Beijing Forest Studio
北京理工大学信息系统及安全对抗实验中心

安卓原生系统库和系统服务的漏洞挖掘

硕士研究生 杨树

2026年06月14日

问题回溯

- 总结反思
  - 演讲时间较短，语气较为平淡
  - 实验部分不够深入
  - 论文与题目贴合不够好，对论文理解较浅

- 相关内容
  - 2023.07.23 陆永鑫《准确高效地检测安卓APP中的第三方库》
  - 2022.11.20 陆永鑫《Android第三方库检测》
  - 2022.04.17 陆永鑫《Android自定义权限及其设计缺陷》

内容提要

- 预期收获
- 题目内涵解析
- 研究背景与意义
- 研究历史与现状
- 知识基础
- 算法原理
  - FuzzGen++
  - NASS
- 特点总结与工作展望
- 参考文献

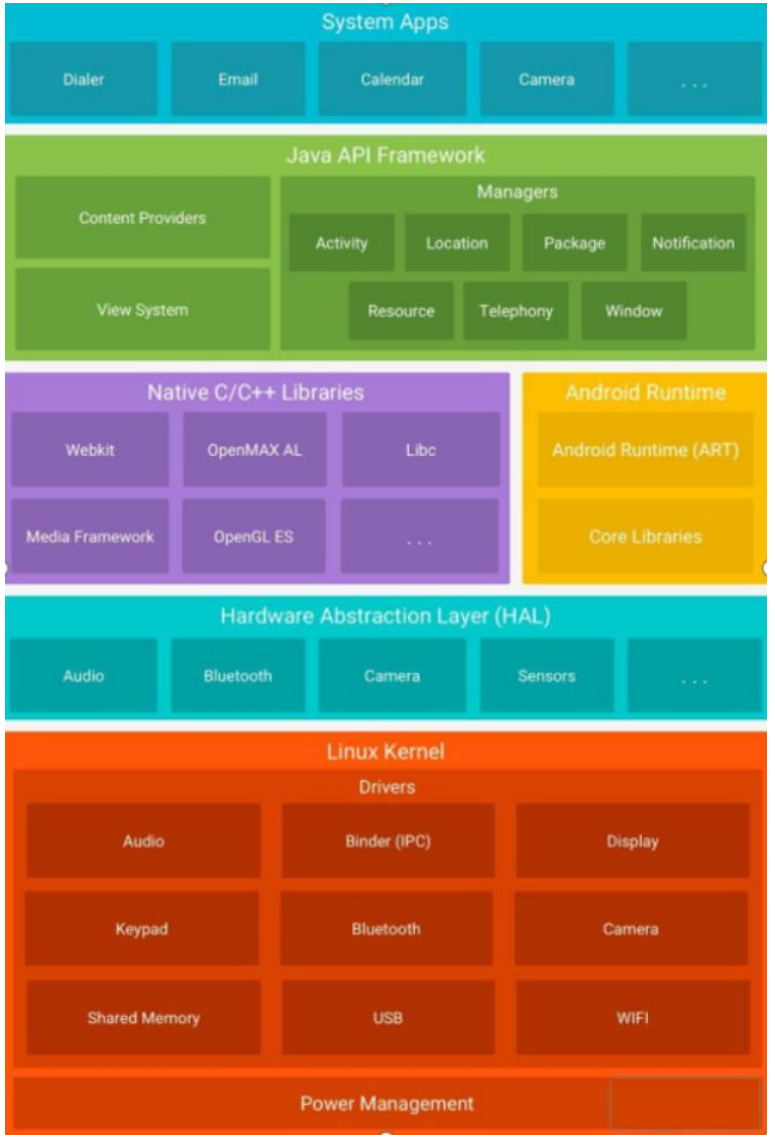
预期收获

- 预期收获
  - 了解Android OEM框架与系统服务的安全威胁
  - 理解模糊驱动生成的基本原理与工程障碍
  - 理解DGIE接口恢复技术的创新思路

题目内涵解析

- 安卓原生系统库
  - Android包含一些原生C/C++库，这些库能够被安卓系统的不同组件使用。它们通过Android应用框架为开发者提供服务
  - Android OEM是手机厂商独有的原生系统库

- 安卓原生系统服务
  - Android 系统服务是高权限的用户空间守护进程，应用程序通过Binder远程过程调用（RPC）与系统服务交互，系统服务代表应用访问硬件



背景意义

- 研究背景
  - OEM Android框架大量使用C/C++库，手工编写fuzz driver成本极高，现有fuzz driver生成工具在OEM环境下面临兼容性、易用性、有效性三类障碍
  - Android系统服务是权限提升的关键攻击面，大量原生服务为专有闭源，传统方法需要源码
- 研究意义
  - 在OEM Android设备上让自动生成的高质量fuzz driver替代手工编写，大幅降低OEM厂商对自己框架库进行安全测试的人力成本
  - 不再依赖源码或被动抓包，仅凭二进制就能自动恢复接口并完成覆盖率引导的灰盒fuzzing，使任意厂商的专有系统服务可被安全审计

