的应用程序。应用程序中嵌入了一系列的策略执行器（PE, policy executor），这些 PE 能够在应用触发权限请求行为时向安全决策中心发出事件通知，并向安全决策中心索取安全策略指定的授权操作以执行具体的策略。

3.1 安全策略的生成

要实现能够满足细粒度控制需求的安全策略生成机制，首先需要对用户的行为模型进行抽象。行为模型抽象的目的是梳理用户实体和应用程序实体间的交互方式，并将实体间的详细交互流程进行抽象以便于工程实现。本文所提的行为模型能够提供数据机密性、用户的感知与控制、访问权限控制和信任管理等多层次的安全功能。为了实现上述需求，行为模型需要对数据流、事件时间、用户身份与角色、环境与场景、信任关系等因素进行逻辑学抽象。

安全策略生成机制使用事件−条件−动作（ECA, event-condition-action）规则来提取抽象策略和详细执行策略。ECA 模板的抽象语法如算法 1 所示。SSG 持续监视从模板实例化的 ECA，PE 表示特定组合的所有条件，$C_i \in \text{PE}$ 表示特定模式的事件使特定组合的所有条件得到满足，此时执行相应的授权操作。

算法 1 事件条件操作规则

输入 事件模式集合 E，用户指定的条件 C，用户指定的时延 D，信任类型 T，应用程序控制单元个数 n，规则模板 R

$T \leftarrow T_{\text{Indirect}} \cup T_{\text{Direct}}$

$D \leftarrow D_{\text{amount}} \cup D_{\text{Unit}}$

for $i \in \{1,2,\cdots,n\}$ do

  if $T_i == \text{true}$ then

    if $C_i \in \text{PE}$ then

      $R_i ::= C_i D_i$

    end if

  else

    $R_i ::= \text{deny}$

  end if

end for

return R

上述活动模式是指与临时或规律的行为或交互相匹配的模式，指定这种模式能够支持检测性和预防性的安全策略。上述条件可能对应各种复杂的执行流程和分支场景，如事件模式、外部行为、定时条件、执行条件、上下文、用户身份或角色、用户和行为可行度等。上述权限操作的执行具体是指对临时操作的授权或拒绝，也可以更细粒度地根据安全需求有选择地修改行为的属性或延迟行为的执行。规则模板甚至支持将行为进行模块化，这样就能将不同的行为按照一定规则进行组合、循环或分支判断，以支持额外的活动或动态地更新模型的信任等级，如提升或降低模型的信任度。例如，行为模块化支持实现以下操作，在特定应用程序触发了某些指定的隐私敏感行为时通过 SMS 或电子邮件通知用户。

3.2 虚拟字节码注入

本文使用中间层的 Dalvik 字节码，而不使用反编译源代码，这是因为近年来 Google 官方鼓励开发者使用 Kotlin 语言进行 Android 开发，但是不管是 Kotlin 还是 Java 编写的应用程序，在编译阶段都将统一编译 Android 系统专有的基于寄存器的 Dalvik 字节码。另外，成熟的 Android 应用程序开发者出于应用安全角度和知识产权保护角度，通常会对应用程序源代码进行代码混淆、加固或代码加壳等操作，这极大地增加了将二进制代码或字节码还原到 Java 或 Kotlin 源代码的难度。即使将二进制文件或字节码还原为原始代码，其因代码混淆带来的极低可读性也会使这样的努力得不偿失。

代码 1 展示了对 Android 应用程序进行反编译的一个实例，该实例将一个应用程序的二进制文件反编译为 Dalvik 字节码的 smali 表示。

代码 1

```
invoke−virtual {v0},
TelephonyManager;−>getDeviceId()Ljava/lang/String;
move−result−object v1
iput−object {v1, v0},
Lcom/crumblejon/MainActivity;−>imei:Ljava/lang/String;
```

在代码 1 中，应用程序试图调用 TelephonyManager 类中的 getDeviceId(String)方法。该方法能够得到用户设备的唯一标识符，该标识符通常被应用程序用来对用户进行追踪和识别，以对用户行为

进行建模或分析用户的行为偏好。

1) 安全策略的实例化

本系统会根据用户或组织制定的安全策略向应用程序植入特定的安全控制代码。此外，为了实现动态性，在代码植入时，除了用户或组织指定的安全策略，本系统还会检测应用程序中尚未被用户指定的敏感 API 调用或危险权限请求。这样，在嵌入了安全策略的 Android 应用程序运行时，当到达敏感 API 调用方法时，安全决策中心将首先检查用户或组织是否已经为该 API 调用定义了安全策略。如果用户针对该 API 调用定义了安全策略，安全决策中心会通知策略执行器执行相应的安全策略代码；如果用户未定义相关策略，安全决策中心会略过该 API 调用并继续检测其他的 API 调用。部分敏感的 API 调用如表 1 所示。

