面向跨语言的 Android 应用静态污点分析工具的设计与实现

摘　要

随着 Android 应用分析技术的发展，出现了大量仅需要 APK 安装包即可进行分析应用 Java 代码的 DEX 字节码分析框架，然而这些框架往往忽视了安装包中，通过 JNI 接口注册为 Java 侧函数的二进制代码。现有的此类分析框架也往往存在着分析速度较慢，二进制分析逻辑和 Java 侧分析逻辑耦合严重的问题。本论文设计实现了一种更加高效的 Android 应用跨语言污点分析框架，通过引入 Value-Set Analysis 等二进制静态分析技术，增加了分析的效率。为了解决分析逻辑的耦合问题，本框架将二进制侧的 JNI 接口调用序列转换为等价的 Java 代码，通过重打包技术打包为新的 APK 文件，在提取二进制侧语义的同时实现了对现有的 Dex 字节码污点分析框架的兼容。在精确率上，本框架与 FlowDroid 的组合能够在 NativeFlowBench 数据集上和现有同类工作较为接近，同时在真实应用中的分析速度也较快。

关键词　移动应用安全　静态分析　Java 本地接口　二进制代码分析

THE DESIGN AND IMPLEMENTATION OF AN INTER-LANGUAGE STATIC TAINT ANALYSIS FRAMEWORK FOR ANDROID APPS

ABSTRACT

As the development of static analysis of Android applications, many static analysis frameworks have been developed. These frameworks usually only require apk file to perform analysis. However, apps can include native code by using Java Native Interface, whose analysis is usually ignored by most of these analysis frameworks. Only a few recent works can handle native code, but most of them face the problem of being slow and non-scalable. We propose a new inter-language Android application static analysis framework using static analysis(eg. Value-Set Analysis) instead of symbolic execution to make binary code analysis more efficient. To make binary analysis results more generic, we translate the sematic summary extracted from binary code to java bytecode, inject the generated method body into the corresponding native method on the java side, and repack it as a new apk file. In this way, we achieve compatibility among existing java bytecode analysis frameworks. Our framework reaches a comparable precision with the current state-of-the-art on NativeFlowBench. In real world apps, our framework is also relatively fast.

KEY WORDS mobile security static analysis java native interface binary code analysis

目　录

第一章　引言

1.1　研究背景

1.1.1　Android 应用中的代码

因此，Android 应用的生态研究，包括应用市场生态，恶意应用检测，移动应用欺诈等，也渐渐成为了研究的热点。然而，在大部分场景中，应用的源码往往是难以获取的。为了能够更好地分析 Android 应用，势必需要进行无源码分析，对编译后的应用安装包（即以 APK 为后缀名的文件）进行分析。

Android 应用安装包中代码主要包含为两部分。以 DEX 字节码为代表的 Java 侧代码，和基于 JNI 接口的二进制本地汇编指令代码（以下简称为本地代码或汇编指令代码）。

本地代码在现有安卓应用中分布广泛。早在 2009 年，谷歌就为 Android 发布了 NDK（Native Development Kit），使应用开发者能够将本地 C/C++ 代码集成到 Android 应用中。本地代码可以由 Java 侧应用代码加载，通过 JNI（Java Native Interface）接口与 Java 侧代码交互。开发者使用 NDK 的主要的动机是能够复用现有 C/C++ 代码，或是增加应用运行速度，提升性能。此外，不少 Android 平台的第三方库、第三方 SDK 等都带有一定量的本地代码。

然而现有的分析技术往往忽视了本地代码的分析，本文的工作基于这样的背景展开。

1.1.2　汇编分析与字节码分析

在计算机发明初期，由于手动编写机器的汇编指令代码过于繁琐复杂，人们发明了编译技术，将高级语言转化为机器语言即汇编语言执行。在此后很长一段时间里，编译后的汇编指令往往被认为是人类不可读，且难以分析的。因此各种商业软件可以通过只发布编译后程序的方式，让用户使用软件的同时不泄露自己的源代码。二进制汇编语言代码的分析一直是一个较为困难的研究方向。汇编语言代码往往是由编译器自高级语言生成后用于运行，相比于源码丢失了大量信息，即使是调试器也需要借助编译器额外生产的调试信息去理解二进制文件。分析编译后汇编指令的技术被称作软件逆向工程技术。逆向工程师往往需要耗费大量的人力才能手工恢复出部分源代码逻辑。虽然随着软件逆向工程技术的发展，越来越多的自动化逆向工程技术不断出现。但如今，对汇编指令自动化逆向工程依然远远难以达到理想的效果。

Java 语言不同于现有的直接编译语言，使用了虚拟机技术使代码能够跨平台运行，但因此程序不再是编译为二进制汇编语言，而是 JVM（Java 虚拟机）字节码。在各个平台上由 Java 虚拟机运行。也因此，Java 字节码（如 DEX 字节码）的分析和汇编语言分析遇到的挑战是完全不同的。Java 字节码作为一种支持反射等机制的“托管”语言，相比于直接在 CPU 执行的汇编指令多了非常重要的类型信息。此外汇编语言指令和数据

都混合存放在内存中，同时存在较多的动态计算跳转地址的情况，因此即使是恢复控制流结构也非常困难。

1.1.3 Android 应用静态分析框架现状

随着 Java 字节码分析技术的发展，现有的以 FlowDroid 为代表的对 Android 应用中的 DEX 字节码的静态分析框架（以下简称为 Java 侧分析框架）已经较为成熟，在各种研究中被广泛使用。但最近有研究[1–5]表明，恶意软件作者经常通过 JNI 接口调用本地二进制代码（Native Code）以隐藏他们的恶意操作（如，泄漏用户隐私等）。

当一个 Java 方法声明为本地函数时，该方法将带有“native”标记，同时没有函数体，而是由 C 语言侧（编译为汇编指令存放于 APK 文件中）实现函数逻辑。如果不分析汇编指令代码，则无法知晓函数内部逻辑。由于跨语言分析的难度和汇编指令代码的分析难度，现有 Java 侧分析框架不得不将本地代码看作黑盒，只考虑返回值和参数之间的关系，忽视了本地代码调用 Java 侧其他函数的可能。如图 1-1 中所示，当恶意应用在本地代码中隐藏相关逻辑时，仅分析 Java 侧代码将难以得到正确的结果。

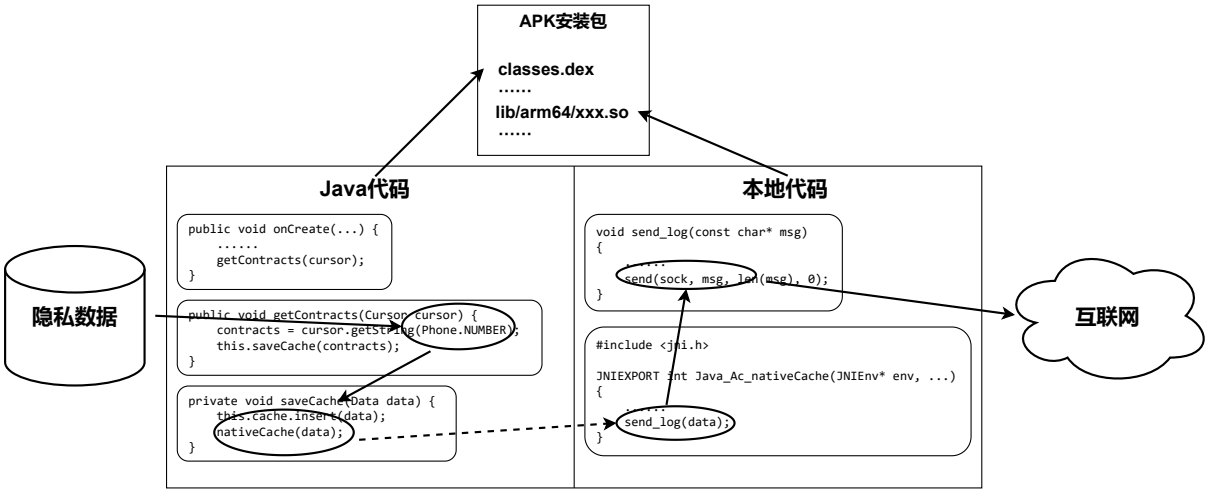

图 1-1 本地代码信息泄露示例

对 Android 应用本地代码的静态分析技术能够支持大量的上层研究更好地开展。如恶意软件检测的研究中，需要检查本地代码中隐藏的恶意行为。基于污点分析的隐私数据泄露检测，需要检测到本地代码中对 Java 侧函数的调用，本地代码中的数据流等等。

1.2 Android 应用静态跨语言污点分析

目前主流的 Android 应用静态分析工具并不支持本地代码的分析，因此很多基于静态分析的研究（例如隐私泄露检测）并不能得到准确的结果。现有的少量对 Android 本地代码的研究里，往往存在着三个问题：

1. 部分工作（如 C-Summary）需要本地代码相关的源代码才能分析。不能做到现有

Java 侧分析框架类似，只通过应用安装包（APK 文件）进行分析的效果。这一要求较大地限制了分析框架的应用场景。

2. 部分研究的功能和单种 Java 侧静态分析框架深度耦合，导致二进制分析结果不能通用的问题，如 JN-SAF[5]、μDep[6] 等工作。

3. 分析时间过长，分析效率过低的问题。对于含有较多的本地代码的应用，现有工具 (如 JN-SAF[5]、JuCify[7]) 在分析本地代码函数时往往出现分析时间过长的现象，导致在分析应用数据集时往往被上层超时逻辑提前结束分析，无法得到有用的结果。

之所以现有的工作的效率不高，是因为现有工作在二进制侧直接使用了符号执行（Symbolic Execution）作为分析技术。符号执行是一种用符号值代替输入，随着程序的执行将符号值迭代为表达式，最后使用约束求解器求解可能的值。该技术在分析复杂逻辑，尤其是次数较多的循环时，往往存在着路径爆炸的问题，即需要分析的路径数量指数级增长，极大地增加了分析时间，无法完全遍历。现有的解决符号执行路径爆炸问题的方案也往往只能略微缓解这种情况，而并不能有效解决。因此对 Android 本地代码使用基于符号执行的分析技术可能并不是一个很好的选择。

