基于符号执行的 Android 原生代码控制流图提取方法

颜慧颖，周振吉，吴礼发，洪征，孙贺
（解放军理工大学指挥信息系统学院，江苏 南京 210000）

摘　要：提出了一种基于符号执行的控制流图提取方法，该方法为原生库中的函数提供了符号执行环境，对 JNI 函数调用进行模拟，用约束求解器对符号进行求解。实现了控制流图提取原型系统 CFGNative。实验结果表明，CFGNative 可准确识别样例中所有的 JNI 函数调用和原生方法，并能够在可接受的时间内达到较高的代码覆盖率。

关键词：控制流图；Android 应用软件；原生代码；符号执行
中图分类号：TP309
文献标识码：A
doi: 10.11959/j.issn.2096-109x.2017.00178

Symbolic execution based control flow graph extraction method for Android native codes

YAN Hui-ying, ZHOU Zhen-ji, WU Li-fa, HONG Zheng, SUN He
(Institute of Command Information System, PLA University of Science and Technology, Nanjing 210000, China)

Abstract: A symbolic execution based method was proposed to automatically extract control flow graphs from native libraries of Android applications. The proposed method can provide execution environments for functions in native libraries, simulate JNI function call processes and solve symbols using constraint solver. A control flow graph extraction prototype system named CFGNative was implemented. The experiment results show that CFGNative can accurately distinguish all the JNI function calls and native methods of the representative example, and reach high code coverage within acceptable time.

Key words: control flow graph, Android application, native code, symbolic execution

1　引言

Android 系统的普及使 Android 应用的数量和种类呈爆发式增长。据 Statista 的数据统计，2017 年 3 月，仅 Google play 上的 Android 应用就达 2.8×10^6 多个，比 2015 年增长了近 1×10^6 个[1]。面对海量的 Android 应用，安全人员可能需要分析程序寻找漏洞或判断程序是否有恶意行为；开发者倾向于复用已有的程序模块。而大多数情况下，分析者接触到的都是编译后的应用，需要分析程序安装包（APK，Android package）中的代码文件（字节码文件和原生库文件）获取有用的信息。因此，人们对高效的 Android 应用分析方法和技术的需求更加迫切。

控制流图在程序逆向和分析领域中应用十分广泛。编译后的程序丢失了很多源代码中数据和

收稿日期：2017-05-07；**修回日期**：2017-06-09。**通信作者**：吴礼发，wulifa@vip.163.com
基金项目：国家重点研发计划基金资助项目（No.2017YFB0802900）；江苏省自然科学基金资助项目（No. BK20131069）
Foundation Items: The National Key Research and Development Program of China (No.2017YFB0802900), The Natural Science Foundation of Jiangsu Province (No. BK20131069)

代码结构的信息，间接跳转、不可执行的数据块、代码段对齐等使反汇编变得困难。控制流图有助于解决这些棘手的问题[2,3]。并且控制流图是数据流分析、数据依赖分析、程序切片等其他分析方法的基础[4,5]，也可作为区分恶意和非恶意代码的特征[6,7]。

传统的控制流图提取方法正被逐渐移植到 Android 应用程序的分析过程中。但由于 Android 应用程序与 PC 程序的结构和运行环境存在较大差别，这些方法[8~17]不能直接用于构造 Android 应用的控制流图。目前，针对 Android 应用的工作[18~20]主要关注 Dalvik 字节码（简称字节码）的控制流图提取方法，缺乏对原生库代码的研究。而越来越多的 Android 程序在原生库中完成一些重要和复杂的功能，如加密、加壳等。恶意代码也开始倾向于隐藏在原生库中以逃避检测，如“百脑虫”“蜥蜴之尾”等木马都将其恶意逻辑隐藏在原生库中，因此仅分析字节码并不能完整地了解 Android 应用的真实行为。针对上述问题，本文深入分析了 Android 应用原生代码的特点，提出了一种基于符号执行的控制流图提取方法，设计并实现了原型系统，主要贡献如下。

1) 提出了一种基于符号执行的控制流图提取方法，符号执行过程基于中间表示 VEX。该方法与平台无关，可以用来分析为多种平台编译的 Android 原生代码。

2) 深入分析系统的 JNI 特性，对系统 JNI 相关的结构和 JNI 函数进行模拟，使本文方法能准确识别原生代码中的原生方法和 JNI 函数调用。

3) 基于 angr 设计并实现了用于提取 Android 原生库控制流图的原型系统 CFGNative，该系统能自动识别导出函数和注册的原生方法，并将其作为控制流图的起点。CFGNative 为用户及现有的 Dalvik 字节码分析工具提供了接口，以便于分析者结合 CFGNative 和字节码的分析工具提取全部应用的控制流图或基于该系统实现其他的程序分析方法。

2 背景

2.1 Android 应用和 JNI 函数调用

本文分析的对象是编译打包后的 APK。APK 中主要包含 Java 代码编译后生成的 Dalvik 字节码文件（后缀为 dex），C/C++代码编译后的针对不同平台的原生库文件（后缀为 so）、资源文件和 AndroidManifest.xml 文件。APK 中的可执行文件为字节码文件和原生库文件。字节码运行在 Dalvik 虚拟机上，原生代码直接运行在 Linux 系统上。

Android 应用中的字节码和原生代码可以通过 JNI 通信和相互调用。JNI 是 Java 程序设计语言功能最强的特性，它允许 Java 类的某些方法原生实现，同时让它们能够像普通 Java 方法一样被调用[21]。Android 的 Dalvik 虚拟机也支持 Java 的 JNI 特性。Dalvik 字节码需调用 System.loadLibrary 方法加载包含原生方法的原生库文件，并且用关键字 native 声明原生方法，之后才能调用原生方法。

原生库按照注册规则向 Dalvik 虚拟机注册暴露给字节码的原生方法。原生方法需在字节码中用关键字 native 进行声明，Dalvik 虚拟机调用原生方法时，在原生库中找到与这些声明对应的方法实体，并传递参数 JNI 接口指针的地址（如 JNIEnv 指针）和实例引用或类引用（若该原生方法是实例方法则传入实例引用，若为静态方法则传入类引用）。

原生库中的代码通过指向 JNI 接口的指针获取 JNI 函数（或 JNI 接口函数）的地址，然后调用 JNI 函数与字节码进行数据交换，如访问 Java 类的字段、创建类的实例、调用类和实例的方法等。JNI 函数调用过程示例汇编代码如下。

```
.text:00000D34      MOV   R4, R0
.text:00000D38      LDR    R3, [R0]
.text:00000D3C      BEQ   loc_E60
.text:00000D40      LDR    R1, = (aAndroid-Content - 0xD50)
.text:00000D44      LDR   R3, [R3,#0x18]
.text:00000D48      ADD   R1, PC, R1 ; "android/content/Context"
.text:00000D4C      BLX   R3
.text:00000D50      SUBS   R5, R0, #0
.text:00000D54      BEQ   loc_E74
.text:00000D58      LDR   R12, [R4]
```

```
.text:00000D5C      MOV    R0, R4
.text:00000D60      LDR    R2, = (aGetsys-
temservi - 0xD74)
.text:00000D64      MOV    R1, R5
.text:00000D68      LDR    R3, = (aLjava-
LangStrin - 0xD7C)
.text:00000D6C      ADD    R2, PC, R2 ;
"getSystemService"
.text:00000D70      LDR    R12, [R12,#0x84]
.text:00000D74      ADD    R3, PC, R3 ;
"(Ljava/lang/String;)Ljava/lang/Object;"
.text:00000D78      BLX    R12
```

