南京大學

研 究 生 毕 业 论 文

（申 请 硕 士 学 位）

论 文 题 目 定制 **Android** 系统服务测试技术研究

作 者 姓 名 廖祥森

学科、专业名称 计算机技术

研 究 方 向 软件方法学

指 导 教 师 曹春 教授

2021 年 **05** 月 **29** 日

学　　号：-

论文答辩日期：**2021** 年 **05** 月 **25** 日

指 导 教 师：　　（签字）

Research of Service Testing in Vendor Android

by

LIAO Xiangsen

Supervised by

Professor CAO Chun

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfilment of the requirements for the degree of
MASTER
in
Computer Technology

Department of Computer Science and Technology
Nanjing University

May 29, 2021

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：定制 Android 系统服务测试技术研究

计算机技术 专业 2018 级硕士生姓名：廖祥森

指导教师（姓名、职称）：曹春 教授

摘　要

当前 Android 系统已成为最主要的移动互联网平台，并且 Android 的技术开放性为移动设备厂商对官方系统进行定制化开发提供了便利。相关研究表明，这些定制化的 Android 系统，特别是其引入的大量服务扩大了 Android 系统的攻击面。为了保障定制 Android 系统的安全性，研究并实现对其中服务的有效测试成为必要。针对现有服务模糊测试工作中因缺乏服务接口签名导致的测试效率低下问题，提出一种服务接口签名逆向提取方法，以指导测试用例的生成。在服务模糊测试过程中，通过模拟 Android 系统上下文提高测试深度，生成异常 IBinder 类型更有效地测试服务的边界情况。

具体而言，本文的工作主要包括：

- 提出一种使用逆向工程自动化提取服务接口签名的方法，可从 Java 与 Native 服务的编译产物中，通过对 Binder 通信序列化函数结构的还原，逆向推测出服务接口签名，并实现了服务接口签名提取工具 RevExtractor。
- 实现了一个服务模糊测试工具 CASFuzzer：基于服务接口签名指导测试用例的生成与变异；通过 Binder 通信劫持技术，模拟影响服务执行的 Android 系统上下文，提高测试深度；动态地构造通用服务对象，以模拟 Android 系统中特有的 IBinder 类型的异常行为，更好地测试服务的边界情况。
- 在两个主流的定制 Android 系统上进行实验评估。实验结果表明，服务接口签名提取工具 RevExtractor 具有较高的准确性，服务模糊测试工具 CASFuzzer 中对 Android 系统上下文的模拟使得测试过程更加深入有效，测试用例中构造的异常 IBinder 类型可以触发特定的服务缺陷。

关键词： Android；模糊测试；逆向工程；Binder

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Research of Service Testing in Vendor Android

SPECIALIZATION: Computer Technology

POSTGRADUATE: LIAO Xiangsen

MENTOR: Professor CAO Chun

Abstract

Android is the most popular operating system of smartphone. It's convenient to customize the Android for mobile device manufacturers. Related research shows that the Vendor Android, which adds a large number of Services, has expanded the attack surface of Android. In order to ensure the security of the Vendor Android, it is necessary to test its Services effectively. Most related works are mutation-based fuzzers, they have low code coverage of target Services due to the lack of type signature. So we design a reverse engineering method to extracte Service signature in closed-source Vendor Android. We implement a generation-based Service fuzzing tool, to improve the test depth through mocked Android Context, and to test the Service boundary conditions more effectively through abnormal IBinder type. The main work is as follow:

- A reverse engineering method to extract the Service signature automatically. From the compiled artifacts of Java and Native Services, the Service signature can be reversely inferred by restoring the structure in the Binder serialization function.
- We implement a Service fuzzing tool, CASFuzzer: it can generate and mutate test case based on the Service signature; through Binder IPC interception technology, it can mock Android Context that affects the execution of the service to improve the depth of testing; through dynamic general ServiceStub, it can generate abnormal IBinder type for boundary condition testing of the Service.
- We evalute our tools with two popular Vendor Android OS. The experimental results show that the Service signature extraction tool RevExtractor has high accuracy. The simulation of the Android Context makes the testing more in-depth and effective. The abnormal IBinder type can trigger specific Service defects.

keywords: Android, Fuzzing, Reverse Engineering, Binder

目　　录

插图清单

附表清单

第一章　绪论

1.1　研究背景及意义

作为移动互联网时代的主要终端设备，智能手机得到了广泛的应用，在当前智能手机市场份额中，Google 公司开发的 Android 系统占据了主导地位（根据 IDC 对移动端操作系统的最新统计 [1] 显示，Android 系统的市场占有率高达 84.1%）。除了智能手机以外，Android 系统同样也在物联网终端，如智能电视、智能汽车等设备上得到了广泛应用。相较于其他操作系统，Android 系统最大的优势在于它的开放性，Google 公司允许任何设备厂商加入到 Android 开发联盟中来。

Android 系统的开放性吸引了越来越多的终端应用开发者，Android 生态圈在短时间内积累了丰富的软件资源和大量的用户。随着用户日常生活与手机越来越紧密相连，手机中存储着用户大量的个人隐私信息，这也吸引了越来越多攻击者的注意。当前 Android 生态圈的安全问题不容忽视：根据 CVE Detail 的统计 [2] 显示，Android 系统常年位居各类操作系统漏洞数量榜前列；2019 年，360 安全大脑共截获移动端新增恶意程序样本 180.9 万个，平均每天新截获 5000 个恶意程序样本 [3]；腾讯安全发布的手机安全报告 [4] 中指出，2020 上半年手机病毒感染用户数 3058.95 万，新增 Android 病毒包 308.05 万个，同比涨幅达 62.24%；漏洞收购平台 Zerodium 更是给 Android 漏洞完整利用链开出了 250 万美元的“悬赏”，远超其他操作系统 [5]。

移动设备厂商为了提高自家手机的产品竞争力，不仅会在硬件配置上体现出差异与特色，还会对 Android 系统本身进行二次开发以提供更丰富的功能。除此之外，为了使得手机用户可以开箱即用，厂商定制的 Android 系统中往往会预装自己的一套系统应用，比如应用商店、手机钱包、视频播放器、桌面美化软件等等。出于性能和用户体验考虑，还可能采取安全性弱化的软件配置。相关研究表明，相较于 Google 官方的 Android 系统，厂商对 Android 系统的定制化引入了更多的安全隐患 [6][7][8]。

Android 生态圈的碎片化现象是导致其安全隐患的重要因素。具不完全统

计，市面上有超过 200 家移动设备厂商提供的超过 7000 种 Android 智能设备[9]，设备上搭载的定制 Android 系统也不尽相同。这使得上游的 Google 公司很难对这样一个开放、多样化的软件生态系统进行很好的质量管理。如何保证这些定制 Android 系统的安全性，发掘其中的安全漏洞具有重要的研究价值。

服务（Service）是 Android 系统的核心组成部分，服务除了以组件的形式广泛分布于 Android 应用之中，Android 系统的绝大多数功能也是通过系统服务的形式暴露给上层使用，对于攻击者来说，服务是一个很好的攻击面。定制 Android 系统中不仅包含了大量的预装应用服务组件，而且引入了更多的系统服务（如 Samsung Experience 8.0 系统中添加了多达 82 个系统服务 [10]），这些具有系统关键权限的服务进一步扩大了 Android 系统的攻击面。因此，对服务进行测试有助于保障定制 Android 系统的安全性，具有重要的研究价值。

1.2 研究现状

软件漏洞挖掘技术自软件诞生之初一直是一个热门的研究领域。根据是否执行待测程序，漏洞挖掘技术可分为静态分析、动态分析与混合分析三类[11]。静态分析直接对程序源码进行检测，可以获得较高的代码覆盖率，并容易发现程序的边界情况，但会面临路径爆炸以及约束求解开销过大等问题；动态分析能获得程序的执行状态，但对代码分支路径的覆盖不够全面。

基于静态程序分析的符号执行（symbolic execution）与基于动态执行的模糊测试（fuzzing）是当前主流、也是最有效的漏洞挖掘技术。

符号执行技术最早由 King 等人提出 [12]，目前 KLEE [13] 是使用最为广泛的符号执行引擎。符号执行技术通过分析待测程序来得到程序特定执行路径的输入，其关键思路是维护一个程序执行路径的约束条件集合，以符号代替具体数值作为输入，遇到程序分支时将分支判断条件加入到约束条件集合中，最后将约束条件集合输入约束求解器，生成与此路径所对应的具体程序输入。

模糊测试技术最早由 Millor 等人提出，用于测试 UNIX 系统中的工具类程序 [14]。模糊测试技术是一种黑盒或灰盒的测试技术，通过自动化生成并输入大量的随机测试用例以检测待测程序中的漏洞。

在 Android 领域的相关研究中，Luo 等人实现的 CENTAUR [15] 首次将符号执行技术应用到 Android 系统服务的测试之上，并结合了具体执行（concrete execution）收集的 Android 系统上下文状态，以避免符号执行陷入系统服务初

始化阶段。然而对于 Android 这样一个复杂并且组件化的系统来说，符号执行等静态程序分析技术会面临控制流图不完整、状态爆炸等问题。模糊测试技术由于其自动化程度高、可拓展性强被广泛应用于 Android 领域的服务漏洞挖掘之中。

模糊测试技术主要分为两类 [16]：基于变异（mutation-based fuzzing），根据已有的测试用例样本，通过变异的方法生成新的测试用例；基于生成（generation-based fuzzing），根据待测程序的协议或接口规范进行建模，基于模型生成测试用例。已有对 Android 服务的测试工作大多使用了模糊测试技术，以下介绍五个代表性工作以及他们的局限性。

Zhang 等人的研究 [17] 以 Intent 为服务通信接口，对 Android 应用中的 Java 后台服务进行测试。研究指出提高测试效率的关键是构造可成功启动服务的 Intent 对象。对待测服务的入口函数 onStartCommand 进行局部的静态分析，识别出 Intent 中 action、component、category 等字段的值，利用 NLP 方法推断出 extras 的 value 类型。然而只能用于启动式服务的测试，并不适用于以 AIDL 为通信接口 [18] 的绑定式服务。

BinderCracker [19] 以 Binder 为服务通信接口，对 Android 中的系统服务进行黑盒测试。通过对通信过程的监控，记录下 Binder 驱动收到的消息体，对消息体进行变异以构造新的测试用例。纯粹基于变异的方法很难构造有效的测试用例，测试的效率不高。

Gu 等人的研究 [20] 将关注点放在 Android SDK 中 Service Helper（Service Helper 封装了系统服务的功能，上层应用往往通过 Service Helper 间接与系统服务进行通信）与系统服务接口在权限检查上的不一致性，使用静态分析的方法比较这两者在实现逻辑上的差异，进而识别出 Service Helper 的安全隐患。

Chizpurfle [10] 以 AIDL 为服务通信接口，对定制 Android 系统中的 Java 系统服务进行测试。通过 Java 反射获取待测服务的接口签名，以生成合法的输入参数；并结合二进制插桩技术统计待测服务的代码覆盖率，使用覆盖率指导的模糊测试技术（coverage-guided fuzzing）对服务进行测试。局限在于只考虑了 Java 类型的服务，事实上某些定制 Android 系统中有将近一半的系统服务以 C++ 代码的形式实现。

FANS [21] 以 AIDL 为服务通信接口，对 AOSP 中的 Native 系统服务进行测试。对通信部分的 C++ 代码进行静态分析，以获取待测服务的接口签名，并尝试从变量类型、变量名称等属性推测不同接口的依赖关系，以此构造多层次

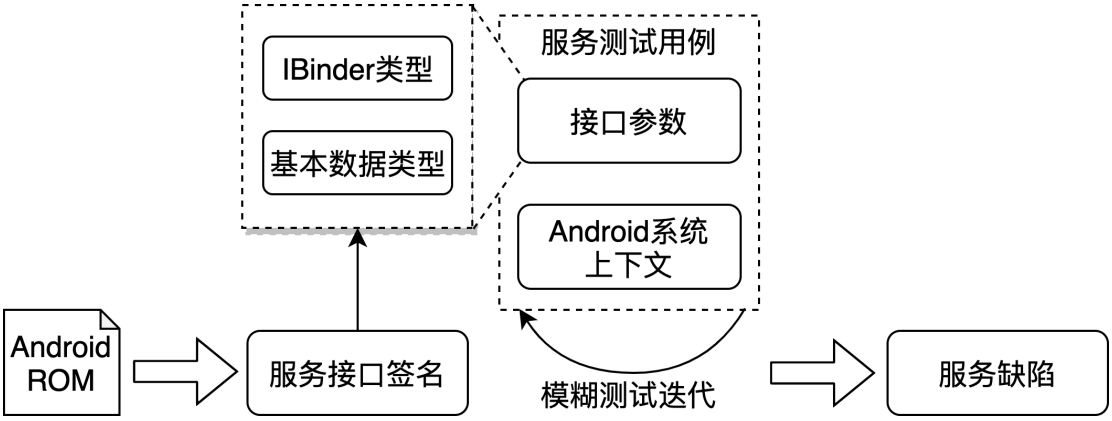

图 1-1: 定制 Android 系统服务测试方法

的类型模型，指导测试用例的生成。局限在于必须获得待测服务的源码，不能适用于定制 Android 系统中的闭源服务。

1.3 本文主要工作

基于生成式策略的模糊测试方法由于对程序输入有更精确的类型模型，减少了不合法输入，在针对具有高度结构化特征的 Android 服务接口输入时，相较于基于变异策略的测试方法更为有效。

生成式策略的关键在于获取程序输入的类型模型，然而针对于定制 Android 系统中的 Java 以及 Native 服务来说，以往基于 Java 反射 [10]、基于 AIDL 接口定义 [20]、基于 C++ 源码静态分析 [21] 等类型模型提取方法并不完全适用。本文提出一种服务接口签名自动化提取方法，从逆向工程的角度分析服务的编译产物，识别其中通信序列化过程的代码结构特征，以此推断出服务接口签名。接着基于生成式策略实现了一个服务模糊测试工具，输入服务接口签名以指导测试用例的生成与变异，通过 Binder 通信劫持技术以模拟其中的动态类型信息，测试迭代时通过动态二进制插桩技术收集基本块覆盖信息，以评估测试效果。

本文提出的整套定制 Android 系统服务测试方法如图 1-1 所示，研究内容主要包括：

- 提出一种使用逆向工程自动化提取服务接口签名的方法，可从 Java 与 Native 服务的编译产物中，还原 Binder 通信序列化函数的结构特征，以推测出服务接口签名。

- 实现了一个服务模糊测试工具 CASFuzzer：基于服务接口签名指导测试用例的生成与变异，提高模糊测试效率；通过 Binder 通信劫持技术，模拟影响服务执行的 Android 系统上下文，提高测试深度；动态地构造通用服务对象，以模拟 Android 中特有的 IBinder 类型的异常行为，更好地测试服务的边界情况。
- 在两个主流的定制 Android 系统上进行实验评估。实验结果表明，服务接口签名提取工具 RevExtractor 具有较高的准确性，服务模糊测试工具 CASFuzzer 中对 Android 系统上下文的模拟使得测试过程更加深入有效，测试用例中的异常 IBinder 类型可以触发特定的服务缺陷。

1.4 本文组织结构

本文组织如下：

第二章介绍了本文涉及到的背景知识与相关工作。首先介绍了 Android 系统及服务的相关背景知识；接着介绍了服务底层的 Binder 通信机制，包括其架构、通信流程以及通信协议；然后介绍了移动设备厂商如何定制 Android 系统以及这些定制化带来的安全隐患；最后介绍了 Android 组件间通信的相关工作。

第三章详细介绍了服务接口签名逆向提取工具 RevExtractor，提取过程包含三个步骤：从 Android ROM 中提取服务相关编译产物；结合 Android 系统运行时状态，筛选出待测服务的通信序列化函数；从通信序列化函数的字节码与 ARM 汇编中，识别出序列化过程结构特征，以此推测出服务接口签名。

第四章详细介绍了服务模糊测试工具 CASFuzzer：首先介绍测试用例中引入 Android 系统上下文的动机；接着介绍整体的服务模糊测试流程，包括如何启动待测服务；然后介绍测试用例生成策略，特别是对 IBinder 类型以及 Android 系统上下文这两种动态数据类型的模拟；最后介绍了如何评价测试用例，包括待测服务是否出现异常以及基本块覆盖率两个指标。

第五章是实验评估部分，对本文提出的服务接口签名逆向提取工具 RevExtractor 与服务模糊测试工具 CASFuzzer 进行了实验评估，验证了本文工作的有效性。