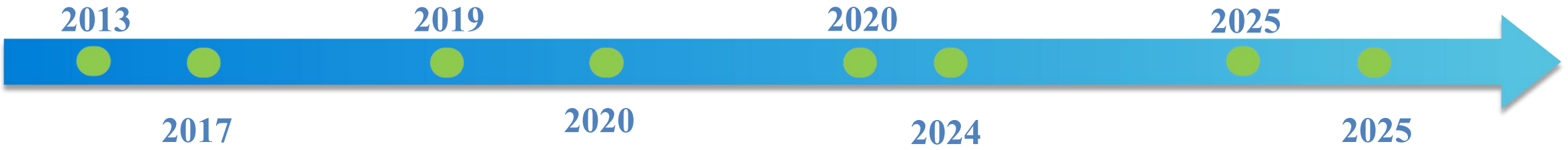
研究历史

Michal等人提出的AFL在编译时对程序插桩，记录每次执行经过的边覆盖信息存入共享bitmap，通过比较前后bitmap差异，保留触发新路径的输入作为种子继续变异，形成覆盖引导的进化式模糊测试闭环

Google提出FUDGE，通过静态分析Google内部代码库中大量调用目标库的消费者代码，提取API调用序列与参数使用模式，自动合成可用于libFuzzer的驱动程序

Kyriakos等人提出FuzzGen，将驱动生成从手动变为自动，核心思路是从现有代码中找到调用目标库的消费者代码，学习其API使用模式，自动生成符合正确调用语义的模糊驱动，主要面向Android开源库

Zhang等人提出针对使用JNI接口的原生系统库模糊测试方法，通过动态构建数据依赖图来理解库函数之间的调用关系与数据流向，从而生成更合理的API调用序列，比静态分析更能捕捉运行时的真实依赖



Google LLVM团队提出libFuzzer，以库的形式直接链接进目标程序，在同一进程内运行模糊测试，利用LLVM的SanitizerCoverage插桩收集覆盖信息，并可与AddressSanitizer等内存错误检测工具无缝集成

Liu等人提出的FANS是第一个系统化测试Android原生系统服务的工具，通过静态分析开源服务的源码自动提取RPC接口定义，再用黑盒方式生成测试输入，但完全依赖源码，无法处理闭源专有服务

Peng等人提出FuzzGen++，针对OEM Android环境的三大障碍——工具链不兼容、API规则难提取、C++高级库支持不足——提出解决方案，并增加了驱动质量评分与筛选机制

Mao等人提出NASS，无需源码、通过动态监控服务处理RPC请求时调用的反序列化函数来自动恢复接口定义，再针对接口类型定向变异，达到对闭源服务的灰盒模糊测试

研究现状

- 现有模糊驱动生成工具（FDG）在OEM环境的三大障碍
  - OEM使用定制的LLVM工具链，与标准LLVM不兼容
  - OEM库缺乏规范的接口文档等信息，API调用逻辑难以获取
  - 现有FDG方法难以支持OEM中常用的高级库

- 现有系统服务测试对闭源服务无能为力
  - 大量原生系统服务为专有闭源二进制，现有方法大多需要源码才能提取接口定义，仅能捕获正常使用中触发的一半左右RPC函数
  - 现有方法均为黑盒模糊测试，无覆盖率反馈，变异盲目、效率低下

- OEM Android编译工具链
  - LLVM是一套模块化的编译工具链
  - OEM Android使用定制LLVM工具链，与标准LLVM不完全兼容；通过补丁文件维护修改；对静态分析工具造成兼容性障碍