针对以上现有工作的问题，我们提出相应的解决方案：

1. 针对需要源码的问题，我们直接分析安装包中本地代码的二进制文件。因此需要深入学习现有二进制分析技术，逆向工程技术，反编译技术等，将相关领域的领先技术应用于 Android 本地代码的分析中。

2. 针对部分工作与单个分析框架耦合的问题，我们提出从本地代码中提取出对应语义信息，与应用的 DEX 字节码融合，重打包后输入上层框架进行分析的方案。该方案不但能支持现有的 Java 侧静态分析框架，而且未来出现效果更好的分析框架时也能直接支持。

3. 针对现有工作分析时间较长，分析效率较低的问题，我们通过使用基于抽象解释的静态二进制分析技术提升效率。调研发现，现有工作耗时较多的原因是选用的符号执行技术所带来的路径爆炸问题。符号执行在分析复杂逻辑，尤其是次数较多的循环时，往往存在着路径爆炸的问题，即需要分析的路径数量指数级增长，极大地增加了分析时间，无法完全遍历可能的情况。为了避免这一问题，我们选用新兴的基于抽象解释的二进制分析技术，Value-Set Analysis。基于抽象解释的静态分析技术不仅自身分析效率就较高，而且可以通过调节敏感程度参数，动态地平衡计算能力和分析精确度之间的取舍。

整个系统的工作流程如图 1-2 所示，首先二进制代码分析模块分析应用安装包中的二进制代码文件，依次分析通过 JNI 接口注册到 Java 侧的每个二进制函数，并提取得到语义信息。接着将语义信息和原应用安装包（APK 文件）输入语义信息翻译与注入模块

中，为对应的 Java 侧本地函数生成函数体，并重打包为新的 APK 文件。新的 APK 文件与正常的 APK 文件无显著差异，因此输入各类分析框架分析时，不需要对分析框架做出任何修改。

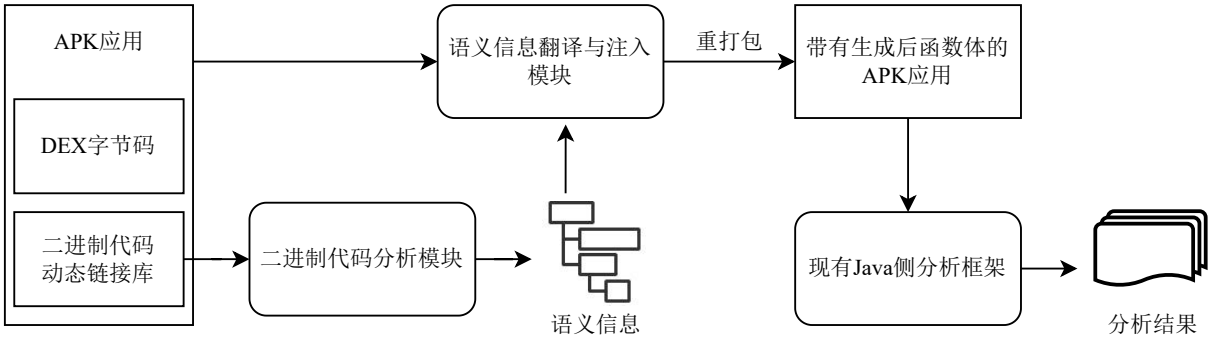

图 1-2 系统流程图

1.3 分析示例

我们通过一个例子说明本系统的分析流程。这个例子来自 JN-SAF[5] 开源的 Native-FlowBench 测试应用数据集。

假设有如代码1-1所示的 Android 应用代码。该代码实现了一个 Activity 类（对应了一个应用界面），并且在界面创建时，即 OnCreate 函数中请求读取手机状态的权限。Android 系统在权限申请成功后通过调用应用的 onRequestPermissionsResult 函数通知应用。该应用得知权限申请成功后，将调用 leakImei 函数，这个函数首先获取了敏感信息，即手机的 IMEI 号，然后将敏感信息传入了 send 函数。

然而可以看到，send 函数并没有函数体，而是带有 native 标记。结合类中对 System.loadLibrary 的调用，可以发现该函数是通过 JNI 接口使用 C 语言实现，编译为二进制代码放在应用安装包的 libleak.so 文件（根据命名规定）中。

现有的 Android 应用分析框架往往只能分析 Java 侧代码，在遇到 Native 函数时无法分析内部逻辑。由于仅获取敏感信息，而没有找到敏感信息的泄露点，并不能说明应用存在着信息泄露的问题，因此现有的 Android 应用分析框架将无法找到该应用中的信息泄露。

代码 1-1 跨语言分析 Java 侧代码示例

```
1 public class MainActivity extends Activity {
2
3     static {
4         System.loadLibrary("leak"); // "libleak.so"
5     }
6
7     public static native void send(String data);
```

```
8
9       @Override
10      protected void onCreate(Bundle savedInstanceState) {
11          super.onCreate(savedInstanceState);
12          setContentView(R.layout.activity_main);
13
14          if (checkSelfPermission(Manifest.permission.READ_PHONE_STATE) !=
                PackageManager.PERMISSION_GRANTED) {
15              requestPermissions(new String[]{Manifest.permission.READ_PHONE_STATE}, 1);
16          }
17      }
18
19      @Override
20      public void onRequestPermissionsResult(int requestCode, String[] permissions,
            int[] grantResults) {
21          switch (requestCode) {
22              case 1: {
23                  leakImei();
24                  return;
25              }
26          }
27      }
28
29      private void leakImei() {
30          TelephonyManager tel = (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
31          String imei = tel.getDeviceId(); // source
32          send(imei);
33      }
34
35  }
```

代码1-2是该应用的 C 语言部分的代码片段，这部分代码编译后会生成 libleak.so 动态链接库文件，被打包到应用安装包内。当 Java 侧代码通过 System.loadLibrary 加载该库时，由于通过 JNIEXPORT 标记导出了按照命名规范命名的函数名，因此该函数会自动对应上 Java 侧 MainActivity 类的 send 函数。当调用到 send 函数时，Java 虚拟机会依据 JNI 规范，调用该 C 语言函数。

可以看到，send 函数的有一个 string 类型的参数，对应 C 语言侧的 jstring 类型参数。该函数将该参数转换为字符串类型后，通过 LOGI（即 __android_log_print）函数打印了出去。这是一个泄露点，如果该字符串内保存着敏感信息，则此处会导致敏感信息的泄露。

为了解决本地代码分析的问题，本系统对应用中的二进制动态链接库进行分析。首

代码 1-2 跨语言分析本地代码示例

```
#define LOGI(...)  __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)

extern "C" {
JNIEXPORT void JNICALL
Java_org_arguslab_native_1leak_MainActivity_send(JNIEnv *env, jobject thisObj,
    jstring data);
}

const char *getCharFromString(JNIEnv *env, jstring string) {
    if (string == NULL)
        return NULL;

    return env->GetStringUTFChars(string, 0);
}

JNIEXPORT void JNICALL
Java_org_arguslab_native_1leak_MainActivity_send(JNIEnv *env, jobject thisObj,
    jstring data) {
    LOGI("%s", getCharFromString(env, data)); // leak
    return;
}
```

先分析静态绑定关系，找到需要分析的二进制代码入口点。其次依次分析绑定的每个二进制函数。这一部分分析无需提供应用源代码，而是直接分析动态链接库中的汇编指令代码。分析后得到 JSON 格式的语义信息，如代码1-3所示

代码 1-3 跨语言分析语义信息示例

```
{
    "apk_name": "native_leak.apk",
    "globals": [],
    "mth_logs": {
        "Lorg/arguslab/native_leak/MainActivity;\tsend\t(Ljava/lang/String;)V": {
            "4195940\tGetStringUTFChars": [
                null,
                [
                    [
                        "arg",
                        4195912,
                        2
                    ]
                ],
                [
                    0
                ]
            ],
```

```
            "4195980\tAndroidLogPrint": [
                [
                    4
                ],
                [
                    "leak"
                ],
                [
                    "%s"
                ],
                [
                    [
                        "ret",
                        4195940,
                        "GetStringUTFChars"
                    ]
                ]
            ]
        }
    }
}
```

可以看到，语义信息能够反映，函数的参数被传入 GetStringUTFChars 函数，而 GetStringUTFChars 函数的返回值被传入 AndroidLogPrint 中被泄露出去。最后经过语义信息转 Java 模块，我们为 send 函数生成函数体，如代码1-4所示。可以看到，虽然代码的局部变量的命名出于调试考虑较为复杂，但能够正确反映字符串泄露的语义。代码中将 __android_log_print 转换为了 Log.d 函数，该函数作为日志相关函数，也是一个信息泄露点。函数体的生成利用了静态分析时不考虑分支判断条件，而是直接假设所有分支都可能执行的特点，使得局部变量在 if 分支后，静态分析框架认为它有多种可能取值。分析到 Log.d 函数时，由于参数可能取值为 send 函数的 String 类型参数，因此反映了该参数可能被泄露的语义。

代码 1-4 语义信息转 Java 代码示例

```
public static void send(String $GetStringUTFChars4195940) {
    String $AndroidLogPrinta4195980;
    int opaque = getOpaqueInt(); // 生成不透明值，使多个IF分支都可能被执行
    if (opaque == 1) {
        $AndroidLogPrinta4195980 = "leak";
    } else if (opaque == 2) {
        $AndroidLogPrinta4195980 = "%s";
    } else if (opaque == 3) {
        $AndroidLogPrinta4195980 = $GetStringUTFChars4195940;
```

```
10      }
11      Log.d($AndroidLogPrinta4195980, $AndroidLogPrinta4195980);
12  }
```

1.4 论文工作

本论文的主要贡献如下：

- **提出了一种兼容现有 Java 侧分析框架的二阶段跨语言分析方法。**通过提取语义信息，重打包原有 APK 并置入生成后的 Java 代码的方法，不仅实现了对现有分析框架的支持，当未来出现效果更好的分析框架时也能直接支持
- **通过使用静态分析技术分析汇编语言代码，相比于符号执行技术，提升了分析效率。**现有工作在二进制分析上大多使用了符号执行技术。随着二进制的静态分析技术的成熟，自动化静态分析技术也渐渐出现在现有的开源二进制分析框架中。本工作将现有的二进制静态分析技术带入 Android 应用静态跨语言分析的场景下，相比于符号执行技术，提升了分析效率。
- **详尽地分析了 Android 跨语言分析场景下的特有问题。**通过提取控制流图形状的方法，解决了如何保留控制流的语义的问题。通过参考 JNI 接口的操作语义（Operational Sematics）较为准确表述了语义信息的提取转换过程。