第六章是总结与展望，对本文工作进行简单的总结，并对未来可能的研究方向进行展望。

第二章　背景知识

本章首先介绍了 Android 系统架构，以及 Android 系统中的重要组成部分—服务。为了更好地理解服务的运行模式，接着介绍了服务底层的 Binder 通信机制，包括其架构、通信流程以及通信协议。然后介绍了第三方定制 Android 系统相关的背景，包括移动设备厂商如何定制 Android 系统以及这些定制所引入的安全问题。不少以往研究从 Android 组件间通信的角度对 Android 服务进行测试，最后介绍这个领域的相关研究进展。

2.1　Android 系统架构

Android 是 Google 公司专为移动设备开发的基于 Linux 内核的开源操作系统，与大多数软件系统一样，采用分层的系统架构。如图 2-1 所示，自下到上分别是 Linux 内核层、Android 虚拟机及系统运行库、应用框架层、应用程序层，各层次的功能如下：

- Android 系统底层基于 Linux 内核，但与上游的 Linux 内核存在不少显著差异。由于大量性能和安全方面的改动造成的不兼容性，Google 一直在 Linux 内核下游维护着单独的 Android 内核分支。Android 内核中，除了包括 Google 引入的 Binder、Ashmem、Logger 等通用设备驱动以外，还包括各家移动设备厂商引入的诸如摄像机、屏幕、编解码器等定制硬件设备驱动。
- Android 虚拟机是一个基于寄存器架构的虚拟机，运行 Dalvik 字节码，为应用提供运行环境并保证了不同应用间的隔离，自 Android 5.0 之后，原有的 Dalvik 虚拟机被 ART 虚拟机所替代，并结合 AOT（Ahead Of Time）等编译技术提高运行速度；系统运行库以 C/C++ 动态链接库的形式分发，以 JNI 的形式暴露给上层应用使用，负责连接应用框架层和 Linux 内核层。
- 应用框架层以 Android SDK 的形式提供给应用开发者一系列 Java API，便于开发上层应用。
- 应用程序层是指手机用户日常接触到的诸如地图、相机、购物软件等移动应用程序，这些应用程序既包括手机厂商预装的系统级应用，也包括用户

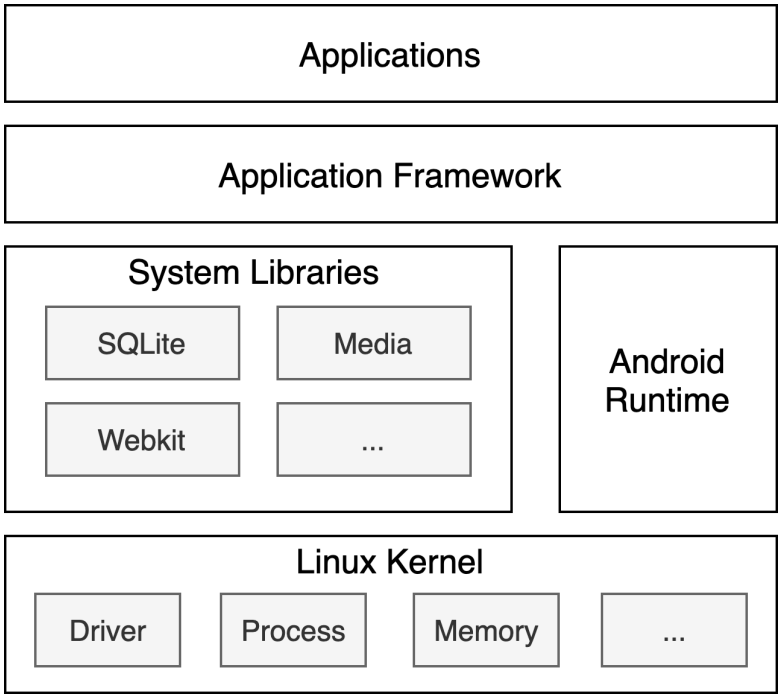

图 2-1: Android 系统架构

从第三方应用市场下载的应用。

Android 框架定义了四大基本组件 Activity，Service，BroadcastReceiver，ContentProvider：Activity 负责组织图形用户界面，与用户进行交互；Service 主要运行在后台，执行一些耗时逻辑，如播放音乐、执行文件 I/O 等等；BroadcastReceiver 用来接收应用和系统发送的广播事件；ContentProvider 提供不同应用间的数据共享功能。

2.2 Android 服务

作为 Android 四大基本组件之一，服务是 Android 系统的核心组成部分。服务除了以组件的形式在 Android 应用中被广泛使用外，Android 系统绝大多数的功能也是通过系统服务（System Service）的形式暴露给上层使用，如提供应用管理功能的 PackageManager、提供网络连接管理功能的 NetworkManagementService、管理摄像机的 CameraService 等等。

从服务的生命周期来看，可分为启动式服务（Started Service）和绑定式服务（Bound Service）两种类型 [18]。对于启动式服务来说，应用组件通过构造 Intent 对象，调用 startService 接口显式启动服务，启动式服务的生命周期独立

于调用者，即使调用者被销毁，启动式服务也仍可继续运行。对于绑定式服务来说，应用组件通过 bindService 接口获得对应的服务 IBinder 对象，之后应用组件可通过该 IBinder 对象与绑定式服务进行交互，绑定式服务的生命周期依赖于应用组件，若应用组件被销毁，则绑定式服务也随之被 Android 系统所销毁。但这两者的关系也并不绝对，有些服务既可以是启动式服务，也同时提供了绑定接口，在绑定的应用组件被销毁之后，仍会继续运行。

从服务的可见性角度来看，可分为实名服务和匿名服务两类。实名服务的基本信息存储在 Service Manager 中，外部组件可通过服务名称从 Service Manager 中检索对应的服务引用，系统服务都属于实名服务；匿名服务只对调用者双方可见，服务对象存储在服务端进程的 LoadApk 对象之中，无法在 Service Manager 中检索到，例如应用中的绑定式服务就属于匿名服务。

服务底层通过 Binder 通信机制与其他应用组件发生交互，了解 Binder 通信机制有助于我们更好地分析服务的行为与运行状态。

2.3 Binder 通信机制

Binder 是 Android 系统特有的进程间通信机制，与 Linux 中常见的进程间通信机制（如 Named Pipe、Message Queue、Signal、Share memory、Socket 等方式）相比，Binder 通信更为安全（支持对通信双方进行身份校验），数据传输也更为高效（只需要一次数据拷贝）。作为 Android 系统最基础的通信方式，Binder 通信机制在 Android 系统的版本演化中很少变化，保证了上层应用在不同版本系统中的兼容性。

2.3.1 Binder 系统架构

Binder 系统架构如图 2-2 所示，在 Binder 通信系统中，所有可被访问的服务实体对应于一个内核空间中的 IBinder 对象，由服务端负责初始化，该对象将内部功能封装为一套自定义接口，就像类的成员函数。客户端持有这个 IBinder 对象的引用，通过该引用访问远程服务接口。由于 IBinder 对象实际存在于共享的内核空间中，打破了位于不同进程中用户态内存的隔离，使得客户端与服务端可以像对待本地对象一般，使用远程对象。

从组件的视角来看，Binder 系统中除了客户端与服务端以外，还包括了管理各种服务的 Service Manager，以及运行在内核态的 Binder 驱动。Service

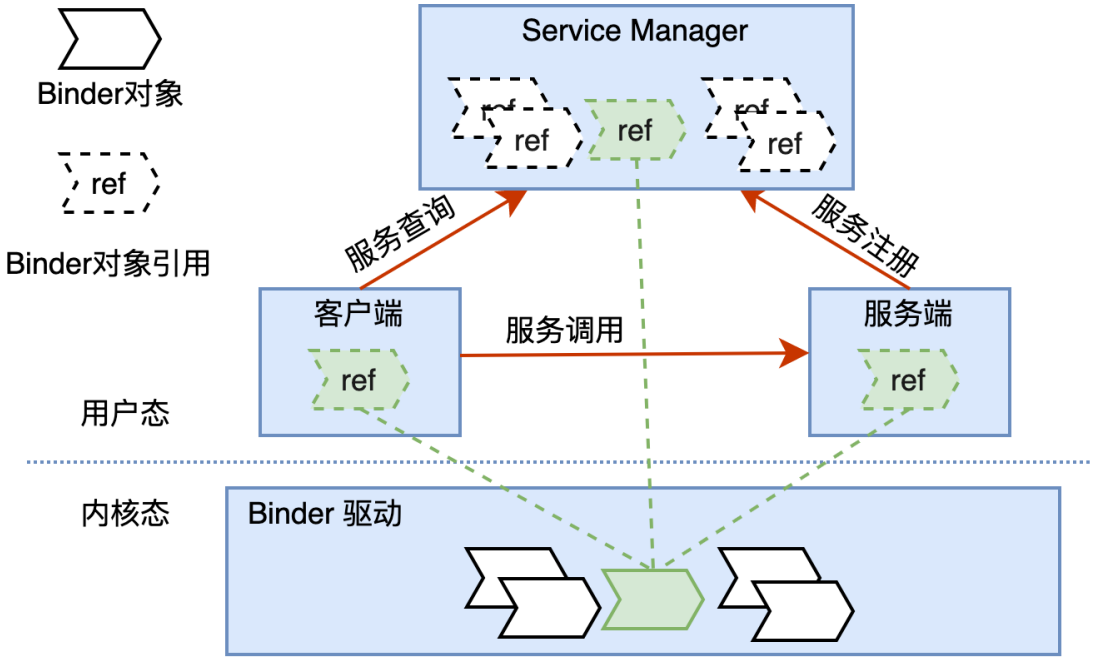

图 2-2: Binder 系统架构

Manager 是整个 Binder 系统中的管理者，对于实名服务来说，服务端初始化服务并注册到 Service Manager 中，客户端在通信前需要先从 Service Manager 中查询对应服务对象的引用。对 Linux 内核来说，Binder 是一个特殊的字符型设备/dev/binder，内部实现遵循 Linux 设备驱动模型，提供 open、close、mmap、ioctl 系统调用，用户态的客户端、服务端与 Service Manager 通过上述系统调用与 Binder 驱动进行交互。

2.3.2 AIDL 接口

为方便上层的 Android 应用开发者使用 Binder 机制进行跨进程通信，Android 系统设计了一套通信接口定义语言 AIDL [22]（Android Interface Definition Language）。一个简单的 AIDL 接口示例如 2.1 所示，其中 IFoo 服务内部包含 doBar 接口方法，Baz 对象代表自定义 Parcelable 复杂结构体。

代码 2.1: AIDL 接口示例

```
1 import my.package.Baz;
2 interface IFoo {
3     String doBar(int a, Baz obj);
4 }
```

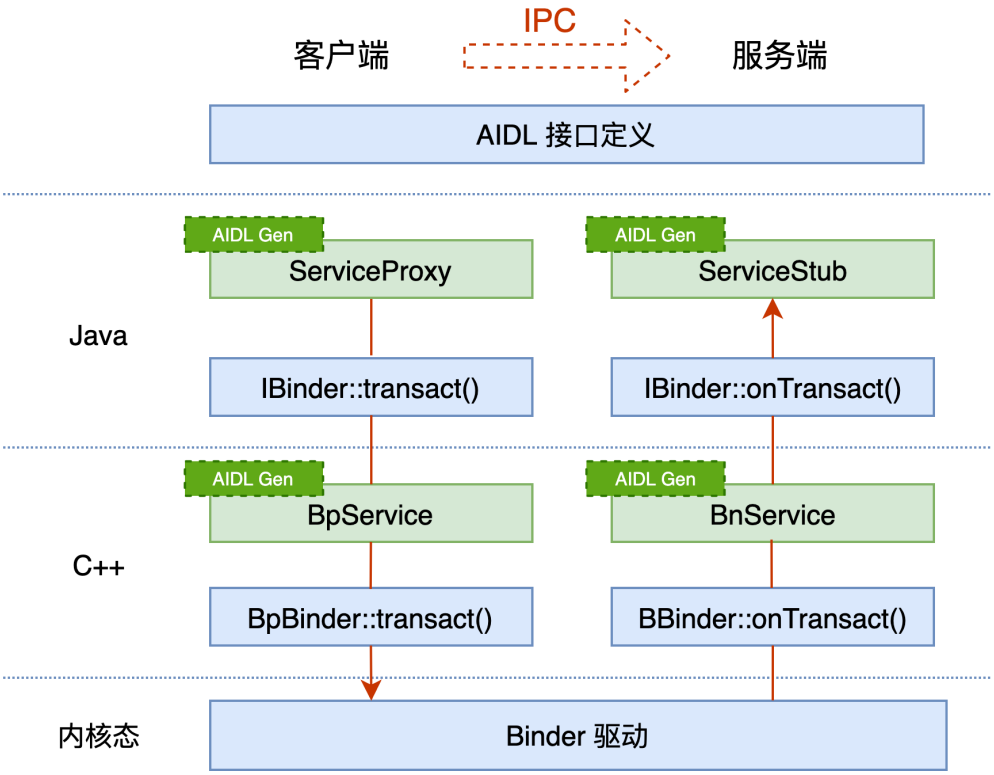

图 2-3: Binder 通信流程

Android 编译工具链中包含 AIDL 代码生成工具，将 AIDL 接口定义转化为客户端与服务端均认可的编程接口（编程语言支持 Java 与 C++），编程接口采用了经典的 Proxy-Stub 设计模式。应用开发者只需按照 AIDL 的语法规范描述服务接口，继承并实现自动生成的代理对象（Java 中对应于 ServiceProxy 与 ServiceStub 对象；C++ 中对应于 BpService 与 BnService 对象），即可利用 Binder 机制实现跨进程通信。上层应用在通过 AIDL 接口进行通信时，就像进行本地函数调用一样方便，AIDL 可以在 Android 任何进程之间使用，既可用于跨应用间的通信，也可用于 Android 系统服务间的通信。

2.3.3 Binder 通信流程

Binder 通信流程如图 2-3 所示：客户端的 IPC 请求通过 Java 层 ServiceProxy 对象与 C++ 层的 BpService 对象，按照 Binder 通信协议序列化为 Parcel 数据，通过 ioctl 系统调用通知 Binder 驱动；Binder 驱动根据服务对象索引，查询服务端所处进程号，通过 ioctl 系统调用通知服务端；服务端收到通知后，通过 C++ 层的 BnService 对象与 Java 层的 ServiceStub 对象，将 Parcel 数据反序列化为相

应接口的调用参数，调用真正的服务接口函数，函数的返回值也按照同样流程序列化并返回给客户端。

在 Binder 通信机制下，客户端与服务端既可以用 Java 实现，也可以用 C++ 实现，不同编程语言间数据结构的兼容性由 AIDL 自动生成的序列化及反序列化层来保证。

2.3.4 Binder 通信协议

传统的 IPC 方式在传输数据时，需要两次用户态与内核态间的数据拷贝：客户端将序列化的数据存放在发送缓冲区中，通过系统调用陷入内核态，内核中断处理例程使用 copy_from_user 函数将客户端进程中发送缓冲区的数据拷贝到内核空间；内核解析通信数据并查找到服务端所在进程，使用 copy_to_user 函数将内核空间的通信数据拷贝到服务端进程中的接收缓冲区。

Binder 系统为提高数据传输效率，使用一种全新的策略，仅需一次用户态与内核态间的数据拷贝。Binder 系统中，服务端利用 mmap 系统调用，从内核空间中直接映射出一片接收缓冲区，这块内存区域既可被服务端用户态进程直接访问，也被 Binder 内核驱动直接使用。当客户端向服务端传输数据时，Binder 驱动会根据数据的大小，从映射出的缓冲区中找到大小合适的内存空间，使用 copy_from_user 函数将数据从发送缓冲区直接拷贝到接收缓冲区。

从内核驱动的角度来看，Binder 客户端与服务端主要通过 ioctl(fd, cmd, arg) 系统调用进行交互，进一步可分为系统控制和数据传输两类交互请求：

系统控制 对应的 cmd 参数为 BINDER_VERSION、BINDER_SET_CONTEXT_MGR、BINDER_SET_MAX_THREADS、BINDER_THREAD_EXIT。其交互请求的语义分别对应于：获得 Binder 驱动的版本号；将当前进程注册为 Service Manager；服务端通过多线程的方式并发处理器客户端请求，客户端告知服务端所需的最大工作线程数，以避免资源的浪费；告知 Binder 驱动当前线程即将退出，Binder 驱动清理内核对应的数据结构。

数据传输 对应的 cmd 参数为 BINDER_WRITE_READ。此时的 arg 参数指向 binder_write_read 结构体，结构体的内部格式如图 2-4 所示，分为两种情况：如果 write_size 不为 0 就意为将数据写入 Binder 驱动；如果 read_size 不为 0 就意为从 Binder 驱动读取数据。其中结构体内的 write_buffer 和 read_buffer 字段指向 binder_transaction_data 结构体数组，结构体中的 code 字段代表请求的目标服务接口编号，data.ptr.buffer 字段指向 Parcel 序列化数据体。序列化数据体由