表 1　　部分敏感的 API 调用

权限	相关 API 调用
MANAGE_ACCOUNTS	invalidateAuthToken()
VIBRATE	NotificationManager.notify()
READ_CONTACTS	ContentResolver.openInputStream()
WAKE_LOCK	MediaPlayer.start()
ACCESS_NETWORK_STATE	MediaPlayer.stop()
INTERNET	getActiveNetworkInfo()

需要说明的是，未定义策略时嵌入的安全代码并不会对应用程序的性能造成过大的影响，应用程序将以几乎可以忽略的时延继续运行。基于这样的设计，本系统提出的字节码嵌入技术能够在用户或组织想要更改安全策略时动态地响应用户的需求，而不需要重新嵌入安全策略字节码或对应用程序进行重打包，因为本文在初次进行代码静态分析和字节码植入时已经考虑了所有可能的安全控制需求。

注入应用程序的安全代码，更具体地说就是一系列的策略执行器，将以代理模式被添加到应用程序的相应位置中。通过将策略执行器以安全代理的形式封装起来，本系统可以极大地减少对原始应用程序代码的侵入性，并为修改后的应用程序提供更高的稳健性和可用性。在静态分析阶段，本系统将通过分析应用程序的 AndroidManifest.xml 文件和应用程序的反编译字节码确定在该应用程序中所涉及的所有敏感操作和权限申请，并针对性地抽取相应的策略执行器组成用于注入程序的运行库。随后，系统将在原始应用程序的敏感操作和权限申请涉及的所有方法前后注入对执行器的调用，这些调用将根据用户或组织的声明来决定是否调用相关的策略执行器，并在相关权限操作之前或之后进行相应的安全操作。以设备定位信息的控制为例，Dalvik 字节码展示了一个应用程序获取定位数据的方法单元，如代码 2 所示。可以看到，开发者预先在类中声明了一个 android.location.LocationManage 类实例 Xl，用于向操作系统的定位管理器申请定位数据；随后，通过 LocationManager 类的 getLastKnownLocation()方法获取设备的最后位置。值得注意的是，该应用程序在打包时使用了代码混淆技术，因此方法名 A()和字段名 Xl 都是无意义的字符串。

代码 2

```
.method private A(Ljava/lang/String;)Landroid/location/ Location;
    iget−object v0, p0, Landroid/support/v7/app/i; −>Xl:Landroid/location/LocationManager;
    invoke−virtual {v0, p1},
    Landroid/location/LocationManager;−>getLastKnownLocation(Ljava/lang/String;)Landroid/location/Location;
    move−result−object v0
```

上述私有方法封装了定位数据的获取方式，当应用程序在业务逻辑中需要请求设备定位时将会申请访问该类的公有方法。Dalvik 字节码( )展示了该类的一个关键公有方法，如代码 3 所示。该方法通过调用上述私有方法向应用程序的其他业务逻辑部分暴露了一个定位数据申请接口。代码 3 定义了一个名为 gn()的公有方法。该方法首先检查了 AndroidManifest.xml 清单中是否声明了 ACCESS_FINE_LOCATION 权限，并确认用户已经向应用授予了该权限；随后，调用上面定义的 A(Ljava/lang/String;)方法以获取设备的定位数据。

代码 3

```
.method public gn()Landroid/location/Location;
    invoke−direct {p0, v0}, Landroid/support/v7/app/i;−>A(Ljava/lang/String;)Landroid/location/Location;
    move−result−object v0
    const−string/jumbo v3,
     "android.permission.ACCESS_FINE_LOCATION"
```

invoke−static {v2, v3}, Landroid/support/v4/ content/f;−>checkSelfPermission (Landroid/content/Context;Ljava/lang/String;)I

move−result v2

if−nez v2, :cond_0

const−string/jumbo v1, "gps"

invoke−direct {p0, v1}, Landroid/support/v7/ app/i;−>

A(Ljava/lang/String;)Landroid/location/Location;

move−result−object v1

显然，公有方法 gn()Landroid/location/Location 访问定位数据的关键接口。为了对该应用程序的定位数据访问行为进行控制，需要在 gn()方法中注入安全策略控制代码。为了保证用户能够根据具体需求对应用行为进行不同程度的限制，本文注入的安全策略代码不会将控制逻辑固化到 gn()方法单元中，而是通过 Java 反射策略调用安全决策中心来实现动态控制。需要注意的是，对 gn()方法的修改将影响虚拟机寄存器的分配。

