PDiOS:iOS应用程序中私有API的调用检测

吴 姝 周安民 左 政
(四川大学电子信息学院 成都 610065)

摘 要 苹果公司对App Store上的每一款应用程序都进行了审核,包括是否存在访问用户敏感信息的私有API调用,但是仍有恶意应用通过了该项审查。针对iOS应用程序中私有API的调用问题,提出了一种动、静态相结合的检测技术PDiOS。通过反向分片和常量传播的静态分析方式来处理大部分API调用,基于强制执行的动态迭代分析来处理剩余API。静态分析包含了对二进制文件的全面分析以及对资源文件中隐式调用的处理,动态分析主要依赖于二进制动态分析框架进行迭代分析。最后通过对比公开头文件中的API来确定私有API的调用。在对官方商店的1012款应用程序的检测中,确认有82款应用程序存在共128个不同的私有API调用。在对企业证书签名的32款应用程序的检测中,确认有26款使用了私有API调用。

关键词 私有API,应用程序审查,反向分片,常量传播,强制执行
中图法分类号 TP309 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2018.04.027

PDiOS:Private API Call Detection in iOS Applications

WU Shu ZHOU An-min ZUO Zheng
(College of Electronic and Information,Sichuan University,Chengdu 610065,China)

Abstract Apple has reviewed every application in App Store,including private application programming interface(API) calls,but some malicious applications still escape from the review. Aiming at the private API call in iOS application,a detection technique combining dynamic and static analysis was proposed. Most of the API call sites were processed by static analysis of backward slicing and constant propagation,and the remaining APIs are dealt with by dynamic iterative analysis based on enforcement. Static analysis includes a comprehensive analysis of the binary file and the implicit call analysis in the resource file processing. Dynamic analysis mainly depends on the binary dynamic analysis framework for iterative analysis. Finally,the existence of private API is determined by comparing the API in the public header file. There are 82 applications with 128 different private API calls during the testing of 1012 applications in App Store,and 26 applications are sure to use private API calls in the 32 applications signed by the enterprise certificate.

Keywords Private application programming interface,Application vetting,Backward slicing,Constant propagation,Forced execution

1 引言

随着移动市场的快速发展,iOS应用的数量迅速增长,研究人员对应用程序的分析也不断深入。作为Apple明令禁止的调用接口,私有API的功能非常强大,对用户隐私以及系统安全性都有一定的破坏性。因此,本文设计了PDiOS检测模型来对私有API的调用进行全面检测,以保护用户的隐私以及iOS系统的安全。

Joorabchi M E等人[1]提出通过钩住应用程序来检查和运行用户界面(User Interface,UI)元素。Kurtz A等人[2]在移动设备指纹识别配置一文中提出的DiOS系统使用自动化分析方式来检索GUI层次结构并与GUI元素交互。以上基于GUI元素交互的设计存在很多限制,比如当转换条件不满足时,UI元素交互无法转移到新的UI状态,会导致代码覆盖率较低。Egele M等人随后提出的PiOS系统采取静态分析的方法来查看应用程序中的隐私泄露[3],但静态分析并不足以解析出应用程序中的所有应用程序编程接口(Application Programming Interface,API)。Deng Z等人[4]提出的iRiS系统在应用审核阶段检测了私有API的使用,该系统对Objective-C方法和C函数都具有较好的审查效果,但缺乏对Swift方法以及storyboard资源文件的检测。

与已有工作不同,本文并非通过GUI元素交互来跳转执

到稿日期:2017-08-21 返修日期:2017-09-13

吴 姝(1994-),女,硕士生,主要研究方向为iOS与macOS系统安全,E-mail:wushu_1994@sina.com;周安民(1963-),男,研究员,主要研究方向为安全防御与管理技术、移动互联网安全、云计算安全,E-mail:2871669252@qq.com(通信作者);左 政(1986-),男,博士,主要研究方向为恶意代码自动化检测、内核安全。

行路径,或无边界条件的强制执行,而是通过捕获事件处理程序的注册方式来触发应用程序,并依据二进制及资源文件静态分析生成的控制流程图来加载动态分析框架,从而强制迭代执行并完善调用图;另外,还增加了 Swift 方法调用以及隐式调用分析。本文第 2 节对 iOS 应用程序中的相关技术进行介绍;第 3 节描述了 PDiOS 私有 API 检测模型的构建以及实现;第 4 节通过测试 1012 款 App Store(iOS 官方应用商店)上的应用程序以及 32 款企业证书签名的程序,对本文提出的 PDiOS 检测模型进行评估;最后总结全文。

