基于模拟执行的 Android 应用 Java 方法与 Native 函数的映射识别

徐贯虹, 傅建明, 聂　宇, 解梦飞

武汉大学国家网络安全学院 空天信息安全与可信计算教育部重点实验室 武汉 中国 430072

摘要　Native code 被广泛应用, 为移动应用提供丰富的功能和开发方面的便利性。然而, Native code 天然的跨层执行行为给 Android 应用数据流分析带来了挑战。由于语言与程序运行机制的差异, 过去针对 Android 应用的数据流分析往往仅关注 Java 层代码行为, 这种跨层分析断点使得隐私泄露和恶意代码行为可以轻易地隐藏在 Native 层中。针对这一问题, 现有工作尝试基于静态分析建立 Java 与 Native 之间的方法调用映射, 从而补全跨层行为分析的断点。然而, 这些方案既无法应对 Native 库中广泛存在的保护机制, 也缺乏对 Native 方法动态绑定机制的理解。在本文中, 我们提出了 JNativeEmu, 一种基于模拟执行的跨层方法调用映射分析工具。JNativeEmu 以跨层调用注册作为解析入口, 在模拟执行过程中补全基本的系统调用与 JNI 依赖。通过符号执行的引导, 它能够准确建模 Android 应用中的跨层映射, 为后续的跨层数据流分析提供可靠的支持。JNativeEmu 的方法增强了对 Native code 跨层执行行为的理解, 解决了现有数据流分析的跨层分析局限。我们对应用市场 50 个流行应用中 1309 个 Native 库的分析结果表明, JNativeEmu 能够正确模拟执行其中 83.2%的 Native 库并且没有发生崩溃。进一步地, 在动态注册 Native 方法的解析成功数量上, JNativeEmu 的识别结果较 Jn-saf 提高了 2.23 倍。此外, 本文还通过案例研究对 Native 库中的函数注册实现和相应的程序保护机制进行了具体分析。

关键词　本地代码; 跨层程序分析; 混淆; 模拟执行

中图法分类号　TP311.5　DOI 号 10.19363/J.cnki.cn10-1380/tn.2025.05.01

Identify the Mapping of Java Method and Native Function for Android Applications Based on Simulated Execution

XU Guanhong, FU Jianming, NIE Yu, XIE Mengfei

Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China

Abstract　Native code plays a crucial role in enhancing the functionality and convenience of mobile applications. However, the inherent cross-layer execution behavior of native code poses challenges for effective data flow analysis in Android applications. Due to the difference in language and program execution mechanism, data flow analysis for Android applications in the past often focuses only on Java layer code behavior, and such cross-layer analysis breakpoints allow privacy leakage and malicious code behavior to be easily hidden in the Native layer. To address this problem, existing work tries to establish call mapping between Java method and Native function based on static analysis, attempting to complement the breakpoints of cross-layer behavior analysis. Nevertheless, these approaches fall short in handling the widespread protection mechanisms within Native libraries, and lack a comprehensive understanding of the dynamic binding mechanism of Native methods. In this paper, we present JNativeEmu, a cross-layer call mapping analysis tool based on simulated execution. JNativeEmu utilizes the cross-layer call registration as the parsing entry point, and complements essential system calls and JNI dependencies during the simulated execution process. Guided by symbolic execution, it is able to accurately model cross-layer mappings within Android applications, so as to provide reliable support for the subsequent cross-layer data flow analysis. JNativeEmu's approach enhances the understanding of cross-layer execution behavior hidden in Native code, addressing the cross-layer analysis limitations of existing data flow analysis. Through the analysis of simulating 1,309 Native libraries from 50 popular apps in the app market, JNativeEmu demonstrates an accurate execution simulation for 83.2% of these Native libraries without encountering crashes. Further, in terms of the number of successful parsing of dynamically registered Native methods, JNativeEmu recognizes 2.23 times better results than Jn-saf. In addition, to provide deeper insights, this paper includes detailed case studies analyzing the implementation of function registrations in Native libraries and the corresponding program protection mechanisms.

Key words　native code; cross-layer(Java and Native) program analysis; obfuscation; simulated execution

---

通讯作者: 傅建明, 博士, 教授, Email: jmfu@whu.edu.cn。

本课题得到重点研发项目(No. 2021YFB3101200)、国家自然科学基金(No. 62272351, No. 61972297, No. 62172308)资助。

收稿日期: 2023-10-20; 修改日期: 2023-12-19; 定稿日期: 2025-02-27

1 引言

移动互联网时代不断发展, 智能移动设备已经取代传统 PC 成为了最常用的上网设备。中国互联网络信息中心在 2022 年发布的第 49 次《中国互联网络发展状况统计报告》显示[1], 截至 2021 年 12 月, 高达 99.7% 的网民使用手机上网。而 Android 作为最主流的移动设备操作系统, 市场占有率维持在 70%以上[2]。

依托于移动设备来提供服务的移动应用, 也被普遍应用在日常生活中。Android 应用作为移动应用的典型代表, 主要由 Java 编写而成, 也常常会应用 C++、C、汇编等, 借助 JNI(Java Native Interface)机制[3], Android 应用能够从 Java 层跨层调用由 Native 动态链接库(以下缩写为 Native 库)提供的 Native 代码。Native 代码常用于实现视频处理、图片处理、人脸识别、身份认证等常用的功能性任务, 也会应用于加解密、软件安全防护(包括加固、动态加载、环境检测等)、日志记录等安全任务。然而, 也正是由于 Java 层与 Native 层程序语言和运行机制的不同, 难以对 Android 应用进行统一的跨层数据流分析, 存在跨层分析断点。如图 1 所示, Java 方法直接由 Java 虚拟机负责调用处理, 而声明在 Java 层的 Native 方法则需要 JNI 的支持, 匹配并调用 Native 库中的 Native 函数具体实现。

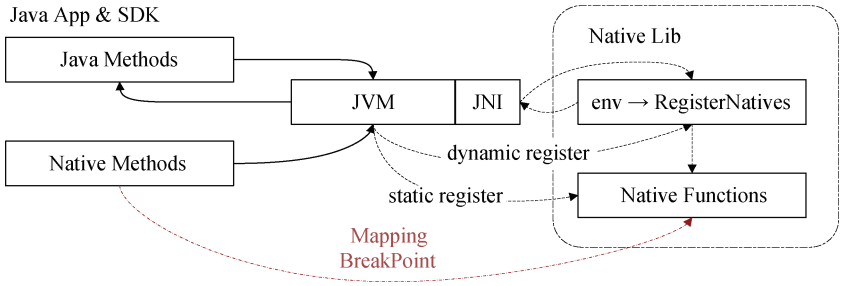


图 1 Native 层跨层断点示意

Figure 1 Schematic of Native cross-layer breakpoints

现有的先进的 Android 程序分析工具(例如 FlowDroid[4]、Iccta[5])在进行分析时, 由于 Android 中 Java 层与 Native 层跨层分析断点的存在, 往往选择忽略 Native 层, 尤其忽略对动态注册的 Native 方法的考虑。这影响了其下游任务, 例如隐私泄漏检测、恶意代码分析、漏洞检测等程序分析应用的完整性与可靠性。

跨层 Native 方法的匹配注册方式包括静态注册和动态注册两种, 静态注册基于函数符号的命名规则, 而动态注册则是通过间接调用 JNI 所提供的 API 指针表中的 RegisterNatives 完成。在跨层本身所带来的分析障碍以外, Native 库本身可能存在的程序保护也增加了跨层分析的难度。静态分析 Android 应用跨层行为一般基于关键的 JNI 调用符号信息与字符串常量信息, 例如 native-scanner[6]。由于静态分析缺乏应用程序具体运行时信息, 导致分析结果与实际结果之间可能存在较大差异, 在遇到复杂的程序实现(网络与本地文件读取等调用)与程序保护手段(加固、混淆加密)时, 难以分析 Native 库中 Java 方法与 Native 函数的映射行为(案例分析见第 6 节)。因此, 通过静态分析的方法分析 Native 方法动态注册行为, 存在着一定的障碍。相对地, 动态分析方法能够捕捉移动应用真实运行时的调用触发, 得到程序运行时的具体数值, 可靠且直观; 但动态污点分析方法往往需要修改系统源码进行插桩, 并且因为复杂应用中存在庞大繁杂的类的实现与方法的调用, 全面完整地获取并触发 Native 库的加载和 Native 方法的调用仍然是有一定困难的。

针对以上问题, 为了提供 Android 应用跨层分析支持, 基于对 Native 库加载与使用机制的理解, 本文提出了模拟执行 Native 库加载过程的跨层分析方法, 并实现了 JNativeEmu, 用于解析 Java 层到 Native 层的 Native 注册方法映射关系, 分析 Android 应用 Native 库跨层行为。

依据动态辅助静态的思想, JNativeEmu 在静态分析 Native 库的基础上, 使用 unicorn 支持的 angr[7] 的动态符号执行技术实现模拟。为了保证模拟执行过程的完整性和分析的可靠性, 我们以满足 Native 库加载过程中的依赖为出发点, 对 JNI 进行了建模, 拦截并实现了 Native 库中常见系统调用的回调钩子, 使得动态符号执行能够模拟正常的运行。

我们使用 JNativeEmu 分析了 Google 应用市场 5 个类别共 50 个 App 所包含的 Native 库, 评估了 Native code 的使用情况, 平均每个 App 使用 29 个

Native 库, 超过半数的 Native 库存在与 Java 层的跨层交互行为。在动态注册 Native 方法的解析效果方面, 因为能够对抗 Native 库自身存在的控制流混淆与数据混淆, JNativeEmu 的表现远远优于 Jn-saf, 解得数量超过 Jn-saf 的两倍, 对 Native 库中的 JNI 跨层交互行为的分析更加高效准确。

本文有以下贡献:

(1) 提出了一种基于模拟执行 Native 库加载过程进行程序分析的方法, 能够分析获取 Java 与 Native 注册方法的映射, 补充 Android 应用中的跨层数据流分析断点, 为跨层数据流分析提供了有力支持;

(2) 实现了 JNativeEmu, 一个基于 angr 动态符号执行实现模拟执行 Native 库加载分析方法的工具, 能够高效准确地分析 Android 应用的 Native 库中的 JNI 交互行为, 并对实际应用进行评估, 探索了 Native 库中混淆的使用情况;

(3) 为处理 Native 库跨层分析依赖中的混淆提供了一种思路, 并对实际 Android 应用进行案例分析, 具体逆向分析了商业 App 中 Native 库采用的代码实现手段以及程序保护措施。

文章剩余的章节如下组织。第二章比较了现有的相关工作及其实现技术, 第三章详细阐述了模拟执行 Native 库加载过程的分析方法, 第四章主要描述 JNativeEmu 的实现细节, 第五章评估了 JNativeEmu 在实际应用市场 App 数据集上的运行与分析效果, 第六章给出了具有代表性的案例分析, 第七章讨论了本文方法的局限性与未来工作。