图 2 展示了在 Android 应用程序的敏感操作前后注入安全策略执行代码的流程。当应用程序执行到该敏感操作时，策略执行器会向安全决策中心请求安全策略。若安全策略中包含了对该敏感操作的限制规则，安全决策中心指示策略执行器进行相应的权限控制和安全操作。例如，若安全策略对该敏感操作涉及的危险权限做出了禁止声明，策略执行器就会直接忽略对该方法的调用；类似地，若安全策略指示对该敏感操作进行延迟或条件修改，策略执行器也会执行相应的安全操作。上述操作都是指在敏感操作之前进行的安全代理，对于在敏感操作之后进行的安全代理，只能对操作进行修改或延迟，因为敏感操作已经执行，无法对其进行禁止。

将安全策略注入 Android 应用程序后，安全决策中心将在应用程序的主活动中的 onCreate()方法中初始化。在有些情况下，会因为设备、版本或其他原因而导致安全决策中心初始化失败而导致安全策略处于不可用的状态，这时被安全强化的应用程序将询问用户是否重试初始化安全决策中心，若用户拒绝重试并试图在此情况下继续运行应用程序，应用程序执行到敏感操作或危险权限调用时会对用户提出安全警告。此外，在应用程序执行期间，软件资源、硬件问题或功耗等原因都可能导致安全策略中心变为不可用状态，为了保证安全性和稳健性，应用程序将在执行到安全敏感操作或权限申请时提示用户，让用户自行决定是否继续执行相关操作，并为应用程序接下来可能遭遇的安全决策征求用户的默认许可或拒绝。用户所进行的决策可以作为后续相关操作出现时的默认操作，系统将会缓存用户所给出的决策，并在下次出现类似决策时提供参考，以实现对于同一安全决策的一致性。

2) 安全策略的动态调整

注入了安全策略的 Android 应用程序已经可以按照用户指定的安全需求完成相应的安全操作。但是随着场景需求、信任关系、用户身份与角色等因素发生变化，用户对安全策略的要求也在随时发生变化，需要对安全策略进行动态调整。如前文所述，以往的研究在这一阶段束手无策，只能针对新的安全需求重新生成安全策略实例并将其重新注入应用程序中，然后将新的应用程序安装至用户的设备中。这种操作方式对安全策略的使用是灾难性的，考虑到个人或组织时常变动的安全需求，这种反复地重新编译与打包所带来的时间成本和操作成本是无法容忍的。这也是之前的研究和设计难以投入使用的根本原因。本系统在初次进行字节码注入时就会通过静态分析手段寻找应用程序中的所

图 2 在敏感操作前后注入安全策略执行代码流程

有敏感数据访问请求和危险权限操作，并在所有相关的 API 调用位置前后都嵌入策略执行器。这就意味着只要安全决策中心给出特定的控制指令，所有敏感 API 调用都能够被本系统限制或直接屏蔽。这种动态控制的实现直接依赖于 Dalvik 虚拟机环境和 Java 语言的反射特性。反射机制利用了虚拟机执行语言的自省能力，能够在程序运行时以数据流的方式提供对代码逻辑的控制。更具体地，基于 Dalvik 虚拟机的反射技术允许在程序运行时动态加载和使用类，并通过字符串形式的控制信息调整对方法、字段、类和接口的调用方式。反射技术可以将程序的以上逻辑单元变量化和参数化，使方法、类和接口可以作为参数传输。因此，只需预先定义所有可能的控制代码，应用程序在运行时就可以根据安全策略选取匹配的方法或接口。

为了帮助用户调整安全策略并向应用程序推送调整后的安全策略，本系统在安全决策中心中添加了网络通信模块，该网络通信模块能够以 HTTP 通信的形式向服务器请求安全策略，服务器也能够以 WebSocket 的方式向所有注入了安全策略的应用程序推送新的安全策略，使所有程序都能够保持和最新安全策略的同步，如图 3 所示。

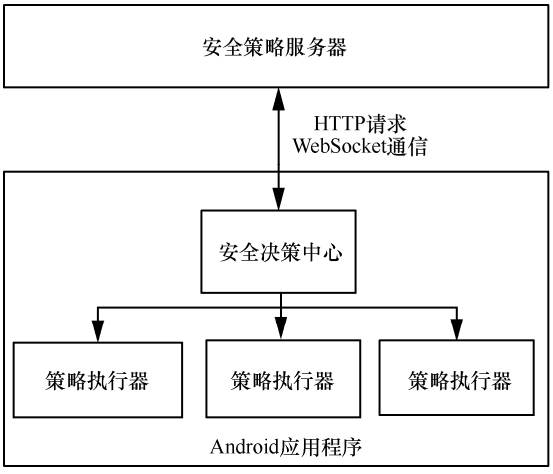

图 3　安全决策中心与服务器通信

4　实验评估与分析

为了对本文所提 Android 应用程序隐私保护系统进行测试和评估，本节从国内各大应用市场中随机采集了一些应用程序，并以此为载体对系统进行了详细的分析。评估主要考察的是注入安全策略后应用程序的可用性和稳健性，以及注入的安全策略对应用程序所产生的性能方面的影响。