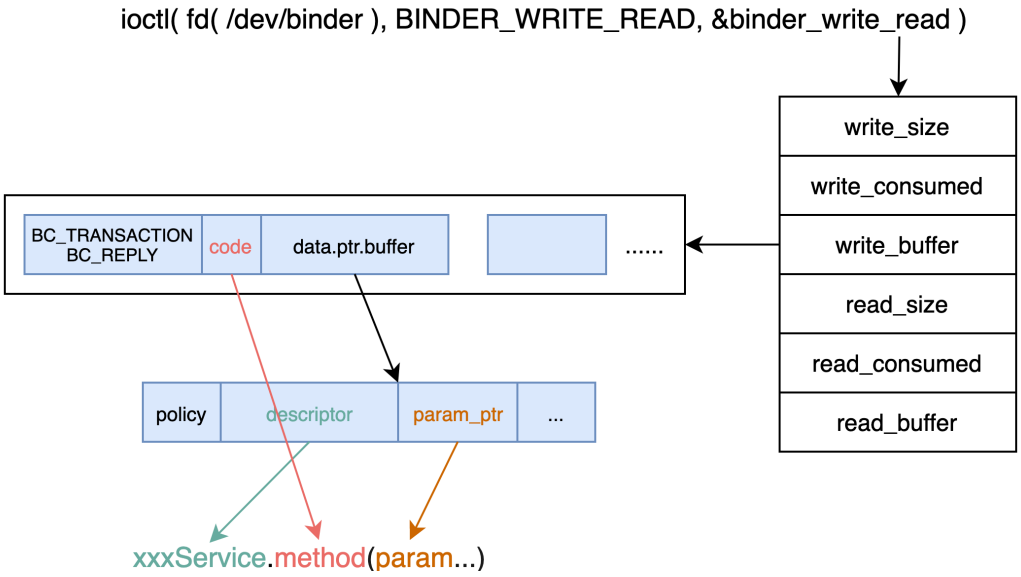

图 2-4: Binder 数据传输格式

policy 传输模式、descriptor 服务标识符、表示接口调用参数的 flat_binder_object 结构体数组构成。Binder 驱动在用户态与内核态间拷贝 flat_binder_object 结构体时，会对结构体内部的字段进行修改，根据 Binder 服务端对象是否位于目标进程中，将其指向 binder_node（Binder 实体）或 binder_ref（Binder 引用）。

2.4 定制 Android 系统

由于 Android 系统的开放性，移动设备厂商往往会对 Android 系统进行深度的定制，以提供更丰富的功能。目前主流的移动设备厂商均推出了自己的定制 Android 系统，如小米的 MIUI，华为的 EMUI，三星的 One UI 等等，这些定制 Android 系统在设备出厂时就预装在手机上，系统的刷写包称为 Android ROM。

与上游 Google 公司的 AOSP 相比，移动设备厂商对 Android 系统的定制化主要集中以下四个方面：

Linux 内核驱动 与 PC 或服务器不同，手机上的操作系统与设备自身的硬件有很强的绑定关系，上游 Linux 内核对硬件设备的支持不能满足移动设备快速迭代的需求。移动设备厂商为了支持更有竞争力的硬件配置（如高像素摄像机、高刷新率显示屏、AI 加速处理器等），会在定制的 Android 内核中添加专用硬件设备驱动。

Android 应用框架 Android 系统在内核设备驱动之上添加了硬件抽象层，向下屏蔽驱动的实现细节，向上提供设备的统一访问接口，这些访问接口又被上层服务所组合，以 AIDL 接口的形式暴露给应用使用。为了适配新添加设备驱动的功能，移动设备厂商会对硬件抽象层与系统服务做修改，改变或拓展 AOSP 中的 AIDL 服务接口定义。

权限配置 Android 系统在 Linux 基于 UID 与 GID 的安全机制之上，实现了 Permission 安全机制，支持更灵活的应用权限管理。Android 应用框架通过系统中的配置文件（主要位于/system/etc/permissions 等目录下）来完成应用权限管理系统的初始化，出于用户体验、管理便利性等方面的考虑，移动设备厂商会对这些权限配置文件进行修改。

预装应用 AOSP 主要包含 Android 基础框架层以下的代码，Google 公司将常见的应用（邮箱、地图、应用商店、浏览器等）打包成 Google 移动应用服务 GMS 单独出售。移动设备厂商基于性价比、产品竞争力、易用性等方面的考虑，往往会在定制 Android 系统中预装自家开发的一套系统应用，而不是选择 GMS。具不完全统计 [23]，定制 Android 系统中 80% 以上的应用来源于设备厂商或其他商业公司。

2.4.1 安全隐患

不少研究表明移动设备厂商对 Android 系统的定制化带来了不少额外的安全隐患：Zhou 等人分析了 2423 个 Android ROM 中 Linux 设备驱动文件的访问权限配置，研究发现有 1290 个 ROM 中存在公开访问权限（比对应的 AOSP 中的文件访问权限更宽松）的设备驱动文件，并基于此成功构造了 Touchscreen Keylogger、Screenshot Capture 等隐私窃取攻击 [24]；Yousra 等人分析了 591 个 Android ROM 中对于应用权限的配置（UID/GID、应用包名、组件可见性、Permission 等配置项），发现定制 Android 系统对这些配置项有较多的修改，并且修改后的权限配置很有可能导致安全漏洞 [7]；Zhang 等人分析了 606 个 Android ROM 中的应用卸载残留问题，通过静态分析的方式收集在应用安装与卸载过程中敏感信息的写入与删除，发现平均每个 ROM 中有 106 处应用卸载残留问题 [25]。

对于定制 Android 系统中的普遍存在的预装应用，Julien 等人的研究 [23] 发现这些应用往往存在如下安全隐患：

- 手机厂商定制的应用仅有少部分可以在公开的应用商店（如 Google Play

Store、Apkpure、酷安等）中获取到，使得这些定制应用被以往的安全研究工作所忽视。

- 相较于 AOSP 的权限配置，很多预装应用被赋予了过多的系统关键权限，这些应用本身的漏洞很有可能导致它们被攻击者所控制，进一步导致用户敏感信息的泄漏。
- 预装应用通过可暴露组件，将其资源和功能共享给其他应用使用，这些暴露组件在处理外部请求时如果没有对调用者权限进行校验，则很有可能被攻击者所利用。
- 这些预装应用中普遍存在收集个人隐私数据，追踪用户等行为。

2.5 Android 组件间通信问题研究

Android 系统及应用通过组件间通信（Inter-Component Communication，ICC）来实现组件的复用，这种机制使得 Android 应用趋向于低耦合高内聚的软件设计模式。由于用户往往会在设备上安装几十甚至上百个应用，在这样一个复杂的系统环境之中，恶意程序同样也可以利用组件间通信的方式，绕过 Android 系统的安全隔离机制，窃取用户敏感信息，执行非法指令等恶意行为。

不少研究从组件间通信的角度，对隐私泄漏、合谋攻击等漏洞进行挖掘。挖掘这类漏洞的挑战在于具有安全隐患的程序执行路径分散在各个应用之中，仅仅分析其中的一个应用无法检测出安全问题，需要从通信关键 API 的调用参数中还原出多个应用组件的调用关系。

已有工作主要以 Intent 为通信载体，研究组件间通信问题。Intent 是一个由目标组件名称以及键值对形式的额外数据构成的消息对象。启动 Activity、启动 Service、广播消息等常见的 Android 应用管理功能皆是通过 Intent 来实现。

Chin 等人第一个指出 Intent 存在的安全隐患，将基于 Intent 的攻击分为两类，Intent 接受者未授权与恶意的 Intent 消息注入，并通过静态分析的方式检测 Intent 的内容是否安全 [26]。

Intent Fuzzer [27] 以 Intent 为测试用例，使用模糊测试的方法对手机上的预装及第三方应用组件进行测试，测试过程中触发了应用反复 crash、甚至 Android 系统重启等漏洞。

FlowDroid [28] 对单个应用利用静态分析的方式，生成虚拟的生命周期和

回掉函数，以构造完整的程序调用图（CG）和过程间控制流程图（ICFG），追踪从 source 到 sink 的数据流，分析出敏感信息流的泄漏路径。

IccTA [29] 利用字节码插桩技术修改应用，以构建跨应用间连续的控制流图，使用静态污点分析的方式，结合 FlowDroid 的方法，对跨应用的数据流进行追踪。

第三章　服务接口签名逆向工程

为了生成有效的服务接口测试用例，在对服务进行模糊测试前，我们需要先提取出服务的接口签名（包括接口编号、接口方法及返回值类型）。已有工作中对于服务接口签名的提取方法有如下三种：

Binder 通信数据监控　BinderCracker [30] 在设备驱动的层次，收集 Binder 通信的底层数据，对该数据结构进行变异，重放该 Binder 通信数据对目标服务进行测试。属于黑盒的 record-and-replay 方法，测试效果很大程度依赖于预先记录通信数据的质量，容易漏过不常使用的接口，而且对通信数据的随机变异很难生成合法的服务接口调用参数，测试效率较低。

Java 反射　在 Java 层，通过 Service Manager 对象的 getService 方法，获取 Java 系统服务的类名，利用反射技术获取该 Java 类的方法签名。该方法效率高、实现简单且不依赖于源码，被不少灰盒测试工具 [10][31] 所使用。但只能应用于 Java 服务，不能应用于不注册于 Service Manager 中的匿名应用服务。

源代码静态分析　绝大多数服务的通信函数是通过 AIDL 代码生成工具自动生成的，Gu 等人通过 AOSP 源码中的 AIDL 接口定义提取出服务接口签名 [20]。为进一步提高接口类型模型的精度，FANS [21] 通过 Native 服务 C++ 源码的静态分析，根据变量类型与名称推测出不同接口间参数与返回值的依赖关系。从源码中提取的接口签名精度高，但不能应用于闭源服务之上。

上述方法要么在接口签名的提取精度上有缺陷，要么依赖于服务的源码，不能满足定制 Android 系统中闭源的 Java 与 Native 服务的测试需求。

我们发现服务接口签名与服务 Binder 通信序列化函数中的代码特征相对应，即使是在编译生成的字节码与 ARM 汇编代码中，这些代码结构特征也得到了一定程度的保留。只需从服务的编译产物中还原出这些代码结构特征，即可推测出服务接口签名。

对于定制 Android 系统中的预装应用服务、Java 系统服务以及 Native 系统服务，这三类待测服务的接口签名提取流程如图 3–1 所示，包括服务编译产物提取、服务通信序列化函数提取、服务接口签名推断三个步骤。

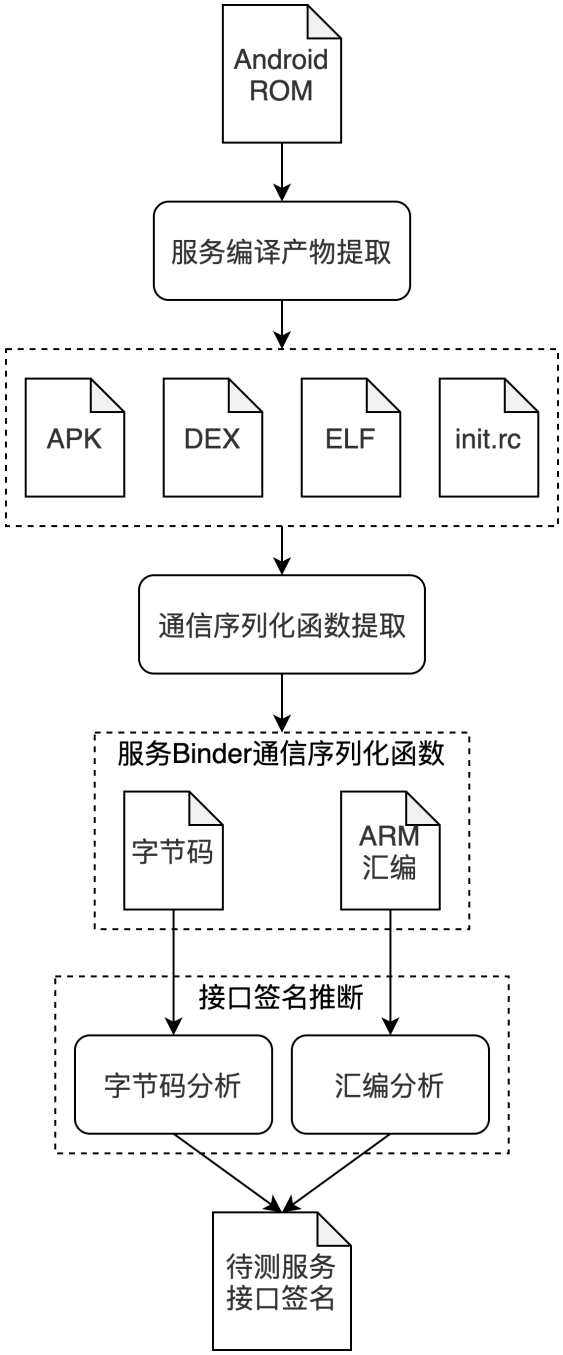

图 3-1: 闭源服务接口签名提取流程

3.1 编译产物提取

对不同类型服务进行逆向分析所需的编译产物不同，它们所处文件的提取规则如表 3-1 所示。

Android ROM 中预装应用的 APK 文件一般位于 /system/app 等目录下，使

表 3-1: 编译产物所处文件提取规则

类型	ROM 中位置	文件格式
应用服务	/system/app /system/priv-app /system/vendor /system/preload	APK
Java 系统服务	/system/framework /system/vendor/framework	JAR/odex/vdex
Native 系统服务	/system/lib /system/lib64 /system/vendor/lib /system/vendor/lib64	ELF
	/system/etc/init /system/vendor/etc/init	rc

用 Apktool [32] 工具从中提取出包含应用配置的 AndroidManifest.xml 文件与包含 Dalvik 字节码的 classes.dex 文件。

Android 系统为了将 Java 实现的系统服务暴露给其他应用使用，应用启动前会预先将依赖的 Java 对象加载到 ART 虚拟机的 Class Loader 中，相关的编译产物以 JAR 文件的形式位于 /system/framework 等目录下。预加载 Java 对象配置信息保存在 /system/etc/preloaded-classes 文件中，由 Zygote 孵化进程读取并加载。

缺少 DEX 的情况 在 Android 8.0 之后，应用 APK 或 /system/framework 下的 JAR 中一般不包含 DEX 文件，而是以 odex 文件（包含 AOT 编译过后的二进制码）与 vdex 文件（包含未压缩的 Dalvik 字节码以及加快验证速度的元信息）的形式存在，我们使用 LIEF 工具 [33] 从这些文件中还原回 Dalvik 字节码。

Android 系统采用动态链接库的形式，将 C++ 系统服务的接口暴露给其他程序使用，这些动态链接库文件一般位于 /system/lib 等目录下，文件名形如”libxxxservice.so”。Native 系统服务的配置信息则以 Android 初始化语言的形式记录在 init.rc 等文件之中，位于 /system/etc/init 等目录下。

3.2 通信序列化函数提取

接下来，结合 Android 系统运行时状态，从上述编译产物中提取出待测服务通信序列化函数的字节码或汇编代码，分为两个步骤：

- 确认待测服务，包括预装应用中暴露的绑定式服务，以及 Java 与 Native 系统服务；
- 根据服务的组件生命周期回调函数、序列化函数签名以及服务标识符等特征，提取待测服务的通信序列化函数。Java 服务的通信序列化函数名为“Boolean onTransact(Integer code, Parcel data, Parcel reply, Integer flags)”，一般位于形如“XXXService$Stub”的 Java 类中；Native 服务的通信序列化函数名为“status_t onTransact(uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)”，一般位于形如“BnXXXService”的 C++ 类中。

3.2.1 应用服务

代码 3.1: 应用服务的配置文件

```
<service android:name=".FooService" android:exported="true">
  <intent-filter>
    <action android:name="FOO_ACTION"/>
    <category android:name="FOO_CATEGORY"/>
  </intent-filter>
</service>
```

Android 应用组件根据可见性可分为私有组件和公有组件（也叫暴露组件）：私有组件只允许与同一应用中的组件，或者具有相同 UID 应用中的组件发生交互；暴露组件允许与其他应用程序发生交互，容易被其他恶意程序所利用，我们关注预装应用中的暴露服务组件。AndroidManifest 文件中包含该应用的组件配置信息，文件格式如代码 3.1 所示。从 android:exported=”true” 属性即可确认为暴露服务；若该属性不存在，当存在至少一条 intent-filter 时则为暴露服务。

