软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(8):3745−3756 [doi: 10.13328/j.cnki.jos.006652]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于 Capstone 和流敏感混合执行的自动化反混淆技术*

鲁 辉[1], 郭润生[1], 金成杰[1], 何陆潇涵[1], 王兴伟[2], 田志宏[1]

[1](广州大学 网络空间先进技术研究院, 广东 广州 510336)
[2](东北大学 计算机科学与工程学院, 辽宁 沈阳 110169)
通信作者: 田志宏, E-mail: tianzhihong@gzhu.edu.cn

摘 要: 经过多年的技术发展和攻防对抗, Android 平台应用加固技术已较为成熟, 防护粒度逐步从通用的 DEX 动态修改发展为高度定制化的 Native 层混淆机制, 通过不断提高逆向分析的难度和工作量, 增强客户端代码防护能力. 针对近期崛起的 OLLVM 混淆加固技术, 提出一种基于 Capstone 和流敏感混合执行的自动化反混淆决方案(CiANa). CiANa 采用 Capstone 引擎分析基本块及其指令结构, 识别散落在程序反汇编控制流程图中的真实块, 并基于流敏感的混合执行确定各真实块间的执行顺序, 最后对真实块汇编指令进行指令修复得到反混淆后的可执行二进制文件. 实验对比结果表明, CiANa 可有效恢复 ARM/ARM64 架构下经 OLLVM 混淆的 Android Native 文件. CiANa 是目前为止首个在 ARM/ARM64 架构中, 支持对全版本 (Debug/Realse 版本) OLLVM 进行有效反混淆并生成可执行文件的框架, 为逆向分析提供了必要的辅助支撑.
关键词: OLLVM 混淆; Android Native 文件; 反混淆
中图法分类号: TP311

中文引用格式: 鲁辉, 郭润生, 金成杰, 何陆潇涵, 王兴伟, 田志宏. 基于Capstone和流敏感混合执行的自动化反混淆技术. 软件学报, 2023, 34(8): 3745–3756. http://www.jos.org.cn/1000-9825/6652.htm
英文引用格式: Lu H, Guo RS, Jin CJ, He LXH, Wang XW, Tian ZH. Automated Anti-obfuscation Technology Based on Capstone and Flow-sensitive Concolic Execution. Ruan Jian Xue Bao/Journal of Software, 2023, 34(8): 3745–3756 (in Chinese). http://www.jos.org.cn/1000-9825/6652.htm

Automated Anti-obfuscation Technology Based on Capstone and Flow-sensitive Concolic Execution

LU Hui[1], GUO Run-Sheng[1], JIN Cheng-Jie[1], HE Lu-Xiao-Han[1], WANG Xing-Wei[2], TIAN Zhi-Hong[1]
[1](Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou 510336, China)
[2](School of Computer Science and Engineering, Northeastern University, Shenyang 110169, China)

Abstract: After years of technical development and attack-defense confrontation, the reinforcement technology for Android applications has matured to the extent that protection granularity has gradually developed from general dynamic Dalvik executable (DEX) modification to a highly customized Native-layer obfuscation mechanism. Client code protection is strengthened by continuously increasing reverse analysis difficulty and workload. For the newly emerged reinforcement technology of obfuscator low level virtual machine (OLLVM) obfuscation, this study proposes an automatic anti-obfuscation solution CiANa based on Capstone and flow-sensitive concolic execution. The Capstone engine is used to analyze the basic block and its instruction structure, thereby identifying the real blocks scattered in the control flow graph of program disassembly. Then, the execution sequence of the real blocks is determined by leveraging flow-sensitive concolic execution. Finally, the real block assembly instructions are repaired to obtain anti-obfuscated executable binary files. The comparative experimental results show that CiANa can recover the Android Native files under OLLVM obfuscation in the ARM/ARM64 architecture. As the first framework that offers effective anti-obfuscation and generates executable files for all versions (Debug/Release version) of OLLVM in the ARM/ARM64 architecture, CiANa provides necessary auxiliary support for reverse analysis.
Key words: obfuscator low level virtual machine (OLLVM) obfuscation; Android Native file; anti-obfuscation