根据函数调用规约，R0 传递参数 JNIEnv 指针，因此代码 00000D38 处的 R3 寄存器和 00000D58 处的 R12 寄存器存储的是 JNIEnv 的值。JNIEnv 是一个接口指针，该接口结构中包含 JNI 函数表，通过 JNIEnv 和相对偏移可以访问函数表。相对 JNIEnv 偏移 0x18 的地址上存储的是 JNI 函数 FindClass 的地址。程序在 00000D44 处获取 FindClass 的地址，并通过 00000D4C 处的间接跳转指令调用该 JNI 函数得到 android/content/Context 类的引用，在 00000D70 处获取存储在相对 JNIEnv 偏移 0x84 地址上的 JNI 函数 GetMethodID 的地址，00000D78 处的间接跳转指令调用 GetMethodID 得到 android/content/Context 的 getSystemService 方法的引用。传统的控制流图提取方法不能解析程序中的 JNI 函数调用过程，因此不能直接用来提取 Android 原生代码的控制流图。

2.2　中间表示 VEX

随着嵌入式平台的发展，Android 系统的底层平台也更加多样，现在市场上主要的架构有 ARM、X86、AMD64 和 MIPS，所以原生库可能是不同平台的目标文件，使用不同的指令集。为了使本文方法与平台无关，将原生库中的指令转化为中间表示，在中间表示上进行符号执行。

本文使用的中间表示是 VEX[22]。VEX 是一种较成熟的平台无关的中间语言，使用广泛，已经被实践证明能较好地兼容多种平台（包括 Android 系统中可能使用的几种平台）。一些经典的二进制分析平台，如 Valgrid[23]、angr[24]，将二进制码转化为中间表示 VEX，再基于 VEX 进行程序分析。第 3.3 节将详细介绍 VEX 在符号执行过程中的作用。图 1 为 32 bit ARM 指令转化为 VEX 的示例，图 1 左侧是 ARM 指令，将寄存器 R2 中的值减 8 再存入 R2 寄存器。图 1 右侧是转化得到的 VEX 语句，先将 R2 的值赋给变量 *t*0，使变量 *t*1 取值 8，再将变量 *t*0 和 *t*1 的差赋给变量 *t*3，然后把 *t*3 的值存储到寄存器 R2 中，最后将下一条指令的地址 0x59FC8 存入 IP 寄存器。

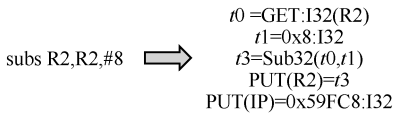

图 1　ARM 指令转化为 VEX 示例

3　基于符号执行的控制流图提取方法

以原生库文件的导出函数（一般包括静态注册的原生方法一般会出现在导出函数表中）和动态注册的原生方法为符号执行的起点，控制流图的提取过程如图 2 所示。首先为每个起点初始化程序状态，在程序状态中模拟 JNI 相关结构（JNIEnv、JavaVM、JNIInvokeInterface 和 JNINativeInterface），将无法确定的参数初始化为符号值；然后从起点开始符号执行。遇到跳转指令时，若跳转地址是 JNI 相关结构中的 JNI 函数地址，则执行模拟 JNI 函数的 SimProcedure，再返回符号执行过程；否则从跳转地址继续符号执行。

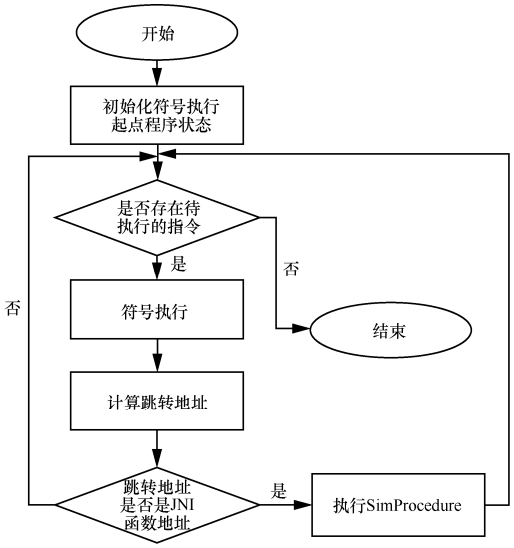

图 2　控制流图提取过程

3.1 程序控制流图

按照冯诺依曼体系结构，无跳转指令时，程序执行完一条指令后，紧接着在内存中取下一条指令继续执行，这种执行顺序被称为顺序执行。程序的执行流程有顺序、分支和循环这 3 种。分支和循环都是由跳转指令破坏当前的顺序执行实现的。把连续的顺序执行的指令集合作为一个基本块，将基本块作为控制流图的节点，跳转语句导致的基本块之间的控制流迁移为控制流图的边。因此，提取控制流图的重点和难点是计算跳转地址（跳转指令实现跳转后，程序继续执行的地址）。

3.1.1 跳转指令

每种指令集中都有跳转指令，这些指令可以使程序接着执行跳转地址处的指令，而非顺序执行下一条指令。按跳转是否需要条件可将跳转指令分为强制跳转指令和条件跳转指令。一定会发生跳转的指令为强制跳转指令，如 X86 指令集中的 CALL、JMP 指令，ARM 指令集中的 B、BL 指令，以及直接给 PC 赋值的指令，如 LDR PC、Expr。当满足特定条件才跳转的指令为条件跳转指令，如 X86 指令集中的 JE、JNE 指令，ARM 指令集中的 BE、BNE 指令。按跳转地址的取值方式可以将跳转指令分为直接跳转指令和间接跳转指令。直接跳转指令将跳转地址直接编码到指令中，而间接跳转指令的跳转地址依赖于寄存器或内存中的值。每一条跳转指令都可以表示为 $jmp_{addr}(target, guard)$的形式，其中，*addr* 是跳转指令所在的地址，*target* 是跳转地址，*guard* 是跳转的条件。控制流图的准确性和完整性很大程度上依赖于跳转地址计算的准确性。

跳转指令将 PC 的值置为跳转地址而非顺序执行的下一条指令地址，同时还可能改变内存或其他寄存器的值，如 CALL 指令在程序跳转前会将顺序执行的下一条指令的地址压入堆栈，BL 指令会将顺序执行的下一条指令存储到 R14 寄存器中。因此可以将跳转指令抽象为改变程序状态（寄存器和内存的取值情况）的函数，第 3.3.2 节会进一步介绍。