以往工作 [17][34] 主要关注以 Intent 为通信接口的启动式服务，忽略了以 AIDL 为通信接口的绑定式服务，我们关注预装应用中的绑定式服务。绑定式服务通过 onBind(Intent) 方法返回给调用者 IBinder 接口对象，之后调用者通过该接口对象与服务进行交互。从配置文件中的 android:name 属性确定服务

类名，若该 Java 类实现了 onBind 方法，则为绑定式服务，通过数据流分析计算出 onBind 方法的返回值具体类型，服务序列化函数所在的 Stub 类一定是 onBind 方法返回值的父类。

3.2.2 系统服务

所有的系统服务都注册在 Service Manager 中，通过它的 listService 接口列举出系统服务名称与服务标识符 descriptor。

每个服务必须实现一个返回值为 descriptor 的特殊接口，一般通过 AIDL 代码生成工具自动生成，因此根据服务序列化 Stub 类中的 descriptor 字符串值可确定该 Stub 类对应于哪一个服务。

绝大多数 Java 系统服务位于 system_server 进程中，如 BackupManagerService、PackageManagerService、PowerManagerService 等等，每个服务以一个单独线程的形式运行。读取该进程的虚拟内存使用情况表 /proc/[pid]/maps，从该表的最后一列中筛选出该进程所加载的字节码文件（以 jar、vdex、odex、oat 为文件后缀名）。在这些字节码文件中根据序列化函数签名以及 descriptor 字符串两个特征，识别出该服务通信序列化函数所在的 Stub 类。

代码 3.2: cameraserver.rc 配置文件

```
1 service cameraserver /system/bin/cameraserver
2   class main
3   user cameraserver
4   group audio camera input drmrpc
```

Native 系统服务一般位于独立的守护进程之中，如 cameraserver、mediaserver、audioserver 等等。这些进程的启动参数以 Android 初始化语言（Android Init Language，AIL）的形式记录在 init.rc 等配置文件之中。以 cameraserver 为例，对应的配置文件如3.2 所示，从中可提取出进程可执行文件、UID、GID 等服务启动配置信息。解析服务进程可执行文件在动态链接过程中所依赖的 so 库文件，在这些动态链接库文件的符号表中，先根据序列化函数签名筛选出所有服务对象的序列化函数，接着根据服务标识符 descriptor 在这些序列化函数的汇编代码中进行特征匹配，以确认服务与其通信序列化函数的对应关系。

我们通过拦截系统服务在 Service Manager 中的注册过程，以获得服务名称、所处进程 PID、Binder 驱动内的服务索引 handle 等运行时信息。Service

Manager 的服务端使用 C/C++ 实现，在 AOSP 中服务注册过程的关键函数定义如代码 3.3 所示。

代码 3.3: 服务注册函数定义

```
int do_add_service(struct binder_state *bs,
                   // 服务名称
                   const uint16_t *s, size_t len,
                   uint32_t handle, uid_t uid,
                   int allow_isolated, pid_t spid);
```

拦截函数调用前需要知道函数在进程中的地址，由于 do_add_service 是一个内部的函数调用，它的地址并不会记录在进程可执行文件的符号表中。根据 do_add_service 函数的参数类型信息，并结合 ARM 中的函数调用规约（calling convention），我们可以生成出对应的汇编函数开场白（prologue）。由于定制 Android 系统很少对 Service Manager 实现进行修改，我们以汇编函数开场白为特征，扫描 Service Manager 服务端可执行文件，即可检索到 do_add_service 的函数地址。

3.3 服务接口签名推断

服务接口签名包括接口编号、接口方法参数以及返回值类型，这些信息都可对应于服务 Binder 通信序列化函数中的代码特征。即使是在编译过后的字节码与 ARM 汇编代码中，这些代码特征也得到了一定程度的保留，我们从编译产物中识别出这些代码特征，以推测服务接口签名。

3.3.1 Binder 通信序列化函数特征

服务的 Binder 通信序列化 onTransact 函数结构如图 3-2 所示。函数的骨架是一个以 code 作为参数的 switch-case，每个 case 分支实现了该服务某个接口的参数与返回值的序列化过程，code 表示服务的接口编号，data 表示接口请求参数的 Parcel 序列化数据对象，reply 表示接口返回结果的 Parcel 序列化数据对象。switch-case 代码结构中，以特殊常量 0x5F4E5446 为条件的分支中，将服务标识符 descriptor 写入 reply 中。通信序列化函数每个分支的序列化过程可进一步分为三段：

- 从 data 中读取接口参数，对应于形如“readXXX”的反序列化 API 调用。

```
ServiceStub::onTransact(code, data, reply) {
    switch (code):
        case 1:
            // 反序列化参数
            param1 = data.readInt()
            param2 = data.readFloat()
            result = foo(param1, param2)
            // 序列化返回值
            reply.writeString(result)
            return NO_ERROR
        case MAGIC_NUMBER:
            reply.writeString(SERVICE_DESCRIPTOR)
            return NO_ERROR
}
```

图 3-2: onTransact 函数结构

- 执行本地函数调用，若调用成功，则调用 writeNoException 函数标记成功，否则调用 writeException 函数写入异常信息，onTransact 函数直接结束。
- 将调用结果写入 reply 中，对应于形如“writeXXX”的序列化 API 调用。

由于通信序列化过程代码往往由 AIDL 代码生成工具自动生成，结构较为稳定（事实上，即使是开发人员手工编写的通信序列化过程函数，也具有十分类似的代码结构）。即使是编译过后高度精简的二进制产物，也可从中识别出这些代码结构特征。

3.3.2 Parcel 序列化 API

根据序列化数据类型的不同，可将 Parcel 序列化 API 分为六类：

基本数据类型 包括 readByte、readInt、readString 等 API。序列化数据的字节序由当前系统的处理器所决定。

基本数据类型数组 包括 readByteArray、readIntArray、readStringArray 等 API。Parcel 序列化后的数组对象由开头表示数组大小的 32 位无符号整数与之后依次排列的数组项构成。为防止歧义，空对象会写入-1 作为占位符。

Bundle 包括 readBundle、writeBundle 等 API。Intent 使用 Bundle 对象携带额外的信息，Bundle 为键值对形式的 Map，key 固定为字符串类型。为防止歧义，空对象会写入-1 作为占位符。

Parcelable 包括 readParcelable、createTypedArray 等 API。用于在进程间传输应用自定义的复杂结构体，这些复杂结构体需要实现自定义的序列化方法，Bundle 对象也是一种特殊的 Parcelable 对象。当匹配到这类对象的序列化 API 时，我们需要递归地进入该对象的 writeToParcel 或 readFromParcel 函数实现中，重复以上所述的序列化 API 识别过程。

FileDescriptor 包括 readFileDescriptor 和 writeFileDescriptor 这两个 API。常常用于实现跨进程文件共享、套接字共享等功能，底层对应于 Linux 中的文件描述符。Parcel 在序列化 FileDescriptor 对象时，将其转换为 Binder 内核驱动中的 flat_binder_object 对象，其定义如代码 3.4 所示，并将 type 设置为 BINDER_TYPE_FD，文件描述符存储在 handler 中。Binder 驱动在传递文件描述符时，在目标进程中重新创建对应的文件描述符。

IBinder 包括 readStrongBinder、createBinderArray、createBinderList 等 API。IBinder 对象本质是一种特殊的跨进程引用，是指向内核空间内 Binder 实体 binder_node 的引用。Parcel 在序列化 IBinder 对象时，将其转换为 flat_binder_object 对象，其中的 type 有四种可能的取值：

- BINDER_TYPE_BINDER 或 BINDER_TYPE_WEAK_BINDER：代表传输的是 Binder 实体，对应于 Binder 驱动内的 binder_node 对象。当 Binder 驱动在传输 flat_binder_object 时，若发现其指向的 binder_node 不存在，则在 Binder 驱动中创建并加入到 binder_proc 所指向的红黑树之中。
- BINDER_TYPE_HANDLER 或 BINDER_TYPE_WEAK_HANDLE：代表传输的是对 Binder 实体的引用，对应于 Binder 驱动内的 binder_ref 对象。即使是指向同一个 Binder 实体的应用，不同进程的 binder_ref 对象互相独立。因此 Binder 驱动在传输 flat_binder_object 时，需要修改 handle 以指向目标进程的 binder_ref 对象，若 binder_ref 对象不存在，则在目标进程中创建之。

代码 3.4: flat_binder_object 结构体定义

```
1 struct binder_object_header {
2   __u32 type;
3 };
4 struct flat_binder_object {
5   struct binder_object_header hdr;
6   __u32 flags;     // 传输方式
7   union {
```

```
8       __u64 binder; // 指向Binder驱动内binder_node结构对象的指针
9       __u32 handle; // 指向Binder驱动内binder_ref结构对象的指针
10    }
11    __u64 cookie;    // 指向C++层BBinder对象的指针
12  };
```

3.3.3 从字节码中提取服务接口签名

Java 服务的编译产物为 Dalvik 字节码，我们借助 Soot 框架 [35] 将其转换为 Jimple 字节码，Jimple 字节码的可读性较好，不难从中推测出服务接口签名：

- Jimple 字节码中通过 lookupswitch 指令与 label 标签来表示 switch-case 结构，lookupswitch 指令中直接记录了相应分支的跳转条件。
- 在字节码中识别 Parcel 序列化 API 调用：对于 staticinvoke、virtualinvoke 方法调用指令，指令的参数直接包含目标函数签名，对应于基本数据类型及数组、Bundle、ParcelFileDescriptor、IBinder 等数据的 Parcel 序列化 API；对于 interfaceinvoke 接口调用指令，需要确定实现了该接口的实例对象类型，通过前向数据流分析找到该实例对象的构造函数调用，以确定实例对象的具体数据类型，对应于 Parcelable 复杂结构体的 Parcel 序列化 API 调用。

3.3.4 从 ARM 汇编中提取服务接口签名

对于 Native 服务，从编译后生成的高度精简的 ARM 汇编中提取出上述特征有一定的挑战，即使是当前最先进的反编译工具 RetDec [36] 与 Hex-Rays [37]，也仍然不能从汇编代码中完美还原回 C++ 代码。由于汇编代码的可读性较差，推测服务接口签名的过程中有以下两点挑战：

- **还原 switch-case 结构** 服务通信序列化 onTransact 函数以 switch-case 为基本骨架，由于 ARM 汇编中不存在 switch-case 指令，分析的第一步需要将汇编代码块还原回源码中的 switch-case 结构。如果按传统方式将汇编函数体转化为控制流图（Control Flow Graph, CFG）来分析，会发现函数体所包含的基本块之间并不相连。
- **识别 Parcel 序列化 API** 在 ARM 汇编中，函数调用存在多种实现方式，对应于 B、BL、BLR 等分支跳转指令。静态函数、虚函数等不同类型的函

数调用被编译后采用的寻址方式又不尽相同，这给从汇编中识别并解析出函数调用的语义带来了挑战。

3.3.4.1 还原 switch-case 结构

Android 系统以 Clang/LLVM 为编译工具链，switch-case 语句编译后对应于三种汇编代码结构：多重分支跳转、跳转表（jump table）、查找表（lookup table）。编译器根据分支跳转的效率以及汇编代码的大小决定生成哪种汇编代码结构。

对于通过 AIDL 代码生成工具生成的通信序列化 onTransact 函数来说，其中接口编码 code 为连续的枚举值。当分支跳转条件为连续的整型值时，Clang/LLVM 会根据分支数量生成多重条件跳转或者 jump table 汇编代码结构：分支数小于 4 时，生成前者，否则生成后者。

多重分支跳转 组合多个分支跳转语句来表示 switch-case 的多个分支。每一个分支的开头指令满足如下特征：

- 首先是 CMP 指令，比较接口编号 code 所处寄存器与常数是否相等；
- 接着是条件跳转指令，基于之前 CMP 指令的比较结果，跳转到对应分支的起始指令或下一条 CMP 指令，如 B.EQ、B.NE 等指令。

jump table jump table 实际上是一个以地址长度为宽度，以 case 数量为长度的数组，一般位于 ELF 的.rodata 部分（对于 ARM32 指令集，会将 jump table 内嵌在函数体的汇编代码段中）。对于 switch-case 结构生成的 jump table 来说，其中表项存储了 case 分支开头指令的地址与表的基址之差，将表项与表基址相加即可计算出 case 分支的起始指令地址。

为了从.rodata 中提取出 jump table，我们还需要从汇编代码中提取出 jump table 的长度与基址两个属性：

- switch-case 结构中的 default 分支，对应的汇编形如代码 3.5 的 1-7 行所示，由于接口编码 code 是从 1 开始递增的枚举型整数，因此 default 分支的判断条件为 code 是否大于该枚举类型的最大值，这个最大值也就是 jump table 的长度。
- 汇编通过间接寻址的方式使用 jump table，如代码 3.5 的 9-16 行所示，其中 10-11 行计算出了 jump table 的基址，对应计算公式为”(ADRP 基址 » 12) « 12 + 0x17000 + 0x700”，0x17000 代表.rodata 基址相对当前代码页的偏移量，

0x700 代表 jump table 基址相对.rodata 基址的偏移量。

代码 3.5: AArch64 汇编中对于 jump table 的使用

```
1  // W1寄存器中存储了onTransct的code参数值
2  SUB          W8, W1, #0x1
3  // 对应于switch-case中的default分支
4  // W8寄存器值无符号大于或等于25时，跳转到default分支
5  // 由此可推断出这个switch-case结构的分支数量的25
6  CMP          W8, #0x19
7  B.HI         45810
8  ...
9  // X9寄存器中为jump table基址
10 ADRP         X9, 0x17000
11 ADD          X9, X9, 0x700
12 // X8寄存器中为对应表项的内容
13 LDRSW        X8, [X9, X8 LSL 0x2]
14 ADD          X8, X8, X9
15 // X8中存放了case分支的指令地址
16 BR           X8
```

3.3.4.2 识别 Parcel 序列化 API

ARM 汇编中，通过 B、BL、BLR 等分支跳转指令来表示函数调用，根据跳转指令的寻址方式，我们将函数调用分为两类：

直接函数调用 指令的参数中包含目标函数地址，根据地址直接在符号表中查询即可知道对应的函数名。对于基本数据类型、基本数据类型数组、Bundle 对象、ParcelFileDescriptor 对象、IBinder 对象这五类 Parcel 序列化 API 调用都属于这种情况。代码 3.6 中的第 5、20 行皆属于直接函数调用。

间接函数调用 指令的参数不直接包含目标函数地址，需要先通过运算将地址保存在寄存器中。最典型的例子就是 C++ 中虚函数的调用，如代码 3.6 的 10-17 行所示。Parcelable 复杂对象的序列化接口 writeToParcel 与 readFromParcel 在基类中定义为虚函数，属于间接函数调用。识别间接函数调用的挑战在于如何计算出目标函数地址，本文参考 VTint [38] 中的方法，并将其从 x86 移植到 ARM 上，从汇编中识别虚函数调用，并确定对应的 Parcel 序列化 API：

- 识别构造函数调用；

- 在构造函数中，会进行虚函数表的初始化，以此特征可从构造函数中识别出该对象的虚函数表基址；
- 根据与虚函数表基址的偏移量计算出实际的函数调用地址；
- 以该地址为检索条件，从符号表中查询出目标函数签名；

代码 3.6: Parcel 序列化过程 AArch64 汇编

```
1  // X19寄存器里为this指针
2  // 将this指针转换为指向IBinder对象的指针
3  ADD          X1, X19, 8
4  MOV          X0, X21
5  BL           0x3b5f8 // android::Parcel::checkInterface
6  // 若checkInterface返回值为0，则跳转到onTransact函数结束
7  TBZ          W0, 0, 0x45dd4
8
9  // X8寄存器里为虚函数表的地址
10 LDR          X8, [X19]
11 MOV          X0, X19
12 // 将虚函数表中的表项值保存到X9寄存器
13 // X9寄存器实际指向了同一类下的成员函数地址
14 LDR          X9, [X8, 0xc0]
15 ADD          X8, SP, 8
16 BLR          X9
17 ADD          X1, SP, 8
18
19 MOV          X0, X20
20 BL           0x3b658 // android::Parcel::writeString8
```

由于 Parcel 序列化 API 的实现统一位于动态链接库 libbinder 中，为了实现运行时的动态链接，所以编译时一定会保留这些函数的符号信息。

3.4 工具实现

按照上述方法实现了服务接口签名逆向提取工具 RevExtractor。工具中包含文件提取、字节码分析、汇编逆向分析、运行时配置信息提取四个主要模块：

文件提取 使用 Python 代码实现。所有的服务相关文件都位于 system

分区下，对应于 Android ROM 中的 system.img，可能会以两种形式存在：完整的 ext4 分区镜像，直接使用 mount 命令挂载即可；稀疏镜像（sparse image [39]），是 Android 中特有的稀疏表示的分区镜像，镜像中去除了为零的填充数据，使用 simg2img 工具将其转化为完整的 ext4 分区镜像。