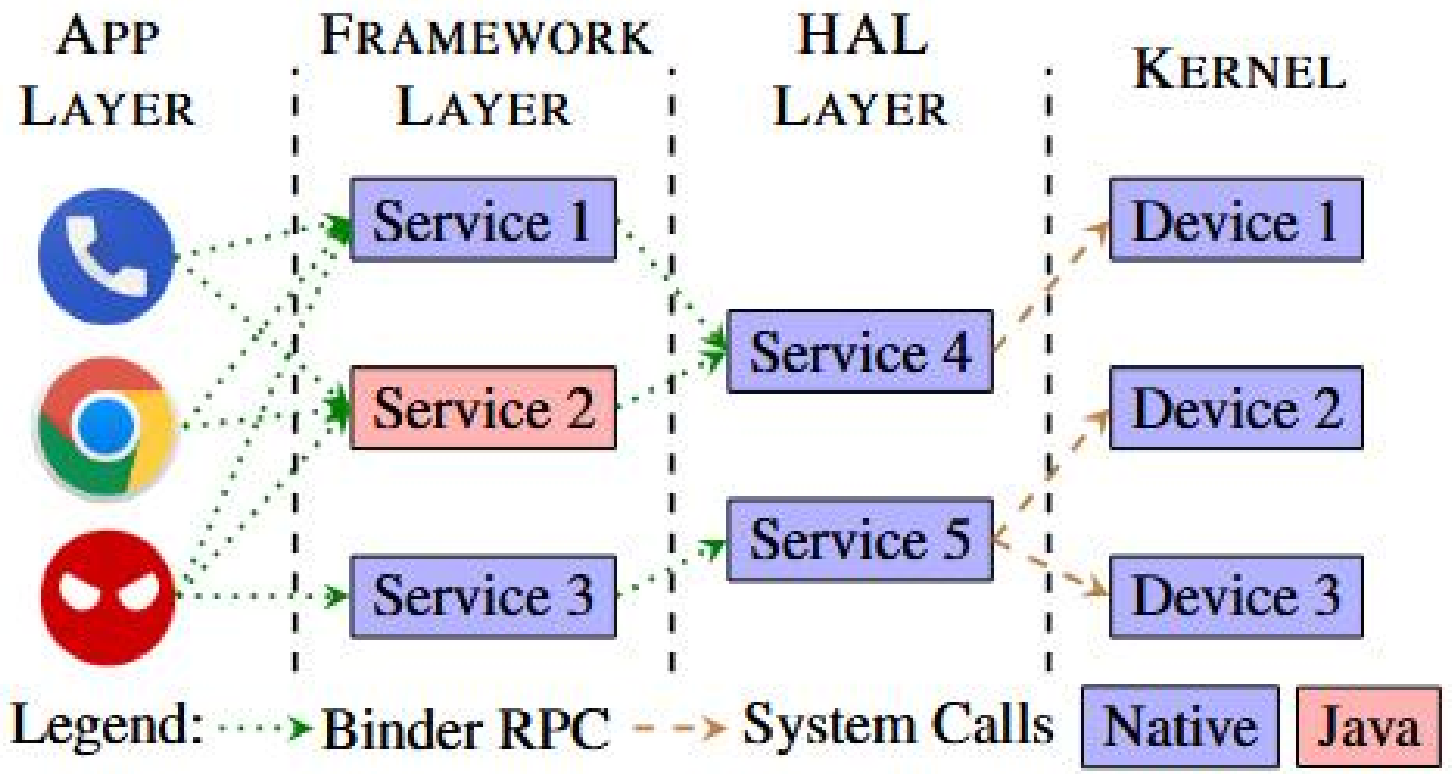
- **LLVM IR**
  - **LLVM IR是编译流程中的中间层——源代码（C/C++）经 Clang 前端处理后，先被翻译成IR，然后在IR上做优化，最后再由后端生成目标机器码**
  - **LLVM IR指令种类少、规则简单、数据流清晰，适合用来分析API调用规则**

	源码	LLVM IR
变量关系	同一变量多次赋值	每个变量只赋值一次
宏/预编译	未展开	已展开，函数调用关系清晰
类型信息	隐式转换多	强类型，操作数类型明确
跨语言	C/C++语法有差异	统一格式
分析工具	需要手写分析工具	LLVM Pass 框架现成可用

- Consumer
  - 目标库 API 的"使用者"——OEM Android框架中已经存在的、调用了目标库API的那些代码
  - 举例：目标库：libmediacodec.so
    - 对外暴露API： Codec_create、configure、decode 等
    - 调用API的模块：相机模块、媒体播放模块、图库模块——consumer

- CFI（Control Flow Integrity）
  - LLVM的一项安全防御机制——编译器为每个间接调用点生成一张白名单
  - 如果实际跳转目标不在白名单内，程序直接崩溃，阻止控制流劫持攻击
  - 可以用来确定正确的API调用关系

- Android Binder RPC架构
  - App Layer → Framework Layer → HAL Layer → Kernel的四层架构；Binder IPC作为RPC传输层；系统服务通过Binder暴露接口供客户端调用



- RPC三大设计原则
  - Ab（IPC绑定代码抽象）： client stub/server stub自动生成，处理流程固定
  - Si（单入口点）： 单一函数Ontransact统一处理所有RPC请求，便于覆盖率计算
  - St（标准反序列化例程）： 使用标准库函数进行参数(反)序列化，便于统一处理

Applying Fuzz Driver Generation to Native C/C++ Libraries of OEM Android Framework: Obstacles and Solutions