1.5 论文结构

本文共分为六章，内容组织结构如下：

第一章为引言，介绍了论文的研究背景和主要工作。

第二章介绍相关工作，包括基于抽象解释的二进制分析技术、相关跨语言分析技术、和现有的同类型工作。

第三章介绍框架的设计考量，相关问题的解决思路。第四章介绍框架的详细实现与使用的算法。

第五章对实现的框架进行了实验与评估。

第六章对其他可能相关的工作进行总结，并包含未来工作的展望。

第二章　相关工作

2.1　现有工作

2.1.1　Android 应用 Java 侧污点分析框架

随着 Java 字节码分析技术的发展，现有的以 FlowDroid 为代表的对 Android 应用中的 DEX 字节码的静态分析框架（以下简称为 Java 侧分析框架）已经较为成熟，不仅出现了大量开源分析框架，而且出现了一些工作评估这些框架的效果。例如：ReproDroid[8] 这篇工作对现有的大量 Android 应用分析框架在各种数据集上进行了对比研究。该工作主要集中在 Java 侧语言的分析，但对 Java 侧框架的选用有较好的参考价值。其中参与对比的框架有：Amandroid[9]、DIALDroid[10]、DidFail、DroidSafe[11]、FlowDroid[12]、IccTA[13]。

由于跨语言分析的难度和汇编指令代码的分析难度，现有 Java 侧分析框架不得不将本地代码看作黑盒，只考虑返回值和参数之间的关系，忽视了本地代码调用 Java 侧其他函数的可能。例如，FlowDroid[12] 这篇工作中的 Implementation 一节对 Native Calls 有这样的描述：“对于基于 Java 的分析，这些不能分析的函数（本地代码函数）只能被看作黑盒处理。”，“FlowDroid 对大多数的常见的本地代码函数有着对应的污点传播规则，例如 System.arraycopy。”，“对于没有这样规则的本地代码，FlowDroid 只能采取常规的策略：如果某个参数带有污点，则把污点传播到所有的参数和返回值中。这种方法既不是完善（Sound）的也不是最精确的，但在黑盒的模型下这可能是最实际的策略。”

2.1.2　Android 跨语言本地代码污点分析框架

随着二进制分析的成熟，不少研究者也投入到 Android 跨语言本地代码污点分析的研究中。其中较为深入、有代码实现的相关工作如下：

1. **JN-SAF**[5]：该框架基于符号执行实现，因此也存在路径爆炸导致分析效率过低的问题。此外代码实现与 Augus-SAF 这一 Java 侧代码分析框架深度耦合，无法通用。

2. **Jucify**[7]：该框架提出通过分析二进制侧可以完善 Java 侧代码的 CallGraph。代码也是基于符号执行实现，因此也存在路径爆炸导致分析效率过低的问题。该工作对二进制侧的分析较为粗浅，同时也未提及 C 语言和 Java 语言在语言特性上的差异所导致的问题。

3. **μDep**[6]：该框架通过 fuzzing 技术，将应用内的二进制库在模拟器中运行起来，并通过不断修改参数和观察变化，得到二进制函数的污点信息传播情况。该框架仅对接了 DroidSafe 这个上层 Java 侧分析框架。此外，这种方法也没有考虑函数调用的情况，该工作通过另一种方式解决函数调用的问题。

4. **C-Summary**[14]：该工作实现的框架为源码级分析框架，基于 Facebook 的 Infer[1]框架实现。该工作无法做到仅通过应用安装包进行分析。

此外这些工作都没有考虑到二进制的控制流对语义的影响，如后文3.2节所述。

2.2 基于抽象解释的二进制分析

2.2.1 二进制程序分析

二进制汇编语言代码的分析一直是一个较为困难的研究方向。汇编语言代码往往是由编译器自高级语言生成后用于运行，相比于源码丢失了大量信息，即使是调试器也需要借助编译器额外生产的调试信息去理解二进制文件。试图理解分析汇编语言的研究往往被归类于逆向工程的范畴。分析汇编语言的基本步骤包括反汇编，CFG（Control Flow Graph）恢复，非直接跳转解析等。更进一步的分析包括类型信息恢复，控制流结构恢复，反编译等等。各种 C/C++ 语言的需求和特性也给汇编语言的分析带来了额外的困难，包括内存管理，面向对象，虚函数等。

此外，Java 字节码（如 DEX 字节码）的分析和二进制汇编语言分析遇到的挑战是完全不同的。Java 字节码作为一种支持反射等机制的“托管”语言，相比于直接在 CPU 执行的汇编指令多了非常重要的类型信息。此外二进制汇编语言指令和数据都混合存放在内存中，同时存在非常多的动态计算跳转地址的情况，即使是恢复控制流结构也非常困难。

在 Android 本地代码的分析场景中，二进制代码对 JNI 接口的使用也是分析的难点。除了具有二进制汇编指令代码分析困难的特点外，Android 本地代码还使用 JNI 接口与 Java 侧进行交互，给分析带来了额外的挑战。JNI 的全称是 Java Native Interface，它允许 Java 侧代码与其他语言编写的程序或库函数进行交互，支持的语言包括 C、C++、汇编等。JNI 接口规定了 Android 本地代码的注册规则，本地代码的入口函数在注册后与 Java 侧声明为 Native 的函数一一对应。此外，本地代码也可以通过 JNI 接口调用 Java 侧函数，传递数据，这也是忽略本地代码为何会导致上层分析数据流缺失的重要原因。如何将本地代码对 JNI 接口的使用，反映给上层分析框架，是本项目主要的研究内容之一。

2.2.2 二进制分析框架

现有的商业化二进制分析框架有 IDA[2]、Jeb[3]、Binary Ninja[4]等。它们虽然较为成熟，但其反编译器并不开源，往往难以基于反编译的中间结果进行深入的分析，最近各个

[1]https://fbinfer.com/

[2]https://hex-rays.com/ida-pro/

[3]https://www.pnfsoftware.com/

[4]https://binary.ninja/

框架都在推出自己的高层次 IR 以缓解这一问题。现有的主流开源二进制分析框架包括 Angr[15]，radare2[1]/rizin[2]，Ghidra[3]等。其中 Ghidra 由于其带有一个较为成熟的反编译器，因而相较于其他框架更为成熟。rizin 框架也从自己开发的反编译器渐渐转向对接 Ghidra 的反编译器。此外还有一些独立的开源反编译器 RetDec[4]等。随着二进制研究渐渐成为热门方向，不少新的二进制分析框架也在不断涌现，如 rev.ng[5]等。现有研究[16] 显示现有的反编译器已经越来越成熟，然而研究者们却鲜有基于反编译器开展更进一步的上层分析。

此外，现有的学术界二进制分析技术相较于开源的二进制分析技术领先很多，但学术界的二进制分析技术的成果往往选择不开源，倾向于商业化。如近期涌现的大量基于 VSA 技术的二进制分析相关的工作：BDA[17]、BinPointer[18]、BPA[19] 等。

2.2.3 抽象解释

现有的主流程序静态分析技术都是基于抽象解释[20,21] 实现的。简单来说，抽象解释是对程序真实运行时可能状态进行抽象合并。通过抽象的方式，可以将单个函数所有可能的执行，抽象为单一的函数对象（以上下文不敏感分析为例）。函数运行实例中的同一变量也被抽象为了单独的一个变量。然后，为每个这种变量维护一个可能取值的集合作为运行时所有可能真实取值的抽象。这种做法将需要分析的对象数量从无限缩减为和函数数量，变量数量相关的有限值。但分析时也有可能因为分析的不精确，而完全无法确定部分变量的值。相关的知识在南京大学的《软件分析》课程[22] 中有较为清楚的解释。

2.2.4 二进制静态分析框架

基本的二进制分析框架往往只提供了基于汇编指令的接口，或是进一步提供字符串识别，交叉引用等功能，支持二进制静态分析的框架较少。基于抽象解释的静态分析技术，由于需要把函数调用时不同的局部变量映射到同一抽象对象上，想要应用到二进制程序上必须基于一定的局部变量的恢复技术，否则变量如果按照汇编指令对栈寄存器的使用去分配空间，则不同的函数栈会分散在不同的地址出，导致在合并状态时无法简单地按地址合并。现有的大量抽象解释的静态分析技术正是基于变量实现，想要将这些技术应用于二进制分析上，类型恢复、变量恢复等技术是十分必要的。Angr 作为新兴的开源二进制分析框架，其最完善的是符号执行技术，而符号执行技术并不需要基于栈变量的恢复。基于笔者对 Angr 的使用，Angr 的 VSA 实现忽视了这一点（截至 2022 年 5 月）。所幸我们的分析可以通过细分极大地减少需要分析的代码数量，Angr 的这一缺陷

[1]https://rada.re/

[2]https://rizin.re/

[3]https://ghidra-sre.org/

[4]https://github.com/avast/retdec

[5]https://rev.ng/

影响不大。可能正因如此，Angr 目前暂时并未积极维护 VSA 的实现，而是重点集中在反编译器的开发上。

Value-Set Analysis，作为一种高效的基于抽象解释的新兴二进制静态分析技术，目前还没有较为完善的开源实现。目前有相关实现的开源分析框架主要有 Angr、BAP[1]。然而目前这两个仅有的开源 VSA 实现也较不完善。BAP 基于 OCaml 语言的 BAP 框架实现，代码量只有 2440 行代码，且分析速度较慢。[2]，而在 BDA[17] 中提到，他们向 Angr 的作者确认过，Angr 的 VSA 无法在大型的复杂程序上跑起来，在 BinPointer 中提到，Angr 的 VSA 即使在单个函数内，效果也不佳。此外，腾讯公司的科恩实验室近日里发布了 BinAbsInspector 框架[3]，该框架基于 Ghidra 反编译器得到的 IR（Ghidra 的 P-Code）和反编译器恢复的栈变量上进行抽象解释分析，有望成为较好的开源二进制抽象解释分析工具之一。