4.1　数据源分布特性

为了保证数据集的随机性和普适性，本节选取了市场份额较高的 4 家应用市场：腾讯应用宝、360 手机助手、华为应用市场、小米应用商店。从这 4 家应用市场中采用随机方式分别爬取了 60 个应用程序，将这些应用程序中的重复项去除后得到了 129 个应用程序。需要强调的是，爬虫在采集应用程序时并未对应用程序做任何限定或要求，而是采用完全随机的方式进行爬取。这也意味着实验中并没有对应用程序的类型、下载量、大小等性质做出假设，这是为了保证数据集的随机性和普适性。

来自不同应用市场的应用程序数量如图 4 所示。需要说明的是，本文在去除重复应用时以应用商店的市场份额作为优先级，即以来自腾讯应用宝中应用程序集合为基础，并去除从其他 3 家应用市场中爬取的应用程序集合中的重复项。然后，对 360 手机助手中获取的应用程序进行类似操作，删除剩余 2 家应用市场中相对应的重复项，依次类推。因此，图 4 中应用市场所提供的应用程序数量是依次减少的。数据源中应用程序的类型分布如图 5 所示。从图 5 可以看到，本文采集的应用程序中，游戏与娱乐类程序占比最高，为 30.23%；网络购物与金融、影视与音乐、生活方式类程序占比也较高，分别为 16.28%、12.41%和 10.85%。上述统计结果与日常生活中的直观体验相符，说明本文采集的应用程序集合基本符合日常生活中应用程序的分布情况。

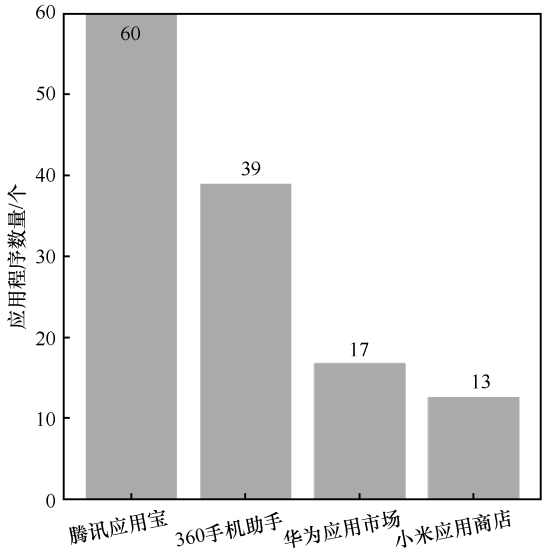

图 4　各应用市场对数据源的贡献

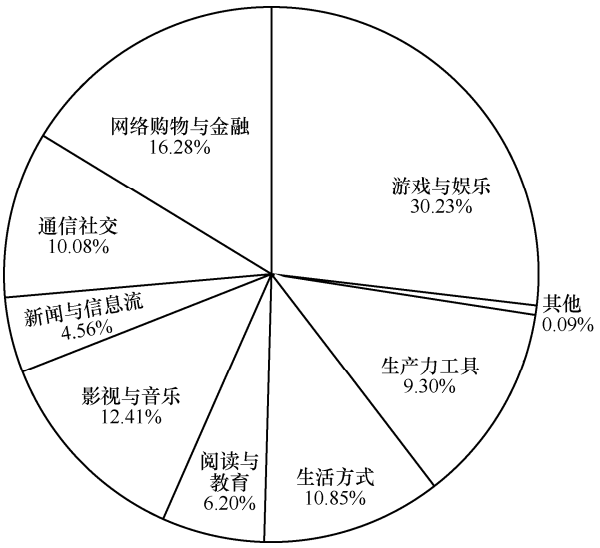

图 5　数据源中应用程序的类型分布

4.2　安全策略的生成与注入

根据用户和组织的不同需求，针对特定的应用场景和安全需求生成相应的安全策略，系统需要首先通过对用户指定的 Android 应用程序进行静态分析，获取该应用程序涉及的所有敏感 API 调用和危险权限请求；然后将获取到的信息转化成策略点，通过对策略点的操作，决定是否同意应用程序的危险请求以及更细粒度的 API 调用。除了特定 Android 应用程序所涉及的具体调用或权限申请，本系统还要考虑用户自身的环境与需求。例如，在一些保密工作场景和会议过程中，不允许应用程序拥有随时访问麦克风的权限；在家庭场景中面对可能的来电需求，则需要允许特定的应用程序拥有访问麦克风的基本权限。