* 基金项目: 国家自然科学基金 (61972108, U20B2046); 国家重点研发计划 (2021YFB2012402); 广东省重点研发计划 (2020B0101120002)
收稿时间: 2021-06-23; 修改时间: 2021-09-28, 2021-12-16; 采用时间: 2022-02-14; jos 在线出版时间: 2023-02-22
CNKI 网络首发时间: 2023-02-23

IDC 报告[1]指出, 2021 年全球智能手机市场总出货量达到 13.2 亿部, 其中三星、苹果、小米、OPPO、VIVO 位居前 5 名. Android 生态框架的快速增长为 Android 经济带来巨大利润的同时, 也使其成为攻击者和复杂恶意软件的首选目标. 360 公司在 Android 恶意软件的专题报告中指出[2], 2020 年第一季度, 360 安全大脑共截获移动端新增恶意程序样本约 39.2 万个, 平均每天截获新增手机恶意程序样本约 0.4 万个, 研发更强大的 Android 恶意软件分析技术[3]是当务之急.

为增加 Android 程序流畅性和高效性, Google 引入 Android NDK 机制[4], 开发人员可在应用程序中加入本地代码二进制文件 (由 C/C++编写, 并生成共享动态链接库文件). 这种效率提升方式已在 Android APP (application, APP) 中被广泛应用[5−7]. 此外, 为进一步提高逆向工程门槛来保护知识产权, 阻止不法分子破解核心功能代码, 代码混淆工具应运而生[8−10], 如 Android 框架中知名的代码模糊处理工具 ProGuard[10]等. 这些工具可在 Java 代码级别、DEX 字节码级别和本地代码级别上运行. 目前, 代码混淆技术已被 APP 厂商广泛采用, 然而这些工具和方法也可为恶意代码提供强有力的保护, 导致逆向分析人员的静态分析过程被严重复杂化. 若本地 Native 层代码[11]也被混淆, 则安全分析难度将进一步加剧.

底层虚拟化机器 (obfuscator low level virtual machine, OLLVM)[12]是瑞士西部应用科技大学信息安全小组开发的代码混淆开源项目, 凭借混淆方案可定制且平台架构无关的特性, 在软件安全领域得到了前所未有的应用场景. 由于 Java 字节码可以很容易地反编译, 因此攻击者开始在本地代码中隐藏恶意载荷和核心功能, 从而逃避杀毒引擎的检测[5,13−15]. 为保护广大软件用户的合法权益不受威胁, 针对潜藏在本地 Native 层的恶意代码因使用 OLLVM 混淆而难以逆向分析等问题, 以消除混淆保护影响为目标, 尽可能还原原始代码并保留完整控制流的通用型反混淆技术正逐步成为当前研究热点.

1 相关工作

Android 应用的反混淆研究大都集中在 Java 级别的反混淆功能上. 例如, 基于命名混淆的 DeGuard[16]反命名混淆工具, 及其升级版本 Debin[17]. Debin 是一个使用机器学习来恢复已剥离二进制文件 (X86, X64, ARM) 调试信息 (例如: 名称/类型) 的框架. 通过对开源软件包中非剥离式二进制文件进行学习, 它能够基于决策树的分类来区分寄存器分配变量和内存分配变量, 并采用概率图模型 (使用 Nice2Predict) 来预测变量和函数的名称及类型. 然而命名混淆是代码混淆中最原始方法, 并不会改变程序的控制流程, 逆向分析人员只需重命名标记即可实现反混淆.

基于 Python 的逆向框架 Miasm[18], 通过符号执行引擎和 Miasm IR (intermediate representation) 中间语言语义解析器, 实现 Windows 平台 PE 文件和 Linux 平台 32 位 ELF 文件等多种文件格式解析, 并支持 X86 和 32 位 ARM 等多种平台的指令集. Miasm 提供的符号执行引擎可将包含汇编指令的控制流图 (CFG) 转换成代码中间表示 (IR) 的 CFG, 通过符号执行引擎执行 IR CFG, 使得寄存器/内存的值表达为 IR 表达式的组合. 但其生成的 Miasm IR Graph 不可执行, 无法验证结果是否与未反混淆前的一致性, 且 Miasm IR Graph 基本块中填充的 Miasm IR 代码晦涩难懂, 最终输出结果既不能重新反编译, 又无法得到源码, 扩展性不强.