3.1.2 基本块

本文将指令转化为中间表示 VEX，因此控制流图的基本块是连续且顺序执行 VEX 语句的集合。

定义 1 指令 *ins* 的长度记为 $len(ins)$，所在地址记为 $addr(ins)$。若 *ins* 为跳转指令，$isjmp(ins) = True$，否则 $isjmp(ins) = False$。若指令使函数结束，$isExit(ins) = True$，否则 $isExit(ins) = False$。指令序列 $iseq = [iseq_0, iseq_1 \cdots, iseq_n]$ 为一个指令基本块，当该指令序列中的指令满足条件

$$\begin{aligned}&\forall iseq_i (i < 0 \leqslant n \to addr(iseq_i)\\&= addr(iseq_{i-1}) + len(iseq_{i-1}))\\&\wedge\ isjmp(iseq_n) \vee isExit(iseq_n)\\&\wedge \exists s((addr(s) = addr(iseq_0) -\\&len(iseq_0)) \wedge isjmp(s))\end{aligned}$$

将 *iseq* 中的指令全部转换为 VEX 语句得到的语句序列为 VEX 基本块。根据定义可以推断基本块（除起始块）的起始地址都是某一条跳转语句的跳转地址，基本块（除结束块）的最后一条指令都是跳转语句。只能从基本块第一条语句开始执行，且一旦开始必然顺序执行到该基本块中最后一条指令。

3.1.3 控制流图

本文的目的是分析程序控制流的迁移，并将其表示为由基本块和边构成的控制流图。

定义 2 任意一个 Android 库文件 *P* 的指令都可以被划分为多个基本块，这些基本块的集合记为 $Blocks_P$。2 个基本块 *b*1 和 *b*2 间存在数据流迁移 $trans(b1, b2)$，当且仅当

$$\begin{aligned}&(ins_{b1\_last} = jmp_{addr(ins_{b1\_last})}(target, guard)) \wedge\\&(addr(ins_{b2\_0}) = target)\end{aligned}$$

其中，ins_{b1_last} 为 *b*1 中最后一条指令，ins_{b2_0} 为 *b*2 中第一条指令。

定义 3 称 $G = (N, E)$ 为 *P* 的一个控制流图（*N* 是控制流图中节点的集合，*E* 是控制流图中边的集合），当且仅当

$$\begin{aligned}&\forall n(n \in N \to \exists b(b \in Blocks_P \wedge vex(b, n)))\\&\wedge\forall e\left(\begin{array}{l} e \in E \to \exists trans(b_p, b_q)\left(\begin{array}{l} e(n_p, n_q)\\ \wedge vex(b_p, n_p)\\ \wedge vex(b_q, n_q)\end{array}\right)\end{array}\right)\end{aligned}$$

在 $vex(b, n)$ 中，*b* 为中指令基本块，*n* 为 VEX 语句基本块，即将 *b* 中的每一条指令转化为 VEX 语句得到的结果。

定义 4 *P* 没有 main 函数，本文从导出函数和不出现在导出表中的原生方法开始提取控制流

图。给控制流图增加 2 个特殊节点 n_{start}、n_{end} 和若干边，控制流图中其他节点和边的约束依然满足定义 3，得到 $G^*=(N^*,E^*)$。

$$N^*=\{n_{\text{start}},n_{\text{end}}\}\bigcup N$$

$$E^*=\{e\,|\,e(n_{\text{start}},n),n\in N_{\text{exFun}}\}\bigcup$$

$$\{e\,|\,e(n,n_{\text{end}}),n\in N\wedge isExit(n_{\text{last}})\}\bigcup E$$

其中，n_{start} 为 G^* 入口节点，n_{end} 为 G^* 的结束节点。P 中导出函数和不出现在导出表中原生方法的第一个基本块的集合记为 N_{exFun}，当 n 中最后一条语句为函数结束语句时，$isExit(n_{\text{last}})=\text{True}$、否则 $isExit(n_{\text{last}})=\text{False}$。本文提取的是满足 G^* 定义的控制流图。

3.2　模拟 JNI 函数调用

如 2.1 节所述，原生方法通过间接跳转调用 JNI 函数，而传统的控制流图提取方法没有将原生方法的第一个参数解析为 JNI 接口指针地址，也不能解析 JNI 接口的结构，因此无法计算出 JNI 函数调用的间接跳转地址。对此，本文方法在程序状态中模拟 JNI 接口指针等 JNI 相关结构，找到向虚拟机注册的原生方法，然后将 JNI 接口指针的地址传递给原生方法。当符号执行遇到 JNI 函数调用的间接跳转指令时，即可根据模拟的 JNI 相关结构计算得到函数表中的 JNI 函数地址。另外，APK 的原生库中没有 JNI 函数的指令，无法符号执行 JNI 函数，本文方法用 SimProcedure 代替符号执行过程。

3.2.1　模拟 JNI 相关结构

JNI 相关结构包括 JNIEnv、JavaVM、JNIInvokeInterface 和 JNINativeInterface。Dalvik 虚拟机将 JNIEnv 的地址作为第一个参数传递给原生方法（JavaVM 的地址是 JNI_OnLoad 的第一个参数）。JNIInvokeInterface 和 JNINativeInterface 是 JNI 接口类型，分别包含一个函数表，函数表中包含的是 JNI 函数的地址。原生代码通过函数表中的函数访问虚拟机或字节码中的内容。JavaVM 指向的正是 JNIInvokeInterface 的起始地址，JNIEnv 指向的正是 JNINativeInterface 的起始地址，因此原生方法通过参数和相对偏移即可找到函数表中某一 JNI 函数表项的地址，从而得到该 JNI 函数的地址。JNIEnv、JavaVM、JNIInvokeInterface 和 JNINativeInterface 在内存中的结构如图 3 所示。

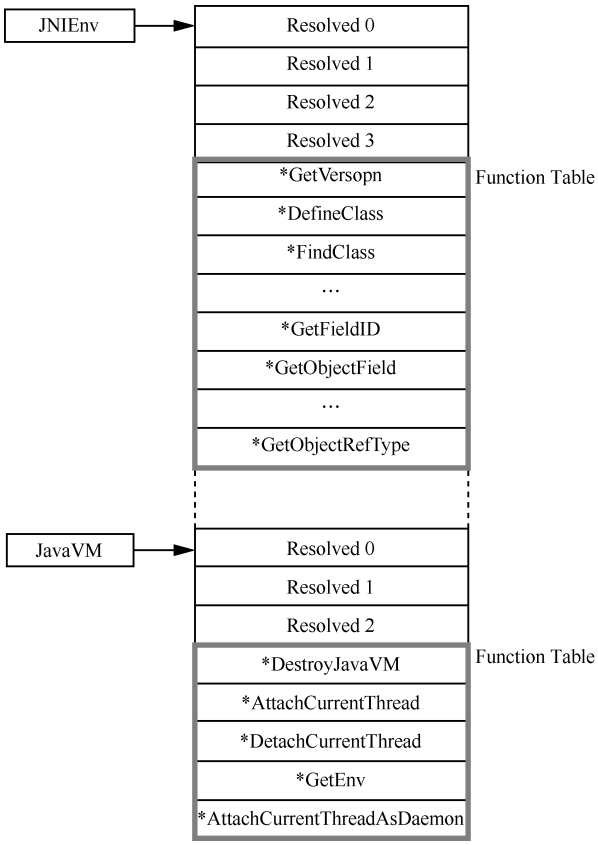

图 3　内存中 JNINativeInterface 和 JNIInvokeInterface 的结构

本文分析的原生库中不包含这些结构，也不会在内存中构造该结构，因此，在符号执行前需要在程序状态的内存中模拟该结构。在内存中开辟一块未使用的空间给 JavaVM、JNIEnv、JNIInvokeInterface 和 JNINativeInterface，在开辟的空间中填入任意未使用地址空间中的地址，执行过程中自动将 JavaVM 和 JNIEnv 的地址作为参数传递给原生方法。