2 相关工作

现有的程序分析方法和工具被广泛地应用在 Android 隐私泄露检测[8-9]、漏洞挖掘、恶意代码检测[10]等领域, 间接影响了使用它们进行程序分析的下游任务的可靠性。存在隐私泄漏或恶意代码的 App 为了逃避检测, 倾向于将其实现在 Native 代码中, 通过跨层操作来隐藏敏感 API 的使用, 使隐私泄露或恶意代码行为难以被检测到。

许多 Android 隐私泄漏检测工作仅针对 Java 层代码[4-5, 11], 无法处理 Native 方法与跨层分析断点。目前最为流行的 Android 静态数据流分析工具是 Arzt S 等人在 2014 年推出的 FlowDroid[4], 它对 Java 代码的静态分析建立在反编译工具 Dexpler[12]和 Java 分析工具 Soot 上; Li 等人设计了 Iccta[5], 通过静态插桩连接各个组件, 扩展了 FlowDroid 的跨组件分析功能, 使其对数据的跟踪能够在不同组件间保持连贯性; 在 2021 年, RAICC[11]同样关注组件间通信, 通过插桩补充达到更全面的隐私泄漏检测效果。

如果不考虑 Android 应用中 Java 与 Native 的跨层分析, 对 Android 应用的程序分析与相应的检测会缺乏完整性。目前对 Android 应用进行跨层程序分析的工作主要从静态与动态两方面出发。

(1) 静态分析

Gordon M 等人在 2015 年实现了面向 Android 应用的对象敏感静态分析工具 DroidSafe[13], 该工具支持跨组件和跨应用的交互, 对大多数 Android Framework 和 Native 方法建立了分析存根, 即语义等价的 Java 表示, 但这种方式不能应用于进行自身 Native 库函数的分析。

随着 Native 代码的使用越来越普遍, 研究者们开始考虑对 Android 应用中用到的 Native 代码进行分析。2020 年, Lee S 等人对跨层程序进行分析, 从 C 代码中提取 JNI 形式化抽象语义信息, 并通过替换 Java 中 Native 方法统一分析不同语言的程序流, 检测跨层语言交互错误[14]。George Fourtounis 等人从静态字符串分析的角度, 提出了一种获取从 Native 层到 Java 层回调的方法, 并实现为 native-scanner, 它通过扫描动态链接库二进制文件中的常量字符串, 静态指针分析 Java 代码结构的交叉引用, 恢复 Native 回调对应 Java 方法签名的静态符号信息[15]。但其对于动态注册的 Native 函数的匹配分析中, 仅考虑了常量字符串, 并且需要函数符号信息的支持, 无法处理混淆字符串; 对于 Native 层中 Java 回调的识别也仅考虑了硬编码的常量上下文。2022 年, Ruggia A 等人关注 Android 平台中 Native 代码进行恶意操作的情况, 设计并开发了静态分析工具 ANDani[10], 该工具分别使用 Soot 和 Ghidra 对 Java 和 Native 代码进行分析, 为 Java 和 Native 代码创建程序间控制流图 IPCFG。然而对于 Native 部分的解析处理, ANDani 仍是纯静态解析方法, 依赖于函数符号信息与常量字符串。

(2) 静态符号执行

为了更好地分析 Android 应用中的 Native 跨层行为, Fengguo Wei 等人在 2018 年设计并实现了静态分析框架 Jn-saf[8], 该框架对 JNI 进行建模, 通过符号执行进行 Native 层的数据流分析, 以实现 Android 应用 Java 层与 Native 层结合的跨层分析。但由于其 Native 层的数据流分析基于符号执行, 存在路径爆炸和难以处理混淆的问题, 导致 Native 层模拟出现失败、超时、崩溃等情况。2022 年, Jordan Samhi 等人设计并实现了 JuCify[9], 该工具同样使用符号执行记录 JNI 调用行为, 通过提取 Native 与 Java 之间的

调用信息, 初步构建了统一的调用图, 并基于启发式规则将 Native code 转化为 Jimple 中间语言, 从而构建用于跨层数据流分析的 Java 与 Native 统一模型。但其对于符号执行的使用与 Jn-saf 具有相同的问题, 并且通过启发式规则转换中间语言, 仅考虑了跨层调用指令, 不够准确和稳定。Jn-saf 和 JuCify 都难以直接应用到真实世界中的实际应用, 在大规模自动化分析中表现不佳。

(3) 动态分析

静态分析难以克服跨层交互、混淆等分析断点, 这些分析断点使得静态分析存在着局限性。而在 Android 应用的动态数据流分析方面, Enck W 等人在 2014 年推出了动态污点分析工具 Taintdroid[16], 通过插桩将隐私数据标记为污点, 并在动态执行的过程中为所有的对象申请额外的空间用于污点标记, 根据污点标记和污点传播规则追踪数据流, 它能够准确解析动态加载和反射调用, 但在程序覆盖率方面存在不足, 且仅支持 Dalvik 虚拟机。LK Yan 等人设计的 DroidScope[17]则是以 QEMU 模拟器为基础实现了动态污点分析。Sun Ms 等人设计的 TaintArt[18]插桩 ART 编译器和 JNI 关联函数, 实现 ART 的多标签污点分析, 但它不支持隐式数据流。2019 年, Xu G 等人设计的 SoProtector[19]通过修改源码的动态记录日志提取反射、动态加载以及 Native 方法用于检测隐私泄漏, 并分析 Android 应用中是否存在恶意 Native 库。虽然动态分析方法在运行效果上有很大优势, 但是往往实现复杂, 需要修改系统源码或进行代码插桩、重打包, 存在反调试、环境检测对抗的问题, 并且对于 Native 库及 Native 方法的调用难以触发完全。

静态分析缺乏运行时信息存在分析断点, 容易产生误报和漏报, 动态分析覆盖率难以保障, 且实现与运行成本高。为了解决静态分析的分析断点, 2021 年 Xueling Z 提出了上下文敏感的动态辅助静态污点分析方法 ConDySTA[20], 通过重打包与插桩, 获取动态污点分析的调用栈上下文信息, 补充静态数据流分析路径。但该方法需要插桩与重打包, 同样存在动态分析所具有的实现与运行成本问题。

(4) 本文工作

与本文设计方案类似的是 Jn-saf, 都使用符号执行技术分析 Native 库, 进行了基本的 JNI 建模, 但我们的实现机制相对于 Jn-saf 的静态符号执行有较大的不同。首先, 我们使用了基于 unicorn 支持的动态符号执行方法, 并大量且细致地建模(包括 JNI 调用与系统调用)用于补齐模拟执行环境与依赖, 动态辅助原本的静态分析, 在提高效率的同时更加完整和可靠。其次, 我们深入分析了 Native 库的加载过程与 Native 方法注册与调用机制, 补全了被 Jn-saf 所忽略的 Native 库初始化部分等依赖, 完善了分析过程, 能够克服静态分析无法解决的混淆等程序保护手段。此外, 我们的方法能够为跨层分析提供有力支持, 使用的范围更加广泛, 应用更加灵活高效。

表 1 现有跨层分析方法与本文工作对比

Table 1 Comparison of existing cross-layer analysis approaches with the work in this paper

分类	相关工作	JNI 建模	修改源码插桩/重打包	对抗混淆
静态指针分析	native-scanner	×	×	×
	ANDani	×	×	×
静态符号执行	Jn-saf	√	×	×
	JuCify	√	×	×
动态分析方法	TaintArt	×	√	×
	SoProtector	×	√	√
	ConDySTA	×	√	√
本文方法	JNativeEmu	√	×	√

3 方法设计

我们首先研究了 Android 应用中 Native 库加载与 Native 方法注册与调用机制, 基于对 Native 层运行机制的理解, 我们提出了通过模拟执行 Native 库加载过程来解析动态注册与静态注册的方法。

3.1 Native 方法调用

Native 方法首先需要将 Native 库中 Native 函数注册到对应的 Java 层类, 在注册完成后 Java 层可以调用相应的 Native 方法。

在 Java 层的 Native 方法与 Native 层中对应实现的 Native 函数进行注册绑定与调用的过程中, 需要使用到一个重要的数据结构 ArtMethod。每个 Java 方法在 ART 虚拟机中都对应有一个 ArtMethod 结构体, ArtMethod 记录了这个 Java 方法的信息, 包括所属类、方法标志、代码执行入口地址等。其中 ArtMethod 结构体 PtrSizedFields 的 data_字段会用来作为 Java 层 Native 方法指向 Native 层 Native 函数的代码入口。

当 Android 的 Java 层调用一个 Native 方法时, 如图 2 所示, 会调用 ArtMethod::Invoke 函数。该函数会进一步调用 art_quick_invoke_stub, 处理调用方法的参数, 并通过汇编代码跳板 art_quick_invoke_stub_internal 调用 ArtMethod 结构中的代码入口字段 entry_point_from_quick_compiled_code_, 根据不同情况, 该字段指向不同地址:

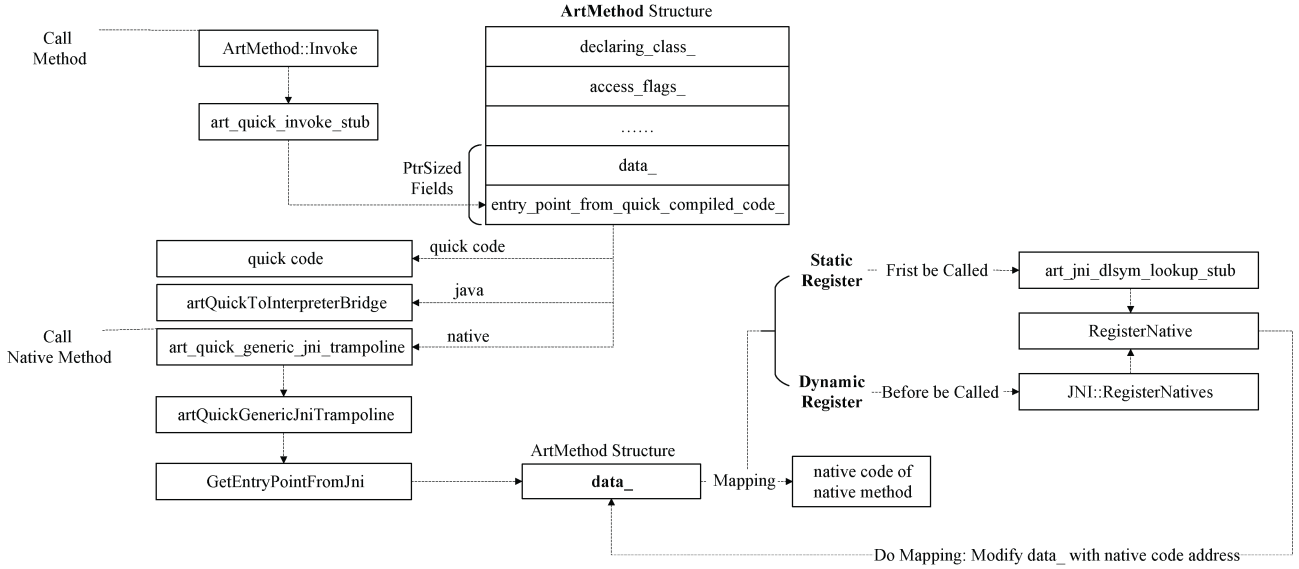


图 2　Native 方法调用过程

Figure 2　The process of Native method invocation

1. 如果方法已经存在相应的 quick code, 则指向 quick code 起始地址;