字节码分析 先使用 dex2jar [40] 工具，将 dex 文件转换为 Jar 文件，之后借助 Soot 框架 [35] 将 Dalvik 字节码转换为 Jimple 字节码进行分析，主要以 Java 代码实现。

汇编逆向分析 借助 Miasm 框架 [41] 以 Python 代码实现，由于 Android 系统可同时支持 ARM32 与 ARM64 两种指令集，对于这两种情况需要分别进行处理。

运行时配置信息提取 其中的函数调用拦截部分借助 Frida 框架 [42] 以 Javascript 代码实现，其余以 Python 代码实现。

3.5 本章小结

本章介绍了服务接口签名逆向提取方法。首先介绍了已有工作中提取服务接口签名的方法，以及这些方法为何不适用于闭源的 Java 与 Native 服务。然后介绍了服务接口签名逆向工程的基本思路及工作流程，包括三个步骤：APK、DEX、ELF 等服务编译产物的提取；结合 Android 系统运行时状态，根据服务配置、服务标识符等特征，提取出服务的通信序列化函数；从服务的编译产物中，识别出 switch-case、序列化 API 调用等代码特征，以此推测出服务接口签名。最后综合以上方法，实现了服务接口签名逆向提取工具 RevExtractor。

第四章　系统上下文感知的服务模糊测试工具

观察 Android 服务的内部实现逻辑，我们可以发现如下特征，如代码 4.1 所示：在执行真正的业务逻辑之前，需要对调用者权限、系统能耗模式等 Android 系统上下文进行检查，如果检查不通过，服务接口调用就提前结束。

代码 4.1: 服务实现中对 Android 系统上下文的检查

```
1  // 检查调用者是否有权限，PMS代表PackageManagerService
2  if(PMS.checkUidPermission(...) != PERMISSION_GRANTED) {
3    return false;
4  }
5  // 检查调用者是否活跃，USS代表UsageStatsService
6  if (USS.isAppInactive(...)) {
7    return false;
8  }
9  // 检查节能模式下是否允许执行，DIC代表DeviceIdleController
10 if (!DIC.isPowerSaveWhitelistApp(...)) {
11   return false;
12 }
13
14 BUG();
```

然而，这些 Android 系统上下文往往与接口的输入参数无关，如果这些检查不通过的话，那么无论怎么生成输入参数，这个接口的测试深度都不会有太大变化，无法测试到真正有问题的代码。因此，对于服务接口的测试中，除了生成接口输入参数以外，还需要考虑 Android 系统的上下文。

实际上，这些对 Android 系统上下文的检查往往都是通过外部服务的接口调用来实现的，可以对这些接口调用进行劫持以模拟 Android 系统上下文。我们在 Binder IPC 层面统一地对这些外部服务接口调用进行劫持，以模拟 Android 系统上下文。

我们基于 Binder IPC 劫持、动态二进制插桩等技术，以服务接口签名指导测试用例的生成与变异，实现了一个系统上下文感知的 Android 服务模糊测试

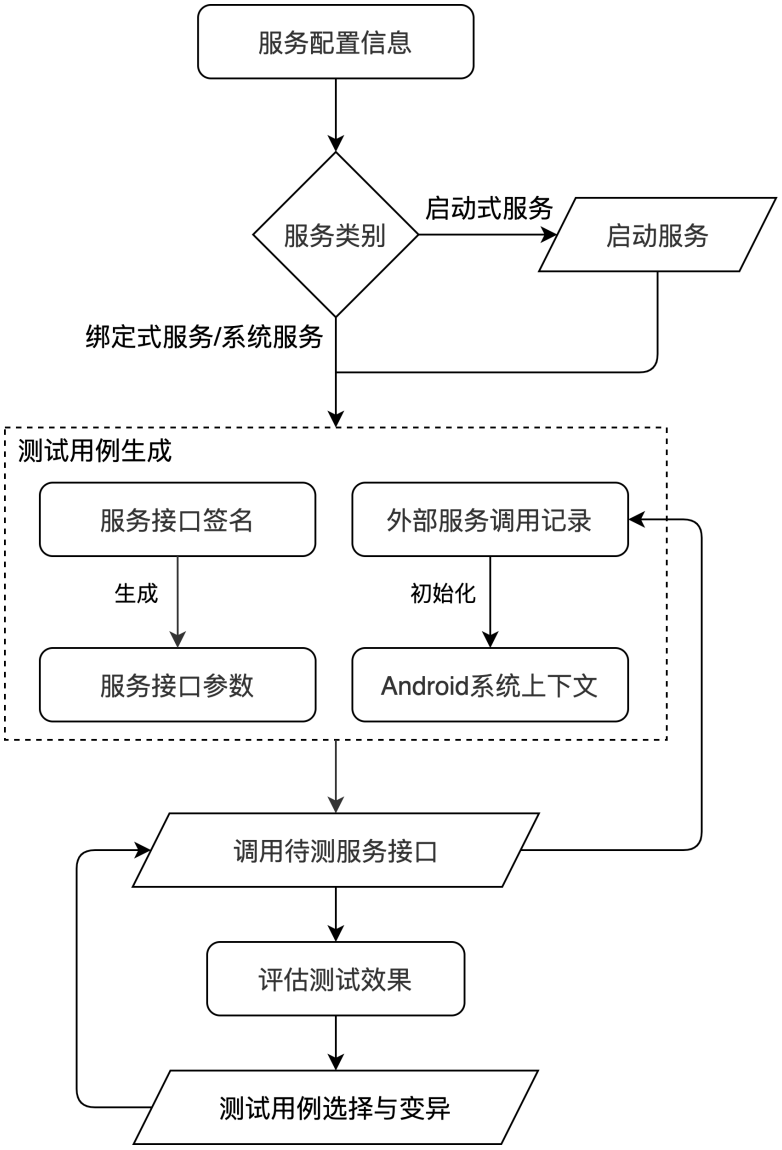

图 4-1: 模糊测试迭代过程

工具 CASFuzzer（Context-Aware Service Fuzzer）。

4.1 服务模糊测试流程

CASFuzzer 定义的服务测试用例包含待测服务标识符 descriptor、接口编码 code、服务接口参数 parameters 以及 Android 系统上下文 MockDroidContext 四个维度的参数，基于服务接口签名对这些参数进行生成与变异，通过基本块覆盖率来评估测试用例的效果，以指导模糊测试过程的迭代。

服务模糊测试的工作流程如图 4-1 所示：

- 选择待测服务接口，根据服务类型及配置启动待测服务；
- 基于服务接口签名生成接口输入参数；
- 随着测试的迭代，收集外部服务调用记录，以初始化 Android 系统上下文；
- 装载 Android 系统上下文，调用待测待测服务接口；
- 根据测试效果，外部服务调用值与接口参数一起进行选择与变异。

4.1.1 启动待测服务

系统服务 对于位于 system_server、mediaserver、cameraserver 等守护进程中的系统服务来说，每个服务以线程的形式运行，只要同一进程中任意服务出错，进程中的 WatchDog 模块会监测到这一异常并退出整个进程。这些守护进程的启动参数记录在 init.rc 配置文件中，由 Android 系统中的 init 根进程所管理，一旦某个服务进程退出则会被 init 根进程自动重启。因此，在测试开始前只需等待对应系统服务自动运行即可，无需手动构造服务启动参数。

应用服务 根据 Android 系统对服务生命周期管理方式的不同，可分为启动式服务与绑定式服务两类，判断的标准是服务是否实现了 onStartCommand 接口，启动式服务需要手动构造启动参数，绑定式服务则无需手动启动：

- 对于启动式服务来说，关键是构造有效的启动 Intent，我们沿用已有工作 [17][34] 中的 Intent 构造方法，对待测服务的 onStartCommand 函数进行轻量的静态分析，提取出 Intent 中 component、action、category 等属性的约束条件，使用约束求解器计算出对应的值，以此构造启动 Intent。
- 对于绑定式服务来说，它的生命周期由 Android 框架来管理：当第一个客户端绑定服务时，该服务会被 Android 框架自动启动；当所有客户端均与服务取消绑定后，Android 框架会自动销毁该服务。Android 框架会检查绑定客户端的应用权限，为提高启动服务的成功率，需要将客户端应用置于 system 用户下。Android 系统通过签名机制来保证敏感应用不会被篡改，所有 system 用户下的应用必须被厂商私钥签名，因此我们无法将应用添加到 system 用户下。为了绕过这一限制，我们拦截待测服务进程中 IPCThreadState::getCallingUid 函数调用，当服务绑定请求来自于客户端应用时，返回 system 用户的 UID。

应用服务启动成功后，它的配置信息由 ActivityManagerService 维护在 ServiceRecord 对象中。通过 ActivityManagerService 服务的 dump 接口，即可获

得所有应用服务的配置信息，从中提取出应用的 PID、UID 等字段，以便在后续的测试中追踪待测服务的运行时状态。

4.1.2 模拟 Android 系统上下文

模拟 Android 系统上下文 MockDroidContext 由一组对 Binder IPC 的劫持规则构成，包括 Binder IPC 匹配规则 BinderIPCFilter 以及接口的模拟返回值 reply。BinderIPCFilter 包含请求的外部服务标识 descriptor、外部服务接口编码 code 以及请求参数匹配规则，匹配规则较为简单，直接对 Parcel 对象逐字节进行比较即可。

用户空间内的客户端进程一般通过 IPCThreadState 与 Binder 驱动进行交互。由于 libbinder.so 导出了 IPCThreadState::transact 函数符号，所有使用 Binder 通信机制的进程一定会链接 libbinder 动态连接库。因此通过 hook 目标进程中 libbinder 动态链接库的 IPCThreadState::transact 函数，即可拦截目标进程的所有 Binder 通信请求。

代码 4.2: hook 目标函数签名

```
1 status_t IPCThreadState::transact(int32_t handle,
2                                   uint32_t code, const Parcel& data,
3                                   Parcel* reply, uint32_t flags);
```

该函数的签名如代码 4.2 所示，在 transact 函数执行结束后，执行 hook 逻辑，根据作为函数参数的 code 以及 data，以及可从 data 对象开头提取出的服务标识 descriptor，我们即可对 BinderIPCFilter 规则逐个进行匹配，一旦满足条件，则将 reply 对象内容替换为 mock 返回值。

MockDroidContext 的迭代更新算法如 4.1 所示。测试工具一开始执行时，MockDroidContext 为空，在测试迭代一段时间后，查询外部服务接口访问记录，以此为种子初始化劫持规则集合。测试迭代过程中，随机选择劫持规则，对其返回值进行变异，若测试反馈效果好，则更新 MockDroidContext。当劫持规则集合的变异次数达到一定上限时，查询获得最新的外部服务接口访问记录，对劫持规则集合进行更新。过于复杂的劫持规则会导致过多 Hook 被加载到待测服务进程上，影响待测服务的执行效率，因此使用 LRU 策略淘汰长时间没有匹配成功的劫持规则。

算法 4.1 MockDroidContext 迭代更新算法

输入： *ctx* 代表 MockDroidContext 的劫持规则集合，*max_mut* 最大变异次数，*s_len* 劫持规则数量限制

```
better ← 0
while better < Min(max_mut, Len(ctx)) do
    <filter, reply> ← RandomSelect(ctx)
    mut_ctx ← ctx ∪ {<filter, Mutate(reply)>}
    if Fuzzing(mut_ctx).cov > Fuzzing(ctx).cov then
        ctx ← mut_ctx
        better ← better + 1
    end if
end while
record ← 更新外部服务接口访问记录
for r ∈ record do
    if <r.descriptor, r.code> ∈ ctx then
        ctx.Update(r)
    else
        ctx.Append(r)
    end if
end for
if Len(ctx) > s_len then
    ctx ← LRU(ctx， s_len)
end if
```

4.2 总体框架设计

服务模糊测试工具 CASFuzzer 主要有以下几个核心模块，其各部分的实现情况如表 4-1 所示：

表 4-1: CASFuzzer 模糊测试工具各模块实现情况

模块名称	编程语言	编程框架	LoC
Injector	C++	Frida	620
CovStalker & Interceptor	C++	Frida	3412
Ececutor & GeneralServiceStub	Java	Android	1528
TCGenerator	Java		5659
总计			11219

其中 Frida [42] 是一款基于动态插桩技术实现的 Hook 框架，支持 Android、

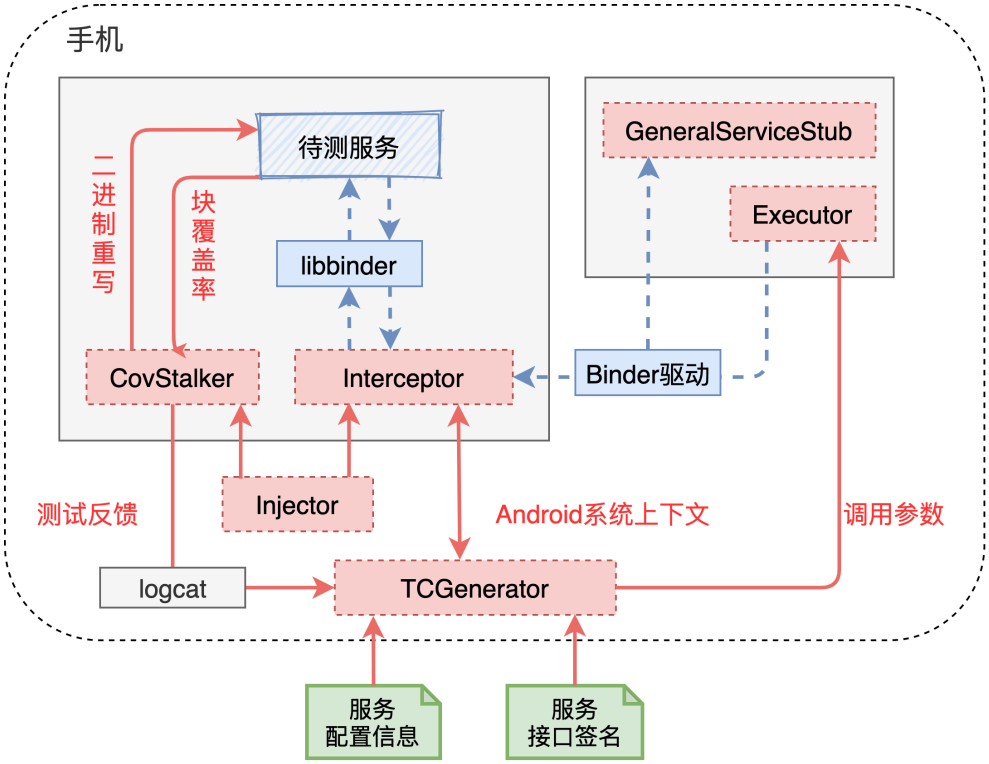

图 4-2: 总体框架

iOS、Windows、Linux、MacOS 等各种主流平台，兼容 x86、ARM 等多种指令集，提供 Javascript、Python、C++ 等多种语言的交互接口。

TCGenerator 是整个工具的核心模块，首先根据配置信息启动待测服务，读取服务接口签名以生成测试用例，确保 mock 的 Android 系统上下文已被 Interceptor 模块所装载后，让 Executor 模块调用待测服务，从 CovStalker 模块和 logcat 中获取测试结果反馈，改进测试用例并重复以上步骤。以单独 Java 进程的形式运行，将执行中间结果保存在 SQLite 本地数据库中。

Injector 负责将 CovStalker 和 Interceptor 模块注入到待测服务进程之中，当待测服务重启后，模块会被重新注入。

CovStalker 被注入到待测服务进程之中，通过动态二进制插桩（dynamic binary instrumentation）的方法修改目标进程的控制流，以记录基本块的访问情况，计算出基本块覆盖率，以 Unix Domain Socket 的形式提供给 TCGenerator 模块覆盖率查询接口。

Interceptor 被注入到待测服务进程之中，在 Binder IPC 层面记录并劫持服务接口调用。TCGenerator 模块会定期向 Interceptor 模块查询外部服务接口调用记录，以指导生成 MockDroidContext。Interceptor 模块会记录下大量的外部

服务接口调用记录，然而系统上下文不是一成不变的，随着测试的执行，对于同一服务接口的调用记录会覆盖旧有记录。由于 Interceptor 模块是以一个线程的模式运行在待测服务进程之中，当待测服务进程退出时，模块内保存的调用记录也会随之清空。

Executor 以 Android 应用的形式存在，从 TCGenerator 模块获取服务参数，根据服务标识符 descriptor 从 Service Manager 或 ActivityManagerService 获取待测服务的 IBinder 对象，手动完成 Parcel 对象的序列化，调用 IBinder 对象的 transact 方法以完成接口调用。