3.2.2　SimProcedure 模拟 JNI 函数

除了模拟 JNI 相关结构，还需要模拟接口函数表中的 JNI 函数。在真实的 Android 原生库运行环境中，原生方法通过 Dalvik 虚拟机调用 JNI 函数，本文符号执行过程中既无虚拟机也无 JNI 函数，需要用 SimProcedure 对 JNI 函数的功能进行抽象和模拟，并且用 SimProcedure hook JNI 函数的地址，使执行到 JNI 函数时，相应的 SimProcedure 能被调用。

SimProcedure 是指用来模拟 JNI 函数的一类函数，概括了 JNI 函数的程序逻辑，是程序执行过程中实现 JNI 函数对程序状态的改变以及相应控制流图节点构造的实体。SimProcedure 不是原

生库中的代码，因此其构造的节点只用于记录该处 JNI 函数调用的信息，不包含真实代码对应的 VEX 语句。

JNIInvokeInterface 和 JNINativeInterface 的函数表中的每个函数都对应一个 SimProcedure。有的 JNI 函数会返回字节码的类、对象、字段、方法等的引用，后续 JNI 函数调用中可能会使用这些返回值。而执行环境中不存在字节码中的结构，这些结构的引用只是标识它们的符号。因此，这些字节码内容的引用是全局的，且在定义 SimProcedure 时，应该考虑 JNI 函数之间的关系，使其他 SimProcedure 也能够准确识别这些标识。如 FindClass 的 SimProcedure 需要返回一个标识以表示类名为传入的第二个参数的类，在符号执行过程中，其他 JNI 函数在遇到这个标识时也应该将其解释为这个类。

先构造图 3 的结构，再用对应的 SimProcedure hook JNI 接口函数表中填入的地址。执行过程中，当原生方法调用函数表中的 JNI 函数时，找到的 JNI 函数地址实际上是被 SimProcedure hook 的地址，因此执行的是 SimProcedure，执行完 SimProcedure 后返回调用 JNI 函数的下一条指令继续执行。

3.2.3 定位注册的原生方法

本文的符号执行要自动将 JNIEnv 的地址作为参数传递给原生方法，因此需要其能够识别注册的原生方法。

注册原生方法的方式分为静态注册和动态注册这 2 种。静态注册需要根据命名规则命名原生方法，要求用包名加上类名再加上字节码中声明的方法名来命名原生库中对应的原生方法，如应将与 android.helloWorld 包的 MainActivity 类中声明的原生方法 helloFromJNI 对应的原生库中的方法命名为 Java_android_helloWorld_ MainActivity_helloFromJNI。符号表中一般都包含静态注册的原生方法，因此可以通过符号表中函数名识别原生函数。而动态注册将原生方法与其在字节码中声明的对应关系记录到 JNINativeMethod 结构中，再通过调用 JNINativeInterface 中的 RegisterNatives 函数将 JNINativeMethod 中的原生方法注册到虚拟机。JNINativeMethod 的结构如下所示。

```
typedef struct {const char* name;
    const char* signature;
    void* fnPtr;
} JNINativeMethod;
```

该结构有 3 个字段，分别为字节码中声明的原生方法的名称、参数和返回值的类型信息以及原生方法在原生库中的地址。若执行过程中调用了 RegisterNatives 函数，从参数中可得到 JNINativeMethod 结构的列表在内存中的地址和列表长度，解析该列表获得动态注册的原生方法，该过程由 RegisterNatives 函数的 SimProcedure 完成。

动态注册需要在字节码调用原生方法之前完成。JNI_OnLoad 在原生库加载时被调用。通常在 JNI_OnLoad 中调用 RegisterNatives，动态注册过程就会在原生库加载时完成。因此，如果原生库中实现了 JNI_OnLoad 方法，应该优先执行 JNI_OnLoad 以找到可能出现的动态注册的原生方法。

3.3 提取控制流图

本文分析的对象是 Android 原生库文件，而原生库不可单独执行，如果直接进行动态分析，则需要实现触发原生库执行的模块。而 Android 应用中调用原生方法的是字节码，因此需要搭建两层执行环境（包括 Dalvik 虚拟机），使环境比较复杂，并且有时需要构造复杂的输入才能触发原生库中代码的执行，分析效率较低。为此本文提出了一种基于 VEX 的符号执行方法，直接分析 Android 原生库，提取控制流图。只要构造了函数的执行环境，该方法便可单独执行某一个函数，无需从 main 函数开始执行。符号执行的环境由 angr[25]提供，angr 将指令转化为中间表示 VEX，符号执行引擎根据 VEX 表达式和语句的语义修改程序状态，并记录下路径的约束条件，当遇到跳转语句时根据程序状态和约束条件计算跳转地址。

3.3.1 程序状态

程序状态主要包含程序在某一程序点内存、寄存器、变量（VEX 中含有变量）的取值情况。将 VEX 中寄存器的集合记为 *Regs*，内存区域记为 *Mems*，临时变量的集合记为 *Tmps*。符号执行的程序状态中有具体值和符号值。将具体值（如立即数 0x10、地址 0x400000 等）的集合记为 *Cons*。符号值表示满足一定约束的值的集合，用符号（如 x、y 等）表示。符号值在执行中也可参与运算，如 $x+y$、x+0x10 等。将符号值的集合记为 *Syms*。

因此程序状态中寄存器、内存和变量取值的值域为 $Vals(Vals = Syms \cup Cons)$。原生库中函数按函数调用规约接收参数，且除 JavaVM 和 JNIEnv 的地址外，其他参数在分析者没有指明的情况下都默认取符号值。只要构造一个函数被调用时的程序状态并将该状态输入符号执行引擎，就能从该函数开始符号执行，因此需要初始化原生函数和其他导出函数开始执行时的程序状态，如将参数赋值给传递参数的寄存器。

定义 5　程序状态 state 用二元组的集合表示，实际上为内存、寄存器和变量到 *Vals* 上的映射。

$$state \in States, States = (Regs \cup Mems \cup Tmps) \times Vals$$
$$\exists (a, val) \in state \ \ iff \ \ a := val$$

当程序状态中有二元组（a，val）时，则说明该状态下 a 的取值为 val。

Mems 理论上是无穷且连续的，但程序执行过程中各类型变量的值一般存储在一块连续的内存区域中。将该段内存中的值作为 *Vals* 中的一个元素，内存即可被看作离散的。另外，运行程序所需的内存空间有限，未使用的内存只是概念上的一块区域，在符号执行过程中并不分配空间以记录其状态，因此记录和读取内存是可实现的。