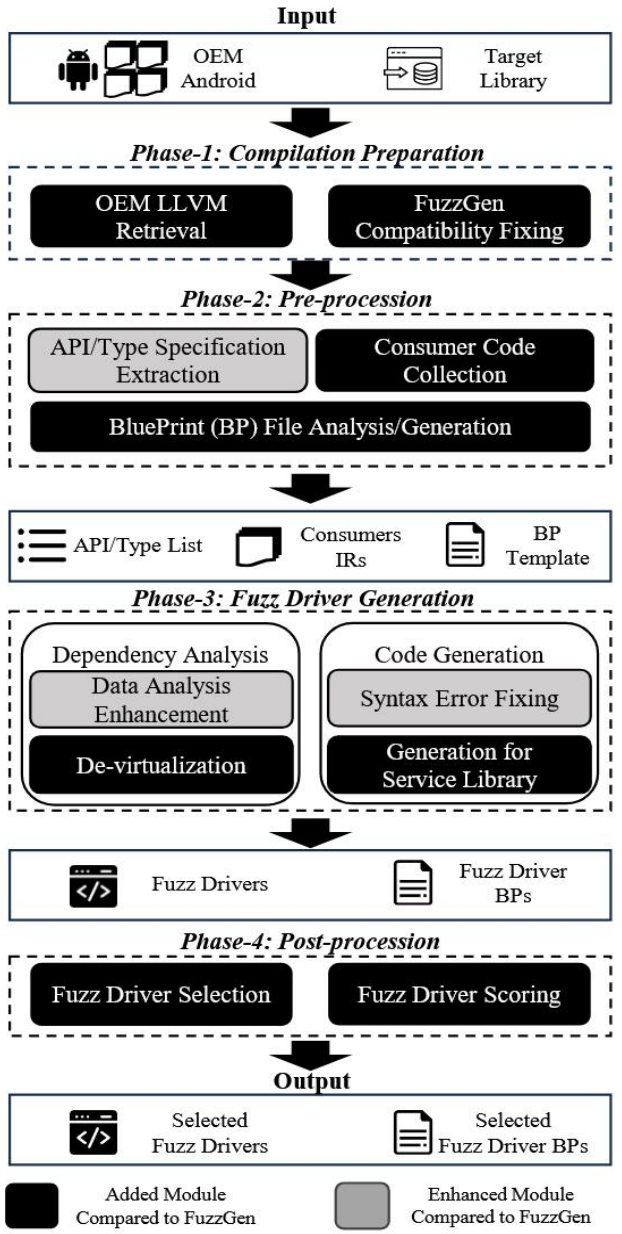
算法原理 TIPO

T	目标	为OEM Android C/C++库生成高质量Fuzz Driver并挖掘漏洞
I	输入	1个包含91个OEM Android C/C++库的数据集
P	处理	1、确定LLVM版本并找到相关补丁文件，根据补丁手动修改FuzzGen 2、对输入进行预处理，提取API符号和类型定义，从IR中收集consumer代码，并从Blueprint文件解析编译配置 3、利用去虚拟化、数据流增强等生成可直接编译的Fuzz Driver源码
O	输出	每个库1组Fuzz Driver源码

P	问题	现有FDG工具在OEM Android中不适用
C	条件	能从LLVM文件中分析版本并获取补丁
D	难点	LLVM工具链不兼容、缺乏consumer代码、高级库不支持
L	水平	ASE '24（CCF A）

算法原理 创新说明

- FuzzGen++
  - 利用编译准备确保FuzzGen兼容OEM Android编译工具链
  - 对原版FuzzGen进行了增强和添加模块以解决将FDG应用于OEM Android的障碍
  - 增强模块（以灰色框示意）表示它们存在于FuzzGen中，并在FuzzGen++中得到了增强
  - 新增的模块（以黑色框表示）在FuzzGen++中是独占的



算法原理 FuzzGen++

- 编译准备
  - OEM LLVM检索
    - 从工具链中的预编译二进制文件中检索LLVM主线版本
    - 根据LLVM工具链中的补丁文件为每个子项目映射并应用必要的补丁
  - FuzzGen兼容性修复
    - 用更新的替代方案替换过时的API，并更新API的使用方式
      - 用getPointerElementType等更新的 API取代已废弃的API
    - 适应OEM Android最新 LLVM 版本的变化
      - 新版本用StringRef数据结构替代直接返回字符串，并集成了对从该结构中提取必要字符串的支持

算法原理 FuzzGen++

- 预处理
  - API/类型提取
    - 将目标库编译为.a文件并用binutil中的nm提取目标库中的API
    - 通过遍历相关AST节点确定字段的类型定义
  - Consumer代码收集
    - 收集所有OEM Android项目的IR并匹配目标库 API
    - 利用LLVM CFI白名单机制，为虚函数的调用点提供候选API
  - Blueprint文件生成
    - Android 编译系统用Blueprint（.bp 文件）描述编译规则
    - 分析并匹配目标库 Blueprint 文件中的依赖关系和编译选项
    - 将这些信息整理并记录为FuzzGen++可读的JSON格式

算法原理 FuzzGen++

- 模糊驱动生成
  - 依赖分析
    - 去虚拟化：用CFI确定IR中虚函数的具体指向
    - 数据流增强：
      - 扩展source point定义，跟踪更全面的信息（函数参数、非API参数返回值等）
      - 字段敏感，精确变异结构体中具体字段
  - 代码生成
    - service库生成：创建模拟环境并随机调用service
      - 原因： service库没有常规consumer，在系统中不会被直接调用
    - 语法修复：修改生成的Fuzz Driver中与OEM工具链不兼容的语句

算法原理 FuzzGen++

- 后处理
  - Fuzz Driver选择
    - 排除低质量Driver
      - 试图析构一个从未被构造过的对象
      - 没有可变参数或有效分支
  - Fuzz Driver评分
    - Score = API调用次数 + 被Fuzzed的参数数量
    - 每库只保留得分前 20 的Fuzz Driver

Algorithm 1 Fuzz Driver Scoring

Input: ❶ Source Code of Fuzz Driver $Source_{fd}$; ❷List of APIs in Target Library $APIs$;
Output: Score of Fuzz Driver $Score_{fd}$;

```
(Callsites, MutationValues) ← Source_fd;
for ((Callee, Arguments) ∈ Callsites ) do
    if Callee ∈ APIs then
        Score_Invocations += 1;
    else
        continue;
    for ((Arg) ∈ Arguments ) do
        Definer_Arg ← (Arg, Source_fd)
        if 0 < |MutationValues ∩ Definer_Arg| then
            Score_Arguments += 1;
Score_fd ← Score_Arguments + Score_Invocations;
```