2.2.5 Reaching-Definition 与 DDG（Data Dependency Graph）

Reaching-Definition Analysis，到达定值分析，是传统的数据流分析之一，基于抽象解释技术。该技术能够分析程序运行过程中，变量的使用点使用了哪些变量的赋值点所定义的值。随着二进制分析技术的发展，现有的二进制分析框架，如 Angr，也对二进制代码实现了 Reaching-Definition 分析。使用该技术，可以从单个汇编指令对寄存器的使用，追溯到寄存器中被使用值的来源。从而得到数据依赖关系。通过进一步对栈上变量信息的恢复，并标记存储和使用变量值的指令，也可以追溯流经栈内存的信息流。

基于汇编函数恢复得到的控制流图，在其上应用 Reaching-Definition 分析，可以进一步得到汇编函数中的数据依赖图（DDG，Data Dependency Graph）。通过在数据依赖图中遍历，可以解析 JNI 接口的参数，解析返回值的数据流来源，并表示到生成的语义信息中，从而使得上层分析框架能够精确地进行数据流分析。这种方式也是一种进行语义信息提取的思路。在本系统的实现中，仅在解析可变参数函数的参数使用情况时使用了到达定值分析技术。

2.2.6 Value-Set Analysis

Value set analysis（VSA）是一种高效且精确的汇编指令的静态分析方式，在各种分析汇编指令的研究中被广泛使用，包括汇编语言中指针地址的别名分析，二进制软件漏洞静态检测（如，检测栈溢出漏洞）等。相比于其他基于抽象解释的汇编指令分析技术，VSA 既不需要假设内存和寄存器都是固定大小的基本单元，也不需要假设内存读写的范围必须不重叠。因此，VSA 更加贴合实际场景，满足真实应用的分析需求。

VSA 同时追踪了寄存器和内存中存储的数值（数值分析），和内存访问时地址的指

[1]实现于 https://github.com/draperlaboratory/cbat_tools

[2]2022 年 5 月 8 日，使用 cloc 工具统计 https://github.com/draperlaboratory/cbat_tools/tree/master/vsa/value_set/lib/src 下的代码

[3]https://github.com/KeenSecurityLab/BinAbsInspector

向（指针分析）。对程序的数值分析促进了地址访问的解析，而对地址访问的精确处理使得程序的数值分析更精确，两者相互促进。使用 VSA 分析汇编指令函数后，可以得到控制流图中某个程序点处，各个寄存器可能的取值范围。该信息对内存相关的分析尤其关键，使得处理动态计算地址的访问成为可能。

VSA 最早由 Gogul Balakrishnan 和 Thomas Reps 提出于《Analyzing Memory Accesses in x86 Executables》[23] 中提出。随后他们对早期的 VSA 做出了不少改进，发布了一系列论文[24]。Thomas Reps 的文章《WYSINWYX: What You See is not What You Execute》[25] 中对 VSA 技术做了较为完整的综述和总结，因此，如今的论文提到 VSA 时，往往引用的是这一篇。

2.3 跨语言分析

2.3.1 JNI 接口的语义

随着编程语言语义学的发展，出现了不少用数学符号证明和表达各种编程语言的语义的工作。然而这种方式目前暂未普及，现有的往往是描述式的规范（prose specifications），通过文字描述各种编程语言操作对应的行为。然而使用语义学的方法形式化地表达语言特性的语义有很多优点，如不容易出现二义性，可以完善地考虑到各种编程语言特性交互时的复杂情况等。此外，基于现有的语义规范，静态分析工具也可以形式化证明他们的分析能力，例如保证自己的分析一定涵盖了所有真实执行的可能性（Soundness）。现有的大多数工作往往只关注单个编程语义内部的语义，鲜有关注跨语言接口（FFI，Foreign Function Interfaces）的工作。所幸对于 JNI 跨语言接口，有 JNI Light[26] 这篇工作。

该类工作对相关的上层研究有着较好的指导作用：首先该类工作对相关操作的语义定义需要建立在合适的抽象模型上，而为了能够更好地开展后续的证明推导工作，该类工作往往会选择较为简洁的抽象模型。这种抽象层次对相关的软件系统设计有着较好的指导作用。其次，本文实现的语言信息到 Java 语句的转换可以参考该类工作定义的语义，从而保证转换前的接口调用和转换后的 Java 语句的语义一致。

2.3.2 C 语言转 Java 语言技术

C 语言到 Java 语言的转换，看似是难以实现的，因为 C 语言和 Java 语言在语言特性上的差异较大，有些转换看似无法实现。比如 Java 侧没有指针，只有引用，用到复杂指针的 C 语言代码如何翻译为 Java 语言的问题。然而，C 语言和 Java 语言历史悠久，在 Java 语言刚出现时，由于其跨底层架构、跨操作系统平台的特性，催生了大量从 C 语言迁移到 Java 语言的需求。因此 C 语言到 Java 语言的转换方面也出现了不少的相关研究[27–29]。详细的转换方法读者可以参看引用的对应论文。

语义信息提取技术，在面对一些复杂的情况时，也往往需要考虑将部分 C 语言对象翻译到 Java 语言中，因此需要用到相关的语言转换技术。如部分程序在 C 语言侧分配

内存后，将指针强制转换为 JLong 类型而返回到 Java 侧作为句柄，在其他 API 调用时传回并转换回对应类型的指针类型使用。这种较难处理的情况就需要将对应的结构体对象提升到 Java 侧的类，从而更完善地分析数据流关系。

第三章　系统设计

3.1　跨语言分析

想要让现有框架能够兼容 C 语言侧二进制代码，就不得不面对跨语言分析的问题。随着使用二进制分析技术恢复越来越多的信息，如变量在内存中的分布后，上层分析就会越来越接近有源码的情况。然而，即使是有源码的分析，由于 Java 代码和 C 语言代码语言特性上的差异，Java 侧分析框架也无法直接理解 C 语言侧的代码，这一点在 JNI Light[26] 这篇文章中也有提到。

Java 侧代码和 C 语言侧代码语言特性上的不同导致出现了很多分析上的问题，如 C 语言侧的指针如何对应到 Java 语言的引用、C 语言指针运算在 Java 语言中没有对应的问题等。对这些问题，有三种可能的解决方式如下：

1. 重新实现一套分析能力足够强的框架，可以在同时兼容 Java 语言侧和 C 语言侧语义的抽象层次进行分析，使用类似于 JNI Light[26] 中提出的块内存抽象模型。然而，从零实现一个分析框架的难度、所需要的工程上的开销都过于巨大，导致在现实中难以实现。

2. 完整地提取语义信息，基于现有的反编译技术恢复出 C 语言代码，再借助现有的 C 语言到 Java 语言的转换编译器（transpiler，source-to-source compiler），以及现有的 C 语言到 Java 语言转换的研究成果，将 C 语言代码完整地转换到 Java 侧，交由上层分析框架分析。然而现有的反编译技术还是不够完善，尤其是在面对复杂的 C++ 语言特性时，反编译的效果很不好。很有可能大量的反编译结果因为结果不够准确（如包含过多的强制类型转换），导致无法进行翻译。

3. 仅提取部分 JNI 接口函数调用序列传递到上层框架。这种方法的思想是，C 语言侧对 Java 侧进行的操作都需要通过调用 JNI 接口函数实现。现有的论文大多采用的都是这种方法（如 C-Summary[14]、Jucify[7] 等）。这种方法的好处是无需基于现有的反编译技术，只需构建控制流图后直接使用符号执行等分析方法。且生成的语义信息也更为简洁。

由于现有的二进制反编译技术还较为不成熟，因此目前本文在这个问题上和现有工作一样，采取了第三种方法，将相关的调用序列转换为 Java 语句。此外，出于生成的 Java 代码的简洁性考量，第二种方法也未必优于第三种方法。正如下文3.2节所述，基于第三种方法，在部分场景下结合第二种方法的思想，或许是解决该跨语言问题的最佳策略。

3.2 控制流的处理

仅提取 JNI 调用的方式，则会导致控制流缺失的问题。下面用一个例子说明该问题。假设使用 JNI 的 C 语言代码如3-1所示，此处暂时不考虑二进制分析，用 C 语言代码表示编译后的二进制指令，同时省略了部分不相关代码。

代码 3-1 控制流处理问题代码示例

```
void Java_Main_nativeMethod(JNIEnv *env, jobject thisObj) {
    jclass cd = env->GetObjectClass(data);
    jfieldID fid = env->GetFieldId(cd, "f", "I"); // I 表示int类型
    jint c = env->GetIntField(thisObj, fid); // c = this.f
    if (c == 1) {
        env->CallObjectMethod(...);
    } else if (c == 2) {
        env->CallVoidMethod(...);
    }
    ...
}
```

在仅提取 JNI 调用后，我们得到如代码3-2所示的调用序列。可以看到，相关的 if 分支结构丢失了，而且翻译后的两个函数调用变成了必然执行，真正的语义应该是两个函数调用只有一个会被执行，而且可能都不被执行。我们选择仅提取 JNI 调用的部分原因是现有的反编译技术、对二进制的控制流分析技术不够成熟。

代码 3-2 控制流处理问题语义信息示例

```
Java_Main_nativeMethod:
    o1 = GetObjectClass(arg1)
    o2 = GetFieldId(o1, "f", "I")
    o3 = GetIntField(thisObj, o2)
    CallObjectMethod(...)
    CallVoidMethod(...)
    ...
```

然而该问题也可以在不进行深入的控制流恢复的情况下解决。注意到在二进制分析中，需要先恢复控制流图（Control Flow Graph，CFG），基于控制流图进行分析。而在基于抽象解释的静态分析中，也是往往是基于控制流图进行分析。也正是控制流图表达了各种循环和条件控制结构。语义信息提取时可以将控制流图的形状导出，生成 Java 代码时首先生成对应的控制流图形状框架，再将对应的调用插入到对应的基本块中。

代码3-1对应的控制流示意图如图 3-1 所示。可以看到，控制流图可以有效地表达 CallObjectMethod 和 CallVoidMethod 这两个调用只会调用一个的同时，也可能都不会调用的语义。通过同时提取控制流图可以较好地解决该问题。但该优化方案暂未在本次工作中实现。