2. 如果一个 Java 方法不存在 quick code, 该字段指向 artQuickToInterpreterBridge 函数, 切换到 Interpreter 模式解释执行 Java 代码;

3. 如果一个 Native 方法不存在 quick code, 该字段指向 art_quick_generic_jni_trampoline 函数的地址, 进一步执行 Native 方法调用。

处理 Native 方法调用的跳板 art_quick_generic_jni_trampoline 根据系统架构以及指令集有不同的汇编实现, 会进一步调用 artQuickGenericJniTrampoline 获取 Native 函数的入口地址。具体地, 通过 (ArtMethod*) called -> GetEntryPointFromJni 调用, 获取 Native 方法在 Java 层虚拟机中对应 ArtMethod 结构体_data 字段所指向的 Native code 地址。

而 ArtMethod 结构体的 Native 代码入口字段 _data 的设置, 由 Native 方法注册完成。动态注册在 Native 方法调用前完成, 通过 JNI 函数设置_data 字段指向所关联匹配的 Native 函数入口地址。静态注册则是在 Native 方法第一次被调用时完成, 注册前 _data 字段指向 art_jni_dlsym_lookup_stub, 该函数在已经加载的 Native 库中按命名规则查找函数符号, 修改 ArtMethod 结构体_data 字段为 Native 函数地址; 静态注册完成后再次调用该 Native 方法时, 跳板汇编代码会直接跳转到 Native 函数入口执行程序。

3.2　解析动态注册的 Native 方法

动态注册方法通过调用 JNI(Java Native Interface) 的 RegisterNatives 函数实现, 其注册时机一般是在 Native 库加载时。因此 Native 库加载是动态注册的前提。

3.2.1　Native 库加载机制

Android 应用通过调用 loadLibrary 函数加载 Native 层的动态链接库, 其实质是调用 dlopen 进行加载。另外, 如果目标库还依赖其他共享库, dlopen 会自动递归加载这些依赖共享库。

动态链接库存在可选的特殊的节.init, 该节存放了函数指针_init。_init 中的代码在 dlopen 加载动态链接库时会被自动执行, 主要用于初始化动态链接库的一些全局变量和状态。此外, 动态链接库中还存在可选的.init_array 节, 该节存放初始化函数指针数组, 在.init 执行之后, 依次执行该节中初始化函数。若 Android 应用的动态链接库还存在 JNI_OnLoad 函数, Android 系统会调用该函数完成 Android 应用 Native 库的初始化, 往往包括实现 Java 层方法与 Native 层函数的动态注册绑定。

Java 层加载 Native 库的具体过程如图 3 所示。在 loadLibrary 的具体实现中, 首先会根据传入的相对路径或绝对路径, 判断 library 的加载状态, 若已经加载则返回并提示, 未加载则查找动态链接库及其依赖库信息。它调用 dlopen 打开动态链接库, 在这个过程中会依次执行动态链接库的.init 与.init_array 的初始化函数。而后会调用 dlsym 获取 JNI_OnLoad 函数符号, 若获取失败表明不存在 JNI_OnLoad 回调函数; 若存在则执行动态链接库的 JNI_OnLoad 回调函数, 并传递 JavaVM 指针给 Native 层。在 JNI_OnLoad 函数中, Native 库会通过 JNI 机制进行与 Java 层的交互。

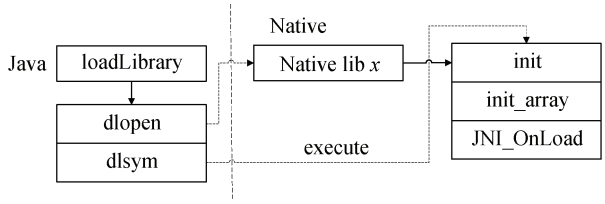


图 3　Native 库加载过程
Figure 3　Native library loading process

3.2.2　解析 RegisterNatives 动态注册

Native 方法动态注册主要体现在 Native 库加载的 JNI_OnLoad 函数中, 通过 JNI 调用 RegisterNatives 实现。RegisterNatives 在注册 Native 方法时会调用 Android 系统框架层中的 RegisterNative 函数注册单个 Native 方法, 沿 Java Class 继承关系向上查找并传入方法名和签名对应的 ArtMethod 对象, 通过函数 SetEntryPointFromJNI 设置 ArtMethod 结构体中的 data_字段, 修改为 Native 方法实际执行的入口地址。

因此, 解析在 Native 库加载过程中动态注册 Native 方法, 即解析 RegisterNatives(env, clazz, methods, nMethods)调用。该调用的函数指针由 JNI 存储, 从图 4 应用 com.pingan.paces.ccms 的 Native 库 libsqlcipher.so 动态注册实例中, 可以直观地看到调用 RegisterNatives 的过程: 首先是通过指向 JNI 结构体的 JNIEnv 指针(寄存器 R4)找到其中 functions 函数指针列表相应偏移 0x35C 所存储的 RegisterNatives 函数地址, 而后传入 JNIEnv 指针 env、Java 类对象 cls(寄存器 R5)、动态注册方法信息 JNINativeMethods (off_210FF0)、注册方法数量 nMethods(0xE)作为参数进行调用。其中, 动态注册 Native 方法最关键的信息是参数 JNINative-Methods, 该参数包含了 Native 方法的方法名、方法签名以及绑定的 Native 函数指针(Native code 地址信息), 如图 4 off_210FF0 所示。因此, 捕捉到 RegisterNatives 调用并解析 JNINativeMethods 参数, 就可以获取动态注册的 Native 方法及其绑定关系。

```
1  jint JNI_OnLoad(JavaVM *vm, void *reserved)
2  {
3      «
4      env->functions->RegisterNatives(env, cls, (const JNINativeMethod *)&off_210FF0, 14);
5      «
6  }
```

```
1  LDR.W    R3, [R4]
2  MOV      R0, R4
3  LDR      R2, =(off_210FF0 - 0xABB92)
4  MOV      R1, R5
5  LDR.W    R6, [R3,#0x35C]
6  ADD      R2, PC  ; off_210FF0
7  MOVS     R3, #0xE
8  MOV      R12, R6
9  POP.W    {R4-R6,LR}
10 BX       R12
```

```
off_210FF0
    DCD aEnablesqlprofi
            ; "enableSqlProfiling"
    DCD aLjavaLangStrin
            ; "(Ljava/lang/String;)V"
    DCD sub_ABAB4+1
    DCD aNativeExecsql    ; "native_execSQL"
    DCD aLjavaLangStrin
            ; "(Ljava/lang/String;)V"
    DCD sub_ABFE0+1
```

图 4　RegisterNatives 调用实例
Figure 4　Example of RegisterNatives call

由于 RegisterNatives 调用及其参数并不总是直观可读, 在 Native 库的手工逆向分析探索过程中, 我们发现, 商业 App 的开发者往往具有较高的安全意识, 会为 App 的 Native 库添加程序保护手段, 比如: Native 库加固加密函数的二进制字节码, 或是控制流平坦化混淆, 导致 JNIEnv 指针的传递以及 RegisterNatives 调用被隐藏, 难以捕捉和解析; 还包括, JNI_OnLoad 或其他 Native 函数中使用到的关键字符串等常量被加密混淆, 无法提取动态注册关联绑定信息。例如代码清单 1 中 com.tencent.portfolio 应用 libqmp.so 库所示, 经过人工逆向, 我们发现 JNINativeMethods 参数 off_1D034 中的 Native 方法名与签名信息, 被混淆为非明文字节。

程序保护手段增加了动态注册解析的难度, 阻碍获取 Java 到 Native 层调用的完整性: 函数加固与控制流混淆会隐藏调用 RegisterNatives 函数的控制流, 而数据加密混淆则会使得 RegisterNatives 调用参数等关键信息不再是明文可分析的。

代码清单 1.数据被混淆保护的动态注册实例

```
1  off_1D034       DCD byte_1D090
2                  DCD aOoooiodsdIdkbJ
3  ; "-OOOOIodsd*idkb*Jgo'fq>Iodsd*idkb*Jgo'f"...
4                  DCD sub_3358+1
5                  DCD byte_1D090
6                  DCD aHhhhnhctcNcleM
7  ; "*HHHHNhctc-ncle-M`hgav9Nhctc-ncle-M`hga"...
```

8	DCD sub_33A4+1

如果采用传统的静态分析方法(例如指针分析)分析 Native 库, 难以对抗真实世界 Android 应用 Native 库中的程序保护手段, 无法可靠地解析动态注册调用获取真实的 Native 方法信息。为了对抗程序保护, 解决静态分析难点, 一个自然的想法是, 在正常加载运行过程中, Native 库执行的 JNI 操作所处理的数据都是真实可靠的, 所以动态思路可以帮助解决上述问题。但是, 动态分析方法需要确保 App 加载了目标 Native 库, 触发条件难以满足; 另一方面动态分析方法需要 hook 系统层 registerNatives 函数, 而安全性意识较高的 App 会对 hook 操作进行检测与对抗, 这也会影响动态注册解析的成功率。因此, 我们使用动态符号执行模拟 Native 库加载这种动态辅助静态的方式, 捕获并解析 RegisterNatives 动态注册调用, 能够在避免 root、hook 等检测的同时, 对抗混淆和加密。

根据动态注册的实现机制, 在模拟执行过程中捕获并解析 RegisterNatives 调用中有 2 个隐含的依赖关系需要满足。一是控制依赖, JNativeEmu 能够模拟执行到 Native 库代码中的 RegisterNatives 调用的情况下, 才能对动态注册调用进行捕获; 二是数据依赖, 完成 Native 库对动态注册关键参数数据的处理, 这样在解析 RegisterNatives 时才能够获取完整的动态注册调用明文信息。

(1) 控制依赖

为了满足控制依赖, 模拟执行调用并捕获到 Native 库代码中的 RegisterNatives 函数调用, 需要对控制流中的分支、跳转、函数调用进行处理, 使其走向正常运行时包含 RegisterNatives 调用的代码执行路径。在实例探索中, 我们发现程序控制流中的分支走向往往是由系统调用或 JNI 调用的返回值所决定的, 但由于缺失自身代码实现, 需要外部输入的处理。此外, 动态符号执行能够根据具体输入确定程序运行时的符号约束, 在输入足够具体时, 相当于模拟 Native 库具体执行, 即使经过控制流混淆, 仍能够不受影响正常运行原本的流程。