实验设计 数据资源

- 数据集：
  - 从OEM Android框架目录中汇编所有文件，确保库的多样化选择
  - 筛选整合进库中的项目，专注于与AOSP不同的OEM特定新增内容
  - 形成了一个包含91个库的数据集，涵盖了媒体解码、纹理解析和网络管理等功能
- 评价指标
  - 代码覆盖率：
    - 测试用例执行代码的数量占代码总数的比例
    - 是衡量测试有效性的核心指标之一
    - 表示测试用例覆盖了多少代码逻辑

工具配置

配置项	设置
Consumer 数上限	20
单 Consumer timeout	10min
Consumer 收集上限	180min
后处理上限	30min
全流程	8h
API 序列合并	关闭

实验设计 对比实验

- RQ1：Fuzz Driver整体生成情况
  - 30,153 consumer → 21,457 driver → 精选 1,695 个，69% 可运行，100% 库至少有 1 个可运行
  - 与手动制作的Fuzz Driver相比，平均覆盖率提升了107.92%

# of Libraries	# of Consumer Functions	# of Generated Fuzz Driver	# of Selected Fuzz Drivers	Average Time Cost for Consumer Collection	Average Time Cost for Fuzz Driver Generation
91	30,153	21,457	1,695	1,534s	1.02s

- RQ5：影响真实世界bug的有效性
  - FuzzGen++已成功识别出6个OEM Android中的已确认错误，涵盖了多个问题，如两个空指针去引用（NPD）漏洞、两个超出界限（OOB）错误、1个内存泄漏以及一个可达断言。这些发现涵盖了多种系统功能，从文本和媒体解析到以更高权限级别运行的服务代码

实验设计 对比实验

- RQ2：克服兼容性障碍的表现
  - 使用了两种编译系统，这两套系统共包含七个基于LLVM的编译器版本，总共应用了291个补丁，包括大约89,010行代码
  - 对FuzzGen应用了4,265行代码修改，全部修改均有效并成功应用

Compiler	LLVM Main Version	Patch	Lines
OEM-Compiler-1	LLVM-11	15	5,078
OEM-Compiler-2		17	5,333
OEM-Compiler-3		20	5,664
OEM-Compiler-4		34	5,354
OEM-Compiler-5		40	5,965
OEM-Compiler-6	LLVM-14	93	34,002
OEM-Compiler-7		72	27,614
Sum	-	291	89,010

NASS: Fuzzing All Native Android System Services with Interface Awareness and Coverage