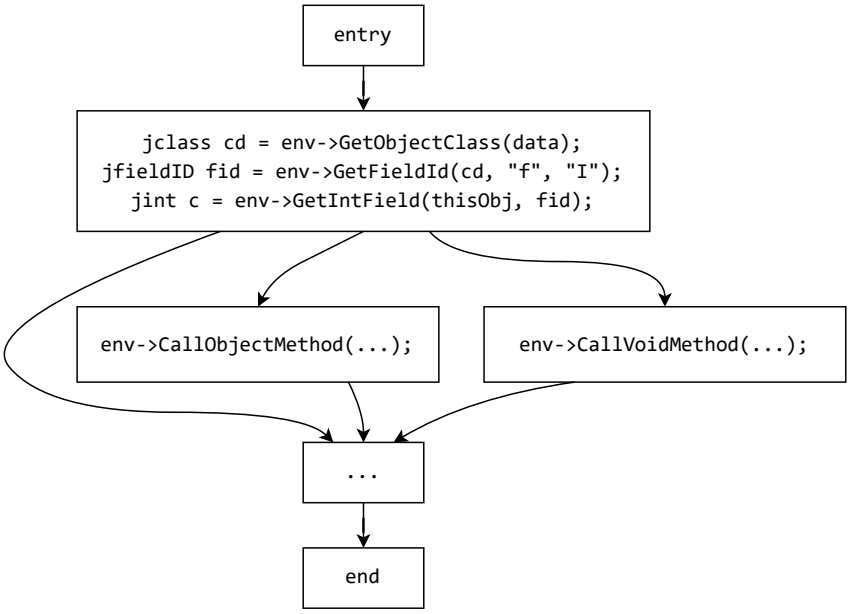

图 3-1 示例代码的控制流示意图

3.3 分析目标的精确化

在《Exception Analysis in the Java Native Interface》[30] 中提到，现有的 JNI 代码往往存在着这样一种编程模式：即为了让 Java 侧代码使用原有 C 语言库函数，在原有 C 语言库基础上编写 JNI 接口代码。因此往往可以将 C 语言侧的代码分为两部分，库函数代码和接口代码。只有接口代码部分有 JNI API 调用，且接口代码往往远少于库函数代码。受到其启发，在本系统的分析场景下也可以采取这种分割策略，不进入复杂的库函数代码内进行分析，而是仅分析接口代码。

虽然接口代码看似没有真正实现功能的内部库函数重要，但是在我们的跨语言污点分析中，往往库函数只是数据转换和处理，真正传播污点数据，影响 Java 侧数据流的往往是 JNI 接口代码。例如，Android 应用为了处理多媒体数据而引入了视频图像处理相关库，处理压缩包而引入压缩算法库，在这些库函数内部都不会造成新的数据流，仅是进行运算。此外接口代码还可能调用 Java 侧函数引入新的调用边，对数据流有潜在的影响。因此，通过只分析接口代码能极大地减少分析的目标，从而极大地增加分析的速度。

3.4 语义信息翻译与注入

3.4.1 需要翻译的 JNI 调用

实现语义信息翻译的初衷，是部分 JNI 代码在语句上能够等效地对应 Java 语句的代码。然而如何更加形式化和严谨地定义这个想法也是非常重要的。JNI Light[26] 这篇工作中基于操作语义定义了 JNI 接口的语义，是非常有价值的参考对象。参考对比文中 JNI 接口的语义和对应 Java 指令的语义，得到重点关注并支持的 API 有：

- 函数调用相关。如 CallObjectMethod 系列[1]函数，以及 CallNonvirtualObjectMethod，CallStaticObjectMethod 系列函数。
- 成员变量相关。如 GetObjectField 系列函数获取成员变量，SetObjectField 设置成员变量，以及 GetStaticObjectField、SetStaticObjectField 系列函数。
- ID 相关。包括 GetMethodID、GetStaticMethodID、GetFieldID、GetStaticFieldID、FindClass。这些 API 虽然没有直接对应的语义，但却是上述两种 API 调用时需要传递的参数。
- 对象创建相关，如 NewObject 会在创建新对象的同时调用构造函数。
- 异常抛出相关，如 Throw、ThrowNew 等 API 用于抛出新的异常。

但是 JNI 中除了有这些能够明确对应到 Java 操作的 API 外，还有一些 JNI 操作特有的 API 需要特殊考虑：

1. 异常处理。包括 ExceptionOccurred、ExceptionDescribe、ExceptionClear、ExceptionCheck。

   值得注意的是 JNI 的异常处理和 Java 的异常处理有较大的差别。Java 语言在遇到异常时将立刻中断当前函数的调用，开始回溯。而在通过 JNI 调用的 C 语言侧函数中如果出现了异常，并不会打断 C 语言侧代码的执行，而是需要程序员手动立刻处理。JNI 标准规定，如果当前出现了异常，C 语言代码将只能调用异常处理相关的 API，将异常处理完毕后才能调用其他大多数 API。容易看出这种要求较容易因为程序员的疏忽而造成错误的调用，因此现在也出现了较多工作研究如何自动化发现这种类型的编程错误，如[30] 等。总之，鉴于两者异常处理的差别，本文的实现中没有处理相关的 API 语义信息的提取和转换。但是 Throw、ThrowNew 等 API 由于其明确的语义，本文的实现中进行了相关处理。

2. 引用计数。包括 NewGlobalRef、DeleteGlobalRef、NewLocalRef、DeleteLocalRef。

   引用计数涉及到 Java 的堆内存管理机制使用的算法。Java 语言中通过使用引用计数来释放不需要的对象，当一个对象引用计数为 0 时，说明该对象无法被程序引用到，因此可以释放所占的内存。然而 Java 侧代码并不知道 Native 侧是否保存了 jobject 的引用，因此如果 Native 侧函数需要跨调用保留对象，防止其在 Java 侧因不需要被垃圾回收销毁，需要用这些 API 告知 Java 虚拟机自己持有的引用。由于没有明确对应的 Java 侧语义，本文的实现中也没有进行处理。但引用计数往往和在 Native 函数中跨调用保存 Java 侧对象搭配使用。可以辅助判定全局变量的产生。

[1]由于 JNI 接口根据不同返回值类型给相同功能的接口定义了一系列函数，如 Call<type>Method 包含 CallObjectMethod、CallBooleanMethod、CallIntMethod 等函数，因此此处称为一系列函数

3.4.2 多取值参数的翻译

二进制静态分析的结果中有可能存在单个函数调用时参数有多种可能的取值的情况。这也给我们的语义信息翻译带来了新的挑战。

1. 函数调用参数存在多个值。

   在静态分析中，函数参数解析时存在多种可能的取值是非常常见的情况。可以在函数调用前为这种参数单独创建局部变量，并使用 if 分支在不同的分支内赋值。当 Java 侧分析框架进行分析时，也同样会在函数调用时分析发现参数可能取多个值。

2. ID 相关 API 的字符串参数错误或不精确。

   如代码3-3所示，在静态分析时，字符串参数无法精确解析为单个值，而是有多种可能的取值。例如翻译类的成员函数调用时，需要基于精确解析的 MethodID 在 Java 侧找到这个函数，才能生成调用语句。如果无法精确解析 ID 则会影响到之后使用到相关 ID 的语句的生成。这种情况较难处理，但并不多见。本文的实现中遇到这种情况时会报出相应的错误信息。同理，当在生成语义信息时若无法在 Java 侧找到对应的对象，则也报出错误信息。

代码 3-3 字符串参数解析示例

```
1 char* a = "class1";
2 if (...) {
3     a = "class2";
4 } else if (...) {
5     a = “class3”;
6 }
7 jclass clz = env->FindClass(a); // 生成时无法精确找到对应的类
8 jmethodID gd = env->GetMethodID(clz, "getData", "()Ljava/lang/String;"); //
      进一步无法精确找到函数
9 ...
```

第四章　系统实现

本系统主要分为两个模块，二进制分析与语义信息提取模块，语义信息翻译与注入模块。整体系统流程如图 1-2 所示，即二进制分析模块分析应用安装包中二进制代码得到语义信息，语义信息翻译与注入模块将语义信息转换为 Java 函数体注入原有 Native 方法，并重打包为新的应用安装包。在细节上系统实现和图示有所差异，如二进制分析时也会从 DEX 代码中提取带有 Native 声明的函数。

4.1　二进制分析与语义信息提取

二进制分析部分由于底层选用了 Angr 分析框架，因此该部分和底层框架使用相同语言，即 Python 语言实现。整个系统实现包含 30 个 python 文件，除去注释、空行等后统计代码行数为 2423 行。

二进制分析与语义信息提取模块的内部逻辑为，首先通过 APK 解包与静态解析模块依据命名规范找到需要分析的导出函数入口地址，然后输入语义信息提取模块依次分析每个函数。

4.1.1　APK 解包与静态注册解析

APK 解包部分在设计上针对效率进行了一系列优化。APK 文件由于在底层类型上属于 ZIP 压缩包，因此我们使用 Python 标准库中的 zipfile 模块高效地进行处理。在解包过程中没有使用临时文件夹等涉及文件系统的方式，而是直接解压到内存中进行分析，进一步提升了效率。

二进制动态链接库解析模块同样有不少针对效率的优化。由于静态解析逻辑仅需要提取导出函数，因此我们没有使用 Angr，而是使用高效的 ELF 文件解析库 pyelftools，直接提取二进制文件中的“.dynsym”段。该段由动态链接机制使用，去除符号信息时不会对去除该段，否则程序将无法正常加载该动态链接库。

DEX 字节码分析使用了现有的 androguard 库实现，读取后，遍历每一个 Java 方法的声明，找出被标记为 Native 的方法，即通过 JNI 接口在二进制库中实现的 Java 方法。

当 Java 侧和二进制侧提取了相关的函数信息后，静态解析模块负责依据 JNI 标准的命名规定，将 Java 侧 Native 方法映射到二进制侧的导出函数中，保存映射关系作为结果，如图 4-1 所示。该模块在分析大量应用数据集时不断完善，支持了重载方法（类名和函数名相同，但函数参数不同）的解析，特殊符号的正确转义等特性。此外，该静态解析逻辑可以单独调用，生成解析结果报告，同时可以指定是否执行耗时较多的 DEX 字节码分析。不分析 Java 侧函数时虽然无法进行映射关系分析，但是可以非常高效地打印出二进制侧相关的待解析 Java 符号。