基于 Python 的二进制分析反混淆框架 BARF[19]支持 32/64 位 Intel X86 及 ARM32 架构. 通过基本块识别、符号执行和指令修复, BARF 在 X86 平台上获得了较好还原效果. 但因为架构不同与编译优化效果差异, 并不适用于 ARM64 架构.

DiANa[20]是国内最先尝试解决 Android 的本地代码混淆问题的框架. 它通过结合静态污点分析和符号执行来消除 OLLVM 中不同混淆技术带来的影响. 其核心是引入污点分析以执行语义级别的反混淆处理, 同时考虑一般情况和编译器优化情况. DiANa 利用流敏感的符号执行来重建严重混淆的控制流. 为解决基本块分割问题, 将原始控制流程切碎分离, 通过静态功能选择分析目标, 并动态调整分析目标序列以最大化保留上下文继承关系, 最终可在 X86 和 ARM32 平台上获得优秀的反混淆效果. 但与 BARF 类似, DiANa 仅支持 ARM32 位架构. 此外, DiANa 无法输出可执行恢复文件来验证反混淆效果[21].

目前, OLLVM 反混淆框架在面对控制流混淆还原时, 存在如下 3 个挑战性问题.

挑战 1. 由于 ARM/ARM64 中存在特殊内联的跳转指令优化, 关键分支的跳转可能具有不同表现形式, 在这样

© 中国科学院软件研究所 http://www.jos.org.cn

的环境下如何准确地识别出真实块?

挑战 2. 当前基于符号执行的反混淆技术能够处理复杂度较低的混淆程序, 但对于高复杂的控制流混淆, 例如在不透明谓词中加入复杂数学猜想, 如何避免路径爆炸问题?

挑战 3. 对于反混淆后 Patch 的行为需要考虑上下文一致性, 并且不能影响 Patch 区域以外的代码偏移地址. 能否在 ARM/ARM64 架构中利用反混淆框架准确输出可验证结果一致性的可执行二进制文件?

致力于解决以上挑战性问题, 本文提出一种反混淆框架 CiANa (capstone-based deobfuscation of Android native binary code). CiANa 借助 Capstone (跨平台的反汇编框架, 能很好地将二进制代码反汇编为 ARM/ARM64 汇编指令)[22]的助记符分析具体指令功能, 并使用 Unicorn[23]模拟执行确定虚假控制流分支的真实流向, 有效解决路径爆炸问题. 并通过符号求解引擎来传入具体的分支位向量, 从而精确指导流敏感符号执行. CiANa 不依赖具体约束表达式, 只需改变其控制条件在内存建模中的临时变量值即可完成求解. 进一步基于分析得到的真实块关系, CiANa 在真实块末尾或在无用代码块构建跳板指令, 自动化形成块间连接, 结合特殊指令控制条件, 形成真实分支, 最终输出可验证结果的可执行二进制文件, 有效弥补了已有方法不能验证结果一致性的缺陷.

本文第 1 节对已有反混淆方法进行梳理与思考. 第 2 节给出具体研究框架, 并对指令替换及其反混淆、虚假控制流及其反混淆以及控制流程平坦化及其反混淆策略展开相关研究. 第 3 节给出实验验证, 最后总结全文, 并对未来值得关注的研究方向进行展望.

2 CiANa 分析框架

CiANa 的分析框架如图 1 所示, 对于给定的 Android 应用程序, 在预处理阶段, CiANa 首先对输入的二进制文件进行混淆技术类型判定; 然后基于关键寄存器污点分析对“指令替换”和“虚假控制流”保护进行指令级反混淆分析; 对于受控制流平坦化保护的二进制文件, CiANa 则执行动态调整型流敏感模拟来重建控制流. 反混淆结果最终被重新绘制并重写为二进制文件. CiANa 采用污点分析和模拟执行重新覆盖 OLLVM 混淆代码. 污点分析用于解决指令优化带来的挑战, 通过全局特征匹配来完整地检测 OLLVM 带来的混淆影响, 并通过跟踪污点寄存器来识别需要重写的指令, 基于 Unicorn 的模拟执行方法则用于虚假控制流检测. 同时采用符号执行针对被控制流平坦化破坏的控制流进行重构, 首先基于静态分析方法识别维持原始操作的基本块, 并通过流敏感混合执行发现块间关系. 最后自动化修复条件传送指令, 并将数据重写为可执行二进制文件, 验证结果一致性.