GeneralServiceStub 以 Executor 应用中的一个 Service 组件的形式存在，负责模拟 IBinder 类型对象，按照该对象的接口签名响应 onTransact 请求。

4.3 测试用例生成策略

我们将测试用例中涉及到的数据类型分为静态和动态两类，其中静态数据类型通过事先约定的规则即可生成，而动态数据类型需要结合程序运行时上下文才可确定。对于 AIDL 接口支持的六类数据类型中：基本数据类型、基本数据类型数组、Bundle 类型、Parcelable 类型、FileDescriptor 类型属于静态数据类型；IBinder 属于动态数据类型。虽然 FileDescriptor 类型代表的是进程中打开的文件描述符，只有在程序运行时才能生成，但决定文件特征的文件名、权限、内容等属性依然是可以按照静态的规则预先生成，因此我们还是将 FileDescriptor 类型归为静态数据类型。

4.3.1 原始数据类型

byte、int、float 等原始数据类型的生成及变异策略如表 4-2 所示。值得一提的是，int 变量常常用于存储 UID/PID 等值，因此将待测服务或者 Executor 模块的 UID/PID 作为候选值之一。

4.3.2 String 类型

String 类型变量有三类取值：应用包名、应用权限 permission、其他常量字符串。应用包名的候选值来自于待测 ROM 中所有的预装应用。对于 permission 的候选值，从对应 Android 系统版本的文档中可提取出 system-permission，从

表 4-2: 基本数据类型的生成策略

AIDL 数据类型	候选值	变异策略
boolean	true,false	取反
byte	0, 1, -1, -128, -127	字节反转、加上随机值
char	0, 0xffff, ASCII	字节反转、加上随机值
int	0, 1, -1, $2^{31}-1$, -2^{31}	取负、加上随机值
	待测服务 PID/UID Executor 模块 PID/UID	无
long	0, 1, -1, $2^{63}-1$, -2^{63}	取负、加上随机值
float	0, 0x7fc00000, 0x1, 0x7f7fffff	取负、加上随机值
double	0, 0x7ff8000000000000 0x1, 0x7feffffffffffff	取负、加上随机值

预装应用 AndroidManifest 可提取出 custom-permission。对于常量字符串的候选值，从 DEX 和 ELF 这两种编译产物中提取常量字符串：

DEX 中字符串 按照 DEX 的文件格式定义 [43]，string_ids 段以数组的格式存储了所有字符串在 data 数据段中的偏移量。根据偏移量从 data 读取到的数据中，第一个字节代表字符串长度，之后内容为字符串的值。

ELF 中字符串 ELF 中把常量字符串的索引保存在.strtab 字符串表中，如果字符串表不存在，则遍历.rodata 从中提取出所有合法的 ASCII 码字符串。

4.3.3 Bundle 类型

Bundle 类型本质上是一个键值对形式的 Map，key 为 String 类型，value 有多种类型可选：bool、int、long、double、String、上述类型的数组以及嵌套的 Bundle 对象。Bundle 被用于 Intent 中用于携带额外的信息。有三个维度的变量需要考虑：内部 Map 的大小、key 字符串的值、value 的类型及具体值。

4.3.4 Parcelable 类型

Parcelable 类型代表的是一个可通过 Binder 通信机制传输的复杂结构体。序列化数据由其内部成员变量的序列化数据依次拼接而成，前文所述的 Parcel

序列化 API 识别方法在遇到 Parcelable 类型时，会从该对象的 writeToParcel 方法中识别出其内部成员变量的类型及序列化顺序。

除此之外，若 Parcelable 对象可在 Class Loader 中用反射机制检索到对应的 Java 层对象，可以直接调用其构造函数生成，通过反射枚举出对象内部的成员变量，随机挑选变量进行变异，最后调用该对象的 writeToParcel 序列化方法即可。

4.3.5 FileDescriptor 类型

代表的是客户端进程中一个合法的文件描述符，这个文件描述符不一定指向的是文件，也有可能指向 Socket（如 IConnectivityManager 的 establishVpn 接口返回值为 FileDescriptor，代表了服务端建立的 VPN 连接，由于这种情况并不常见，我们暂且不考虑文件描述符指向 Socket 的情况）。为使 FileDescriptor 指向的是一个合法的文件，我们使用 Jimfs [44] 在客户端进程内按照 Android 的目录命名规则构建一个虚拟的文件系统，文件有四个维度的变量需要考虑：

文件名及所处目录结构 有两种目录格式：”/data/data/<package>/files/<name>”和”/mnt/sdcard/Android/data/<package>/files/<name>”，对于应用服务来说，package 为其所在的应用包名，对于系统服务来说，package 可为系统上任何已安装应用的包名。

UID/GID Android 系统中使用 UID/GID 来管理应用程序的权限，在应用安装时由系统进行分配，之后不会再改变，应用私有目录的 owner 与应用的 UID/GID 一致。我们只考虑待测服务是否有权限访问文件这两种情况即可，对应于待测服务 UID 与 AID_NOBODY（9999）两个候选值。

POSIX 文件权限 由 Owner 权限、Group 权限、Other 权限三个整型数构成，整型数共有八种候选值。

文件内容 简单起见，只考虑纯文本文件，沿用上述的 String 类型的生成方法即可，随机选用 GBK、UTF-8、UTF-16 三种编码方式。

4.3.6 IBinder 类型

代码 4.3: IBinder 类型 AIDL 接口定义

```
1 interface IFoo {
2   void foo(IFooCallback callback);
```

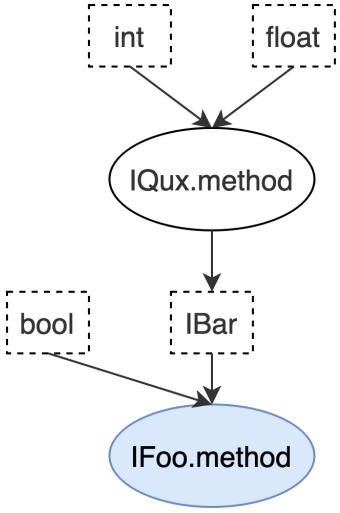

图 4-3: 使用依赖重放的方法生成 IBinder 类型

```
}
interface IFooCallback {
  bool action(int num);
}
```

如代码4.3 所示，IBinder 类型常常被用作在服务接口调用时传递回调函数或其他服务对象引用。对于待测服务接口 IFoo.foo，需要构造实现了 IFooCallback AIDL 接口的 IBinder 服务对象。

已有的测试工作 [19][10][21] 中，有两种 IBinder 类型构造方法：

构造函数 将其作为一般的 Object 对象，通过 Java 反射调用构造函数生成。

依赖重放 解析 IBinder 类型间的依赖，IBinder 类型可作为其他接口的返回值，递归地调用其他接口，直到接口的参数只包含静态数据类型，将整条接口依赖链重放以生成目标 IBinder 类型。原理如图4-3 所示，待测服务为 IFoo.method，需要 IBinder 型参数 IBar，前序遍历依赖，发现 IQux.method 方法可返回 IBar 型对象，并且所有的参数都为静态类型，则构造接口参数调用 IQux.method 方法以生成 IBar 对象。

这两种 IBinder 类型生成方法，都不能保证生成的成功率，而且由于生成的都是行为合法的 IBinder 类型，也不利于对待测服务的边界条件进行探索。

为了克服上述两点不足，受动态类型语言中“鸭子类型”的启发，我们在客户端进程中构造一个通用服务对象 GeneralServiceStub，将所有作为参数的 IBinder 类型代理到 GeneralServiceStub 中，根据之前提取出的 IBinder 类型的内

部服务接口签名，模拟其行为，还可生成行为异常（抛出异常、长时间阻塞等行为）的 IBinder 类型。

IBinder 类型是指向其他服务（包括实名服务与匿名服务）的引用。构造 IBinder 类型的关键在于，对其所指向服务的行为进行模拟，它的行为可分为四种情况:

服务正常处理请求 按照服务接口签名，返回类型合法的返回值。

接口调用抛出异常 Binder IPC 中，服务端所抛出的 RuntimeException 会被传播到客户端。当异常的 IBinder 类型被作为参数传入待测服务接口时，待测服务接口对异常 IBinder 类型的使用可能会导致 uncaught exception 类型的 crash。对应于代码 4.3 中的 action 函数调用抛出异常这种情况。

接口调用阻塞 将调用阻塞的 IBinder 类型传入待测服务时，若对其接口的调用使得待测服务长时间持有某些资源锁，那么有可能会被系统中的 watchdog 机制判断超时，导致孵化进程 zygote 的重启，一般也称之为软重启。对应于 4.3 中的 action 函数阻塞这种情况。

服务被销毁 当已被销毁的服务引用被传入待测服务时，对服务引用的调用会导致底层的 Binder 驱动出错并抛出异常，有可能导致 uncaught exception 类型的 crash。对应代码于 4.3 中 IFooCallback 服务对象已被客户端 Executor 应用所销毁这种情况，对其的调用会导致 Binder 驱动报错。

4.3.6.1 IBinder 类型生成

对于前三种服务行为，使用一个通用服务对象 GeneralServiceStub 来统一处理请求，所有测试用例中的模拟 IBinder 类型都是对 GeneralServiceStub 的引用。GeneralServiceStub 对象的实现逻辑如代码 4.4 所示。

服务正常处理请求 由于我们只有服务的接口签名，并不知道服务接口的内部语义，只能按照接口签名的类型信息模拟合法的服务接口返回值，例如接口签名显示返回值为 String 类型，返回一个固定的字符串即可。

代码 4.4: GeneralServiceStub 实现逻辑伪代码

```
1 class ServiceType {
2   string                  descriptor;
3   Map<int, MethodType>    methods;
4 }
5 class MethodType {
```

```
  List<VarType>          parameters;
  VarType                reply;
}
Map<string, ServiceType>  typeModel;
class GeneralServiceStub {
  onTransact(code, request, reply) {
    // request的头部包含目标服务标识符descriptor
    descriptor = parse(request);
    serviceSig = typeModel.find(descriptor);
    methodSig = serviceSig.methods.find(code);
    // 检查调用参数类型是否匹配
    if(checkType(request, methodSig.parameters)) {
      // 按照前文所述的方法根据类型生成返回值
      reply = generateValue(methodSig.reply);
    }
  }
}
```

接口调用阻塞 onTransact 函数在响应请求时阻塞住即可。

接口调用抛出异常 当服务端抛出异常时，会将异常转换为错误码与对应的文本描述写入 Parcel 对象开头。值得一提的是，并不是所有的 RuntimeException 都会被传播，Android 8.1 中只支持传播 SecurityException、BadParcelableException、IllegalArgumentException、NullPointerException、IllegalStateException、NetworkOnMainThreadException、UnsupportedOperationException、ServiceSpecificException 9 种异常，对应的错误码从-1 到-9。onTransact 函数只需往返回值 reply 对象中写入错误码即可。

服务被销毁 从 Binder 驱动的角度来看，IBinder 对象对应于指向 binder_ref 结构的索引，若索引不合法，则 Binder 驱动会向上抛出异常。通过 Interceptor 模块修改 Parcel 对象中对应 flat_binder_object 对象的 handle 索引字段即可。

除了通过 GeneralServiceStub 来生成 IBinder 对象类型以外，还有一种策略是在运行时记录其他服务接口的返回值，若返回值是同一类型的 IBinder 对象，使用该返回值即可。但通过这种方法获得的 IBinder 对象，我们很难去修改它的行为，不容易探索到待测服务的边界值与异常情况。

4.4 测试用例评估

我们从待测服务或系统是否出现异常、基本块覆盖率是否提高两个指标来评价单个测试用例的执行效果，以此作为反馈来指导测试用例的迭代。

4.4.1 异常日志监控

通过 Android 系统的 logcat 工具监控系统日志中 ERROR 与 FATAL 级别的条目来判断待测服务是否抛出异常，同时为了监控 Native 服务进程的异常，还需监控 /data/tombstones 目录下的核心转储 coredump 文件。

需要注意的是，由于服务之间普遍存在的依赖关系，单个服务的崩溃很有可能会导致大量服务的连锁崩溃（如 system_server 中任一服务的崩溃会导致整个进程重启，致使系统中几乎所有的 Java 服务抛出异常）。从一系列异常日志中很难自动识别出根本原因，因此我们将所有异常日志先持久化到 SQLite 数据库中，后续由人工进行分析。

4.4.2 基本块覆盖率

主流的以覆盖率为指导的模糊测试工具中，AFL [45] 与 libFuzzer [46] 在程序编译期间修改待测程序的中间代码表示，在基本块结尾插入覆盖率收集代码，kAFL [47] 在 hypervisor 层面利用 Intel PT 机制使用硬件记录程序执行的 trace，以计算覆盖率信息。

以往针对 Android 应用的测试工具，大多只考虑了 Java 代码的覆盖率，通过字节码插桩 [48]、类加载时插桩（load-time instrumentation）[49] 等技术实现。这些技术都属于侵入式的方法，需要对应用 APK 或者 ART 虚拟机进行修改。对于不能在模拟器环境下运行的闭源 Android ROM 来说，这些覆盖率收集方法并不适用。我们参考 Chizpurfle 中的方法，借助 Frida 框架提供的动态二进制插桩能力，修改待测程序控制流，以记录基本块的访问情况。

实现原理如图 4-4 所示，可分为如下六个步骤：

- 挂起待测服务所处进程；
- 从当前程序寄存器所指代码位置开始，向后匹配分支跳转指令；
- 修改分支跳转指令的目标地址为 Counter 计数器代码段的首地址，并将跳转指令原本的目标地址保存到事先约定的内存地址中；

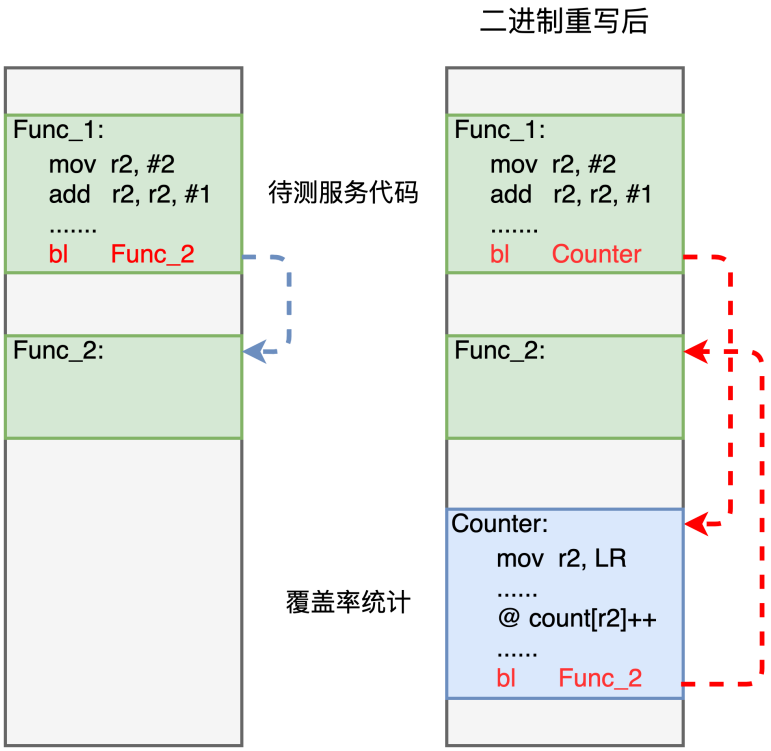

图 4-4: 基于动态二进制插桩的基本块覆盖率统计方法

- 恢复待测服务进程执行；
- 当程序执行到二进制重写过后的基本块结尾跳转指令时，跳转到 Counter 计数器代码段；
- 从 LR 寄存器中读取先前基本块的末尾指令地址，以此为索引标记基本块被覆盖，在 Counter 计数器代码段的结尾，从事先约定的内存地址中读取指令在二进制重写前的跳转目标地址，跳回待测代码段，重复以上步骤；

基于上述方法实现了覆盖率收集模块 CovStalker，以单个线程的形式运行在待测服务进程中，将每个线程的覆盖信息存储在对应的线程局部存储 TLS（Thread-local Storage）中，以减少覆盖率计数器在线程切换时的运行开销。

需要注意的是，在 Android 系统中，单个进程中往往会运行若干个服务，每个服务以单独线程的形式运行，这些线程共享内存空间。因此很难单独计算出某个特定服务所对应的基本块总数，只能以整个进程地址空间所包含的基本块数量来近似计算基本块覆盖率。

4.5 本章小结

本章首先介绍了在测试用例中引入 Android 系统上下文的动机，然后介绍了如何使用 Binder 通信劫持的方式模拟 Android 系统上下文。基于此，我们设计了 Android 系统上下文感知的服务模糊测试工具 CASFuzzer。接着介绍了 CASFuzzer 的基本框架，主要组件的功能以及模糊测试任务的执行流程。