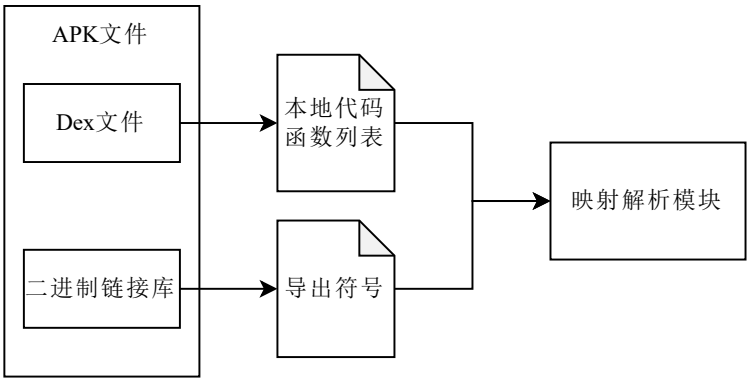

图 4-1 静态注册解析

4.1.2 语义信息提取

首先基于上一子模块的分析结果，我们将成功映射的 Java 方法、二进制导出函数按照所在的二进制动态链接库归类，依次分析。对每个动态链接库，首先创建控制流图，然后基于控制流图分析每个需要的导出函数。如算法1所示。

算法 1 本地函数分析

输入： 按所在二进制文件归类的解析结果

输出： 每个函数的语义信息

```
1: 对每个二进制文件
2:     计算整个二进制模块的控制流图
3:     对每个需要分析的导出函数
4:         基于控制流图对函数进行 VFG 分析
5:         导出得到的语义信息
```

对单个二进制函数的分析基于 Angr 框架的二进制静态分析实现，使用了 Value-Set Analysis 分析（在 Angr 中被称为 VFG，Value Flow Graph）。而我们为了能够让 Angr 的这一分析能够更加高效，做出了大量的微调。包括参数的调整、对内部代码的少量修改。如我们将 VFG 的 max_iterations 参数和 max_iterations_before_widening 参数更改为默认值的一半，分别是 20 和 4。

分析时，基于3.3节提出的思想，将 C 语言侧代码依据是否与 JNI 接口相关，分为了库函数代码和接口代码。因为 C 语言侧想要对 Java 侧交互，必须通过 JNI 接口进行，我们只对最为关键的 JNI 接口的相关代码进行分析即可得到较好的结果，也因此，我们得以在不处理复杂的 C++ 虚函数等特性的同时不增加分析的复杂度。

具体分析单个函数时，我们首先依据 Java 侧函数签名，和 JNI 标准，为二进制函数创建相应的参数，并创建一个 Angr 的 Call State 表示调用前的内存和检查器状态。接着在这个 State 上启动 VFG 分析。注册的二进制函数的第一个参数往往是 JNIEnv 结构体指针，该结构体内保存着 JNI 接口函数的函数指针，用于调用 JNI 接口函数。对于这个参数，需要在内存中布置好函数指针后传入起始地址，函数指针指向实现的 SimProcedure

类模拟 JNI 接口函数的效果，同时记录下调用情况。启动后，当函数使用 JNIEnv 参数内部的函数指针时，就会调用到我们的建模函数。整体分析流程如图 4-2 所示。

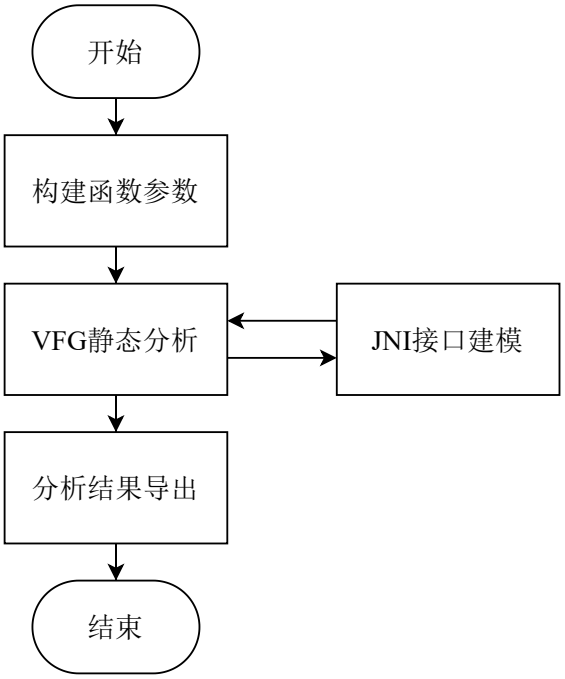

图 4-2 VFG 分析流程

4.1.3 JNI 接口建模

由于在静态分析过程中可能存在迭代的情况，即程序中存在复杂循环时会沿着循环路径分析多次，因此我们的建模函数也可能被调用多次。为了解决这一问题，我们通过获取函数的 callsite，即汇编 call 指令所在的地址，唯一地区分函数调用。

Angr 框架中，对函数的建模通过继承 SimProcedure 类实现。JNISimProcedure 类继承自 SimProcedure 类，保存通用处理逻辑。每个 JNI 接口函数都对应着 JNISimProcedure 的一个子类。大多数子类只需要重载父类的表示建模的函数的参数和返回值的成员变量即可，直接使用继承自父类的通用的运行逻辑。如代码4-1所示，其中“reference”代表 JNI 接口返回的不透明对象类型，如 jclass，jobject 等。

代码 4-1 建模函数类代码

```
1 class GetMethodID(JNISimProcedure):
2     arguments_ty = ('jenv', 'reference', 'string', 'string')
3     return_ty = 'reference'
```

JNISimProcedure 类中的 run 函数实现了大多数建模函数的通用运行逻辑。类实例化时会根据子类重载的“arguments_ty”成员设置好当前函数的原型，被调用时 Angr 框架会依据二进制代码的调用约定，从对应的寄存器中获取参数，然后调用 run 函数。run 函数内部逻辑如图 4-3 所示，如果当前函数是可变参数类型，如 CallObjectMethod 函数，

则需要动态解析额外的参数数量，并获取相应参数。获取参数后依次对每个参数代表的抽象值进行解析，同时获取 callsite 地址，将解析结果按照 callsite 保存，如果有上一次的分析结果则需要进行合并。最后根据函数的返回值类型，准备相应的抽象值返回，Angr 框架会按照调用约定放到寄存器中。

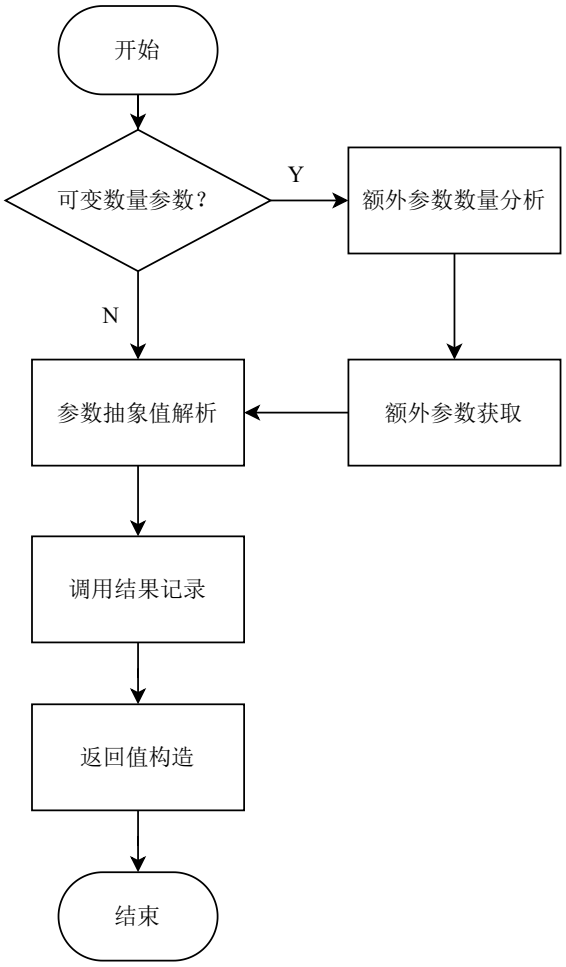

图 4-3 JNISimProcedure 类 run 函数流程图

动态参数数量解析基于 Angr 的到达定值（Reaching-Definition）分析实现。该分析会分析在该调用点前，二进制函数写入了哪些与调用约定有关的寄存器，从而得到可能的参数数量。当遗漏部分参数时，用户也可以通过运行参数手动配置最少解析的参数数量。

最终记录下来的每个二进制函数的调用信息和相关的参数会被导出成为 JSON 格式的语义信息文件。

4.2 语义信息翻译与注入

该模块基于 Soot 框架实现，因此在编程语言上选择了使用 Java 语言。用户指定 JSON 格式的语义信息和应用安装包作为输入，该模块首先利用 Soot 框架加载 APK 文件中的 DEX 字节码为 Jimple 中间表示，然后将读取的语义信息也生成为 Jimple 中间

表示，最后利用 Soot 框架的编译和重打包功能，将生成的 Jimple 中间表示重新编译为 DEX 字节码并重打包为新的 APK 文件。

各个类与相应的功能如下：

- APKRepacker：这个类负责命令行参数的解析，Soot 框架的初始化，APK 的加载，最终使用 MethodBuilder 类为 APK 文件注入函数体。
- MethodBuilder：这个类负责读取包含语义信息的 JSON 格式文件，对每个需要构建函数体的方法实例化 BodyBuilder 类创建函数体。
- BodyBuilder：这个类负责单个函数的函数体的创建。遍历该函数的语义信息中每一条 JNI 接口的调用记录，生成相应的 Java 语句。该类中实现了3.4节所述的生成算法。

由于 Soot 框架的 Android 应用重打包机制目前还不是特别完善[1]，在遇到部分 Java 库函数代码时会报错。因此目前的实现中在重打包时过滤了相关的库函数，这部分函数对数据流分析的影响也不大。

[1]https://github.com/soot-oss/soot/issues/1856

第五章　效果与评估

为了说明本地代码分析的必要性，本章首先分析了本地代码在真实 Android 应用中的使用情况。之后的小节针对本系统的分析准确率、分析效率与真实场景下的有效性进行了相关实验。

评估时主要用到的应用数据集是 FDroid 数据集。FDroid 是一个开源 Android 应用的应用市场，它通过爬虫自动化地获取开源应用的代码并进行构建、发布。由于 FDroid 数据集可以很便捷地获取，因此我们选用了该数据集进行分析。值得注意的是，FDroid 上的应用会不断更新，本次使用的数据集下载于 2022 年 4 月 1 日。