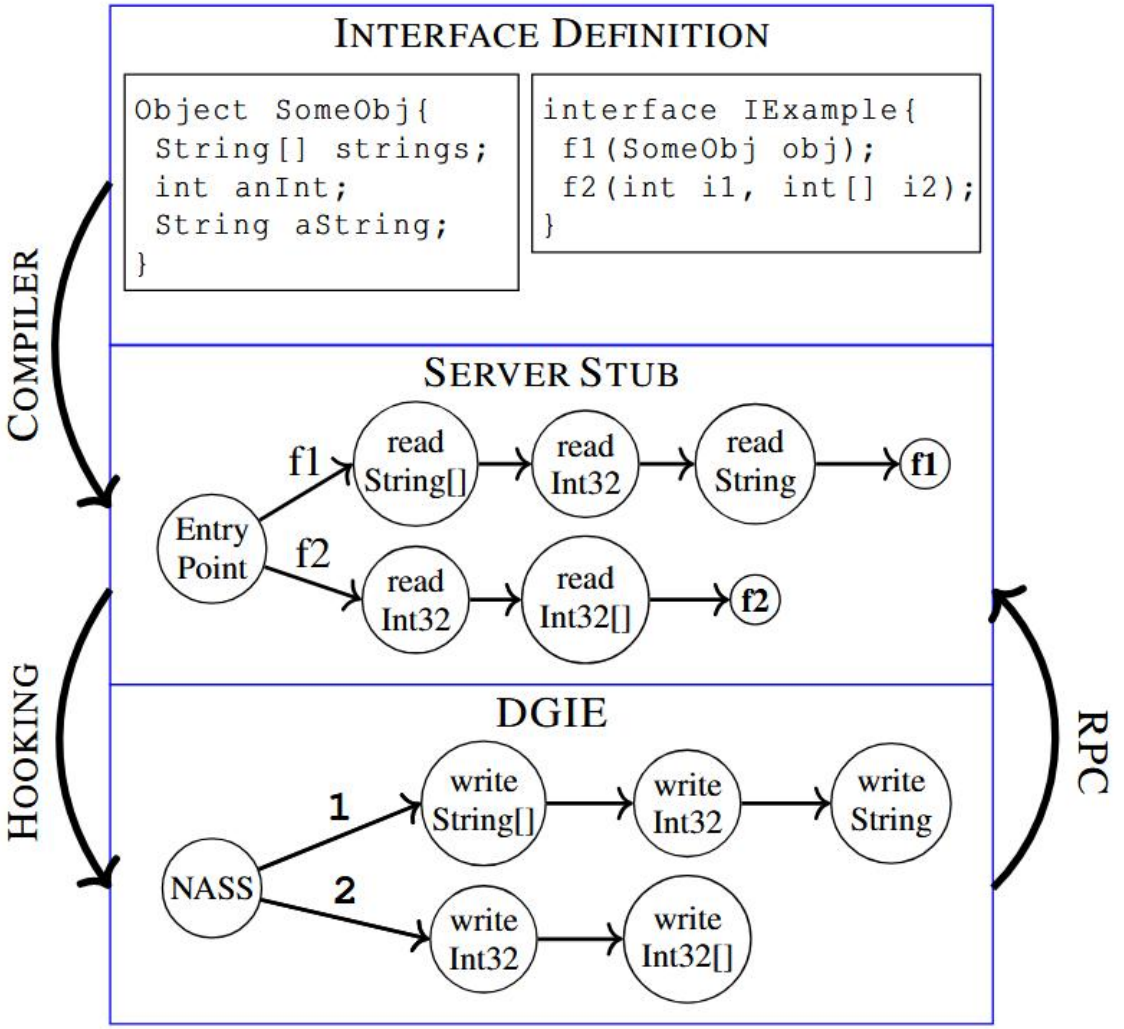
算法原理 TIPO

T	目标	挖掘专有原生Android系统服务中的漏洞
I	输入	316 个专有原生服务
P	处理	1、Entrypoint 识别（拿到 onTransact 地址） 2、DGIE 迭代反序列化确定接口定义 3、根据接口类型变异参数进行模糊测试
O	输出	代码覆盖率和新漏洞

P	问题	专有原生 Android 系统服务是权限提升的关键攻击面
C	条件	服务闭源，没有源码、没有接口文档、binder驱动被重写
D	难点	如何在灰盒条件下提取接口定义
L	水平	usenix security 2025（CCF A）

算法原理 创新说明

- NASS
  - DGIE
    - Hook libbinder.so 中 23 个标准反序列化函数观察参数调用序列
  - 单线程覆盖率收集
    - Hook 单入口点 onTransact，记录单个线程覆盖率
  - 接口感知变异
    - 对 23 种参数类型各设计一种语义变异策略
    - 每个参数用自身类型的策略变异



算法原理 NASS

- DGIE
  - 依据：Ab（IPC绑定代码抽象）+St（标准反序列化例程）
  - 输入：目标服务Binder handle（通信地址）
  - 输出：接口定义（函数 → 参数类型序列）
  - 算法结构
    - 外层循环
    - 内层循环一：随机变异参数字节 → 覆盖率反馈 → 触发新覆盖的种子入队 → 无新覆盖停止
    - 内层循环二：原样重放种子 → Hook 反序列化函数 → 记录类型调用序列 → 取最长序列
  - code 是Binder RPC中写在每个IPC请求头部的整数值，用于唯一标识本次调用要执行目标服务的哪个RPC 函数

算法原理 NASS

- **DGIE**

算法原理 NASS

- 单线程覆盖率收集
  - 依据：Si（单入口点）
    - 所有RPC请求从同一个函数onTransact 进入
    - Hook 这一个点 = 拦截到所有IPC请求
  - IPC流程
    - 拿到onTransact地址：通过Hook libbinder.so中调用onTransact前的分发函数，间接读出专有服务onTransact的运行地址
    - 在onTransact入口设置Hook：每次IPC请求到达 → Hook 自动触发 → 检查PID：如果PID不是NASS client→ 其他系统组件在调服务 → 放行；如果 PID 是 NASS client → 自己发的请求 → 进入下一步
    - 用Frida Stalker去trace当前这一线程，每碰到一个新基本块 → 更新 coverage bitmap记录覆盖率

算法原理 NASS

- 接口感知变异
  - 字节级变异对RPC无效，参数边界会被破坏
  - 已知签名后按类型单独处理每个参数的变异

Type	Mutators Used
Bool, Char, Short, Int32, Int64, Float, Double	FixedLengthBytes, InsertSpecialNumber
String8, String16, CString	FixedLengthBytes, VarLengthBytes, InsertSpecialString
BoolVector, CharVector, ShortVector, Int32Vector, Int64Vector, FloatVector, DoubleVector	ChangeVectorSize, MutateEntryFixed
String8Vector, String16Vector	ChangeVectorSize, MutateEntryVarLength
StrongBinder	MutateStrongBinder
FileDescriptor	MutateFileDescriptor
Parcelable	All*
ParcelableVector	ChangeVectorSize, All*

实验设计 数据集

- **DATASET**
  - 五款品牌的安卓设备上运行的本地（N）服务（S）。60%的本地服务是专有的（P），其中92%运行在HAL（H）层，8%运行在框架（F）层

	Overall	Google Pixel 9	Samsung S23	Xiaomi RM Note 13	OnePlus 12R	Infinix X670
# S.	1784	371	388	310	437	278
# N.S.	528	146	110	61	162	49
# P.N.S.	316	102	67	20	117	10
# F.P.N.S.	23	10	0	0	13	0
# H.P.N.S.	293	92	67	20	104	10

实验设计 对比实验

- RQ1：DGIE 能否从二进制服务中完整恢复接口?
  - 方法：14 个开源服务 × 手工逆向得到真值 → 对比DGIE vs 消息捕获
  - 结果：
    - DGIE发现 265/276 个RPC函数（96%），完整恢复参数序列 242 个（88%）
    - 6 个使用AIDL服务完整恢复，消息捕获仅发现 148/276 个（53%）
  - 结论：DGIE对遵守Ab原则的服务恢复精度极高；消息捕获严重低估攻击面