2 相关技术

2.1 函数调用方式

Objective-C 与 Swift 是用于构建 iOS 应用程序的主要编程语言。图 1 显示了在 Objective-C 中,如何通过 objc_msgSend 系列函数之一向对象发送消息,从而实现方法调用。消息由接收者(receiver)和选择子(selector)组成[5]。在图 1 中,接收者为 myDemo 对象,选择子为需要调用的方法名 myMethod 以及对应参数 0。objc_msgSend 函数首先定位到对象的 isa 结构体,找到该类的实例方法列表,进而搜索到方法的实现 imp,从而完成整个消息的转发过程。在编译期间,所有的方法调用都被编译为消息传递模型,而模型中的接收者以及选择子等参数直到运行时才被动态确定。由于 Objective-C 是 C 语言的超集,因此在 Objective-C 中使用 C 语言是完全合法的。

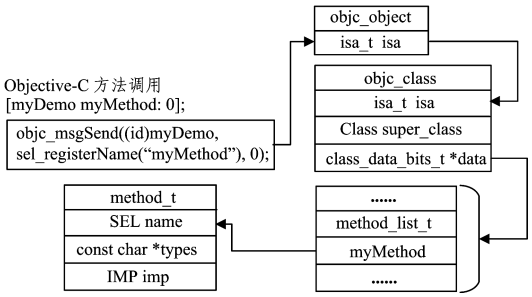


图 1 Objective-C 的消息传递方式

Fig. 1 Message passing way of Objective-C

由 swift 解释器或 swiftc 编译器链接的 Swift 方法,可以使用与 Objective-C 相同的运行时(runtime),被苹果公司形容为"没有 C 的 Objective-C"。相比于单独的 Objective-C 语言编程,Swift 与 Objective-C 的混编程序大大增加了研究人员的逆向和分析难度。

2.2 Mach-O 文件和资源文件

Mach-O 是 iOS 系统上的可执行文件格式,由文件头(Header)、加载命令(Load Command)、具体数据段(Segment)和节(Section)组成[6]。文件头可以帮助系统快速定位其运行环境以及文件类型。加载命令对处理随后的二进制数据、系统内核加载器以及动态链接器起指导作用。Mach-O 文件有多个段,每个段拥有不同的功能,也被分为很多小节,节用来存放具体的数据。对于包含多种架构的复合二进制文件,本文使用 MacOS 系统自带的 lipo 命令来提取 ARM64 架构的文件进行测试。

资源文件主要用于创建视图。xib,nib 以及 storyboard 都是由 Interface Builder 图形界面设计工具生成的资源文件。nib 和 xib 文件在经过 Xcode 编译后转化为编译后的 nib 文件,本文不对两者做过多区分。nib 文件描述了由 Interface Builder 创建的对象以及对象间的关系,其中对象包括接口对象(如视图、控件等)和占位符对象(如 First Respinder 等);对象间的关系包括文件所有者、属性方法绑定以及代理绑定等。一个 nib 文件对应着一个视图控制器和多个视图,storyboard 是多个 nib 文件集合的描述文件。

2.3 事件处理程序

应用程序通过将事件处理程序注册到对应按钮来处理用户输入。iOS 实现事件处理程序的两种主要设计模式,即目标-行为模式和委托模式[7]。目标-行为模式,即当某个事件发生时,应用程序会调用已设置好的对象的对应方法。开发者调用 addTarget:action:forControlEvents。接口为对象的控件事件注册一对目标和动作,action 参数是触发事件时应该调用的方法名称,target 参数是调用该方法的对象[5]。委托模式主要是为了降低对象的复杂度和耦合度,方便代码管理和代码实现。当事件发生时,视图会向其对应的代理对象发送一个消息,以此调用相应的事件处理程序。

2.4 框架中的私有 API

iOS 为开发者提供了丰富的以 .framework 为扩展名的动态框架。这些框架包含了库的二进制文件、头文件以及有关资源文件,用以共享代码和资源。位于集成开发工具 Xcode 中的软件开发工具包(Software Development Kit,SDK)目录下的 Frameworks 文件夹内的框架,是可以用于第三方 iOS 应用程序的公共框架。位于 PrivateFrameworks 文件夹内的框架为私有框架,这类框架没有提供头文件,仅被 iOS 内置应用或公共框架使用。私有框架内的 API 为狭义的私有 API,与公共框架内未被文档记录的 API 共同组成广义的私有 API,如图 2 所示。本文中提到的私有 API 均为广义私有 API。