5.1　Android 应用中的本地代码使用情况

本地代码在真实应用中的使用相关的调研在同类工作[5–7] 的评估部分中都有涉及，读者可以参看对应文章中的结果。在本次使用到的 FDroid 数据集中有 19.3% 的应用至少包含了一个二进制动态链接库（后缀名为.so）的文件。在 Jucify[7] 这篇工作中提到，在包含 2641194 个良性应用的数据集中，44% 的应用至少包含一个.so 文件，43% 的应用至少包含一个带有 native 修饰的 Java 方法。在 174342 个恶意应用的数据集中，这两个数据来到了 73% 和 70%。

对 FDroid 数据集中的二进制库文件按照文件名分类，可以分析得到最常用的本地代码库。结果如表 5-1 所示。

表 5-1　数据集中本地代码第三方库使用情况

文件名	出现次数	描述
libgdx.so	33	编写游戏的底层图形库
libpl_droidsonroids_gif.so	26	GIF 处理库
libSDL2.so	24	底层图形库
libmain.so	23	使用 Python 编写 Android 应用的相关库
librealm-jni.so	20	可以替代 sqlite 的数据库
libgojni.so	19	Go 语言使用 JNI 接口生成的本地代码
libjingle_peerconnection_so.so	16	WebRTC，视频流
libsentry-android.so	12	运行监控，错误日志上传

5.2　分析精确率

由于真实应用无法得到真正准确的结果作为参照，该部分的评估基于 JN-SAF[5] 框架开源的 NativeFlowBench 数据集[1]。该数据集是为了分析 Android 跨语言分析框架而

[1]https://github.com/arguslab/NativeFlowBench

构造的，每个应用代表一种可能的数据流传递方式。该数据集中包含的部分跨组件调用（ICC，Inter-Component Communication）的三个样例和本地代码实现 Activity（Native Activity）的两个样例被剔除了。由于用本地代码实现 Activity 这种方式在现实应用中使用较少，因此本框架暂不支持。跨组件调用的情况本框架也可以直接支持，但是对上层分析框架要求较高，因此也没有被纳入进来。此外，本框架暂未支持本地方法通过 RegisterNatives 函数的动态注册机制，因此也没有列出动态注册的两个样例。但动态注册的解析和 JNI 接口调用序列提取分析过程并无较大差异，可以较为容易地复用相关逻辑实现。

对于成功注入和重打包这个层面，我们将分析后重打包得到的应用使用 Jadx[1]反编译器手动查看重打包后的 APK 中的 Java 代码。其中 native_complex_data 应用重打包后用 Jadx 反编译器查看其中一个生成函数结果如图 5-1 所示。可以看到，我们的框架能够正确地生成对 getData 函数的调用，同时也反映了，返回值可能被日志打印函数泄露的语义。

```
public static void send(ComplexData $CallObjectMethoda4196300) {
    String $ALPa4196376;
    String $GSUC4196332 = $CallObjectMethoda4196300.getData();
    int $opred = NativeSummaryAux.opaqueInt();
    if ($opred == 1) {
        $ALPa4196376 = "data";
    } else if ($opred == 2) {
        $ALPa4196376 = "%s";
    } else if ($opred == 3) {
        $ALPa4196376 = $GSUC4196332;
    }
    Log.d($ALPa4196376, $ALPa4196376);
}
```

图 5-1 complex_data 应用生成结果示例

在这个数据集上，除了 native_leak_array 和 native_noleak_array 这两个应用由于目前对数组运算的支持不够完善导致无法正确生成对应 Java 字节码外，其他应用均正确生成了代表其语义的 Java 字节码。

首先，我们使用 Jadx 反编译工具手动检查了每个重打包后的应用，查看是否能够生成正确的字节码。经过查看后发现所有应用的反编译结果的 Java 代码正确反映了本地代码的语义。然后，我们对重打包后的应用分别使用 Argus-SAF 框架和 FlowDroid[12] 框架进行污点分析，寻找隐私泄露的情况。Argus-SAF（也称 Amandroid[9]）框架是 JN-SAF[5] 这篇工作中对接的上层分析框架，但 JN-SAF 与该框架耦合严重，无法通用与其他分析框架。相关结果如表 5-2 所示。

可以看到，重打包后的应用都能够被两个 Java 侧分析框架正常分析，证明了我们的框架能够通用于各种上层分析框架。但由于 Java 侧分析框架的分析能力不同，导致不同分析框架能够检测出的隐私泄露数量也不同。例如，native_source 应用可以被 Argus-SAF

[1]https://github.com/skylot/jadx

表 5-2 NativeFlowBench 数据集上的结果，其中右侧两列数据来源于[5]

应用名称	本框架 + Argus-SAF	本框架 + FlowDroid	JN-SAF	仅 FlowDroid
native_source	O	X	O	X
native_nosource				
native_source_clean	*	*		*
native_leak	O	O	O	X
native_noleak				
native_method_overloading	O	O	O	X
native_multiple_interactions	O	O	O	X
native_multiple_libraries	O	O	O	X
native_complexdata	X	O	O	X
native_complexdata_stringop			*	
native_heap_modify	O	O	O	X
native_set_field_from_native	XX	OO	OO	XX
native_set_field_from_arg	OO	OO	OO	XX
native_set_field_from_arg_field	XX	OO	OO	XX
总结				
正确报出 O	8	12	13	0
误报 *	1	1	1	1
漏报 X	5	1	0	13
精确率 p = O/(O + *)	88.9%	92.3%	92.9%	0.0%
召回率 r = O/(O + X)	61.5%	92.3%	100%	0.0%

O：正确报出（真阳性） *：误报（假阳性） X：漏报（假阴性）

框架检出，但 FlowDroid 无法检测出泄露，而 native_complexdata 应用则能被 FlowDroid 检出，没有被 Argus-SAF 检测出泄露。总体而言，我们的框架能够正确生成反映 Native 语义的 DEX 字节码，并正确重打包为新的应用。

此外，我们还使用了 DroidSafe[11] 框架对重打包后的代码进行分析。但是该框架无法检测到 onRequestPermissionsResult 回调函数的调用，而该数据集中大量使用了该方法作为信息泄露的触发点，导致大量信息泄露无法被检测到。因此我们没有将相关结果列出。但 DroidSafe 框架确实能够正常分析重打包后的应用，这进一步证实了本框架对现有的 Java 侧分析框架的兼容性。

其他值得注意的是，在本次实验过程中发现，生成的 Java 指令中包含冗余 Cast 指令时会影响部分污点分析框架的效果。而如果完全不包含 Cast 指令有时也会影响到上层分析框架解析函数调用的目标。

5.3 分析效率与真实场景下的有效性

前述实验中用到的 NativeFlowBench 数据集由于其仅为人工构造的代表各种底层分析情况的应用，代码量过小，因此并不适合用于评估本框架的效率。由于本框架目前仅支持 arm64 架构的汇编指令的分析，我们从 FDroid 数据集中选取了本地代码架构为 arm64 的 516 个应用作为数据集。由于语义信息翻译模块效率较高，主要的时间开销在于二进制分析侧，本次仅运行了二进制分析和语义信息提取模块。我们为 CFG 构建和每个注册的 native 函数的 VSA 分析分别设置了 3 分钟和 2 分钟的超时时间。此次实验基于 CPython 版本的 Angr，在 CPU 型号为 Ryzen 5800U 的笔记本上进行单线程分析，CPU 频率在 4GHz 左右。值得注意的是，Angr 主页提及使用 PyPy 替代 CPython 可以较大幅度地提升运行速度。

数据集中，平均每个应用的分析耗时为 137.95 秒。分析时间关于 APK 文件大小的散点图如图 5-2 所示。可以看到，针对少数较难分析的应用容易出现分析时间过长的现象。

在这些应用中，我们能为 58.5% 的应用成功提取至少一个函数，总提取的函数数量平均到每个应用上为 18.4 个。动态链接库的控制流图构建上，分析失败占比 1.75%，分析超时占比 10.72%。对每个本地函数的分析，分析失败占比 13.15%，分析超时占比 1.19%。其中分析失败代表分析过程中 Angr 抛出了难以解决的异常。

Angr 分析时会出现遇到部分复杂函数时超时的现象，经调查发现似乎不是静态分析逻辑导致，例如 Angr 在静态分析时依然有部分符号执行逻辑在保存路径约束，而该约束可能由于复杂循环等原因，随着迭代分析变得极度复杂，处理耗时较大，从而阻止了分析的继续进行。这可能是由于 Angr 的 VSA 分析不够完善导致。此外，部分函数在分析时会抛出异常。如引用栈变量时使用 OR 指令取部分比特位时，Angr 会报出 Strided Interval 和 Value Set 这两种数据类型之间不支持 OR 运算的错误。这可能是 Angr 的底层抽象域支持的运算种类不够完善所致。

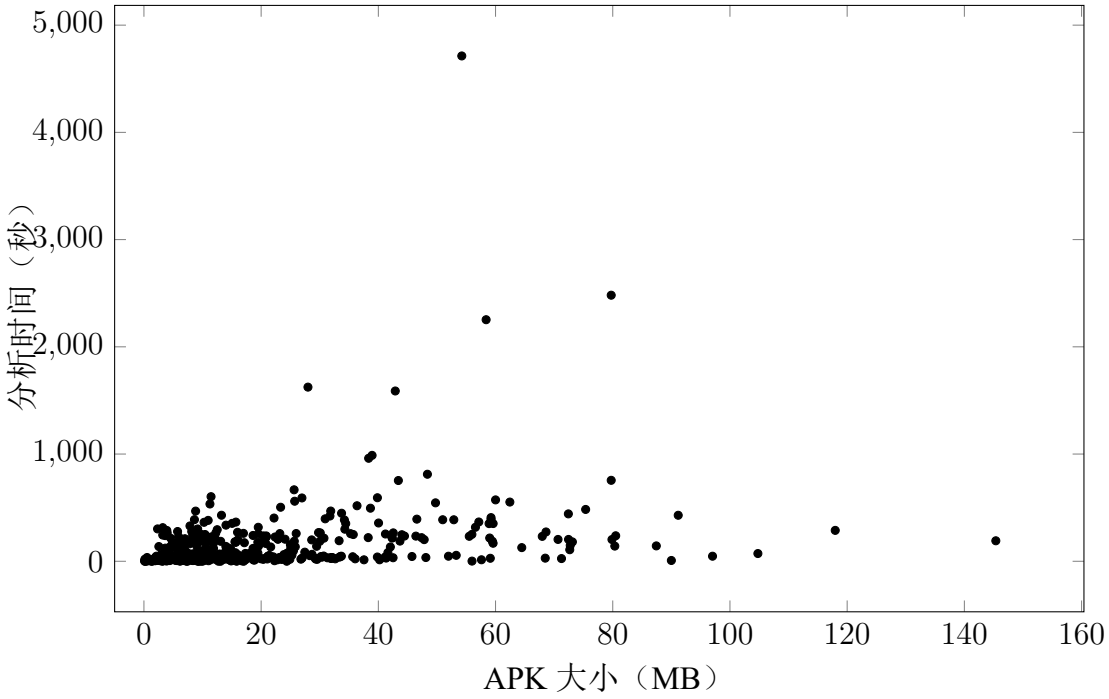

图 5-2 运行时间与 APK 文件大小散点图

分析时间占比上，DEX 分析平均耗时为 28.00 秒，占比 20.30%，CFG 构建平均耗时为 48.46 秒，占比 35.13%，VFG（即 VSA）分析平均耗时为 25.60 秒，占比 18.56%，其他耗时平均为 35.87 秒，占比 26.00%。其中，其他耗时包含了分析失败的本地函数所消耗的时间。