图 5 展示了动态注册控制依赖的模式。虽然初始化构造函数 Initializer(.init 和.init_array)执行时还未传入 Java 虚拟机环境相关指针, 不会存在 JNI 操作, 但由于其往往涉及不同功能的初始化, 例如线程操作、文件读写、代码和数据解混淆、网络连接等, 会影响到后续 JNI_OnLoad 的执行, 在模拟执行时需要处理 Initializer 可能涉及的系统调用。

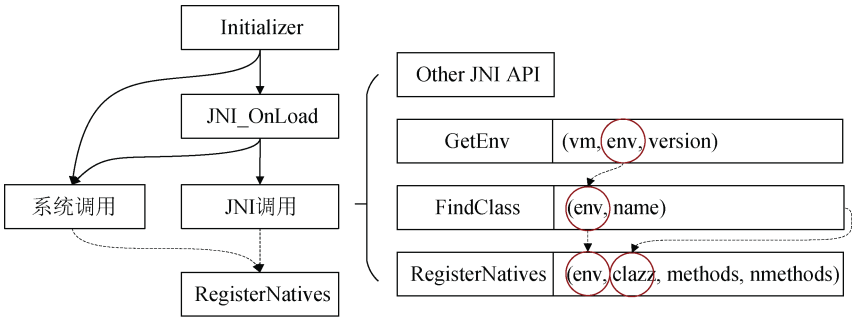


图 5 动态注册控制依赖模式图

Figure 5 Control dependency model diagram for dynamic registration

JNI_OnLoad 则是在系统调用之外, 还会使用 JNI 调用。Java Invoke Interface 与 Java Native Interface 提供了 Java 虚拟机与 Native 代码之间的接口, 接口结构体中定义了大量的 JNI 函数指针, 用于实现 Java 对 Native 的调用和 Native 对 Java 层信息的访问。前置 JNI 调用可能会影响后续执行 JNI 调用的分支条件或调用参数值, 存在控制依赖关系。因此, 需要对 JNI 数据结构进行建模, 保证 JNI 调用的可靠性与 JNI 调用序列的完整性。在模拟调用 JNI_OnLoad 函数时, 将建模得到的 JNI 接口实例化指针, 作为参数传入, 这样在遇到 JNI 调用时能够根据 JNI 数据结构中的偏移识别, 并执行相应的模拟实现与分析处理。

(2) 数据依赖

在 Native 库实现中, 为了保证原本的代码或数据在被使用之前已完成解密解混淆, 能够正常运行, 消解加密与混淆的代码往往编写在 Native 库加载过程的.init、.init_array 或者 JNI_OnLoad 中, 这些先于 Native 库中其他功能性 Native 函数运行的部分, 往往承担了一些必要的相关前置工作。

Native 库自身的解混淆, 可能发生在 Initializer (.init 与.init_array)初始化函数中, 也可能在 JNI_OnLoad 中动态注册之前解混淆, 如图 6 所示。为了满足数据依赖, 需要模拟加载 Native 库的过程, 顺序执行.init、.init_array、JNI_OnLoad 的过程, 保证解混

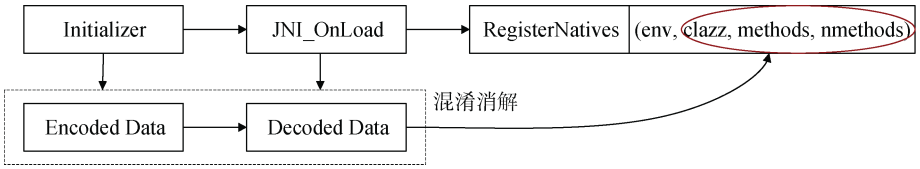


图 6　动态注册数据依赖模式图

Figure 6　Data dependency model diagram for dynamic registration

淆代码得以在使用混淆关键量之前被执行。经过解混淆的处理, 能够获得动态注册中被混淆加密的关键字符串的明文信息。

3.3　解析静态注册的 Native 方法

静态注册方法的注册时机是在第一次调用该 Native 方法时。第一次调用该 Native 方法时, Java 虚拟机会通过 artFindNativeMethod 函数, 调用 FindNativeMethod 搜索到的对应 Native 函数实现。方法的注册关联过程中, 首先根据 JNI 函数命名规范[21], 获取该方法对应的长名称与短名称, 在已经加载的动态链接库中查找与 Native 方法长名称或短名称匹配的函数符号, 获取函数地址。进一步地, 会调用 Android 系统框架层中的 RegisterNative 函数注册该 Native 方法, 与动态注册步骤类似, 查找到对应 ArtMethod 结构体后, 通过 SetEntryPointFromJNI 设置 Native 函数入口地址。

算法 1.静态注册 Native 方法识别算法.

输入: Native 库函数符号表 *symbolTable*
输出: 静态注册的 Native 方法信息列表 *SList*

```
SList = ø
FOR all symbol ∈ symbolTable do
    IF symbol[:5] "Java_" AND isJNIName(symbol) do
        sig = None
        IF "__" in symbol do
            sig, shortSymbol = ParseMethodSig(symbol)
            symbol = shortSymbol
        END IF
        clsName, methodName = ParseMName(symbol)
        SList.Put({clsName, methodName, sig})
    END IF
END FOR
```

根据 Native 方法静态注册机制, 解析同样根据 JNI 函数命名规范, 扫描待分析 Native 库符号表 *symbolTable* 中的 Native 函数, 如果函数符号名 *symbol* 前缀为“Java_”, 就继续判断其是否符合 JNI 函数名命规范, 符合则解析 Native 方法所在类名 *clsName*、方法名 *methodName* 以及长名称可能存在的方法签名 sig, 记录在静态注册的 Native 方法信息列表 *SList* 中, 具体过程由算法 1 所示。

4　JNativeEmu 实现

为了分析 Android 应用中 Java 与 Native code 的交互, 我们构建了 JNativeEmu 工具, 从 Native 层开始对 Native 库的二进制文件进行分析。

4.1　预处理 Android 应用

JNativeEmu 中输入的文件格式支持 APK 和 XAPK 两种类型。XAPK 文件与 APK 同样是一种 Android 应用安装包格式, 用于解决应用安装包大小限制, 可以拆分出 APK 文件进行分析。

Android 应用通过 IDE 工具将 Java 或 Kotlin 代码编译打包获得了 APK 文件, 在 APK 文件中包含了 Java 层编译的 dex 字节码文件与 Native 层的动态链接库 ELF 可执行文件。其中, dex 字节码由 Java 虚拟机(DVM 或 ART)解释和执行, Native 层代码借助 JNI 运行。由于 dex 字节码是与平台架构无关的, 当访问硬件和操作系统的底层时(例如访问文件系统、网络操作等), 就需要使用 Native 代码实现。而为了能够让移动应用运行在不同的操作系统和硬件的移动设备上, Native 层编译得到的动态链接库有多种架构, 如: armv5(armeabi)、armv7(armeabi-v7a)、armv8(armeabi-v8a)、x86、x86_64、mips、mips64, 其中常见的是 arm 架构的 3 种, armeabi、armeabi-v7a 和 armeabi-v8a[22]。

在对 Android 应用进行跨层分析时, 我们首先对 APK 或 XAPK 文件进行解包处理, 扫描其中的库文件目录, 筛选出 ELF 文件作为待分析的 Native 库文件。对于同名不同架构的 Native 库, 我们优先选取 arm32 位架构的 Native 库进行分析。

在得到 Native 库动态链接库文件后, 通过 Android 逆向分析工具 Androguard[23]分析获取应用 Java 层中 Native 方法信息, 以供后续关联分析。

4.2　JNativeEmu 模拟 Native 库加载过程

JNativeEmu 使用 unicorn 支持的 angr 进行动态

符号执行, 模拟 Native 库的加载过程, 获取 Native 层动态注册与JNI调用信息, 具体工作流程框架如图 7 所示, 主要包括 Native 库初始化处理、JNI 建模与底层调用模拟两部分。

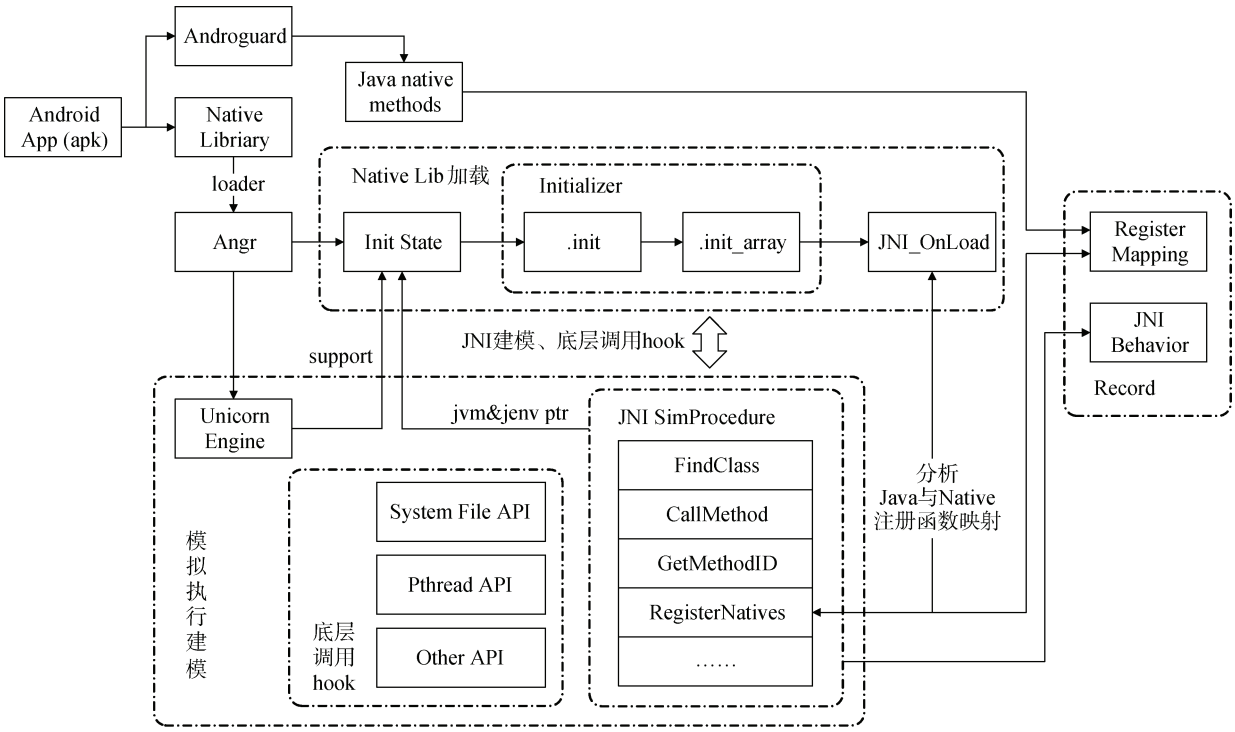


图 7 JNativeEmu 模拟执行工作流程框架

Figure 7 Workflow framework for JNativeEmu simulation execution

(1) **Native 库初始化处理**。通过 unicorn 引擎支持的符号执行框架 angr, 动态符号执行模拟 Native 动态链接库加载初始化部分。

使用 unicorn 引擎的 angr 符号执行是一种动态符号执行方法, 结合了具体执行与符号执行, 根据是否存在符号约束表达切换具体执行与符号执行引擎。在模拟执行时, angr 跟踪和处理程序中的符号变量并维护符号约束, 同时 unicorn 引擎会动态解析执行指令处理程序行为。unicorn 引擎不需要进行中间层代码的转换和解释, 因此能够更快地模拟执行, 并获取更精确的结果。这种结合具体执行的方法, 相对于只依靠符号执行引擎的静态符号执行更加灵活, 可以处理更加复杂的指令序列, 并且通过 unicorn 的模拟执行能够简化符号表达式, 减少不必要的分支, 降低了路径爆炸的可能性; 在遇到难以确定具体内容的符号时, 例如 Java 对象信息等, 仍然可以依靠 angr 的符号约束执行模拟过程。