3.3.2　表达式和语句

符号执行是在 VEX 上进行的，因此需要告知执行引擎 VEX 表达式和语句的语义，执行引擎才能按照语义执行程序。

定义 6　将表达式的集合记为 *Exprs*，语句的集合记为 *Stmts*。表达式的语义为表达式和程序状态到 *Vals* 上的函数 *fe*，语句的语义为语句和程序状态到新的程序状态上的函数 *fs*。

$$fe : Exprs \times States \to Vals$$
$$fs : Stmts \times States \to States$$

$fe(expr, state) = val$ 表示表达式 *expr* 在当前程序状态 *state* 取值 *val*。$fs(stmt, state_1) = state_2$ 表示语句 *stmt* 将当前状态 $state_1$ 变成了新的程序状态 $state_2$。

VEX 语句和表达式的语义要根据具体的表达式和语句定义，如表达式 RdTmp(*t*10)的语义为变量 *t*10 的取值，语句 WrTmp(*t*1)=(IR *expr*)的语义为将表达式 *expr* 的值赋给 *t*1 得到新的程序状态。VEX 的表达式和语句的详细介绍参考文献[22]。

3.3.3　求解带符号的跳转地址

符号执行过程中可能存在带有符号的跳转地址。执行过程在程序状态中记录了每一条分支对符号的约束。当遇到地址带有符号的跳转指令时，用约束求解器求解该符号的约束得到符号值的范围，计算得到可能的跳转地址，然后判断这些可能的跳转地址是否为指令的起始地址，将满足条件的值列入最终跳转地址结果的集合中。

3.3.4　控制流图提取算法

基于上述讨论，控制流图提取算法的伪代码如下。

```
1)  Procedure CFGNative()
2)  begin
    /* Initialisation */
3)    export_funcs_addrs = get addresses
      of exported functiosns
4)    on_load_addr = get the address of
      JNI_OnLoad
5)    static_registerd_NativeMethod =
      get addresses of statically registered native
      methods
6)    dynamic_registered_NativeMethods = ∅
7)    worklist = export_funcs_addrs
8)    CFG = a control flow graph with only
      n_start and n_end
9)    states = ∅
    /* Iteration */
10)   while worklist ≠ ∅ do
11)       select and remove an address(addr)
          from worklist(on_load_addr
          prioritized)
12)       if addr is an address of JNI function
              /* if RegisterNatives is called,
              dynamically registered native
              methods can be found */
13)       then call SimProcedure
14)           while dynamic_registered_
              NativeMethods≠ ∅ do
15)               remove an address from
                  dynamic_registered_
                  NativeMethods
                  and add it to worklist
```

```
16)            end
17)          else
18)          irsb = lift the binary code started at
             addr to a vex block
19)          CFG.addNode(irsb)
20)          src_nodes = get the nodes which
             jumped to addr
21)          for node in src_nodes
22)              CFG.addEdge(node,irsb)  end
23)          sim_state = get the state for irsb from
                         states, if there is not one,
                         initilize a state
24)          sim_irsb = get the smentics of
             vex statements
25)          sim_irsb.symbolicExecute
             (sim_state)
26)          successors = compute the target
             addresses of irsb's jump statement
27)          for successor in successors
28)              states = states ∪ {< successo,
                         sim_state >}
29)              worklist = worklist∪
                         successors end
30)     end
31)  end
```

代码第 3)~9)行为算法初始阶段，其中，第 3)行从导出表中获取导出函数地址；第 4)行获取导出函数中 JNI_OnLoad 的地址，若不存在则为空；第 5)行识别导出函数中静态注册的原生方法；第 7)行将导出函数地址加入到工作列表中。

第 10)~30)行代码开始循环，从工作列表中取出地址，如果该地址为 JNI 函数的地址，第 13)行代码调用 JNI 函数对应的 SimProcedure，若调用的 JNI 函数是 RegisterNatives，则可以找到动态注册的原生方法，第 15)行将新找到的动态注册的原生方法的地址加入工作列表中；如果该地址不是 JNI 函数的地址，则执行第 18)~29)行。第 18)行得到起始地址为 *addr* 的 VEX 基本块。第 23)行获取 VEX 基本块开始符号执行时的程序状态，需要为函数的起始块初始化程序状态，从程序状态集合中获取其他 VEX 基本块对应的程序状态。如果基本块属于原生方法，初始化程序状态时要在内存中模拟 JNI 相关结构。第 24)行得到 VEX 表达式和语句的语义。第 25)行为符号执行过程，第 26)~29)行计算跳转地址，将以跳转地址为起点的基本块对应的程序状态加入程序状态集合，并将跳转地址加入工作列表。循环整个过程直到工作列表为空。

4 原型系统实现与实验

4.1 系统实现

为了验证本文所提方法的有效性，基于 angr 实现了提取 Android 原生库的控制流图的原型系统 CFGNative，其结构如图 4 所示。系统中，模块 loader 解析输入的原生库文件，并将代码段和数据段加载到内存中；lifter 从给定的地址开始识别并翻译 VEX 基本块；execution simulator 在 lifter 识别的 VEX 基本块上进行符号执行并将计算得到的跳转地址给 lifter，lifter 从新的地址开始继续识别 VEX 基本块，迭代该过程直到没有新的 VEX 基本块生成；state 记录执行各阶段的程序状态；core engine 是符号执行的核心模块，逐条解析 VEX 语句的语义；JNI struct constructor 在内存中构造 JNIInvokeInterface 和 JNINativeInterface 等结构；simprocedures 被 hook 到 JNI 函数的地址上；constraint solver 根据约束求解带符号跳转地址。

4.2 实验及结果分析

本文设计了 2 个实验：实验 1 用 CFGAccurate

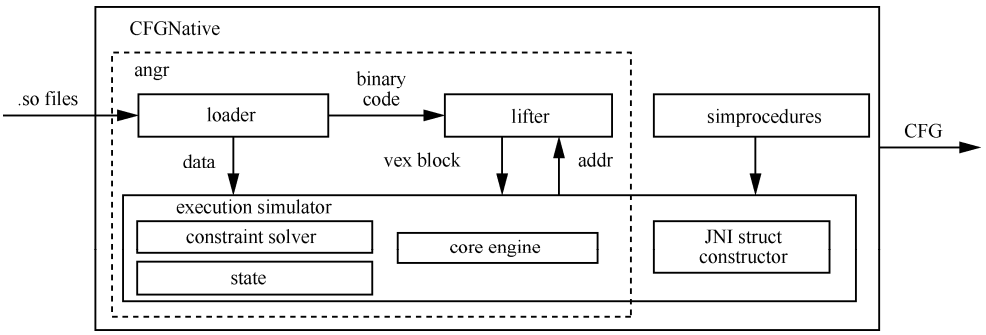

图 4 CFGNative 体系结构

提取本文构造的 Android（源码见附录 A）应用 JNITest 中原生库的一个控制流图，目的是检测 CFGAccurate 是否能准确识别静态和动态注册的原生函数和 JNI 函数调用。实验 2 对比 angr 的 CFGAccurate、IDA 和 CFGNative 提取控制流图的结果，目的是将 CFGNative 的指令覆盖率和时间耗费与现有经典分析工具进行对比，检测 CFGNative 增加的功能是否对其他性能造成过大影响。实验硬件配置为 Intel(R) Core(TM) i7-3770 处理器（8 核 3.40 GHz），32 GB RAM。操作系统为 64 bit 的 Ubuntu16.04。