Service	AIDL	# RPC functions	# NASS disc. RPC funcs.		# NASS extr. RPC funcs.		# Capt. RPC funcs.	
keystore	no	42	42	100%	36	85%	27	64%
gatekeeper	no	6	6	100%	3	50%	4	66%
perfprofd	no	5	5	100%	4	100%	0	0%
stats	no	18	17	94%	17	94%	8	44%
wificond	yes	14	14	100%	14	100%	9	57%
surfaceflinger	no	50	42	84%	32	64%	23	46%
netd	yes	37	37	100%	37	100%	17	45%
installd	yes	39	39	100%	39	100%	27	69%
vold	yes	51	51	100%	51	100%	25	49%
gpu	no	1	1	100%	0	0%	0	0%
media.metrics	no	2	1	50%	1	50%	1	50%
storaged	yes	3	3	100%	3	100%	3	100%
thermalservice	yes	4	4	100%	4	100%	1	25%
incident	no	4	3	75%	1	25%	3	75%
Overall	6/14	276	265	96%	242	88%	148	53%

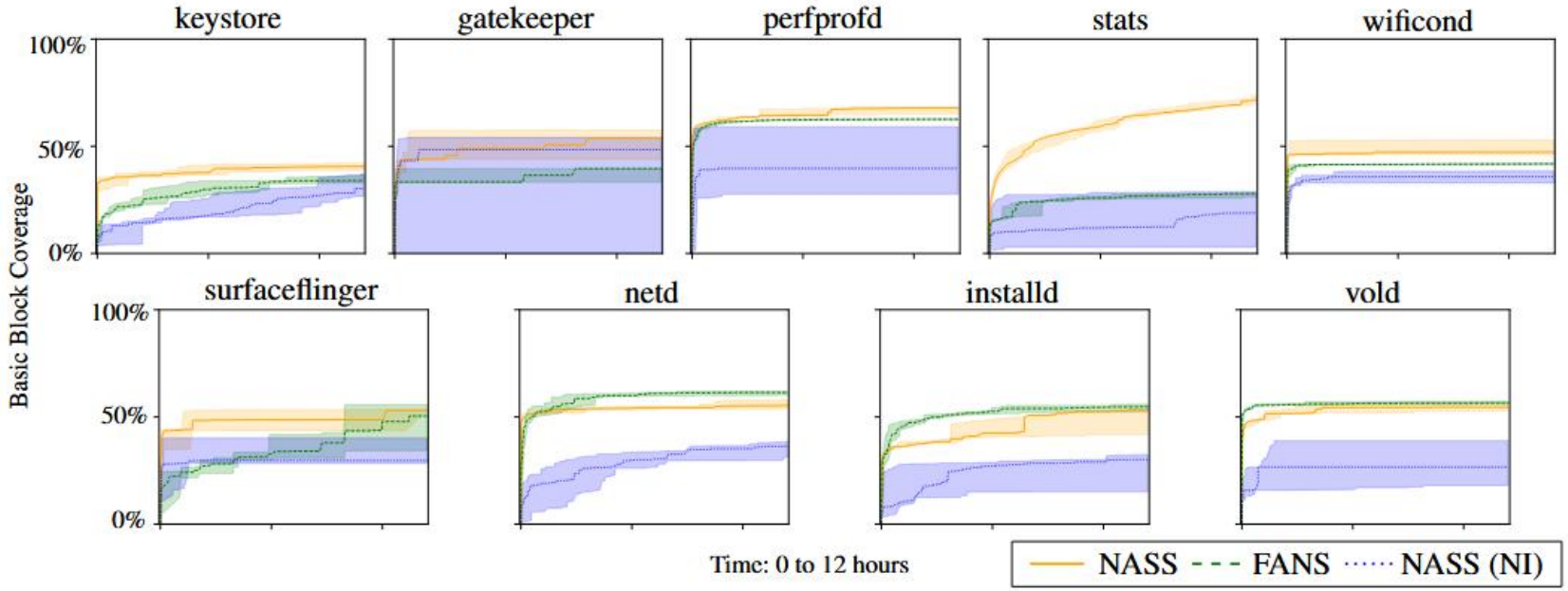
实验设计 对比实验

- RQ2：消融实验，接口感知是否优于非接口感知
  - NASS vs NASS(NI)（关闭接口感知）
  - 结果：
    - NASS覆盖率 12/14 服务优于或持平 NI
    - NASS发现19个崩溃，NI发现8个；
    - NASS发现9个独特崩溃，NI发现0个
  - 结论：接口感知显著提升了覆盖率和bug发现；字节级变异大部分算力浪费在stub反序列化失败上

Service	# Crashes / # Unique Crashes					
	NASS		FANS		NASS(NI)	
wificond	0	0	0	0	0	0
netd	0	0	2	2	0	0
installd	4	0	11	7	2	0
storaged	0	0	0	0	0	0
thermalservice	0	0	0	0	0	0
gatekeeper	2	0	2	0	2	0
incident	1	0	2	0	2	0
media.metrics	0	0	0	0	0	0
gpu	0	0	0	0	0	0
surfaceflinger	0	0	3	3	0	0
perfprofd	1	1	0	0	0	0
stats	2	0	4	2	2	0
keystore	1	1	0	0	0	0
vold	8	7	1	0	0	0
Sum	19	9	25	14	8	0