为了完整地模拟 Native 库的加载过程, 我们构建了模拟执行的起始地址, 用于动态链接库的初始化处理。在动态链接库的加载过程中, 动态链接器会处理.dynamic 段中记录的.init 段与.init_array 段信息, 而后依次执行段中的构造函数。相应地, JNativeEmu 在模拟执行 JNI_OnLoad 之前, 使用 angr 的 SimProcedure 功能 hook 起始地址为初始化部分 Initializer。Initializer 通过解析 ELF 文件格式 dynamic 段获取.init 初始化函数和.init_array 函数数组, 并依次模拟执行, 完成动态链接库的初始化部分。Initializer 初始化处理完成后, JNativeEmu 跳转到 JNI_OnLoad 函数进行后续 Native 库加载的模拟。

(2) **JNI 建模**。模拟 Java Native Interface 与 Java Invoke Interface 中各 API 调用, 启发式构造所需的 Java 对象信息; 底层调用模拟, 为保证模拟执行的顺利进行, 需要模拟 Native 层底层系统调用, 包括 pthread、文件读写、Android 系统信息获取等。

在 JNI_OnLoad 函数中, 涉及到 Java 层和 Native 层之间的交互操作, 这些操作是通过 JNI 调用实现的。为了对 JNI 调用进行模拟, 记录关键 API 的调用和调用时参数等信息, 我们使用 JNativeEmu 对 JNI 进行建模。

建模得到的JNI接口实例化指针, 存储于模拟执行 Native 库加载过程的起始状态中, 作为参数在执行 JNI_OnLoad 调用时传入。JNativeEmu 根据偏移识别 JNI 函数调用, 通过扩展 angr 的 SimProcedure 功能模块为 JNISimProcedure, hook 建模的 JNI 数据结

构中的函数指针, 模拟实现对应 JNI 函数的回调处理。针对不同JNI调用, JNativeEmu有不同的模拟策略。

GetEnv: 用于获取 Java 虚拟机当前线程的 JNIEnv 指针。识别与捕获 JNI 函数调用, 依赖于该指针在程序中的流动与使用。模拟 GetEnv 的 JNISimProcedrue 返回当前符号执行状态中存储的 JNIEnv 指针, 即 JNI 建模接口实例化指针。

FindClass: 获取 Java 类对象的引用, 作为 Native层与Java类的关联, 是后续许多JNI交互操作的前提。因此我们构造并返回 JClass 结构, 记录类名等信息, 由符号执行进行处理。

RegisterNatives: 用于注册 Native 函数到 Java 层中 Native 方法, 是获取动态注册 Native 方法与 Native函数映射关系, 获取Native层程序分析入口点的关键调用。根据动态注册原理, RegisterNatives 的 JNISimProcedure 按照其函数原型解析模拟执行时具体的函数参数, 获取RegisterNatives具体调用时信息, 解析得到动态注册的 Native 方法的方法名、方法签名与对应 Native 函数地址。

其他函数: 根据功能与返回值类型进行具体实现与返回值构造。对于返回值为基本数据类型的 JNI 函数, 根据具体函数功能返回构造的符号变量; 对于返回值为 java/lang/String 类型的, 构造字符串并返回为其分配的内存地址; 对于返回值为 Java 对象包括自定义类对象的, 构造 JObject 结构, 记录 Java 对象类型信息, 依靠 angr 的符号表达进行模拟执行。

5 实验评估与分析

为了研究真实应用市场中Android应用的Native跨层行为, 理解 Native 库跨层执行的使用动机与使用情况, 我们对来自 Google Play 的 5 个类别(金融、游戏、购物、社交、视频), 每类爬取了下载量最多的 100 个 APP 的 APK 或 XAPK 文件, 并随机抽取了 10 个样本。

我们构建的工具 JNativeEmu 基于 angr(版本为 9.2.19), 并集成了 unicorn(版本为 1.0.2rc4)。以下实验数据均是运行于 Ubuntu20.04 子系统中得到的。

本节中使用 JNativeEmu 在上述环境与数据集上进行了实验评估与分析, 探索了以下几个问题:

RQ1. **Native code 在 Android 应用中的使用情况如何?** Native 库是否被广泛使用? Native 方法不同注册方式的使用情况如何?

RQ2. **JNativeEmu 模拟执行运行情况如何?** 模拟执行能否成功完成 Native 库加载过程? 模拟并捕捉到了多少 JNI 调用与系统调用?

RQ3. **动态注册 Native 方法解析效果如何?** 与 Jn-saf 对比是否有提升? Initializer 依赖部分对模拟执行与解析有什么影响?

RQ4. **使用动态注册的 Native 方法内部有哪些值得关注的行为?**

5.1 Native code 使用情况

首先需要回答的问题是 Native code 在 Android 应用中的使用情况是怎样的。为此我们使用 JNativeEmu探索Native库在Android App中的使用数量, 关注其中是否存在 JNI_OnLoad 函数以及静态或动态注册的 Native 方法, 结果如表 2 所示。

so 文件/Native 库。在 50 个 App 中, 仅有 Shopping 类别的 LimeRoad(包名: com.shopping.limeroad)App 不存在 Native 库, 该应用中使用了 ReactNative 框架, ReactNative 框架能够提供丰富的 JavaScript 接口和内置功能, 笔者判断应用可能没有底层的操作系统和硬件访问的需求, 依靠 Java 的跨平台性以及丰富的库和框架就能够满足该应用程序的功能要求。除此之外, 其他应用中均存在 Native 库, 每个 App 平均使用 29 个 Native 库。各个类别均使用了较多数量的 Native 库, 其中 Video 类别使用 Native 库的总量最多, 大多数是多媒体处理相关 Native库; 除Game类别中Native库相对使用比例较

表 2 Android 应用中的 Native 使用情况

Table 2 Native usage in Android apps

分类	存在 Native 库的 App 数量	使用的 Native 库总数量	存在 JNI_OnLoad 函数的 Native 库数量 (存在 D 的 Native 库数量)	D&S / D Only / S Only / Neither 的 Native 库数量
金融	10	275	120(78)	11 / 67 / 114 / 83
游戏	10	104	62(24)	3 / 21 /27 / 53
购物	9	235	177(104)	10 / 94 / 127 / 4
社交	10	301	169(84)	13 / 71 / 151 / 66
视频	10	394	205(124)	12 / 112 / 164 / 106
总体	49	1309	733(414)	49 / 365 / 583 / 312

(注: D 表示动态注册方法, S 表示静态注册方法)

少以外(平均每个 app 使用 10.4 个 Native 库), 其他类别均有相似的 Native 库数量。经初步统计表明, 平均一个 app 使用 30 个左右的 Native 库。由此可见 Native 库在 Android 应用中的使用已经十分普遍和广泛。

JNI_OnLoad 函数。经研究表明, 约有 60%的 Native 库存在 JNI_OnLoad 函数, 这意味着有超过半数的 Native 库直接接收了来自 Java 层的 JavaVM、JNIEnv 等信息, 并用其进行了自身加载时、使用前的初始化。JNI_OnLoad 的存在, 意味着 Java 层与 Native 层交互存在的可能性。此外, 根据表中数据可以知道, 存在 JNI_OnLoad 函数的 Native 库不一定存在 Native 注册方法, JNI_OnLoad 函数的作用并不局限于动态注册 Native 方法。

Native 注册方法。表 2 结果表明, 超过半数的 Native 库存在 Native 注册方法, 占 Native 库总量的 76.2%, 这意味着超过半数的 Native 库存在 Java 到 Native 的入口, Native 会直接与 Java 层交互。需要注意的是, 一个 Native 库可能既存在静态注册 Native 方法也存在动态注册 Native 方法。在各个类别大多数的 App 中, 静态注册的 Native 方法明显多于动态注册的 Native 方法, 存在静态注册的 Native 库也多于动态注册的 Native 库, 这是因为静态注册实现相对简单, 使用范围更加广泛, 只需要按照相应的命名规则定义函数名, 即可注册 Native 方法提供给 Java 层使用。

5.2 JNativeEmu 运行情况

5.2.1 NativeEmu 模拟执行情况

JNativeEmu 的工作过程首先是获取各个 APP 的 Native 库 so 文件, 而后进行加载过程的模拟执行, 依次执行 Initializer(.init 与.init_array)和 JNI_OnLoad 函数, 在模拟执行过程中对抗混淆等程序保护手段, 最终解析得到注册函数入口等 JNI 交互信息。我们评估了 JNativeEmu 的模拟执行情况, 结果如表 3 所示。

根据表 3 结果, 约有 83.2%的正常 Native 库(文件格式正确)能够正常且成功地模拟执行, 但仍有少量的 Native 库模拟执行失败。经分析表明 Native 库模拟执行失败的原因主要分为以下三类:

一是 JNI 模拟不完全导致模拟执行失败。由于 JNativeEmu 的构建与分析更加关注 Native 层, 缺乏 Java 层信息, 在构造 Java 对象时难以进行完整充分的模拟。对于个别 Native 库, 其控制依赖或者数据依赖中关键信息从 Java 层获取, NativeEmu 无法进行完整模拟。

二是底层系统调用与底层指令无法正确模拟导致运行超时(每个 Native 库 180s)。不完全的或被模拟忽略的底层系统调用, 可能使得 JNativeEmu 一直处于执行该指令的模拟状态中, 导致运行超时。