1) 实验 1：JNITest

原生方法分为静态和动态这 2 种注册方式，因此 JNITest 中包含了 2 个原生方法，分别用 2 种不同的方式注册，同时包含多个 JNI 函数调用，具有一定代表性。

Java_com_example_test_MainActivity_getIMEI 为静态注册的原生方法，通过 JNI 函数 FindClass 找到 android.telephony.TelephonyManager 类，然后调用 JNI 函数 GetMethodID 获得该类 getDeviceId 方法的引用，最后通过 CallObjectMethod 调用该方法获取设备的 IMEI。该方法中的 JNI 函数会使用其他 JNI 函数返回的结果，如 FindClass 返回的 Java 类的引用会作为 GetMethodID 的参数，因此可以测试 SimProcedure 的定义是否正确。helloJNI 为动态注册的原生方法，通过在 JNI_ONLoad 中调用 RegistaerNatives 进行注册，该方法返回 JNI 函数 NewStringUTF 构造的字符串“Hello from JNI”。

CFGNative 提取的控制流图显示其准确识别了这 2 个原生方法和所有的 JNI 函数调用，得到的控制流图中共有 8 个代表 JNI 函数的节点，38 条包含表示 JNI 函数节点的边。代表 JNI 函数调用的节点和包含这些节点的边的详细信息见附录 B。

2) 实验 2：对比

实验 2 将 Android 应用市场 APPChina 上下载量排名靠前的 16 个应用作为样本，过滤样本原生库中的广告包、音视频处理等第三方工具包，共收集了 61 个原生库文件。表 4 为从每个应用中收集的原生库的个数，由于有些原生库存在于多个包中，所以个数的总和大于 61。从每个包中获取的具体的原生库见附录 C。用 CFGNative 提取这些原生库的控制流图，并与当前最流行的二进制分析工具 IDA 和 angr 的 CFGAccurate 对比。IDA、CFGAccurate 和 CFGNative 都以导出函数和不出现在导出函数表中的原生方法为起点构造控制流图，去掉编译过程加入的一些函数，如_gnu_Unwind_Resume。统计每种方法提取每个原生库的控制流图平均耗费的时间和平均每个图中节点个数、边数和覆盖的指令数，结果如表 2 所示。

表 1　　样本中收集的原生库个数

包名	原生库个数
com.qihoo360.mobilesafe	1
com.baidu.BaiduMap	4
com.baidu.input	2
com.storm.smart	9
com.pplive.androidphone	4
com.kugou.android	6
com.sankuai.meituan	7
com.qiyi.video	7
com.tencent.mtt	11
com.tencent.qqmusic	17
com.baidu.searchbox	2
com.tencent.qqpimsecure	1
com.baidu.homework	1
com.UCMobile	1
com.sina.weibo	4
com.tencent.mm	1

表 2　　控制流图提取结果

方法	平均数			
	节点/个	边/条	指令/条	时间/s
IDA	559.80	1 413.41	3 592.31	0.45
CFGAccurate(angr)	740.16	1 322.19	3 057.91	110.52
CFGNative	1 050.5	1 805.79	3 602.49	319

CFGAccurate 和 IDA 不具备分析 JNI 的能力，因此不能识别原生方法和 JNI 函数调用。CFGNative 比其他 2 种方式识别了更多的节点，其中包含表示 JNI 函数的节点，这些节点的指令条数为 0，从表 2 中可以看到，CFGNative 比 IDA 和 CFGAccurate 覆盖了更多的指令，这是因为 CFGNative 识别了 IDA 和 CFGAccurate 无法识别的间接跳转，从而发现了新的基本块中的指令。实验结果表明，CFGNative 不仅能够识别

JNI 函数调用，且在可接受的时间内具有较高的代码覆盖率。

5　结束语

本文提出了一种基于符号执行的控制流图提取方法，用于自动提取 Android 应用原生代码的控制流图。该方法可以识别原生方法和 JNI 函数调用，准确计算间接跳转地址，且具有较高代码覆盖率。下一步研究包括以下 2 个方面。

1) 符号执行用 SimProcedure 代替真实的 JNI 函数调用，因此对 SimProcedure 定义的准确性会影响符号执行的结果。目前对 SimProcedure 的定义过于依赖经验。未来工作将深入分析真实环境下 JNI 函数的行为并对其进行更系统的建模。

2) 符号执行过程中包含的符号值过多或约束求解器的求解能力不足都可能导致求解所得的符号值范围过大。当包含符号的跳转地址取值范围过大时，可能会产生路径爆炸问题，需要进一步研究该问题的解决方案。

附录 A　本文构造的 Android 原码

```
#include <string.h>
#include <jni.h>
#include <android/log.h>
#include "jnitest.h"
#define LOGV(...) __android_log_print(ANDROID_LOG_VERBOSE, "com.exaple.test", __VA_ARGS__)
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG, "com.exaple.test", __VA_ARGS__)
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, "com.exaple.test", __VA_ARGS__)
#define LOGW(...) __android_log_print(ANDROID_LOG_WARN, "com.exaple.test", __VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR, "com.exaple.test", __VA_ARGS__)
JNIEnv *g_env;
JavaVM *g_vm;
jclass native_class;
#ifndef NELEM //计算结构元素个数
# define NELEM(x) ((int) (sizeof(x) / sizeof((x)[0])))
#endif
//静态注册的原生方法 getIMEI
JNIEXPORT jstring Java_com_example_test_MainActivity_getIMEI
(JNIEnv *env, jobject mContext){
    if(mContext == 0){
        return (*env)->NewStringUTF(env,"[+]Error : Context is 0");
    }
    jclass cls_context = (*env)->FindClass (env, "android/content/Context");
    if(cls_context == 0){
        return (*env)->NewStringUTF(env,"[+] Error: FindClass <android/content/Context> Error");
    }
    jmethodID getSystemService = (*env)->GetMethodID(env,cls_context,"getSystemService","(Ljava/lang/String;)Ljava/lang/Object;");
    if(getSystemService == 0){
        return (*env)->NewStringUTF(env,"[+] Error : GetMethodID failed");
    }
    jfieldID TELEPHONY_SERVICE = (*env)->GetStaticFieldID(env,cls_context,"TELEPHONY_SERVICE","Ljava/lang/String;");
    if(TELEPHONY_SERVICE == 0){
        return (*env)->NewStringUTF(env,"[+] Error : GetStaticFieldID failed");
    }
    jstring str = (jstring)(*env)->GetStaticObjectField(env,cls_context, TELEPHONY_SERVICE);
    jobject telephonymanager = ((*env)->CallObjectMethod(env,mContext, getSystemService, str));
    if(telephonymanager == 0){
        return (*env)->NewStringUTF (env,"[+] Error: CallObjectMethod failed");
    }
    jclass cls_TelephoneManager = (*env)->FindClass(env, "android/telephony/TelephonyManager");
    if(cls_TelephoneManager == 0){
        return (*env)->NewStringUTF (env,"[+] Error: FindClass TelephoneManager failed");
```

```
        }
    jmethodID getDeviceId = ((*env)->
GetMethodID(env,cls_TelephoneManager, "getDeviceId",
"()Ljava/lang/String;"));
        if(getDeviceId == 0){
            return (*env)->NewStringUTF (env,
"[+] Error: GetMethodID getDeviceID failed");
        }
    jobject DeviceID = (*env)->CallObjectMethod
(env,telephonymanager,getDeviceId);
    return (jstring)DeviceID;
  }
 //动态注册的方法 helloJNI
 JNIEXPORT jstring helloJNI(JNIEnv* env, jclass
clazz){
  const char * chs = "Hello from JNI";
  return (*env)->NewStringUTF(env, chs);
 }
 static JNINativeMethod methods[] = {
  {"helloJNI", "()Ljava/lang/String;", (void*)helloJNI}
 };
 jint JNI_OnLoad(JavaVM* vm, void* reserved){
  if(JNI_OK != (*vm)->GetEnv(vm, (void**)&g_env,
JNI_VERSION_1_6)){
    return -1;}
  LOGV("JNI_OnLoad()");
  native_class = (*g_env)->FindClass(g_env, "com/
example/test/MainActivity");
  if (JNI_OK ==(*g_env)->RegisterNatives(g_env, //动
态注册原生方法
      native_class, methods,
NELEM(methods))){
    LOGV("RegisterNatives() --> helloJNI() ok");
  } else {
    LOGE("RegisterNatives() --> helloJNI() failed");
    return -1;}
  return JNI_VERSION_1_6;
 }
```