实验设计 对比实验

- RQ3：NASS相较于FANS如何?
  - 覆盖率：NASS 8/14优于FANS，4/14持平；FANS仅在需要严格语义约束（文件路径、IP 地址）的服务上占优
  - Bug：NASS 19个vsFANS 25个；NASS有9个独特 bug，FANS有14个，但FANS需要源码——NASS用纯二进制分析做到了不弱于源码分析的性能
  - 结论：NASS用纯二进制动态分析达到了与源码静态分析可比甚至更优的水平。



实验设计 对比实验

- RQ4：NASS能在真实设备上发现真实bug吗?
  - 结果:
    - 2,590个崩溃 → 去重后72个 unique → 60个 DoS + 12个内存破坏
    - 发现的漏洞中5个 CVE 已分配

Device	Service	Bug Type	Disclosure Status	Assigned Severity	CVE
Pixel 9	vendor.google.battery_mitigation.IBatteryMitigation	OOB read	fixed	high	CVE-2025-0085
Pixel 9	android.hardware.secure_element.ISecureElement	OOB read	fixed	medium	CVE-2024-56186
Pixel 9	android.hardware.radio.sap.ISap	UAF	fixed	high	CVE-2024-47040
Pixel 9	android.hardware.boot.IBootControl	OOB read	fixed	high	CVE-2024-47039
Pixel 9	android.hardware.radio.config.IRadioConfig	stack overflow	fixed	high	CVE-2025-26459
Pixel 9	android.hardware.radio.sim.IRadioSim	OOB read	disclosed	low	N/A
Pixel 9	android.hardware.radio.sim.IRadioSim	heap overflow	fixed	high	pending
Samsung S23	vendor.samsung.hardware.radio.network.ISehRadioNetwork	heap overflow	disclosed	high	pending
Redmi Note 13	miui.whetstone.klo†	invalid unmap	disclosed	none	N/A
OnePlus 12R	vendor.oplus.hardware.fido.fidoca.IFidoDaemon	OOB read	disclosed	N/A*	N/A
OnePlus 12R	vendor.oplus.hardware.engineer.IEngineer	OOB write	disclosed	none	N/A
OnePlus 12R	vendor.oplus.hardware.urcc.IUrcc	OOB write	disclosed	pending	N/A

实验设计 对比实验

- RQ5：专有服务是否遵守三项RPC原则？
  - 方法：手工逆向分析全部316个专有服务的Server Stub
  - 结果：
    - Si：100%遵守（Binder 框架强制）
    - Ab：288/316遵守（91%），28个违反——stub中混入权限检查、日志等
    - St：289/316遵守（91%），27个违反——在标准反序列化外自定义对象实例化
    - 三项全遵守：281/316（89%）
  - 结论：DGIE 的设计前提（三项原则）在接近九成专有服务上成立，适用范围广泛

Device	# Services	# (Si)	# (Ab)	# (St)	# Compliant	
Pixel 9	102	102	95	95	95	93%
Samsung S23	67	67	62	63	61	91%
Redmi Note 13	20	20	16	13	12	60%
OnePlus 12R	117	117	106	109	104	88%
Infinix X670	10	10	9	9	9	90%
Overall	316	*316	288	289	281	89%

特点总结与未来展望

算法总结

- 算法创新
  - FuzzGen++：利用patch重建OEM LLVM使FDG工具可运行，全量IR扫描自动收集consumer，结合虚函数处理和字段级数据流增强生成高质量driver
  - NASS：通过Hook反序列化函数并迭代恢复专有服务接口，过滤PID并结合线程级trace提取覆盖率，对各个RPC函数依据各自语义变异
- 算法优势
  - FuzzGen++：全链路自动化，覆盖率远超，适用于OEM Android
  - NASS：纯二进制分析，无需源码，适用任意厂商闭源服务
- 未来工作
  - 编译时静态 IR 分析与运行时动态Hook分析互补，全面覆盖Android原生代码的不同攻击面

参考文献

[1] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. FANS: Fuzzing Android Native System Services via Automated Interface Analysis. In 29th USENIX Security Symposium (USENIX Security 20)[C]. USENIX Association, 2020.

[2]Shiyan Peng, Yuan Zhang, Jiarun Dai, Yue Gu, Zhuoxiang Shen, Jingcheng Liu, Lin Wang, Yong Chen, Yu Qin, Lei Ai, Xianfeng Lu, and Min Yang. Applying Fuzz Driver Generation to Native C/C++ Libraries of OEM Android Framework: Obstacles and Solutions. In 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)[C]. New York: ACM, 2024.

[3] Philipp Mao, Marcel Busch, and Mathias Payer, et al. NASS: Fuzzing All Native Android System Services with Interface Awareness and Coverage. In 34th USENIX Security Symposium (USENIX Security 25)[C]. USENIX Association, 2025.

道德经

道可道，非常道。名可名，非常名。无名天地之始。有名万物之母。故常无欲以观其妙。常有欲以观其徼。此两者同出而异名，同谓之玄。玄之又玄，众妙之门。

谢谢！