以应用程序访问设备定位信息为例，这一行为对应着多种具有 API 调用以及应用程序要求在位置改变时执行回调，或者要求操作系统返回设备的最后位置，或者以一定时间间隔轮询定位传感器等。则对应的安全策略可以描述为

$$\begin{aligned}&\text{Policy::=(GSS=\{Permit,Forbid,DelaySetting\},}\\&\text{RLU=\{Permit,Forbid,DelaySetting\},}\\&\text{GLKL=\{Permit,Forbid,DelaySetting\})}\end{aligned}\tag{1}$$

其中，策略由是否获取系统服务权限（GSS, get system service）、是否允许获取位置更新权限（RLU, request location updates）、是否获取最后的位置信息（GLKL, get last known location）组成；每个权限的回答存在以下 3 种可能，Permit 表示一直允许，Forbid 表示一直禁止，DelaySetting 表示一定时间间隔刷新权限。不同权限的不同值构成不同的安全策略。将抽象的安全策略进行实例化，生成一个包含安全方法的字节码安全运行库。针对具体的应用程序，对其 APK 安装包进行解包，获取 DEX 格式文件、Manifest 文件以及各种静态资源，如图像和颜色配置等。将实例化后的安全策略嵌入 DEX 文件中，使用 ApkTool 对应用程序进行冲编译和打包。因为只有签名的应用程序安装包才能通过 Android 操作系统的检测并成功安装到用户设备中，所以需要使用 signAPK 对得到新的 APK 文件进行签名。

4.3　安全策略的稳健性

稳健性是指在对应用程序进行修改和重新打包后，应用程序能否正常运行。对于安全策略的稳健性，本文从 2 个维度进行评价：一个维度是本文系统执行特定安全策略对应用程序正常运行的影响；另一个维度是对于不同的操作系统版本和不同的硬件平台，注入了安全策略的应用程序能否正常工作。

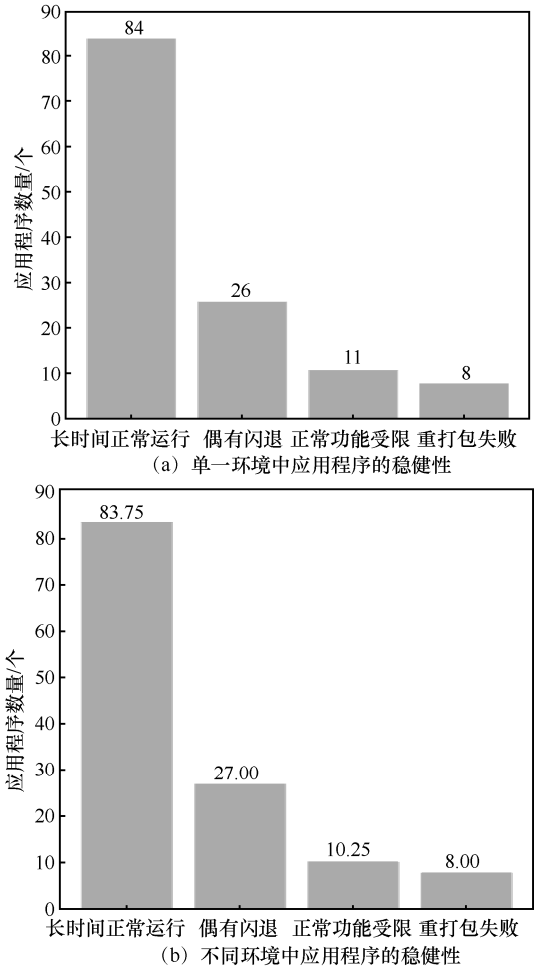

图 6　稳健性测试结果

图 6(a)展示了当运行环境限制在同一操作系统和同一硬件环境时，本系统对 Android 应用程序稳健性的影响。本实验使用搭载 Android 10 系统的华为 P30 Pro 手机对数据集合中的所有应用程序进行了测试。由图 6(a)的数据可以看出，在限制了软硬件环境的情况下，85.27%的应用程序基本能够正常运行，其中大部分应用程序能够保持长时间的正常运行，20.16%的应用程序在执行特定安全策略时偶有闪退情况发生；8.53% 的应用程序发生了无法打开或者正常功能受到影响的情况；6.2%的应用程序在嵌入安全策略字节码后的重编译过程中发生异常而导致无法重打包。

图 6(b)展示了不同软硬件条件下安全策略对应用程序稳健性的影响。除了图 6(a)实验中所使用的搭载 Android 10 系统的华为 P30 Pro 手机，本实验还使用了搭载 Android 10 系统的小米 MI 9 手机、搭载 Android 9 的华为 Mate 20 手机，以及搭载 Android 9 系统的小米 Redmi K20 Pro 手机。实验结果表明，本系统在不同的软硬件环境下并未表现出太大的差别。这在一定程度上体现了本系统对应用程序稳健性的影响不大。