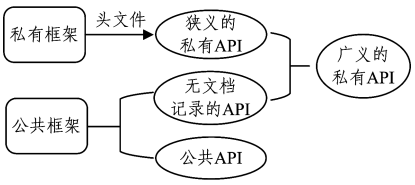


图 2 API 分类图

Fig. 2 Classification of API

私有 API 提供了许多强大的功能,如直接访问设备信息,读取应用程序的 Bundle ID 和用户手机号码等。如果私有 API 被第三方应用调用,则可能会威胁系统的安全性,泄漏用户的隐私。因此,苹果公司明确规定禁止第三方应用程序使用私有 API,并只对公共框架中的公开 API 进行了文档记录和公开。尽管如此,攻击者仍然可以通过逆向手段从框架的动态共享库中获得私有 API 的原型,进而调用私有 API[8]。研究表明,有许多使用私有 API 的应用程序通过了 App Store 的审查机制,如使用 dlopen 和 dlsym 函数来加载调用私有 API,或运行时构建选择子等[9]。为此,本文设计了一种私有 API 检测模型 PDiOS,以对应用程序中的私有 API 调用情况进行更加全面的审查。

3　PDiOS 检测模型的构建与实现

本文设计通过确定 iOS 应用程序二进制文件中的所有 API 调用目标，来检测是否存在私有 API 调用。Objective-C 的运行时特性，使得仅仅通过静态分析的方式并不能确定所有 API 的调用目标。而完全使用诸如符号执行[10]或强制执行[11]的动态分析方式，也因为 iOS 应用程序的大型化和复杂化，而变得不可行。因此，本文采用了静态分析和动态分析相结合的方法进行检测。iOS 应用程序中的大部分调用目标都可以通过静态分析来解决，并且速度较快。静态分析的结果为下一阶段的动态分析提供了必不可少的基础，同时也大大减少了动态分析所需的时间。

本文设计的 iOS 应用程序中的私有 API 检测模型的流程如图 3 所示。首先，从 iOS 官方应用程序商店 App Store 中下载 iOS 应用程序包，即扩展名为 .ipa 的压缩文件，其包含应用程序可执行文件、资源文件和其他元数据文件。nib 以及 storyboard 资源文件作为隐式调用分析的输入文件，通过资源文件分析以及事件处理程序分析生成隐式调用目标。可执行文件为 Mach-O 格式，已被 App Store 官方加密，解密后的文件可通过内存 dump 的方式来获得[7]。然后，使用反汇编软件 IDA Pro[12]打开下载的 iOS 应用程序包，同时加载隐式分析结果进行静态分析，生成调用图。最后，利用动态二进制分析框架加载应用程序，通过自然执行和强制执行来解决调用图中未知的调用目标。每次迭代后的已解析目标将被合并到调用图中，以便于在稍后的迭代中解析更多的调用目标。所有迭代完成后，将最终调用图中的调用目标与 iOS SDK 中的公开头文件 API 进行对比，以检测是否存在私有 API 调用。

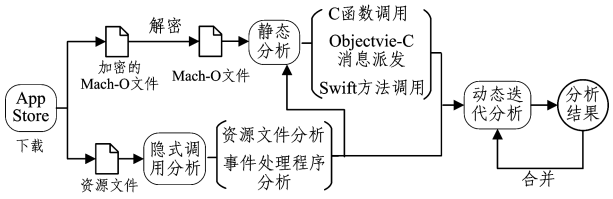


图 3　PDiOS 模型的检测流程

Fig. 3　Testing flow of PDiOS model

3.1　隐式调用分析

3.1.1　资源文件分析

应用程序加载 nib 文件时，首先将 nib 文件内容以及相关资源加载到内存，包括已被序列化的 nib 对象图数据、自定义的图片和声音资源等；然后对对象图数据进行反序列化操作，通过 initWithCoder 方法或者 init 方法初始化对象；接着，重建对象的联系，如 action，outlet 等；最后向被 nib 加载代码初始化的对象发送 awakeFromNib 消息，显示可见对象。storyboard 被加载时，首先会实例化对应的视图控制器，并将对应视图加载到 UIWindow 对象中，通过调用 didFinishLaunchingWithOptions 方法来进入程序入口。