附录 B

附录表 1　　JNI 函数调用的节点和包含这些节点的边的信息

节点	边缘
<CFGNode NewStringUTF (0x6000688)>	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x11c (0x400e48) >, <CFGNode NewStringUTF (0x6000688) >)
	（<CFGNode NewStringUTF (0x6000688) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0x160 (0x400e8c) >)
	(<CFGNode NewStringUTF (0x6000688) >, <CFGNode helloJNI+0x18 (0x400d24) >)
	(<CFGNode NewStringUTF (0x6000688) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0x1a8 (0x400ed4) >)
	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x1ac (0x400ed8) >, <CFGNode NewStringUTF (0x6000688) >)
	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x198 (0x400ec4) >, <CFGNode NewStringUTF (0x6000688) >)
	(<CFGNode helloJNI (0x400d0c) >, <CFGNode NewStringUTF (0x6000688) >)
	<CFGNode Java_com_example_test_MainActivity_getIMEI+0x164 (0x400e90) >, <CFGNode NewStringUTF (0x6000688) >)
	(<CFGNode NewStringUTF (0x6000688) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0x130 (0x400e5c) >)
	(<CFGNode NewStringUTF (0x6000688) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0x144 (0x400e70) >)
	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x180 (0x400eac) >, <CFGNode NewStringUTF (0x6000688) >)
	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x148 (0x400e74) >, <CFGNode NewStringUTF (0x6000688) >)

续表

节点	边缘
	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x134 (0x400e60) >, <CFGNode NewStringUTF (0x6000688) >)
	(<CFGNode NewStringUTF (0x6000688) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0x1c4 (0x400ef0) >)
	(<CFGNode NewStringUTF (0x6000688) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0x194 (0x400ec0) >)
	(<CFGNode NewStringUTF (0x6000688) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0x17c (0x400ea8) >)
<CFGNode Findclass (0x6000404) >	(<CFGNode Findclass (0x6000404) >, <CFGNode JNI_OnLoad+0x60 (0x400f90) >)
	(<CFGNode Findclass (0x6000404) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0xd4 (0x400e00) >)
	(<CFGNode Findclass (0x6000404) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0x24 (0x400d50) >)
	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x14 (0x400d40) >, <CFGNode Findclass (0x6000404) >)
	(<CFGNode JNI_OnLoad+0x48 (0x400f78) >, <CFGNode Findclass (0x6000404) >)
	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0xc4 (0x400df0) >, <CFGNode Findclass (0x6000404) >)
<CFGNode GetMethodID (0x6000470) >	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x2c (0x400d58) >, <CFGNode GetMethodID (0x6000470) >)
	(<CFGNode GetMethodID (0x6000470) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0xfc (0x400e28) >)
	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0xdc (0x400e08) >, <CFGNode GetMethodID (0x6000470) >)
	(<CFGNode GetMethodID (0x6000470) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0x50 (0x400d7c) >)
<CFGNode GetStaticFieldID (0x600062c) >	(<CFGNode GetStaticFieldID (0x600062c) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0x7c (0x400da8) >)
	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x58 (0x400d84) >, <CFGNode GetStaticFieldID (0x600062c) >)
<CFGNode GetStaticObjectField (0x6000630) >	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x88 (0x400db4) >, <CFGNode GetStaticObjectField (0x6000630) >)
	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x88 (0x400db4) >, <CFGNode GetStaticObjectField (0x6000630) >)
<CFGNode CallObjectMethod (0x6000474) >	(<CFGNode CallObjectMethod (0x6000474) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0xb4 (0x400de0) >)
	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x98 (0x400dc4) >, <CFGNode CallObjectMethod (0x6000474) >)
	(<CFGNode Java_com_example_test_MainActivity_getIMEI+0x108 (0x400e34) >, <CFGNode CallObjectMethod (0x6000474) >)
	(<CFGNode CallObjectMethod (0x6000474) >, <CFGNode Java_com_example_test_ MainActivity_getIMEI+0x118 (0x400e44) >)
<CFGNode GetEnv (0x60007a8) >	(<CFGNode GetEnv (0x60007a8) >, <CFGNode JNI_OnLoad+0x24 (0x400f54) >)
	(<CFGNode JNI_OnLoad (0x400f30) >, <CFGNode GetEnv (0x60007a8) >)
<CFGNode RegisterNatives (0x6000748) >	(<CFGNode JNI_OnLoad+0x60 (0x400f90) >, <CFGNode RegisterNatives (0x6000748) >)
	(<CFGNode RegisterNatives (0x6000748) >, <CFGNode JNI_OnLoad+0x90 (0x400fc0) >)

附录表 1 中节点表示为<CFGNode nodeName (nodeAddr)>的形式，其中 nodeName 为该节点的名称，当节点基本块为函数的起始块时，以函数名作为节点的名称，否则以函数名加上节点基本块起始地址相对于函数起始地址的偏移作为节点的名称。表示 JNI 函数的节点以 JNI 函数的名称作为节点的名称。nodeAddr 为节点基本块的起始地址。表示 JNI 函数的节点的 nodeAddr 是本文 3.2.3 节提到的函数表中

填入的地址。边表示为（<CFGNode nodeName1 (nodeAddr1)>，<CFGNode nodeName2 (nodeAddr2)>）的形式，代表连接名为 nodeName1 的节点和名为 nodeName2 的节点的边。