然后介绍了 CASFuzzer 中的测试用例生成策略，包括静态数据类型和动态数据类型两类。静态数据类型的生成与其他模糊测试方法类似，主要通过预先定义的规则生成。动态数据类型包含 IBinder 类型，通过一个通用服务对象 GeneralServiceStub 将对 IBinder 类型对象的调用代理到本地的服务对象上，以模拟出接口调用异常、接口执行阻塞等异常行为。

最后介绍了如何收集测试结果反馈，包括异常日志的监控以及基于动态二进制插桩对基本块覆盖率的获取。

第五章　实验评估

5.1　研究问题

为了评估服务接口签名逆向工具 RevExtractor 对于不同类型服务的接口签名提取精度，我们以 FANS 和 Chizpurfle 中的服务接口签名提取方法为比较基准，在两个主流的定制 Android 系统以及两个版本的 AOSP 系统上进行实验评估；同时为了评估我们的服务模糊测试工具 CASFuzzer 的测试效果，我们在 MIUI 系统下进行实验评估。通过对实验结果的分析来回答如下问题：

- 与 FANS 中基于 C++ 源码静态分析的提取方法，以及 Chizpurfle 中基于运行时 Java 反射的提取方法相比，RevExtractor 服务接口签名逆向工程的提取精度如何？
- CASFuzzer 模拟的 Android 系统上下文，是否提高了模糊测试的效率？
- CASFuzzer 生成的异常 IBinder 类型是否能发现特有的服务异常？

5.2　实验配置

表 5-1: 测试设备配置

设备	Android 系统
小米手机 5	MIUI 10.8.11.22（基于 Android 8.0）
三星 Galaxy C5 pro	Samsung Experience 9.0（基于 Android 8.0）
/	AOSP 8.0.0_r51（aosp_arm64-user）
/	AOSP 9.0.0_r61（aosp_arm64-user）

实验中使用到的测试设备以及 Android 系统如表 5-1 所示，包括 MIUI 与 Samsung 两款定制 Android 系统与两个版本的开源 AOSP 系统。小米与三星

手机均使用该设备能兼容的最新版本的 ROM[①②]进行刷机并 ROOT。除此之外，我们在工作站上运行 RevExtractor 与 FANS 的静态分析组件，工作站配置如下：

- 处理器：Intel(R) Xeon(R) CPU E3-1265L @ 2.40GHz
- 内存：16G
- 存储：500G SSD 硬盘
- 操作系统：Ubuntu Server 18.04.5

5.3 实验设计

5.3.1 服务接口签名提取

从原理上来看，RevExtractor 与 FANS 均使用静态分析的方法提取服务接口签名，而 Chizpurfle 在系统运行时通过 Java 反射提取服务接口签名。由于 FANS 的方法依赖于完整的 Android 系统源码，不能应用于闭源的定制 Android 系统之上，我们只能在开源的 AOSP 上比较 RevExtractor 与 FANS 针对 Native 服务的接口签名提取精度。为了评估 RevExtractor 对于 Java 服务接口签名的提取精度，我们与 Chizpurfle 在 MIUI 与 Samsung Experience 这两个定制 Android 系统上进行比较。

5.3.2 模糊测试

我们以小米手机 5 为测试设备，输入 RevExtractor 从 MIUI ROM 中提取出的服务接口签名，运行模糊测试工具 CASFuzzer，进行实验评估。

为了理解 Android 系统上下文对服务运行的影响，我们将 Interceptor 模块装载到 MIUI 系统中的 CameraService 与 AudioService 进程中（这两种 Native 服务也往往会被移动设备厂商深度定制，并与 Android 系统硬件进行频繁地交互，具有一定的代表性），统计 Interceptor 模块拦截到的 Binder IPC 请求。为了模拟这两种服务的真实运行负载，在统计服务发出的 Binder IPC 请求的过程中，在前台运行相关应用（对于 CameraService，运行相机应用；对于 AudioService，运行音频播放器应用），并使用 Monkey 工具 [50] 生成随机的

[①] https://bigota.d.miui.com/8.11.22/miui_MI5_8.11.22_f9ead04910_8.0.zip
[②] https://samfw.com/firmware/SM-C5010/CHC/C5010ZCU2CRK1

GUI 输入以模拟应用的使用。

为了评估模拟 Android 系统上下文对测试覆盖率的提升，我们以 C++ 实现的 CameraService 与 Java 实现的 SysoptService（MIUI 新引入的服务，负责管理后台同步设置）为待测服务，分别在是否模拟 Android 系统上下文两种情况下，运行模糊测试工具 15 分钟，统计待测服务的基本块覆盖情况。

5.4 结果分析

5.4.1 服务接口签名提取精度

表 5-2: 系统服务接口签名提取结果

工具	Android 系统	系统服务		服务接口		内部服务		接口准确性	
		N_j	N_n	N_j	N_n	N_j	N_n	A_j	A_n
RevExtractor	MIUI	134	22	3662	592	488	365	99.6%	/
Chizpurfle	MIUI	116	40	3099	/	330	/	基准	/
RevExtractor	Samsung	174	34	5316	641	616	482	99.4%	/
Chizpurfle	Samsung	159	52	4875	/	551	/	基准	/
RevExtractor	AOSP 8.0	111	36	2532	505	157	255	/	84.2%
FANS	AOSP 8.0	/	36	/	505	/	260	/	基准
RevExtractor	AOSP 9.0	124	42	2677	518	160	255	/	86.5%
FANS	AOSP 9.0	/	42	/	518	/	266	/	基准

在四种 Android 系统上的服务接口签名提取结果如表 5-2 所示。其中内部服务表示不能直接通过 Service Manager 检索到的匿名服务对象，对应于各种 IBinder 类型所指向的服务对象。N_j 与 N_n 分别表示系统服务数量、内部服务数量、服务接口数量这三个指标在 Java 与 C++ 层面的数量。A_j 与 A_n 表示在 Java 与 C++ 层面上暴露服务以及内部服务所包含接口的准确性，判断标准为接口编号、输入参数及返回值类型是否匹配，判断数据类型是否匹配时：

- 若为原始数据类型、Bundle 类型以及 FileDescriptor 类型，则类型名称匹配即可；
- 若为 Parcelable 类型，则需递归地比较其内部成员的数据类型及序列化顺序是否匹配；

- 若为 IBinder 类型，则需递归地比较它对应的具体服务对象接口签名是否匹配；

从提取出的服务及接口数量指标来看：

Java 层面 RevExtractor 比 Chizpurfle 可以获取到更多的服务对象与接口。这是因为有的 Java 系统服务是由预装应用所初始化的，这些 Java 对象不会被预加载到 Zygote 的 Class Loader 中，因此 Chizpurfle 不能通过反射获取到对象。

C++ 层面 与 FANS 相比，RevExtractor 可以提取出所有服务对象，可以提取出 94.6% 的服务接口。这是因为有些服务的实现不是由 AIDL 自动生成的（如 SurfaceFlinger 等），这些服务的通信序列化函数并不是以 switch-case 结构组织。

从提取的接口签名准确性指标来看：

Java 层面 Chizpurfle 通过调用 Service Manager 的接口以获取服务在 Java 层的对象，并通过反射获得对象内部方法的签名。显然这样获得的类型信息一定是准确的，因此我们以 Chizpurfle 为比较基准，评估 RevExtractor 从字节码中逆向提取 Java 服务类型信息的精度。对于 Chizpurfle 不能获得的 Java 服务，我们通过 DexClassLoader 将应用 APK 手动加载到 Class Loader 中，反射获取类型信息，作为比较的基准。结果显示在 MIUI 与 Samsung 两种定制 Android 系统上，RevExtractor 提取的 Java 服务接口签名的准确性为 99.6% 与 99.4%，具有很高的精度。

C++ 层面 FANS 的类型提取方法借助于 Clang/LLVM 的插件，重新编译 Android 系统以生成服务相关代码的抽象语法树，指导之后的静态分析过程。FANS 基于源码的方式，提取的类型信息准确度很高。我们以 FANS 提取的接口签名为比较基准，评估 RevExtractor 从汇编中逆向提取 Native 服务接口签名的精度。结果显示在 AOSP 8.0 与 9.0 两个版本上，RevExtractor 提取的 Native 服务接口签名的准确性为 84.2% 与 86.5%，这是因为在代码编译过程中，会丢失很多类型信息。例如：对于 read(dst, sizeof(Type)) 这个 Parcel 序列化 API 调用，sizeof 运算符在编译期被常量展开，从汇编中无法推断出 Type 对应的具体类型；虚函数调用可能会被编译器优化为直接跳转指令。

RevExtractor 不仅考虑了系统服务，还对 ROM 中预装应用服务的接口签名进行了提取，结果如表 5-3 所示，其中 U_s、U_p、U_m 与 U_o 分别表示应用属于 android.uid.system、android.uid.phone、android.media 与其他用户。可以看出

表 5-3: 预装应用服务接口提取结果

Android 系统	预装应用数及 UID 分布（U_s / U_p / U_m / U_o）	暴露服务数	内部服务数	服务接口数
MIUI	216（19 / 7 / 0 / 190）	243	116	2316
Samsung	276（47 / 10 / 2 / 217）	319	252	5515
AOSP 8.0	78（5 / 2 / 2 / 69）	97	32	1615
AOSP 9.0	79（5 / 2 / 2 / 70）	99	32	1626

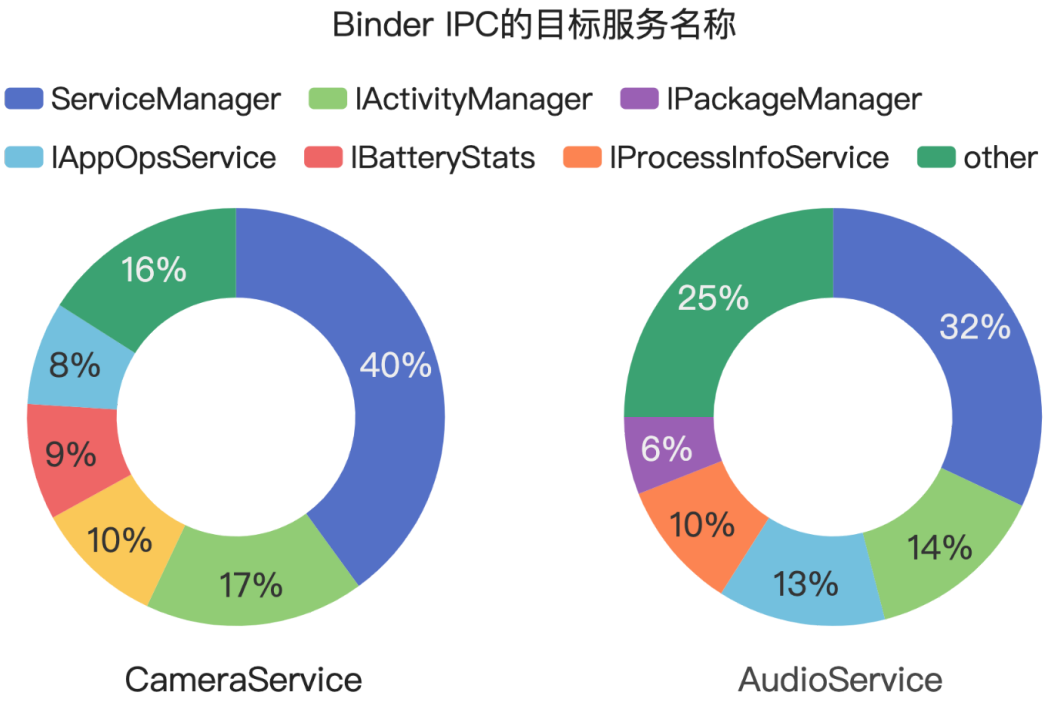

图 5-1: 不同服务拦截到的 Binder IPC 分布

定制 Android ROM 中引入了大量的预装应用以及服务，这些服务组件有很多都属于暴露服务，容易被攻击者所利用。不仅如此，这些预装应用有不少位于 system 等系统用户下，具有很多敏感的应用权限，而以往研究很少关注于这类应用，我们将预装应用中的服务组件也纳入到测试对象中。

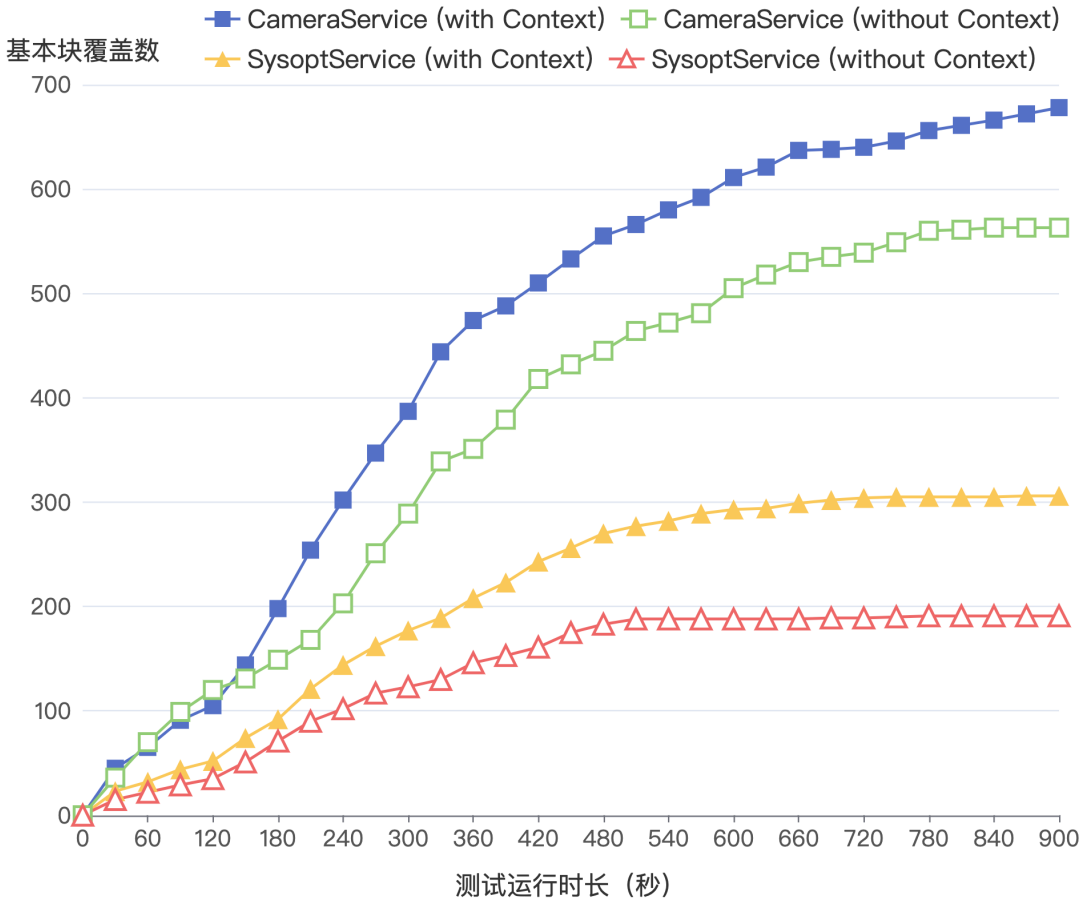

图 5-2: 模拟 Android 系统上下文对测试覆盖率的提升

5.4.2 服务模糊测试效果

5.4.2.1 模拟 Android 系统上下文对测试效果的提升

对于 MIUI 系统中的 CameraService 与 AudioService 服务，提取 Interceptor 模块拦截到的 Binder IPC 请求中包含的目标服务名称，其分布情况如图 5-1 所示。

拦截到的 Binder IPC 请求中，对 Service Mangnager 的访问请求占比最大，这是因为 Service Manager 是所有服务的注册中心，需要先从中获取到其他服务的 IBinder 引用，才可使用这些外部服务接口，对于这类 Binder IPC 请求我们不进行拦截，以便待测服务能继续执行。

除此之外，其他服务请求的对象主要包括 ActivityManager、PackageManager 以及 AppOpsService 等与应用权限管理有关的服务，PowerManager 等与能

耗管理有关的服务，ProcessInfoService 等系统状态有关的监控服务。对于权限管理服务请求的劫持，可以在不同应用权限配置情况下，测试待测服务的行为是否异常。对于 PowerManager 服务的请求主要集中在 acquireWakeLock、acquireWakeLockWithUid 这几个接口，如果劫持这些接口请求并模拟调用异常，则待测服务往往会直接退出。