表 3 JNativeEmu 模拟执行运行情况

Table 3 JNativeEmu simulation execution run case

分类	运行成功	模拟错误	超时	VFP 指令
金融	103	11	4	2
游戏	45	8	7	2
购物	165	8	4	0
社交	135	25	7	2
视频	162	21	14	8
总体	610	73	36	14

三是在执行底层浮点操作(VFP, Vector Floating Point, 向量浮点运算)指令时, 由于所使用的动态符号执行工具 angr 实现的局限性, 无法正确模拟该指令, 导致模拟执行失败。

5.2.2 JNI 调用与系统调用使用情况

为了更完备地模拟 Native 库加载过程, JNativeEmu 实现了 93 个 JNI 调用和 117 个其他系统调用(其中有 54 个线程系统调用)的模拟。

我们统计了 JNI 调用与系统调用的使用情况, 如表 4 所示, 可了解到这两类调用在 Native 库的初始化与加载过程中, 被大量地调用, 平均每个存在 JNI_OnLoad(存在 JNI_OnLoad 的 Native 库会被模拟执行加载过程)的 Native 库中运行时调用 26.7 个 JNI 函数, 调用 25.1 个线程函数。

用户为满足工作、生活、学习等场景下的各类需求, 在移动端会使用许多不同功能的 App, 这需要 App 的前端 UI 与后台任务之间能够流畅无感知地分离执行, 因此移动应用拥有多线程的特性, 能够异步执行, 更充分地使用 CPU 资源提高程序的运行效率, 所以在 Native 库中也往往会有大量的线程实现。

表 4 JNI 调用与 Pthread 线程调用使用情况

Table 4 Usage of JNI calls with Pthread thread calls

分类	金融	游戏	购物	社交	视频	总体
JNI 调用	2097	988	2872	4584	9038	19579
Pthread 调用	2961	3989	1686	1993	7756	18385

5.2.3 Initializer 模拟执行情况

在存在 JNI_OnLoad 的 Native 库中, 有一半以上的 Native 库都会编写并使用初始化构造函数, 这些 Initializer 多是进行一些全局变量的初始化, 往往存在线程操作与文件操作。其复杂的实现与系统调用, 使得 Initializer 的运行成功率低于主要是 JNI 操作的

JNI_OnLoad, 有97.3%的JNI_OnLoad模拟执行成功, 而 Initializer 的运行成功率是 72.4%, 如表 5 所示。

表 5 Initializer 模拟执行评估

Table 5 Initializer simulation implementation evaluation

分类	Initializer 运行成功	JNI_OnLoad 运行成功
金融	54	117
游戏	17	60
购物	70	174
社交	59	165
视频	81	197
总体	281	713

5.2.4 JNativeEmu 模拟执行效率与内存消耗

我们对 JNativeEmu 的运行效率与内存消耗进行了评估, 如表 6 所示。

表 6 模拟执行运行时间与内存消耗评估

Table 6 Simulation execution runtime and memory consumption evaluation

分类	Native 库平均运行时间(s)	Native 库平均内存消耗(MB)
金融	8.48	116.68
游戏	23.99	592.41
购物	8.35	32.44
社交	12.87	68.45
视频	17.93	81.71
总体	13.54	72.58

JNativeEmu 运行 Native 平均所需用时 13.54 s, 平均内存消耗为 72.58 MB。抽取金融类别应用与 Jn-saf 对比执行速率, 有 8 个 Native 库出现执行超时情况, 40 个 Native 库出现崩溃错误, 平均执行时间为 10.15 s, 长于 JNativeEmu 在金融类别的平均执行时间 8.48 s。

其中游戏类别应用的 Native 库平均执行时间最长, 平均内存消耗最大, 这是由于很多游戏类应用会使用游戏引擎, 比如游戏引擎 Unity3D 会使用到 libunity.so 和 libil2cpp.so 这两个体积较大的 Native 库, 因此分析时间较长, 内存消耗较大。

5.3 动态注册 Native 方法解析评估

对于解析动态注册 Native 方法, JuCify 构造了 16 个 apk 作为样本, 涵盖了多种 Java 到 Native 层调用的实现, 以评估其 Native 层分析模块 NativeDiscloser 的跨层调用提取情况。在 JuCify 构建的样本数据集上, NativeDiscloser 在 Native 方法解析方面达到了 100%的精确率与 95.59%的召回率, 而本文工作 JNativeEmu 能够达到 100%的精确率与 100%的召回率。然而 JuCify 不能够有效地运行于本文的真实世界 apk 数据集, 存在大量崩溃与超时。

而通过模拟执行 Native 库的加载过程, JNativeEmu 能够较为准确高效地解得动态注册的 Native 方法, 在 50 个真实世界 Android 应用上分析的结果如表 7 所示。

表 7 JNativeEmu 对比 Jn-saf 动态注册方法解析数量

Table 7 JNativeEmu vs. Jn-saf on number of dynamically registered Native method resolutions

分类	JNativeEmu	Jn-saf
金融	1465	687
游戏	515	171
购物	2970	1194
社交	2896	584
视频	5309	1426
总体	13155	4062

Video 类别使用动态注册 Native 方法最多, Game 类别使用动态注册 Native 方法最少, 其余类别平均每个 App 使用 252.8 个动态注册 Native 方法, Video、Shopping、Social 类应用由于功能丰富, 所以相应 Native 库、Native 注册方法数量也明显较多。

Native 方法的动态注册, 一般情况下是通过在 JNI_OnLoad 中调用 JNI 中的 RegisterNatives 函数实现。对于 RegisterNatives 调用的解析, JNativeEmu 能够通过模拟执行满足绝大多数的控制依赖与数据依赖需求, 成功解析动态注册 Native 方法。从表中结果可以看到, JNativeEmu 解析得到平均每个 Android App 使用上百个动态注册的 Native 方法, 远远超过 Jn-saf 所解析的动态注册 Native 方法。

5.3.1 对比 Jn-saf

我们使用 JNativeEmu 与 Jn-saf 进行对比, 对金融类别的动态注册 Native 方法解析结果进行了具体分析, 归纳总结并抽取了其中几种典型的 Native 库, 结果如表 7 所示。

相较于 Jn-saf, JNativeEmu 加入了对 64 位以及对 Thumb2 指令集的支持, 采用 unicorn 支持的动态符号执行方法, 提供更多的具体输入, 包括但不限于 JNI 建模结构体指针、系统调用, 使得模拟过程更贴合真实运行时的程序状态, 兼具运行速度与准确性, 能够减少符号执行路径, 降低因为路径爆炸而无法正常模拟执行与解析的可能性。

具体地, 如表 8 所示。一般模式中 libijkplayer.so 的动态注册解析, Jn-saf 由于 Thumb2 指令集导

致 angr 反汇编切片错误, 符号执行失败; 在对 libBugly.so 进行分析时, Jn-saf 则是由于模拟不全导致路径爆炸解析失败; libpaic_asr_new.so 中 Jn-saf 没有处理 qmemcpy 使得动态注册信息缺失; 而对于 libgifimage.so, Jn-saf 则是由于记录动态注册信息基于方法名而并非具有唯一性的 Native 函数地址, 去除了同名但实际不同的匹配。

表 8 JNativeEmu 对比 Jn-saf: 典型 Native 库动态注册 Native 方法解析数量

Table 8 JNativeEmu vs. Jn-saf on number of dynamically registered Native methods resolution for typical Native libraries

分类	Native 库	JNativeEmu	Jn-saf
一般模式 (无程序保护)	libijkplayer.so	38	0
	libBugly.so	10	0
	libpaic_asr_new.so	6	0
	libgifimage.so	23	19
复杂实现 (JNI 调用、系统调用)	libfbjni.so	3	0
	libhce-engine.so	70	0
控制流混淆	libSparta.so	2	0
	libwcdb.so	107	0
数据混淆	libyaqstub_trade.so	11	11
控制流混淆&数据混淆	libnetsecsdk-4.1.2.so	3	0
	libqmp.so	4	0

复杂实现中的代表 Native 库, libfbjni.so 是 Facebook 所提供, 具有鲜明的线程调用和跨库调用的编程模式, 我们在第六节案例分析中对这类 Native 库进行了详细分析; libhce-engine.so 库中则是存在反调试与模拟器检测行为, 它通过 popen 执行 ps 命令判断是否存在可信安全机制相关进程, 如果不存在就通过 ptrace 附加到自身进程使得其他进程无法附加进行调试, 并通过路径检测是否存在 qemu 模拟器, 这些环境监测与对抗行为不会影响 JNativeEmu 的模拟执行, 但 Jn-saf 会因为 Native 库的复杂实现而符号执行失败。

此外, 一些安全意识较强的 Native 库会对关键数据(例如 RegisterNatives 的参数)进行混淆加密处理, Native 库自解密的过程往往会在 Native 库的初始化函数中完成。Jn-saf 的符号执行并非具体执行, 没有考虑 Initializer 对后续 JNI_OnLoad 可能造成的影响, 无法获取解密后的关键数据, 导致传入 RegisterNatives 或其他 JNI 调用的参数并非明文, 因而遗漏了其中动态注册的 Native 方法信息以及其他 JNI 调用参数信息; 而 JNativeEmu 基于对 Native 库加载过程的理解, 使用 Initializer 完整地模拟执行了初始化构造函数, 能够消解在 Native 库自身进行运行初始化解密处理的数据混淆。

表 8 中选取了控制流混淆、数据混淆以及二者皆有的 Native 库。控制流混淆中, libSparta.so 使用了控制流平坦化, Jn-saf 对于这类案例出现了符号执行超时的情况; 而 libwcdb.so 则是在 init 部分初始化了 JNI_OnLoad 中所调用的函数列表, 由于 Jn-saf 忽略了 init 部分, 并未真实调用到原本的实现就结束了符号执行。在数据混淆案例 libyaqstub_trade.so 中, JNativeEmu 因为 Native 库中调用了 Java 层方法处理控制流分支判断值, 缺失了 3 条匹配, 而 Jn-saf 则是因为数据混淆, 未能成功解得这 3 条动态注册的明文信息。在两类混淆都有的 Native 库中, Jn-saf 也表现不佳, 无法处理 libnetsecsdk-4.1.2.so 中的动态赋值与异或混淆数据, 在 libqmp.so 中也因为控制流平坦化以及模拟不完整导致路径爆炸符号执行超时; JNativeEmu 则可以处理这些情况, 其中 libqmp.so 的动态注册数据在初始化部分进行了混淆解密, JNativeEmu 模拟执行 Initializer 后能够成功解析动态注册得到明文信息, 若不考虑初始化部分, 则仅得到乱码。

在所有动态注册解析案例中不存在 Jn-saf 能够解析而 JNativeEmu 无法解析的情况, Jn-saf 无法处理复杂实现、混淆加密的 Native 库, 运行实际效果远不如模拟执行。