4.4　字节码注入前后应用程序性能对比

对于手机应用程序来说，另一重要的评估是字节码的注入对应用程序性能的影响。在实验仿真中，性能考虑的是注入字节码对应用程序启动时间、大小（即所占内存）以及功耗的影响。

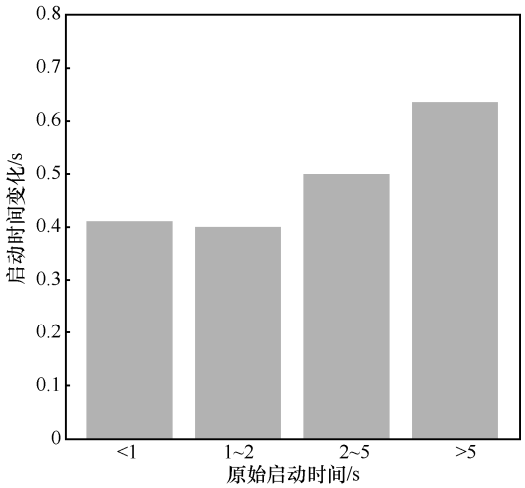

图 7　安全策略对应用启动时间的影响

图 7 反映了本系统注入的安全策略对 Android 应用程序启动时间的影响。实验结果表明，尽管不同应用程序的启动时间分布广泛，但注入安全策略对启动时间的影响与应用程序原始的启动时间并无明显关联。对于原始启动时间小于 1 s 的应用程序，安全策略对启动时间的平均影响约为 0.411 s，实验数据的标准差为 0.080 4 s；对于原始启动时间大于 5 s 的大型应用，安全策略的平均影响仅为 0.635 s，标准差仅为 0.069 9 s。尽管如此，需要特别注意的是，对于那些原始启动时间较短的应用程序来说，注入安全策略所造成的用户直观影响更大。因为在这种情况下安全策略带来的开销所占的比例更高，而这种对比会使用户明显感觉到程序变慢。事实上，注入的安全策略并非在程序启动同时被立即调用，而是在程序执行到特定位置或进行敏感操作时才会被调用。因此，理论上程序启动时会对启动时间产生额外开销的只是安全决策中心的初始化过程，而这个过程本身对程序资源的消耗相对程序自身的初始化过程所产生的开销要小得多。实验的结果符合理论分析。

图 8 展示了注入了安全策略前后应用程序大小的变化。从实验结果可以很直观地看到，注入安全策略对应用程序的大小带来的影响相比于应用程序原始大小几乎可以忽略不计。对于原始体积小于 10 MB 的应用程序，注入安全策略所带来的平均体积影响仅为 246.13 KB；对于原始体积大于 200 MB 的大型应用，注入安全策略所带来的平均体积影响也仅为 1 296.95 KB。但是需要说明的是，安全策略对应用程序大小的影响和应用程序原始大小呈现出明显的正相关。

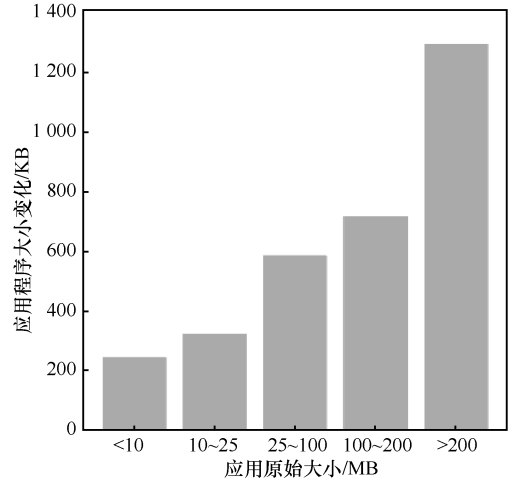

图 8　安全策略对应用大小的影响

从图 8 可以看到，尽管不同应用程序本身的大小分布广泛，但是安全策略注入所带来的空间开销

占重打包后应用程序总大小的比例始终保持在一定范围内。换言之，随着应用程序规模的增大，注入的安全策略字节码规模也增多了。从理论上分析，注入的字节码可分为 2 个部分，即安全决策中心和各个策略执行代理。安全决策中心的大小一般是相对固定的，策略执行代理与具体的应用程序有关。更具体地，当应用程序中涉及更多的敏感操作和权限调用时，所需的策略执行器就相对更多。因此，与其说安全策略带来的空间开销和应用程序原始体积有关，不如说是和应用程序中包含的敏感操作和权限调用数量有关。