如上所述，资源文件的加载过程涉及许多 API 的隐式调用。隐式调用分析使用 UIKit 框架来模拟应用程序启动，从而加载资源文件。由于存储在资源文件中的对象大部分为应用程序自定义的类，因此须利用应用程序自身提供的上、下文环境来加载对应的资源文件。隐式调用分析以动态库的形式将加载 API 的代码添加到构造函数中，然后设置 DYLD_INSERT_LIBRARIES 环境变量，在应用程序执行其他初始化操作之前加载资源文件。在加载分析资源文件之后，立即调用 exit 方法来终止应用程序，从而避免执行不相关的代码。

3.1.2　事件处理程序分析

事件处理程序在资源文件加载期间被注册，但并不被直接调用，因此本文将其作为隐式调用对象进行处理。在资源文件加载期间，应用程序可以通过 Objective-C 运行时注册函数的参数，以获取事件处理程序的入口地址。通过观察方法在目标对象中是否实现了动作选择器，来判断其是否为目标-行为事件处理程序。其中，无法解析文件所有者对象的 setter 方法将会被直接添加为隐式调用目标，以待后续分析。另外，隐式调用分析还将枚举所有在代理或数据源对象的类中实现的方法。在资源文件的加载过程中，隐式调用分析通过检查框架间的内部交互中使用的私有 API 是否在应用程序二进制代码的范围内，来判断其是否为应用程序中函数的调用目标。

3.2　静态分析

静态分析阶段的目标是生成控制流程图（Control Flow Graph，CFG），解决尽可能多的调用目标，以生成详细的调用图。CFG 是以基本块为节点的有向图[14]。反汇编程序 IDA Pro 可以自动生成 CFG，但能够解决的调用目标有限。首先，对于传统的 C 函数调用和 Swift 方法调用，IDA Pro 只能识别指令中嵌入常量相对地址的直接调用目标，无法解析存储在寄存器中的间接调用目标。其次，对于分析 Objective-C 方法，IDA Pro 无法解析存储在寄存器或堆栈变量中的函数参数，这正是静态分析 Objective-C 运行时特性的难点所在。为了解决这一问题，本文采用 IDA 插件的形式来构建分析组件。

3.2.1　C 函数调用

在 ARM 架构中，函数调用由 BL 指令（带链接跳转）和 BLX 指令（带链接和状态切换跳转）完成。枚举二进制文件中的这些指令，并检查其操作数。常数操作数已被 IDA Pro 识别。而寄存器操作数则采用反向分片[13]和转发常量传播[4]的方式来解析，即向后遍历汇编代码，并递归识别、记录与调用目标相关的寄存器或堆栈变量的指令片段，从而解析指令引用的操作数。当遍历到函数的开始地址或获取到确定的值时，停止切片。对于未解析的操作数，将相应的调用目标标记为未知。如果目标地址在导入符号节中，或是作为调用外部 API 的蹦床（trampoline）时，则将其标记为外部 API。

3.2.2　Objective-C 消息派发

ARM64 架构下的 Mach-O 二进制文件格式如图 4 所示，其中__TEXT 以及__DATA 段包含大部分的解析数据，如__objc_classlist节包含开发者自己实现或静态链接的类列表，__objc_classrefs 节中存储了引用的类列表等[2]。对于非静态链接的动态链接库中的类和方法，本文通过分析 iOS 系统中对应的 dyld_shared_cache_x（x 为 7/7s/64）文件得到。最终，可以获取各个类的函数声明以及类层次结构中的相应方法。

对于不同结构的 Objective-C 消息，采取相对应的分析方法。

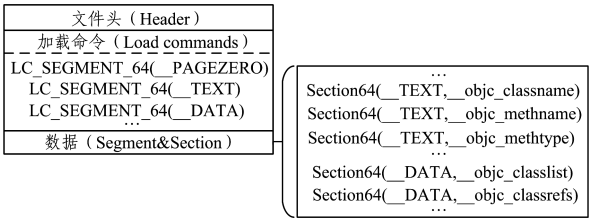


图 4 Mach-O 文件格式

Fig. 4 File format of Mach-O

block 结构：在静态分析中，只存在位于栈上的_NSConcreteStackBlock 和全局静态的_NSConcreteGlobalBlock。在获取栈 block 的 isa 指针之后，紧接着获取该 block 的函数地址。全局 block 以结构体的形式存在于静态常量区。对此，本文以 IDA 插件的形式将 block 符号信息导出为一个文本文件，并将其添加至原 Mach-O 文件的符号表以供解析。