5.3.2 Initializer 的影响

为评估 Initializer 对 Native 库加载过程模拟执行的控制依赖与数据依赖的影响, 我们统计了数据集中存在 Initializer 初始化函数、存在数据混淆、由 Initializer 解混淆的 Native 库数量, 结果如表 9 所示。

存在动态注册 Native 方法的 Native 库中, 有 52.9%的 Native 库存在初始化操作, 这些初始化操作可能会在控制流或数据流方面影响到后续的动态注册过程。我们发现, RegisterNatives 动态注册函数参数并非明文数据的数据混淆样例约占所有存在动态注册 Native 库的 31.9%, 并且在这些混淆样例中有 11.8%是在 Initializer 部分进行解该数据混淆的处理。

5.4 动态注册 Native 方法调用行为

由于跨层分析所具有的对抗分析的属性, 我们根据人工逆向分析探索以及 Ruggia A 等人的工作[10], 进一步探索 Native 方法调用, 挖掘了在 Native 库中值得关注的代码行为, 如表 10 所示。

在 JNativeEmu 得到的动态注册 Native 方法中, 我们关注并分出表中八类调用, 分别代表了不同的

表 9 Initializer 对动态注册解析的影响评估

Table 9 Evaluation of Initializer's Impact on dynamic registration resolution

分类	JN(DN)	IN(DN)	DON	IDON
金融	120(78)	68(46)	17	2
游戏	62(24)	33(12)	1	0
购物	177(104)	79(43)	23	1
社交	169(84)	89(45)	19	2
视频	205(124)	119(67)	8	3
总体	733(414)	388(213)	68	8

(注: JN 表示存在 JNI_OnLoad 函数的 Native 库数量, DN 表示存在动态注册的 Native 库数量, IN 表示存在 Initializer 的 Native 库数量, DON 表示动态注册过程中存在数据混淆的 Native 库, IDON 表示 Initializer 影响混淆的 Native 库数量)

代码行为, 包括动态加载、命令执行、文件读写、内存保护等。

商业 App 的 Native 库中会使用动态加载、命令执行、文件读写、内存保护、进程管理相关调用, 来保证 App 的安全性, 例如通过命令执行获取相关进程信息用于反调试, 增加逆向分析难度等。网络操作、设备属性与日志输出有可能是隐私数据在 Android 应用中流动的 Source 与 Sink 点。

6 案例分析

6.1 Native 库间调用

通过平安口袋安卓 App(包名: com.pingan.paces.ccms), 我们注意到了 libreactnative- jni.so、libfbjni.so 是由 Facebook 提供用于实现 Android 应用与 Facebook 平台之间的交互和通信的 Native 库, 例如 libreactnativejni.so 是用于实现使用 Facebook 提供的 ReactNative 框架的应用与平台交互。这些由 Facebook 提供的第三方 Native 库, 有统一的开发风格与代码模式, 同样也经常出现在许多其他应用中。在该 Native 库中, JNI_OnLoad 的主要功能与代码实现内容写在了函数指针列表 off_B2830 中, 通过 libfbjni.so 中的函数 facebook::jni::initialize 调用这个列表中的函数而不是直接调用; 根据 facebook::jni::initialize 的实现模式, 最终调用到了编写在 libreactnativejni.so 中的 sub_7B60C 函数, 在该函数中

表 10 Native 库中动态注册 Native 方法调用行为

Table 10 Calling behaviour of dynamically registered Native methods in Native library

分类	Native 库	实例
动态加载	dlopen(436), dlsym(442)	dlopen("libopenCL.so"); dlsym("clEnqueueReadBufferRect")
命令执行	execve(110), system(955), popen(39)…	popen("cat /proc/net/tcp |grep:5D8A", "r");
文件读写	open(1326)…	open("/proc/%d/cmdline", 0);
内存保护	mmap(643), mprotect(16)	mmap(0, 0x1000u, 3, 34, -1, 0);
进程管理	kill(19), ptrace, fork(11)…	pid = fork(); kill(pid, 9);
网络操作	socket(88), listen(2), connect(22)…	fd = socket(2, 2, 0);
设备属性	__system_property_get(72), __system_property_find, __system_property_set	_system_property_get)("ro.build.version.sdk",val);
日志输出	__android_log_print(1541), android_set_abort_message(113)	_android_log_print(3,"IJKMEDIA", "ijkmp_shutdown_l()");

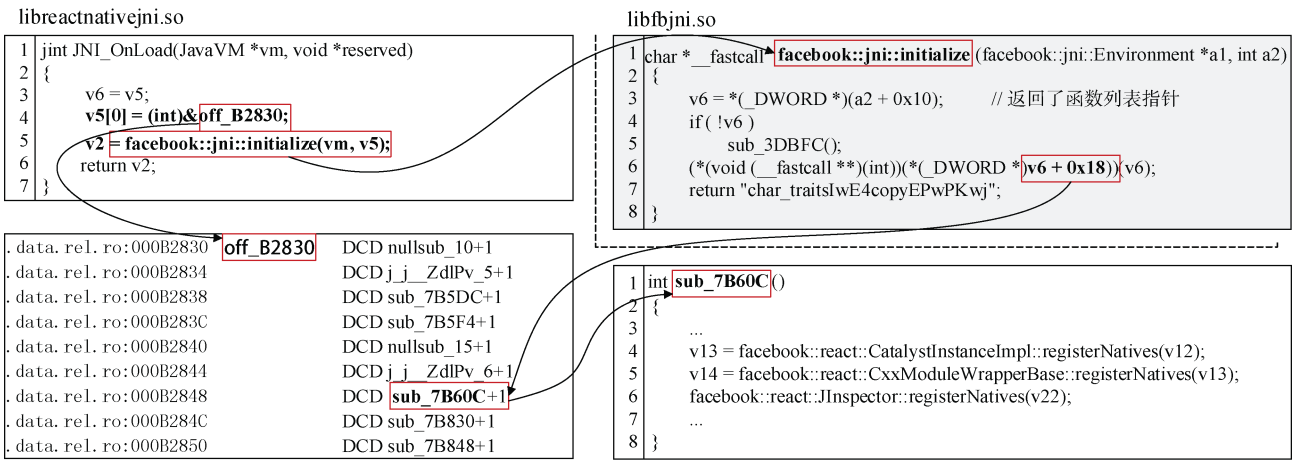


图 8 跨库调用实例示意图

Figure 8 Example of a cross-library call

调用了一系列 JNI API, 包括 RegisterNatives 等。应用市场商业 App 中的 Native 库的实现往往更加系统、耦合与复杂, 这种实现的复杂性使得代码控制流变得更为复杂, 静态分析单个 Native 库难以真正应用到实际。

6.2 Init 预处理数据依赖

在 Android 应用腾讯自选股(包名: com.tencent.portfolio)的 Native 库 libqmp.so 中, 我们发现该 Native 库中的函数大多存在控制流平坦化混淆。在消除控制流平坦化后, 其 JNI_OnLoad 中动态注册 Native 方法的过程如图 9 所示。其中动态注册的核心代码所使用到的关键字符串数据(动态注册关联 Java 类的名称和 JNINativeMethods 动态注册 Native 方法的信息)均被混淆, 在 Native 库的实际加载使用过程中, 会通过 init 调用的初始化构造函数 tencent11532023018238489601 26 进行异或解混淆。JNativeEmu 模拟执行过程中完整地模拟了 Native 库加载过程时的初始化, 执行了解混淆部分, 因此在 JNI 调用时能够拿到满足数据依赖的完整明文信息。

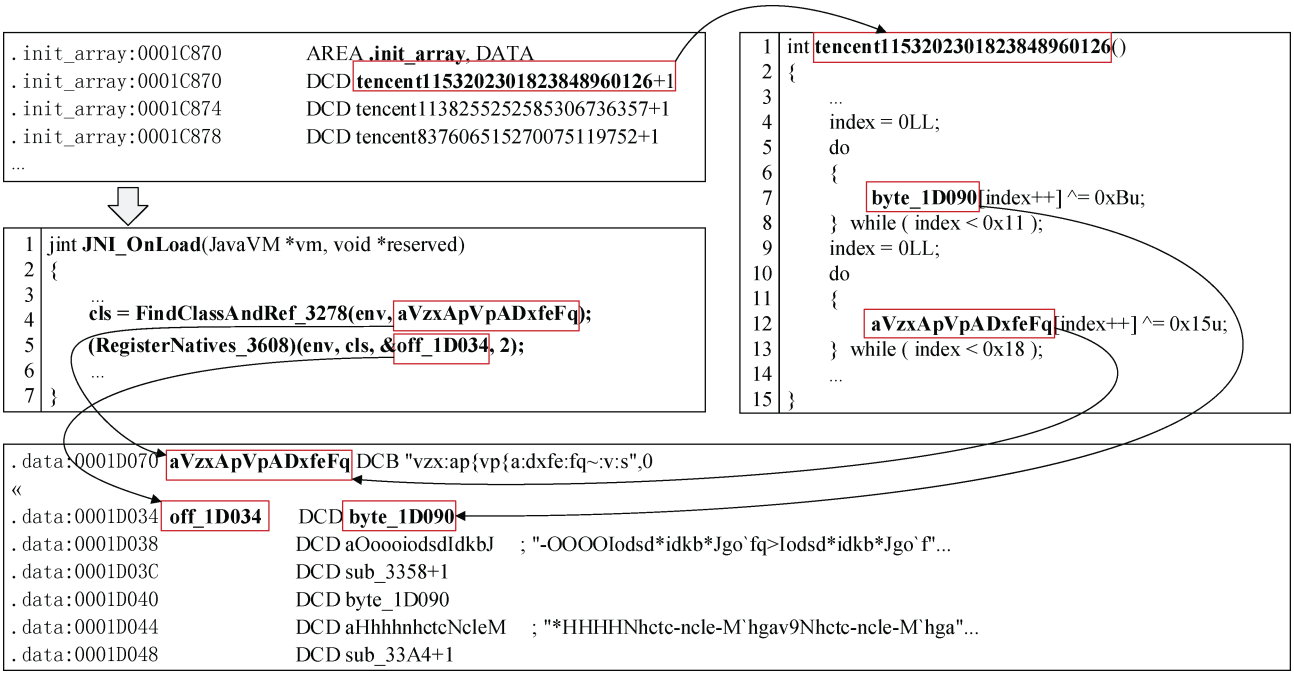


图 9 Init 预处理数据依赖实例示意图

Figure 9 Example of Init preprocessing data dependency

6.3 Init 预处理控制依赖

与此同时, 我们还注意到腾讯自选股 App(包名: com.tencent.portfolio) 中的非第三方数据库框架 libwcdb.so 存在初始化部分影响后续 JNI_OnLoad 调用的控制依赖。