对于安全策略对应用程序功耗的影响，本文以数据源中 2 个常用应用程序为基础进行了详细测试。用一台搭载 Android 10 系统的华为 P30 Pro 手机进行相关操作。在每次测试开始之前，将手机充电至 80%，随后要求受试者使用手机打开注入了安全策略的特定应用程序，并依照程序的正常功能连续使用 60 min。在使用的第 5 min、第 10 min、第 20 min、第 30 min、第 45 min 和第 60 min 分别记录此时手机的剩余电量，并依照记录数据绘制功耗曲线。为了和原始情况进行对比，对于每个应用程序，受试者按照上述流程使用未经安全策略注入的原始程序，在使用应用程序过程中分别同时运行市场上现有的安全类 App（Bitdefender 和 Avria）进行用电量损耗对比，2 个应用程序都能够提供权限的监控。每个应用程序在不同的条件下重复实验 4 次。图 9 揭示了这 2 组实验的实验结果。从实验数据可以看出，本系统所注入的安全策略对应用程序所产生的功耗影响较小，在长达 60 min 的使用过程中最多给功耗造成 2%的差距。现有的市场上的安全类应用程序的功耗有 8%的差距，这从说明了本系统对应用程序的性能并不会产生用户可感知级别的影响。需要说明的是，由于实验过程中没有对更多的应用程序进行功耗测试，因此本部分所进行的测试只能部分反映本系统对功耗的影响，更全面的分析仍然需要依赖对更多应用程序进行的测试。

5 结束语

Android 系统提供了基于权限控制的安全机制，但是该机制的粗粒度控制方式不能满足相当一部分用户或组织的安全需求，而且存在可以被恶意攻击者利用的系统性漏洞。为了防止应用程序滥用危险权限和用户敏感信息，本文设计并实现了基于字节码注入的 Android 应用程序隐私防护系统。该系统的安全策略注入机制利用了基于虚拟字节码的安全策略生成与注入技术，该技术能够在 Android 应用程序中植入对危险权限和敏感 API 调用的控制策略，并根据用户或组织的需求进行动态的调整。在大多数情况，安全策略的注入不会对应用程序的可用性造成影响，并在此基础上可对 85.33% 的敏感 API 调用和危险权限请求进行有效限制。

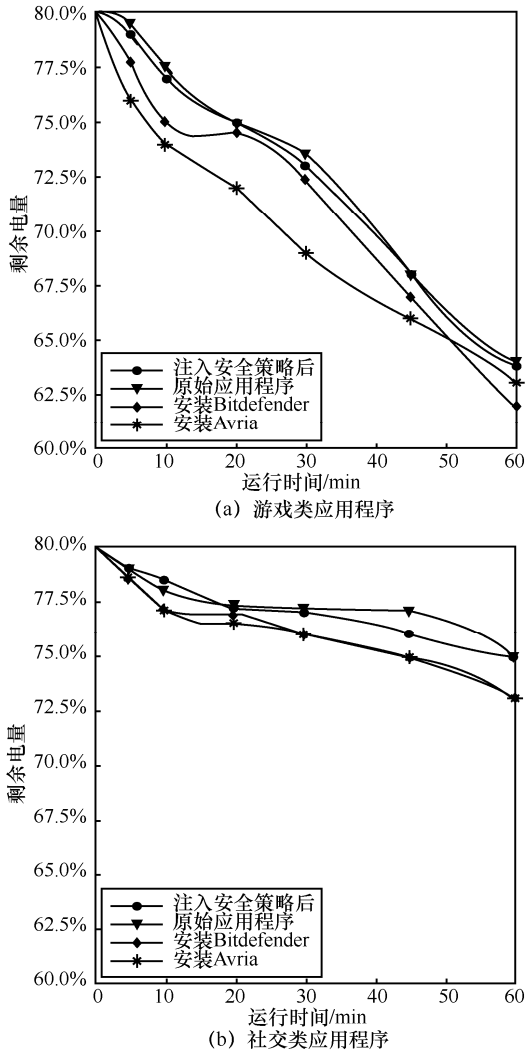

图 9 不同方法对应用功耗的影响

参考文献：

[1] TALAL M, ZAIDAN A, ZAIDAN B B, et al. Comprehensive review and analysis of anti-malware Apps for smartphones[J]. Telecommunication Systems, 2019, 72(2): 285-337.

[2] NAUMAN M, KHAN S, ZHANG X W. APEX: extending Android permission model and enforcement with user-defined runtime con-

straints[C]//The 5th ACM Symposium on Information, Computer and Communications Security. New York: ACM Press, 2010: 328-332.

[3] MAHINDRU A, SANGAL A L. DeepDroid: feature selection approach to detect Android malware using deep learning[C]//2019 IEEE 10th International Conference on Software Engineering and Service Science. Piscataway: IEEE Press, 2019: 16-19.

[4] BLÄSING T, BATYUK L, SCHMIDT A D, et al. An Android application sandbox system for suspicious software detection[C]//2010 5th International Conference on Malicious and Unwanted Software. Piscataway: IEEE Press, 2010: 55-62.