其他消息派发函数：对于显示调用父类方法的其他消息派发函数，如 objc_msgSendSuper，需要应用两轮切片方法。第一轮解析指向 objc_super 结构的参数，第二轮解析结构中父类的名称。

大中央调度器（Grand Central Dispatch，GCD）：iOS 上调度并发代码执行的运行时系统。GCD API 的参数为函数指针或块对象，本文同样采取反向切片来获取函数地址，添加调用目标。对于嵌套的消息传递，如 NSObject 根类提供的 performSelector 家族方法，本文采用递归解析函数参数的方式，直至提取到最内层的消息，然后将其添加为实际的调用目标。

Method Swizzling：Objective-C 方法的实现可以在运行时进行替换，即 Method Swizzling。恶意应用程序的开发者可以定义一个合法的占位符，然后在运行时将其替换为私有 API 的实现。为防止这种攻击，本文会检索 Method Swizzling 中与替换方法实现相关的 Objective-C 运行时方法。虽然使用这种方法并不一定意味着调用私有 API，但考虑到静态方法的复杂性，本检测系统仍然将这类行为归为不合法行为。

3.2.3 Swift 方法调用

在编译时，Swift 函数会按照特定规则被重新编译为新的 C 函数，通过 Xcode 自带的 xcrun swift-demangle 命令可以还原其 Swift 函数原型。对于 dynamic 修饰的函数，与 Objective-C 类似，通过调用 objc_msgSend 系列函数，进入 objc runtime。因此，大部分的 Swift 方法调用可以进一步拆分为 C 函数调用以及 Objective-C 消息派发。

3.3 动态分析

3.3.1 动态分析框架

本文通过移植 Valgrind 环境[15]到 iOS 系统来构建动态分析框架。Valgrind 虽然支持 ARM 架构以及 MacOS 系统，但并不兼容 iOS。在 iRiS[4]的基础上，本动态分析框架修改了部分代码，主要包括：

1）修改系统调用约定。首先构建一个空白程序，通过自行设计的参数执行系统调用的封装函数。在运行过程中，结合 LLDB 调试器在指定函数上设置断点，并在断点处执行单步调试操作，直到抵达 SWI 指令部分。最后，通过观察参数值的存储位置，如关键指令、寄存器、堆栈位置等，来判断调用约定。通过这种方式，可以导出 iOS 内核中的 3 种形式的系统调用约定，分别为 BSD，MACH 以及 MDEP。相比于 iRiS 中采用的 GDB 调试器，本组件采用的是 LLDB 调试器，其对 iOS 系统的调试性能更好。

2）获取缓存符号表。Valgrind 维护了一个用于将地址转换为可读的 API 名称的符号表。但对于 iOS 而言，不存在单个的系统动态链接库。自 iOS 3.1 以后，所有的共享库被组合保存在/System/Library/Caches/com.apple.dyld/目录下的名为 dyld_shared_cache_x（x 为 7/7s/64）的大型缓存文件中。应用程序被加载时，这个缓存文件也被内核映射到对应的地址空间中。可以通过调用 shared_region_check_np 系统调用来获取共享缓存的起始地址，从而得到系统动态链接库的符号。

3）解决运行时的内存限制。在 iOS 上，Apple 引入了 Jetsam 机制[16]。系统使用多种内核 API 来设置其阈值和被 kill 进程的顺序。本框架采用 memorystatus_control API 中的 SET_JETSAM_TASK_LIMIT 和 SET_PROPERTIES 参数来调整内存限制。

4）修改运行权限。对于 GUI 应用程序，在启动时，该应用程序必须向 SpringBoard 进程发送含有应用程序 Bundle ID 的启动请求。但是，SpringBoard 启动的应用程序的所属用户为 mobile，不具备执行所需的 root 权限。因此，本框架增加了 setuid 代码，并通过另一 mobile 用户的正常应用程序来启动修改后的 Valgrind 程序，分别设置两个可执行文件的权限以及用户组，以此来获取 root 权限，并逃避 SpringBoard 权限检测。