附录 C

附录表 2　　样本应用中获取的原生库

包名	原生库名	包名	原生库名
com.qihoo360.mobilesafe	libnzdutil-jni-1.0.0.2002.so	com.tencent.qqpimsecure	libNativeRQD.so
com.tencent.mm	libmm_gl_disp.so	com.baidu.homework	libweibosdkcore.so
com.baidu.input	libbdinput_gif_v1_0_10.so libgaussblur_v1_0.so	com.baidu.searchbox	libbitmaps.so libmemchunk.so
com.tencent.mtt	libbeso.so libbitmaps.so libblur_armv7.so libcmdsh.so libcommon_basemodule_jni.so libdaemon_lib.so libFdToFilePath.so libFileNDK.so libgif-jni.so libmemchunk.so libtencentpos.so	com.tencent.qqmusic	libckey.so libdalvik_patch.so libdesdecrypt.so libexpress_verify.so libfilescanner.so libFormatDetector.so libframesequence.so libLPConvert.so libmApptracker4Dau.so libmresearch.so libNativeRQD.so libnetworkbase.so libqalmsfboot.so libqav_graphics.so libRandomUtilJni.so libtmfe30.so libweibosdkcore.so libwnsnetwork.so
com.storm.smart	liba.so libdaemon.so libgetuiext2.so libMMANDKSignature.so libmresearch.so libpl_droidsonroids_gif.so libpl_droidsonroids_gif_surface.so libstpinit.so libweibosdkcore.so	com.UCMobile	libucinflator.so
com.kugou.android	libkgkey.so libkguo.so libLencryption.so libMMANDKSignature.so librtmp.so libweibosdkcore.so	com.sankuai.meituan	libdpencrypt.so libdpobj.so libnetworkbase.so libnh.so libPayRequestCrypt.so libRectifyCard.so libuploadnetwork.so
com.baidu.BaiduMap	libcpu_features.so libgif.so libufosdk.so libweibosdkcore.so	com.sina.weibo	libamapv304ex.so libmemchunk.so libutility.so libweibosdkcore.so
com.qiyi.video	libblur.so libdaemon.so libmediacodec.so libMMANDKSignature.so libmresearch.so libpl_droidsonroids_gif.so librtmp.so	com.pplive.androidphone	libbreakpad_util_jni.so libmeet.so libVideoSdkMd5.so libweibosdkcore.so

参考文献：

[1] Statista. number of apps available in leading app stores as of march 2017[EB/OL].https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[2] SCHWARZ B, DEBRAY S, ANDREWS G. Disassembly of executable code revisited[C]//The Working Conference on Reverse Engineering. 2002:45-54.

[3] KRUEGEL C, ROBERTSON W, VALEUR F, et al. Static disassembly of obfuscated binaries[C]//Usenix Security Symposium. 2004: 255-270.

[4] REPS T, HORWITZ S, SAGIV M. Precise interprocedural dataflow analysis via graph reachability[C]//The 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1995: 49-61.

[5] HORWITZ S, REPS T, BINKLEY D. Interprocedural slicing using dependence graphs[J]. ACM Sigplan Notices, 2004, 23(7):35-46.

[6] BRUSCHI D, MARTIGNONI L, MONGA M. Detecting self-mutating malware using control-flow graph matching[C]//The International Conference on Detection of Intrusions and Malware and Vulnerability Assessment, 2006:129-143.

[7] CESARE S, XIANG Y. Malware variant detection using similarity search over sets of control flow graphs[C]//The International Conference on Trust, Security and Privacy in Computing and Communications. 2011:181-189.

[8] CIFUENTES C, VAN EMMERIK M. Recovery of jump table case statements from binary code[C]//The International Workshop on Program Comprehension.1999: 192-199.

[9] MENG X, MILLER B P. Binary code is not easy[C]//The 25th International Symposium on Software Testing and Analysis. 2016: 24-35.

[10] SUTTER B D, BUS B D, BOSSCHERE K D, et al. On the Static Analysis of Indirect Control Transfers in Binaries[C]//The International Conference on Parallel & Distributed Processing Techniques & Applications. 2000:1013-1019.

[11] KINDER J, ZULEGER F, VEITH H. An abstract interpretation-based framework for control flow reconstruction from binaries[C]//The International Workshop on Verification, Model Checking, and Abstract Interpretation. 2009: 214-228.

[12] TROGER J, CIFUENTES C. Analysis of virtual method invocation for binary translation[C]//The Working Conference on Reverse Engineering. 2002: 65-74.

[13] JOHNSON R, STAVROU A. Forced-path execution for android applications on x86 platforms[C]//The International Conference on Software Security and Reliability Companion. 2013: 188-197.

[14] XU L, SUN F, SU Z. Constructing Precise Control Flow Graphs from Binaries[J]. University of California. 2012.

[15] ZHAO, J. Analyzing control flow in Java bytecode[C]//The 16th Conference of Japan Society for Software Science and Technology. 1999: 313-316.

[16] 胡刚, 张平, 李清宝,等. 基于静态模拟的二进制控制流恢复算法[J]. 计算机工程, 2011, 37(5): 276-278.
HU G, ZHANG P, LI Q B, el al. Control flow restoring algorithm for binary program based on static simulation[J]. Computer Enginerring, 2011, 37(5): 276-278.

[17] 张雁, 林英. 程序控制流图自动生成的算法[J]. 计算机与数字工程, 2010, 38(2):28-30.
ZHANG Y, LIN Y. Automatic generation algorithm of the control flow graph[J]. Computer & Digital Engineering, 2010, 38(2):28-30.

[18] 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.

[19] LI L, BARTEL A, BISSYANDE T F, et al. Iccta: Detecting inter-component privacy leaks in android Apps[C]//The International Conference on Software Engineering. 2015: 280-291.

[20] WEI F, ROY S, OU X. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps[C]//The 2014 ACM SIGSAC Conference on Computer and Communications Security. 2014: 1329-1341.

[21] 辛纳（美）. Android C++高级编程——使用 NDK[M]. 北京: 清华大学出版社, 2013.
CINAR O. Pro Android C++ wih the NDK[M]. Beijing: Tsinghua University Press, 2013.

[22] Angr. Intermediate Representation[EB/OL]. https://docs.angr.io/ docs/ir.html.

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

[24] YAN S, WANG R, HAUSER C, et al. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware[C]//The Network and Distributed System Security Symposium, 2015.

[25] SHOSHITAISHVILI Y, WANG R, SALLS C, et al. SOK: (State of) the art of war: offensive techniques in binary analysis[C]//Security and Privacy. 2016: 138-157.

作者简介：

颜慧颖（1993-），女，江西吉安人，解放军理工大学硕士生，主要研究方向为软件安全。

周振吉（1985-），男，江苏沭阳人，博士，解放军理工大学讲师，主要研究方向为软件安全。

吴礼发（1968-），男，湖北蕲春人，博士，解放军理工大学教授、博士生导师，主要研究方向为网络安全。

洪征（1979-），男，江西南昌人，博士，解放军理工大学副教授、硕士生导师，主要研究方向为网络安全、人工智能。

孙贺（1990-），男，黑龙江齐齐哈尔人，解放军理工大学博士生，主要研究方向为软件逆向工程。