如图 10 所示, .init_array 中初始化函数都是相似地调用函数 sub_11DF0, 该函数动态生成了函数调用列表 funclist 即 dword_E8220[0], 而 dword_E822C[0] 存储了 dword_E8220[0]列表的长度 funclist_size, 函数列表项 funclist_item 的具体函数指针值则是由 sub_11DF0 函数的第二个参数所决定, 为 sub_12900 函数。该函数用于处理与第一个字符串参数“CursorWindow”Java 类相关的 JNI 操作, 例如调用 FindClass 访问相应类对象, 之后进行 GetFieldID 访问类中 Field 和 RegisterNatives 动态注册 Native 方法的 JNI 交互。

该案例中, 初始化部分动态生成函数列表, 在 JNI_OnLoad 中进行间接调用, 而 Jn-saf 等静态分析时无法确定 JNI_OnLoad 中函数调用地址, 不能解决这样的控制依赖。JNativeEmu 则可以通过模拟执行, 准确高效地分析该 Native 库。

7 局限性与未来工作

我们的工作模拟执行了 Native 库加载过程, 提供了较为精确地分析 Native 库跨层交互行为的思路。JNativeEmu 目前仍存在一些不足和有待进一步研究之处。

一是由于我们的方法缺乏 Java 层信息, 对于 Java 对象的构造仍然是粗粒度的启发式方法, 存在影响模拟执行完整性的因素, 探索如何补充更加有效的 Java 对象信息将对 Native 库的分析大有帮助。

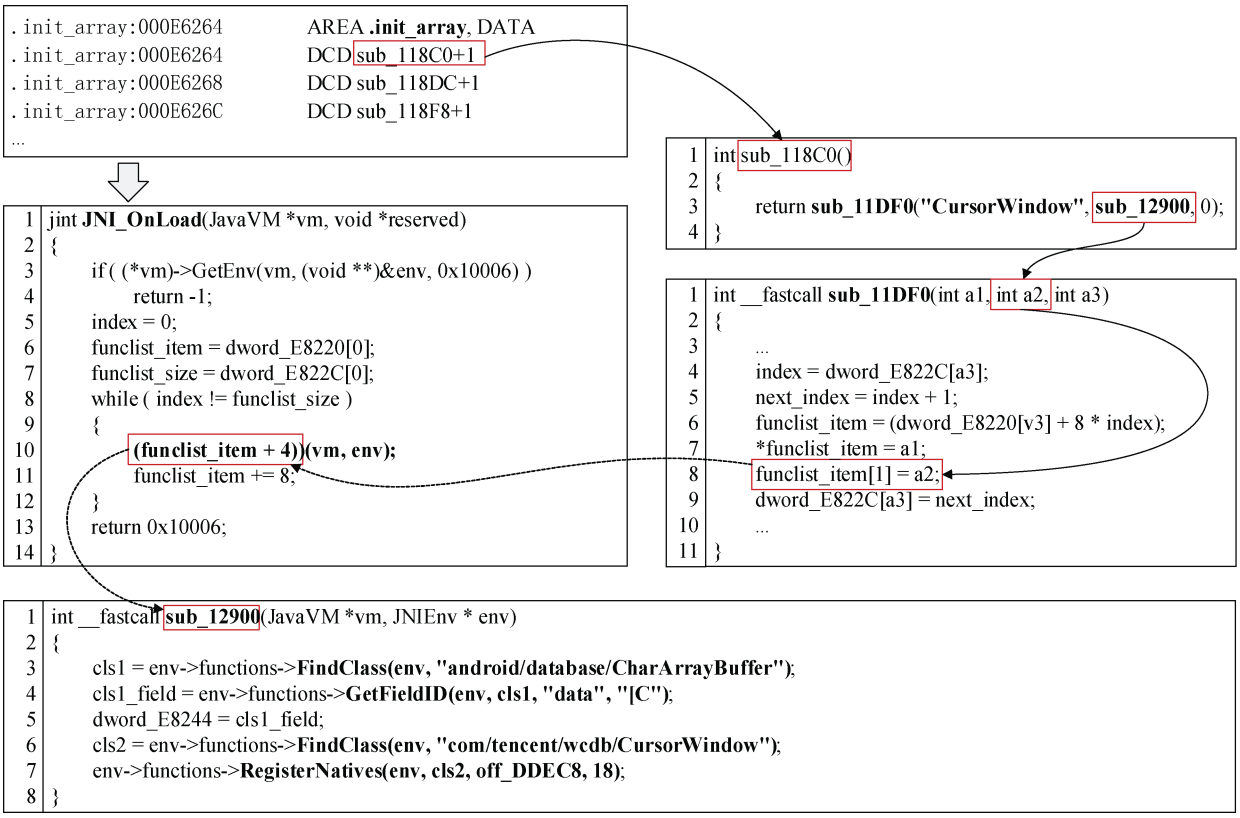


图 10　Init 预处理控制依赖实例示意图

Figure 10　Example of Init preprocessing control dependency

二是 JNativeEmu 的运行比较依赖系统调用等模拟的完整度, 目前没有考虑部分不常见的系统调用以及浮点操作指令, 但实际运行过程中存在这样的样例, 有待后续补充完整。

三是尽管我们已经处理了许多情况, 符号执行工具 angr 本身的不稳定性, 仍可能会导致运行的崩溃, 这使得 JNativeEmu 的适用范围受到限制。此外对于加壳 Native 库, angr 也存在无法正确识别格式进行加载的可能。

8　总结

本文中我们提出了一种基于模拟执行的 Native 库跨层行为分析方法, 为 Android 应用跨层数据流分析及其下游任务提供必要支撑。该方法通过分析 Native 库跨层执行过程, 对抗 Native 库自身的控制流混淆与数据混淆, 可以弥补跨层分析断点。相较于现有先进工作, 我们的方法在对抗混淆提高 Native 库分析完整性的同时, 能够更加高效准确地对 Native 库进行进一步深入分析。

参考文献

[1] CNNIC.中国互联网络发展状况统计报告[R]. 2022.02.25:16-17.

[2] statcounter.[EB/OL].https://gs.statcounter.com/os-market-share/mobile/worldwide.2021.01.

[3] Java Native Interface.[EB/OL].https://docs.oracle.com/javase/8/docs/technotes/guides/jni/.

[4] Arzt S, Rasthofer S, Fritz C, et al. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps[J]. *ACM SIGPLAN Notices*, 2014, 49(6): 259-269.

[5] Li L, Bartel A, Bissyandé T F, et al. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps[C]. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015: 280-291.

[6] plast_lab/native-scanner[EB/OL].https://github.com/plast-lab/native-scanner.

[7] Wang F, Shoshitaishvili Y. Angr - the Next Generation of Binary Analysis[C]. *2017 IEEE Cybersecurity Development* (*SecDev*), 2017: 8-9.

[8] Wei F G, Lin X W, Ou X M, et al. JN-SAF: Precise and Efficient NDK/JNI-Aware Inter-Language Static Analysis Framework for Security Vetting of Android Applications with Native Code[C]. *The 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018: 1137-1150.

[9] Samhi J, Gao J, Daoudi N, et al. JuCify: A Step towards Android Code Unification for Enhanced Static Analysis[C]. *2022 IEEE/ACM 44th International Conference on Software Engineering*, 2022: 1232-1244.

[10] Ruggia A, Possemato A, Dambra S, et al. The Dark Side of Native

Code on Android[J]. *ACM Transactions on Privacy and Security*, 2025, 28(2): 1-33.

[11] Samhi J, Bartel A, Bissyandé T F, et al. RAICC: Revealing Atypical Inter-Component Communication in Android Apps[C]. *2021 IEEE/ACM 43rd International Conference on Software Engineering*, 2021: 1398-1409.

[12] Bartel A, Klein J, Le Traon Y, et al. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot[C]. *The ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, 2012: 27-38.

[13] Perkins J, Gordon M. DroidSafe[R]. Massachusetts Institute of Technology Cambridge United States, 2016.

[14] Lee S, Lee H, Ryu S. Broadening Horizons of Multilingual Static Analysis: Semantic Summary Extraction from C Code for JNI Program Analysis[C]. *2020 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020: 127-137.

[15] Fourtounis G, Triantafyllou L, Smaragdakis Y, et al. Identifying Java Calls in Native Code *via* Binary Scanning[C]. *The 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020: 388-400.

[16] Enck W, Gilbert P, Han S, et al. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones[J]. *ACM Transactions on Computer Systems*, 2014, 32(2): 1-29.

[17] Yan L K, Yin H. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis[C]. *21st USENIX security symposium,* 2012: 569-584.

[18] Sun M S, Wei T, Lui J C S. TaintART: A Practical Multi-Level Information-Flow Tracking System for Android RunTime[C]. *The 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016: 331-342.

[19] Xu G Q, Wang W Z, Jiao L T, et al. SoProtector: Safeguard Privacy for Native SO Files in Evolving Mobile IoT Applications[J]. *IEEE Internet of Things Journal*, 2020, 7(4): 2539-2552.

[20] Zhang X L, Wang X Y, Slavin R, et al. ConDySTA: Context-Aware Dynamic Supplement to Static Taint Analysis[C]. *2021 IEEE Symposium on Security and Privacy*, 2021: 796-812.

[21] Compiling, Loading and Linking Native Methods-Resolving Native Method Names [EB/OL]. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/design.html#resolving_native_method_names.

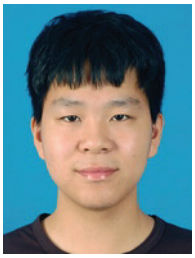
[22] Android ABI.[EB/OL]. https://developer.android.com/ndk/guides/.

[23] androguard.[EB/OL]. https://github.com/androguard/androguard.

徐贯虹 于 2021 年在中南大学信息安全专业获得工学学士学位。现在武汉大学网络空间安全专业攻读硕士学位。研究领域为软件安全、移动安全。研究兴趣包括: 软件安全、系统安全、移动安全。CCF 会员。Email: guanhongxu@whu.edu.cn

傅建明 于 2000 年在武汉大学获得博士学位。现任武汉大学国家网络安全学院教授, 博士生导师。研究领域为系统安全、软件安全。研究兴趣包括: 系统安全、软件安全、AI 安全、移动安全。CCF 会员。Email: jmfu@whu.edu.cn

聂宇 于 2013 年在南昌大学计算机技术专业获得硕士学位。现在武汉大学网络空间安全专业攻读博士学位。研究领域为移动隐私保护。研究兴趣包括: 机器视觉、自然语言处理。CCF 会员。Email: yu.nie@whu.edu.cn



解梦飞 于 2019 年在中国民航大学信息安全专业获得工学学士学位。现在武汉大学网络空间安全专业攻读博士学位。研究领域为系统安全、软件安全。研究兴趣包括: 系统安全、软件安全。CCF 会员。Email: mfxie96@whu.edu.cn