3.3.2 迭代分析

迭代分析主要通过迭代执行动态分析框架，来解决在静态分析中剩余的调用目标，流程图如图 5 所示。只要应用程序执行函数调用，就可以从调用站点（call site）的执行状态来获取目标和参数。难点在于，如何使应用程序到达特定的调用站点来执行任务。为了解决该问题，本文基于 iRiS 中提出的迭代算法[4]以及 X-Force 中提出的路径探索算法[11]，使用动态迭代分析来查找和探索可能到达目标函数的路径。具体实现如下所述：

1）应用程序直接在构建好的动态分析框架中运行，记录在自然状态下覆盖的静态分析以及隐式调用分析后的调用站点。S 为所有调用站点的集合，并作为第一轮迭代的输入参数。

2）在每轮迭代中，分析组件将循环处理每个未解决的调用站点。例如，给定一个未解析的调用站点 A，计算与站点 A 相关的调用站点。相关调用站点是指属于 S 的任何调用站点到站点 A 的路径上的调用站点。这些相关调用站点可以用于指导应用程序执行至未解析的调用站点。

3）在循环中，如果调用站点 A 的相关站点集合 $RA(n)$ 与前一次的迭代分析结果 $RA(n-1)$ 不同，则使用实现路径探索算法的 XExecute 函数探索新的路径，并更新调用图。如果

相同，则跳出循环。

4）每次迭代更新后的调用站点图都将作为下一次迭代分析的输入参数，直至迭代后调用图不再变化时保存调用图，退出动态迭代分析。

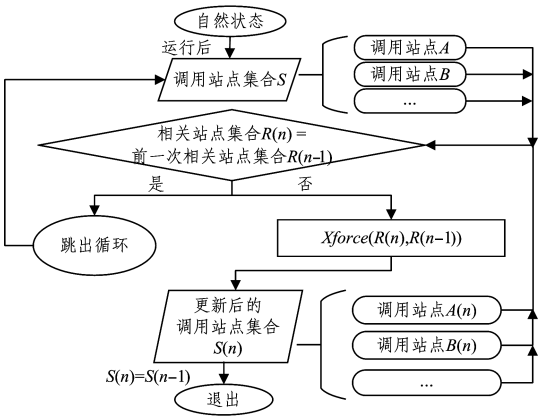


图 5　动态迭代分析流程图

Fig. 5　Flow chart of dynamic iterative analysis

步骤 3）中的 XExecute 函数是基于 X-Force 中的路径探索算法[11]实现的。如表 1 所列，XExecute 中的调用站点与 X-Force 中提出的基本块类似，调用站点之间的转换与基本块末尾的分支类似，并同样在分支上强制控制流，以探索程序路径。不同点在于，在没有边界的地方，XExecute 限制了调用站点到相关调用站点间的转换，使得分支向需要解析的调用站点处执行。假设当前的调用站点为 A，需解决的相关调用站点为 B，函数 f 包含调用站点 B，且为 A 的调用目标之一。强制调用站点 A 转换到调用站点 B，意味着需要强制函数 f 中的分支使得控制流从函数入口了转到调用站点 B。该算法将确定从函数 f 的输入基本块到包含调用站点 B 的基本块之间的控制流路径，并计算其中的所有基本块，将其指定为必要基本块。也就是说，如果执行其他基本块，将无法到达调用站点 B。从函数 f 开始，在每个分支处，如果分支目标不属于必要基本块的集合，则强制执行分支目标。因此，在迭代过程中，动态分析框架将使用尽可能少的分支到达调用站点 B。动态分析完成后，通过对比公开头文件中的 API，确定私有 API 的调用情况。

表 1　XExecute 与 X-Force 的对比

Table 1　The contrast of XExecute and X-Force

	XExecute	X-Force
不同点	无边界处限制转换	无边界处不做处理
相同点	1）调用站点与基本块概念类似 2）分支处强制控制流转换	

4　评估

本文测试了 App Store 上的 1012 款免费应用程序，它们分别来自以下类别：教育、报刊杂志、体育、商务、天气、生活、图书、娱乐和美食佳饮。本文通过 iTunes 下载应用程序并解密，在 iPhone 5s，iOS9.3.2 以及 iPhone 6s，iOS10.2.1 环境中运行。首先，将资源文件作为隐式调用分析的输入文件，并将其生成的隐式分析结果与 Mach-O 文件一同进行静态分析，生成调用图。利用动态分析框架加载应用程序以强制迭代执行，最后将调用图中的调用目标与公开头文件中的 API 进行对比。