对于以 C++ 实现的 CameraService 与 Java 实现的 SysoptService，在是否模拟 Android 系统上下文两种情况下，15 分钟的模糊测试过程中基本块覆盖率情况如图 5-2 所示。实验结果显示，对于 Java 与 Native 服务，模拟的 Android 系统上下文，可以使得模糊测试过程更快地探索到待测服务新的基本块，还可以提高最终的服务测试深度。

5.4.2.2 异常 IBinder 类型触发的服务异常

表 5-4: 测试中发现的异常

类别	异常类型	数量	异常 IBinder 类型触发
Java	java.lang.NullPointerException	28	2
	java.lang.IllegalStateException	19	0
	android.os.DeadObjectException	15	14
	java.lang.SecurityException	7	0
	java.io.InterruptedIOException	6	6
	android.os.TransactionTooLargeException	3	0
Native	SIGSEGV	16	1
合计		94	23

截止到目前的测试，我们在小米手机 5 的 MIUI 10.8 中发现的异常如表 5-4 所示，总计触发了 78 次 Java 异常，16 次原生代码奔溃。为了确定这些服务异常是否由异常 IBinder 类型所触发，相应的测试用例需要满足两个条件：服务接口参数中包含异常 IBinder 类型；将异常 IBinder 类型替换为正常值，服务异常不能被重现。结果显示，DeadObjectException 与 InterruptedIOException 主要都是由异常 IBinder 类型触发的，而其他服务测试工具并没有发现这类特殊的服务异常。

需要注意的是，由于待测服务内部状态的不同，同一个测试用例并不一定

能 100% 触发服务异常，由于缺乏待测服务源码，并不能确定某个服务异常一定是由某种异常 IBinder 类型所触发。

在测试过程中发现，对于 Native 服务的模糊测试也可能会触发 Java 层面的 Exception，反之亦然，这表明 Java 与 Native 服务之间存在复杂的依赖关系，这也导致我们在没有源码的情况下很难定位异常的根本原因。值得一提的是，有的原生代码崩溃会导致整个系统反复重启，必须人工重新刷机来修复，这不利于自动化测试的进行，并且会丢失 CASFuzzer 模块在设备本地使用 SQLite 保存下的所有中间结果，后续考虑切换到其他外部数据库来保存测试中间结果。

5.5 本章小结

首先为了评估服务接口签名逆向工具 RevExtractor 的有效性, 我们在 MIUI、Samsung Experience、AOSP 三种 Android 系统上，与 FANS、Chizpurfle 中的接口签名提取方法进行比较。实验结果显示，对于 Java 服务，RevExtractor 有很高的准确度，对于 Native 服务，RevExtractror 从高度精简的 ARM 汇编中依然可以提取出较为准确的类型信息。

接着我们使用服务模糊测试工具 CASFuzzer 对 MIUI 中的服务进行实验评估。实验结果显示，CASFuzzer 对 Android 系统上下文的模拟使得测试过程更加深入有效，测试用例中模拟的异常 IBinder 类型可以触发特定的服务缺陷。

到目前为止，我们使用服务模糊测试工具 CASFuzzer 在 MIUI 10.8 中发现了 94 个服务异常。

第六章　总结与展望

6.1　工作总结

本文在 Android 服务测试相关研究工作的基础上，针对定制 Android 系统中的预装应用服务、Java 系统服务与 Native 系统服务，提出包括服务接口签名提取、测试用例生成、测试结果反馈、测试用例迭代的整套自动化服务模糊测试方法。

实现了服务接口签名逆向提取工具 RevExtractor，以逆向工程的方式，从服务的编译产物中推测出接口签名，具体来说包含三个步骤：从 Android ROM 中提取服务相关编译产物；结合 Android 系统运行时状态，筛选出待测服务的通信序列化函数；从通信序列化函数的字节码与 ARM 汇编中，识别出序列化过程结构特征，以此推测出服务接口签名。

基于生成式策略，实现了服务模糊测试工具 CASFuzzer：通过 Binder 通信劫持技术，模拟影响服务执行的 Android 系统上下文，提高测试深度；通过模拟 Android 系统中特有的 IBinder 类型的异常行为，更好地探索待测服务的边界情况；基于动态二进制插桩收集被测服务的基本块覆盖率，使得模糊测试过程迭代更为高效。

经实验分析证实，在不同定制 Android ROM 中，RevExtractor 工具可以逆向提取出较为精确的服务接口签名信息，并使用 CASFuzzer 模糊测试工具发现了 MIUI 10.8 中的 90 多处服务异常。

6.2　研究展望

我们下一步的研究内容可能从以下两方面着手：

提取服务接口间的依赖　Android 系统中的服务往往都是有状态的，同一个服务不同接口调用间有隐含的依赖关系，甚至服务之间也会存在依赖关系，若接口调用顺序出错，服务会忽略该请求。例如 ICameraService 服务中调用 removeListener 接口前需要先调用 addListener 接口、IMediaBrowserService 服务

中调用 disconnect 接口前要先调用 connect 接口等。我们可以在 Android 系统运行时，收集服务接口调用记录，从这些历史记录中推测接口之间的依赖关系。

高效地统计覆盖率 通过动态二进制插桩的方式会大大提高目标程序的运行开销（例如 Valgrind 内存分析框架在 SPEC CPU 2006 数据集上，运行速度平均慢 4.3 倍 [51]），动态二进制插桩下的 Android 服务程序，运行开销至多可达 10 倍以上。通过 CPU 提供的硬件级别的程序执行追踪机制（如 Intel 的 Branch Trace Store、ARM 的 CoreSight STM），以硬件辅助的方式高效地收集待测程序的覆盖率。

致　　谢

三年硕士生涯即将结束，在学位论文即将完成之际，我要向所有在此期间支持、帮助过我的人表达诚挚的感谢。

首先，感谢我的指导老师曹春教授。当初保研时，有幸能得到导师的认可，让我能够跨专业进入计算机系攻读研究生。进入实验室以来，曹老师不仅提供了大量项目实践的机会，也引领我走进了软件工程研究领域的大门，他在为人处事上的淳淳教诲也让我收益匪浅。文本从选题、实验设计、论文撰写都离不开曹老师的指导，使我的毕业论文能够顺利完成。

感谢南京大学计算机系的各位老师，是你们精彩的课程吸引我进入计算机领域，也帮助我掌握了扎实的专业技能。感谢软件所各位老师和同学对我的关心和帮助，让我的研究生生活更加充实。

感谢葛红军、邓靖师兄对我 Android 自动化测试研究领域的引导，感谢陈洁、李青坪师兄对我云计算研究领域的引导。感谢实验室的师兄师弟们以及舍友们，在你们的陪伴下，度过了三年的难忘时光。

感谢我的家人们，在外求学期间，你们的关心和挂念，帮助我跨越求学路上的坎坷。

最后，感谢南京大学，提供了开放的学习氛围与优越的生活环境，伴我度过了愉快的七年求学时光。

此致，感谢！

参考文献

[1] IDC. Smartphone OS Market Share[EB/OL]. 2021 [online]. https://www.idc.com/promo/smartphone-market-share.

[2] DETAILS C. CVSS Score Distribution For Top 50 Products By Total Number Of ”Distinct” Vulnerabilities[EB/OL]. 2021 [online]. https://www.cvedetails.com/top-50-product-cvssscore-distribution.php.

[3] 360 安全. 2019 年 Android 恶意软件专题报告 [EB/OL]. 2020 [online]. https://blogs.360.cn/post/review_android_malware_of_2019.html.

[4] 腾讯安全. 腾讯安全发布《2020 年上半年手机安全报告》，揭示手机安全四大趋势 [EB/OL]. 2020 [online]. https://s.tencent.com/research/report/1136.html.

[5] ZERODIUM. ZERODIUM Payouts for Mobiles[EB/OL]. 2021 [online]. https://zerodium.com/program.html.

[6] GALLO R, HONGO P, DAHAB R, et al. Security and system architecture: Comparison of android customizations[C] // Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks. 2015 : 1 – 6.

[7] AAFER Y, ZHANG X, DU W. Harvesting inconsistent security configurations in custom android roms via differential analysis[C] // 25th {USENIX} Security Symposium ({USENIX} Security 16). 2016 : 1153 – 1168.

[8] FARHANG S, KIRDAN M B, LASZKA A, et al. An empirical study of Android security bulletins in different vendors[C] // Proceedings of The Web Conference 2020. 2020 : 3063 – 3069.

[9] DROIDCHART. Smartphone Brands[EB/OL]. 2021 [online]. https://droidchart.com/en/brands.

[10] IANNILLO A K, NATELLA R, COTRONEO D, et al. Chizpurfle: A gray-box android fuzzer for vendor service customizations[C] // 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE). 2017 : 1 – 11.

[11] JI T, WU Y, WANG C, et al. The coming era of alphahacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques[C] // 2018 IEEE third international conference on data science in cyberspace (DSC). 2018 : 53 – 60.

[12] KING J C. Symbolic execution and program testing[J]. Communications of the ACM, 1976, 19(7) : 385 – 394.

[13] CADAR C, DUNBAR D, ENGLER D R, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.[C] // OSDI : Vol 8. 2008 : 209 – 224.

[14] MILLER B P, FREDRIKSEN L, SO B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12) : 32 – 44.

[15] LUO L, ZENG Q, CAO C, et al. Context-aware System Service Call-oriented Symbolic Execution of Android Framework with Application to Exploit Generation[J]. arXiv preprint arXiv:1611.00837, 2016.

[16] LI J, ZHAO B, ZHANG C. Fuzzing: a survey[J]. Cybersecurity, 2018, 1(1) : 1 – 13.

[17] ZHANG L L, LIANG C-J M, LIU Y, et al. Systematically testing background services of mobile apps[C] // 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). 2017 : 4 – 15.

[18] GOOGLE. Android Service[EB/OL]. 2021 [online]. https://developer.android.com/guide/components/services.

[19] FENG H, SHIN K G. Understanding and defending the Binder attack surface in Android[C] // Proceedings of the 32nd Annual Conference on Computer Security Applications. 2016 : 398 – 409.

[20] GU Y, CHENG Y, YING L, et al. Exploiting android system services through bypassing service helpers[C] // International Conference on Security and Privacy in Communication Systems. 2016 : 44 – 62.

[21] LIU B, ZHANG C, GONG G, et al. {FANS}: Fuzzing Android Native System Services via Automated Interface Analysis[C] // 29th {USENIX} Security Symposium ({USENIX} Security 20). 2020 : 307 – 323.

[22] GOOGLE. Android Interface Definition Language (AIDL)[EB/OL]. 2021 [online].
https://developer.android.com/guide/components/aidll.

[23] GAMBA J, RASHED M, RAZAGHPANAH A, et al. An analysis of pre-installed android software[C] // 2020 IEEE Symposium on Security and Privacy (SP). 2020 : 1039 – 1055.

[24] ZHOU X, LEE Y, ZHANG N, et al. The peril of fragmentation: Security hazards in android device driver customizations[C] // 2014 IEEE Symposium on Security and Privacy. 2014 : 409 – 423.

[25] ZHANG X, AAFER Y, YING K, et al. Hey, you, get off of my image: Detecting data residue in android images[C] // European Symposium on Research in Computer Security. 2016 : 401 – 421.

[26] CHIN E, FELT A P, GREENWOOD K, et al. Analyzing inter-application communication in Android[C] // Proceedings of the 9th international conference on Mobile systems, applications, and services. 2011 : 239 – 252.

[27] SASNAUSKAS R, REGEHR J. Intent fuzzer: crafting intents of death[C] // Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA). 2014 : 1 – 5.

[28] 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.

[29] 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 : Vol 1. 2015 : 280 – 291.

[30] FENG H, SHIN K G. Bindercracker: Assessing the robustness of android system services[J]. arXiv preprint arXiv:1604.06964, 2016.

[31] WU J, LIU S, JI S, et al. Exception beyond Exception: Crashing Android System by Trapping in" Uncaught Exception"[C] // 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). 2017 : 283 – 292.

[32] IBOTPEACHES. Apktool[EB/OL]. 2021 [online]. https://github.com/iBotPeaches/Apktool.

[33] LIEF: Library to Instrument Executable Formats[EB/OL]. 2021 [online]. https://lief.quarkslab.com/.

[34] YANG K, ZHUGE J, WANG Y, et al. IntentFuzzer: detecting capability leaks of android applications[C] // Proceedings of the 9th ACM symposium on Information, computer and communications security. 2014 : 531 – 536.

[35] soot OSS. Soot[EB/OL]. 2021 [online]. https://github.com/soot-oss/soot.

[36] Retargetable Decompiler[EB/OL]. 2021 [online]. https://retdec.com/.

[37] Hex-Rays Decompiler[EB/OL]. 2021 [online]. https://www.hex-rays.com/decompiler/.

[38] ZHANG C, SONG C, CHEN K Z, et al. VTint: Protecting Virtual Function Tables' Integrity.[C] // NDSS. 2015.

[39] GOOGLE. Sparse Image Format[EB/OL]. 2021 [online]. https://source.android.com/devices/bootloader/images#sparse-image-format.

[40] PXB1988. dex2jar[EB/OL]. 2021 [online]. https://github.com/pxb1988/dex2jar.

[41] cea SEC. Miasm[EB/OL]. 2021 [online].
https://github.com/cea-sec/miasm.

[42] FRIDA. Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers[EB/OL]. 2021 [online].
https://frida.re/.

[43] GOOGLE. DEX File Format[EB/OL]. 2021 [online].
https://android.googlesource.com/platform/art/+/android-8.1.0_r2/runtime/dex_file.h.

[44] GOOGLE. Jimfs: an in-memory file system for Java[EB/OL]. 2021 [online].
https://github.com/google/jimfs.

[45] ZALEWSKI M. AFL[EB/OL]. 2021 [online].
https://github.com/google/AFL.

[46] LLVM. libFuzzer – a library for coverage-guided fuzz testing[EB/OL]. 2021 [online].
https://llvm.org/docs/LibFuzzer.html.

[47] SCHUMILO S, ASCHERMANN C, GAWLIK R, et al. kafl: Hardware-assisted feedback fuzzing for {OS} kernels[C] // 26th {USENIX} Security Symposium ({USENIX} Security 17). 2017 : 167 – 182.

[48] PILGUN A. ACVTool (Android Code Coverage Tool)[EB/OL]. 2021 [online].
https://github.com/pilgun/acvtool.

[49] SUN H, ROSA A, JAVED O, et al. ADRENALIN-RV: android runtime verification using load-time weaving[C] // 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). 2017 : 532 – 539.

[50] GOOGLE. UI/Application Exerciser Monkey[EB/OL]. 2021 [online].
https://developer.android.com/studio/test/monkey.

[51] NETHERCOTE N, SEWARD J. Valgrind: a framework for heavyweight dynamic binary instrumentation[J]. ACM Sigplan notices, 2007, 42(6) : 89 – 100.

简历与科研成果

基本信息

廖祥森，男，汉族，1996 年 8 月出生，江西赣州人。

教育背景

2018 年 9 月 — 2021 年 6 月 南京大学计算机科学与技术系 工学硕士
2014 年 9 月 — 2018 年 6 月 南京大学地理与海洋科学学院 理学学士

攻读硕士学位期间完成的学术成果

1. 发明专利：一种基于系统调用代理的安卓虚拟化方法及系统（专利号：202110557154.0）
2. 发明专利：一种基于逆向工程的安卓闭源服务类型信息提取方法（专利号：202110557657.8）
3. 软件著作权：软件 SDK 自动化测试云平台 V1.0（登记号：2021SR0675157）

攻读硕士学位期间参与的科研课题

1. 国家重点研发计划：软件定义的人机物融合云计算支撑技术与平台（2018YFB1004805），2018 年 5 月—2021 年 4 月，负责云计算支撑平台的设计与实现。

《学位论文出版授权书》

本人完全同意《中国优秀博硕士学位论文全文数据库出版章程》（以下简称“章程”），愿意将本人的学位论文提交“中国学术期刊（光盘版）电子杂志社”在《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》中全文发表。《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》可以以电子、网络及其他数字媒体形式公开出版，并同意编入《中国知识资源总库》，在《中国博硕士学位论文评价数据库》中使用和在互联网上传播，同意按“章程”规定享受相关权益。

作者签名：_____

_____年_____月_____日

论文题名	定制 Android 系统服务测试技术研究				
研究生学号	-	所在院系	计算机科学与技术系	学位年度	2021
论文级别	☐ 硕士　☐ 硕士专业学位 ☐ 博士　☐ 博士专业学位 （请在方框内画勾）				
作者 Email	-				
导师姓名	曹春 教授				

论文涉密情况：

☐ 不保密

☐ 保密，保密期 (_____年_____月_____日 至 _____年_____月_____日)

注：请将该授权书填写后装订在学位论文最后一页（南大封面）。