[5] GIBLER C, CRUSSELL J, ERICKSON J, et al. AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale[C]//International Conference on Trust and Trustworthy Computing. Berlin: Springer, 2012: 291-307.

[6] SCHUTTE J, TITZE D, DE FUENTES J M. AppCaulk: data leak prevention by injecting targeted taint tracking into android Apps[C]//2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications. Piscataway: IEEE Press, 2014: 370-379.

[7] CHEN K, WANG P, LEE Y, et al. Finding unknown malice in 10 seconds: mass vetting for new threats at the google-play scale[C]//24th USENIX Security Symposium. Berkeley: USENIX Association, 2015: 659-674.

[8] VIDAS T, CHRISTIN N. Evading android runtime analysis via sandbox detection[C]//The 9th ACM Symposium on Information, Computer and Communications Security. New York: ACM Press, 2014: 447-458.

[9] FERREIRA J, RESENDE R, MARTINHO S. Beacons and BIM models for indoor guidance and location[J]. Sensors, 2018, 18(12): 4374-4384.

[10] FARUKI P, BHARMAL A, LAXMI V, et al. Android security: a survey of issues, malware penetration, and defenses[J]. IEEE Communications Surveys & Tutorials, 2015, 17(2): 998-1022.

[11] FELT A P, CHIN E, HANNA S, et al. Android permissions demystified[C]//The 18th ACM conference on Computer and Communications Security. New York: ACM Press, 2011: 627-638.

[12] LIU X L, DU X J, ZHANG X S, et al. Adversarial samples on android malware detection systems for IoT systems[J]. Sensors, 2019, 19(4): 974.

[13] KANG H, JANG J W, MOHAISEN A, et al. Detecting and classifying android malware using static analysis along with creator information[J]. International Journal of Distributed Sensor Networks, 2015, 11(6):474-479.

[14] PENG H, GATES C, SARMA B, et al. Using probabilistic generative models for ranking risks of Android apps[C]//The 2012 ACM Conference on Computer and Communications Security. New York: ACM Press, 2012: 241-252.

[15] CHRISTODORESCU M, JHA S, SESHIA S A, et al. Semantics-aware malware detection[C]//2005 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2005: 32-46.

[16] LEE J, JEONG K, LEE H. Detecting metamorphic malwares using code graphs[C]//The 2010 ACM Symposium on Applied Computing. New York: ACM Press, 2010: 1970-1977.

[17] SUAREZ T G, DASH S K, AHMADI M, et al. Droidsieve: Fast and accurate classification of obfuscated android malware[C]//The Seventh ACM on Conference on Data and Application Security and Privacy. New York: ACM Press, 2017. 309-320.

[18] DE-MAIORCA D, DE-ARIU D, CORONA I, et al. Stealth attacks: an extended insight into the obfuscation effects on Android malware[J]. Computers & Security, 2015, 51: 16-31.

[19] MING J, XIN Z, LAN P W, et al. Impeding behavior-based malware analysis via replacement attacks to malware specifications[J]. Journal of Computer Virology and Hacking Techniques, 2017, 13(3): 193-207.

[20] SARACINO A, SGANDURRA D, DINI G, et al. MADAM: effective and efficient behavior-based android malware detection and prevention[J]. IEEE Transactions on Dependable and Secure Computing, 2018, 15(1): 83-97.

[21] VINOD P, SHOJAFAR M, KUMAR N, et al. Identification of android malware using refined system calls[J]. Concurrency, Computation Practice and Experience, 2019, 75(2):1-30.

[22] SHABTAI A, KANONOV U, ELOVICI Y, et al. Andromaly: a behavioral malware detection framework for android devices[J]. Journal of Intelligent Information Systems, 2012, 38(1):161-190.

[23] MA W, DUAN P, LIU S, et al. Shadow attacks: automatically evading system-call-behavior based malware detection[J]. Journal in Computer Virology, 2012, 8(12):1-13.

[24] LEE S H, KIM S H, KIM S, et al. Appwrapping providing fine-grained security policy enforcement per method unit in android[C]//2017 IEEE International Symposium on Software Reliability Engineering Workshops. Piscataway: IEEE Press, 2017: 36-39.

[作者简介]

宋宇波（1977− ），男，江苏无锡人，博士，东南大学副教授，主要研究方向为无线网络和移动通信安全、移动终端安全、专有数据安全、区块链安全等。

陈琪（1996− ），女，江苏泰州人，东南大学硕士生，主要研究方向为物联网流量识别、Android 隐私保护等。

宋睿（1994− ），男，江苏宿迁人，东南大学硕士生，主要研究方向为移动终端安全、专有数据安全、区块链安全等。

胡爱群（1966− ），男，江苏南通人，博士，东南大学教授，主要研究方向为无线网络安全、物理层安全技术。