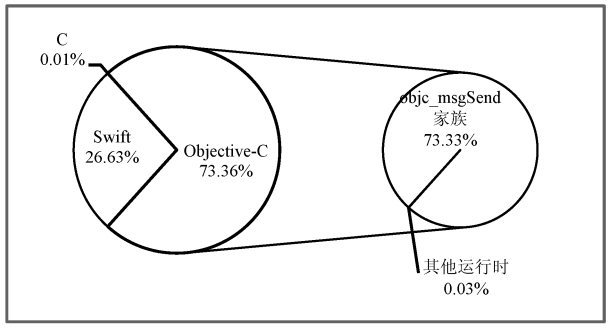
本模型共确定了 1012 款应用程序中的 49814336 个 Objective-C 相关方法调用（包括 49792643 个 objc_msgSend 家族消息的派发函数，21693 个其他 Objective-C 运行时函数），18083610 个 Swift 方法以及 3954 个 C 函数调用。其中，由静态分析确定的调用站点有 67878422 个，约为总调用站点数的 99.97%。对于 Objective-C 相关的调用站点，本文中设计的私有 API 系统能够识别 87%的调用对象类别，iRiS 的识别率为 85%，PiOS 的识别率为 82%，如表 2 所列。另外，iRiS 和 PiOS 中没有对 Swift 方法调用的研究。值得一提的是，在静态分析中，并没有找到任何与私有 API 相关的调用站点。

表 2　OC 调用对象类别识别的对比

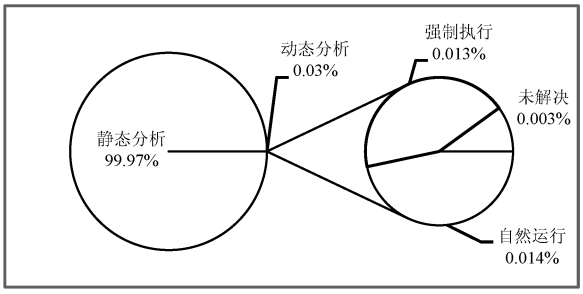
Table 2　Contrast of OC call object category identification

系统模型	Objective-C 调用对象类别识别/%
iRiS	85
PiOS	82
本模型	87

以上应用程序中虽然需要动态分析的调用站点非常少，约占总调用站点数的 0.03%，但是有 961 款应用程序都无法仅靠静态分析解决，约占总应用程序数的 94.96%。对于静态分析中无法解决的 23478 个调用站点，本模型通过动态迭代分析解决了其中的 21012 个调用目标，约为 89.50%。在动态分析中，应用程序的自然运行解决了约 46.12%的调用目标，其余约 43.38%的调用目标被强制执行，如图 6 所示。动态分析的耗时主要取决于待解决的调用站点数量以及应用程序的复杂度。本次分析中，所需最短时间为 321 s，最长时间为 156639 s。



(a)函数调用方式占比



(b)分析方式占比

图 6　分析结果占比图

Fig. 6　Proportion of analysis results

经过分析，在 1012 款应用程序中，有 82 款应用程序存在共 128 个不同的私有 API 调用。为进一步了解这些应用程序逃过 Apple 审查机制的策略，对其进行手动分析，其中最常用的方法是字符串拼接、API 名称混淆以及@selector 调用。有 3 个应用程序为私有 API 的调用定义了专有模块。一部分应用程序使用了加解密的方法对私有 API 名称进行模糊处理等。一部分私有 API 用于实现用户界面功能，如强制设备的显示方向、色调显示，控制亮度等，与设备的安全性无直接关系，因此本文不做详细讨论。剩余的私有 API 所属的框架如表 3 所列。

表 3 检测结果中的私有 API 框架

Table 3 Private API frameworks in detection result

私有 API 框架	APP 数量
StoreServices	2
IOKit	32
AppleAccount	1
SpringBoardServices	8
CoreTelephony	18
IOSurface	1
IOMobileFramebuffer	1
UIKit	15
MobileWiFi	25