我们还统计了每个成功分析的本地函数的耗时。在总计 8908 次成功分析的本地函数中，有 7842 个函数（88.03%）在 1.1 秒内完成了分析，其余函数的耗时分布在 1 到 120 秒的区间内。

总之，我们的框架能够较为高效地为从现有应用的本地代码中提取语义信息，从而让上层分析框架能够分析到更多的数据流。

第六章　总结与展望

6.1　其他相关工作

- NativeScanner[31] 通过简单的二进制分析解析 native 函数的绑定关系，该工作仅讨论了绑定关系的解析问题，使用较为简单的二进制交叉引用分析。
- 《Towards Bridging the Gap Between Dalvik Bytecode and Native Code During Static Analysis of Android Applications》[32] 这篇工作提出 Android 应用的二进制代码分析是一个急需解决的问题，并粗略地提出了可能的实现方式，没有真正去实现。另一篇相关工作[33] 也类似。
- Meizodon[34] 这篇工作测试了在真实恶意样本下，各种开源 Android 应用恶意软件检测器的效果。

6.2　未来方向

随着二进制反编译技术的发展，当反编译结果足够可靠时，可以通过使用反编译得到的 C 语言源码进行有源码的分析。然而这可能也会导致更多的非开源应用转而使用加壳，混淆等反分析手段。此外，等到程序静态分析足够成熟时，针对新的语言特性编写分析算法需要的工程成本也将降低。从零实现一个基于 JNI Light[26] 提出的内存抽象模型的分析框架将成为可能。

本次系统基于 Angr 实现，但 Angr 的 CFG 构建和 VSA 分析在鲁棒性和效率上都不尽人意。基于 Ghidra 上的静态分析框架实现本系统可能会得到更好的效果。此外，语言信息转为 Jimple 语句部分如果能用上现有的编译原理中的技术也许效果会更好一些，项目结构上也将更加利于维护。

本次实现的系统仅支持 arm64 汇编代码的分析，然而现有应用中 32 位本地代码也依然广泛存在，对 32 位代码的支持仍然较为重要。

参考文献

[1] Afonso Vitor, Bianchi Antonio, Fratantonio Yanick et al. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy [C]. In The Network and Distributed System Security Symposium. San Diego, California, USA. 2016 : 1–15.

[2] Lindorfer Martina, Neugschwandtner Matthias, Weichselbaum Lukas et al. Andrubis–1,000,000 apps later: A view on current Android malware behaviors [C]. In 2014 third international workshop on building analysis datasets and gathering experience returns for security (BADGERS). Wroclaw, Poland. 2014 : 3–17.

[3] Qian Chenxiong, Luo Xiapu, Shao Yuru et al. On tracking information flows through jni in android applications [C]. In 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. Atlanta, GA, USA. 2014 : 180–191.

[4] Tam Kimberly, Fattori Aristide, Khan Salahuddin et al. Copperdroid: Automatic reconstruction of android malware behaviors [C]. In Network and Distributed System Security Symposium. San Diego, California. 2015 : 1–15.

[5] Wei Fengguo, Lin Xingwei, Ou Xinming 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]. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA. 2018 : 1137–1150.

[6] Sun Cong, Ma Yuwan, Zeng Dongrui et al. Dep: Mutation-based Dependency Generation for Precise Taint Analysis on Android Native Code [J]. IEEE Transactions on Dependable and Secure Computing. 19 (2). 2022, 3.

[7] Samhi Jordan, Gao Jun, Daoudi Nadia et al. JuCify: A Step Towards Android Code Unification for Enhanced Static Analysis [J]. arXiv preprint arXiv:2112.10469. 2022, 1: 1–13.

[8] Pauck Felix, Bodden Eric, Wehrheim Heike. Do android taint analysis tools keep their promises? [C]. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista, FL, USA. 2018 : 331–341.

[9] Wei Fengguo, Roy Sankardas, Ou Xinming. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps [J]. ACM Transactions on Privacy and Security (TOPS). 21 (3). 2018: 1–32.

[10] Bosu Amiangshu, Liu Fang, Yao Danfeng et al. Collusive data leak and more: Large-scale threat analysis of inter-app communications [C]. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. Abu Dhabi, United Arab Emirates. 2017 : 71–85.

[11] Gordon Michael I, Kim Deokhwan, Perkins Jeff H et al. Information flow analysis of android applications in droidsafe. [C]. In Network and Distributed System Security Symposium. San Diego, California, USA. 2015 : 110.

[12] Arzt Steven, Rasthofer Siegfried, Fritz Christian et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps [J]. Acm Sigplan Notices. 49 (6). 2014: 259–269.

[13] Li Li, Bartel Alexandre, Bissyandé Tegawendé F et al. Iccta: Detecting inter-component privacy leaks in android apps [C]. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Florence, Italy. 2015 : 280–291.

[14] Lee Sungho, Lee Hyogun, Ryu Sukyoung. Broadening horizons of multilingual static analysis: semantic summary extraction from C code for JNI program analysis [C]. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. Melbourne, Australia. 2020 : 127–137.

[15] Shoshitaishvili Yan, Wang Ruoyu, Salls Christopher et al. Sok:(state of) the art of war: Offensive techniques in binary analysis [C]. In 2016 IEEE Symposium on Security and Privacy (SP). San Jose, CA, USA. 2016 : 138–157.

[16] Liu Zhibo, Wang Shuai. How far we have come: testing decompilation correctness of C decompilers [C]. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. Virtual Event, USA. 2020 : 475–487.

[17] Zhang Zhuo, You Wei, Tao Guanhong et al. BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation [J]. Proceedings of the ACM on Programming Languages. 3 (OOPSLA). 2019: 1–31.

[18] Kim Sun Hyoung, Zeng Dongrui, Sun Cong et al. BinPointer: towards precise, sound, and scalable binary-level pointer analysis [C]. In Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction. Seoul, South Korea. 2022 : 169–180.

[19] Kim Sun Hyoung, Sun Cong, Zeng Dongrui et al. Refining indirect call targets at the binary level [C]. In Network and Distributed System Security Symposium. virtually. 2021 .

[20] Cousot Patrick, Cousot Radhia. Static determination of dynamic properties of programs [C]. In Proceedings of the 2nd International Symposium on Programming. Paris, France. 1976 : 106–130.

[21] Cousot Patrick, Cousot Radhia. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints [C]. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. Los Angeles, California, USA. 1977 : 238–252.

[22] PASCAL Research Group. Static Program Analysis [EB/OL]. 2021 [2022-05-08]. https://pascal-group.bitbucket.io/teaching.html.

[23] Balakrishnan Gogul, Reps Thomas. Analyzing memory accesses in x86 executables [C]. In International conference on compiler construction. Barcelona, Spain. 2004 : 5–23.

[24] Reps Thomas, Balakrishnan Gogul. Improved memory-access analysis for x86 executables [C]. In International Conference on Compiler Construction. Budapest, Hungary. 2008 : 16–35.

[25] Balakrishnan Gogul, Reps Thomas. Wysinwyx: What you see is not what you execute [J]. ACM Transactions on Programming Languages and Systems (TOPLAS). 32 (6). 2010: 1–84.

[26] Tan Gang. JNI Light: An operational model for the core JNI [C]. In Asian Symposium on Programming Languages and Systems. Shanghai, China. 2010 : 114–130.

[27] Buddrus Frank, Schödel Jörg. Cappuccino—A C++ to Java translator [C]. In Proceedings of the 1998 ACM symposium on Applied Computing. Atlanta, GA, USA. 1998 : 660–665.

[28] Demaine Erik D. C to Java: converting pointers into references [J]. Concurrency: Practice and Experience. 10 (11-13). 1998: 851–861.

[29] Allan Vicki H, Chen X. Convert2Java: semi-automatic conversion of C to Java [J]. Future Generation Computer Systems. 18 (2). 2001: 201–211.

[30] Li Siliang, Tan Gang. Exception analysis in the java native interface [J]. Science of Computer Programming. 89. 2014: 273–297.

[31] Fourtounis George, Triantafyllou Leonidas, Smaragdakis Yannis. Identifying java calls in native code via binary scanning [C]. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. Virtual Event, USA. 2020 : 388–400.

[32] Lantz Patrik, Johansson Bjorn. Towards bridging the gap between dalvik bytecode and native code during static analysis of android applications [C]. In 2015 International Wireless Communications and Mobile Computing Conference (IWCMC). Dubrovnik, Croatia. 2015 : 587–593.

[33] Li Ziqing, Feng Guiling. Inter-Language Static Analysis for Android Application Security [C]. In 2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE). Dalian, China. 2020 : 647–650.

[34] Rodriguez Sebastiaan Alvarez, van der Kouwe Erik. Meizodon: Security benchmarking framework for static Android malware detectors [C]. In Proceedings of the Third Central European Cybersecurity Conference. Munich, Germany. 2019 : 1–7.

致　　谢

感谢王浩宇、徐国胜老师的支持。感谢现有的开源二进制分析框架，没有它们，现有的二进制分析研究都将更加难以进行。

附　　录