StoreService 为私有框架，有一款应用程序通过 SSAccountStore 类中的 activeAccount API 以及 SSAccount 类中的 accountName API 来获取当前登录的 Apple ID 账户名。IOKit 框架用于与 iOS 设备中的低级硬件通信，虽然属于公共框架，但其中仍包含较多私有 API。有 15 款应用程序通过该框架来访问各种硬件信息，如通过 IORegistryEntryCreateCFProperty API 读取 IOService 对象中的 IOPlatformSerialNumber 属性，从而访问设备的序列号等。SpringBoardServices 框架是 iOS 上管理应用程序的框架，有 8 款应用程序通过它来查询设备上的应用程序状态，如当前运行的程序的 Bundle ID 等。CoreTelephony 框架主要与通信相关，可以通过 CTSettingCopyMyPhoneNumber API 获取 iOS 设备上的手机号码。通过 IOSurface，IOKit，IOMobileFramebuffer 3 个框架可以实现截屏功能。UIKit 框架中也有私有 API 存在类似的功能，在视图中捕获显示的内容并保存为图像。MobileWifi 框架被用于 Wi-Fi 扫描和连接。与手机号码相关的私有 API 在一个天气分类的应用程序中使用。

另外，本文还通过私有 API 检测系统审查了未在 App Store 上架的由企业证书签名的 32 款应用程序，其中有 26 款应用程序被检测出使用了私有 API，约占 81.25%。由此可见，未经 App Store 审核的应用程序使用私有 API 的可能性非常大。

结束语 随着 iOS 应用程序量的迅速增长，私有 API 的强大功能使得恶意 iOS 应用程序极有可能通过调用私有 API 来访问用户的敏感信息，从而对设备的安全性造成威胁。即使上架 App Store 的应用程序已经过审查，但仍有漏网之鱼。本文提出了一种私有 API 检测技术，通过反向分片和常量传播的静态分析，对 nib 和 storyboard 资源文件的隐式调用进行分析，加载动态二进制框架，通过强制执行应用程序进行动态迭代分析来检测 iOS 第三方应用程序中的私有 API 调用。本模型目前还无法捕获外部输入产生的控制流中的私有 API 调用，未来将对此展开进一步研究。

参 考 文 献

[1] JOORABCHI M E, MESBAH A. Reverse engineering iOS mobile applications[C]// 2012 19th Working Conference on Reverse Engineering(WCRE). IEEE, 2012: 177-186.

[2] KURTZ A, GASCON H, BECKER T, et al. Fingerprinting mobile devices using personalized configurations[J]. Proceedings on Privacy Enhancing Technologies, 2016, 2016(1): 4-19.

[3] EGELE M, KRUEGEL C, KIRDA E, et al. PiOS: Detecting Privacy Leaks in iOS Applications[C]// NDSS. 2011: 177-183.

[4] DENG Z, SALTAFORMAGGIO B, ZHANG X, et al. iRiS: Vetting private api abuse in ios applications[C]// Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015: 44-56.

[5] SERIOT N. iPhone Privacy[EB/OL]. [2010-02-03]. http://seriot.ch/resources/talks_papers/iPhonePrivacy.pdf.

[6] iOS Technology Overview[EB/OL]. [2016-05-01]. https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007898-CH1-SW1.

[7] 关东升. iOS 开发指南[M]. 北京：人民邮电出版社，2016.

[8] AGARWAL Y, HALL M. ProtectMyPrivacy: detecting and mitigating privacy leaks on iOS devices using crowdsourcing[C]// 11th Annual International Conference on Mobile Systems, Applications, and Services. ACM, 2013: 97-110.

[9] GARCÍA L, RODRÍGUEZ R J. A Peek under the Hood of iOS Malware[C]// 2016 11th International Conference on Availability, Reliability and Security(ARES). IEEE, 2016: 590-598.

[10] MOU L, LU Z, LI H, et al. Coupling distributed and symbolic execution for natural language queries[J]. arXiv preprint arXiv: 1612.02741, 2016.

[11] PENG F, DENG Z, ZHANG X, et al. X-Force: Force-Executing Binary Programs for Security Applications[C]// USENIX Security Symposium. 2014: 829-844.

[12] Hex-Rays. IDA Pro[OL]. http://www.hex-rays.com/idapro.

[13] WEISER M. Program slicing[C]// International Conference on Software Engineering. IEEE Press, 1981: 439-449.

[14] SABELFELD A, MYERS A C. Language-based information-flow security[J]. IEEE Journal on Selected Areas in Communications, 2003, 21(1): 5-19.

[15] NETHERCOTE N, SEWARD J. Valgrind: a framework for heavy weight dynamic binary instrumentation[J]. ACM Sigplan notices, ACM, 2007, 42(6): 89-100.

[16] LEVIN J. Mac OS X and IOS Internals: To the Apple's Core [M]. England: John Wiley & Sons, 2012.