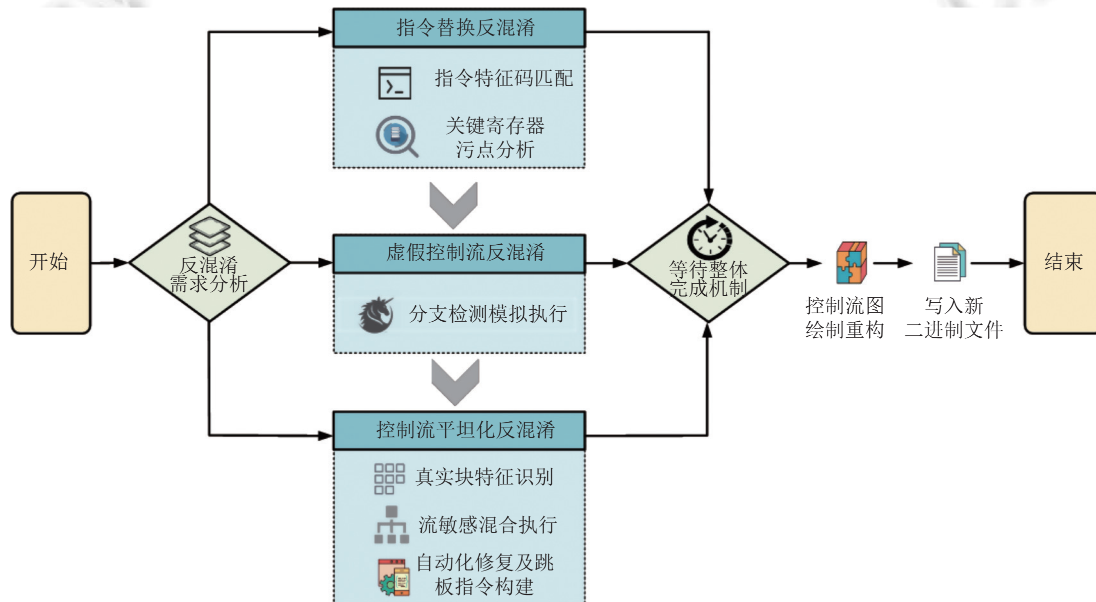


图 1　反混淆框架执行流程

2.1 指令替换及其反混淆策略

指令替换是 OLLVM 中最简单的模糊技术, 其基本思想是用功能等效但更复杂的指令序列替换标准的二进制操作, 例如算术运算 (ADD、SUB) 或布尔运算 (AND、OR 和 XOR). 为避免常数折叠的干扰, CiANa 在分析中将所有变量设置为未知数, 例如等待用户输入的变量等, 以保持指令替换参数激活. 指令替换混淆针对加、减、或、与、异或这 5 种操作进行替换. 图 2(a) 总结了 5 个不同类别, 10 种指令替换, 这些均为 OLLVM 中的常见转换.

Operator 操作指令	Obfuscated Instruction Sequence 混淆替换
$a=b+c$	$a=b-(-c)$ $a=-(-b+(-c))$ $a=b+r;\ a+=c;\ a-=r$ $a=b-r;\ a+=c;\ a+=r$
$a=b-c$	$a=b+(-c)$ $a=b+r;\ a-=c;\ a-=r$ $a=b-r;\ a-=c;\ a+=r$
$a=b\&c$	$a=(b$^$!c)\ \&b$
$a=b\|c$	$a=(b\&c)\ \|\ (b$^$c)$
$a=b$^c	$a=(!b\&c)\ \|\ (b\&!c)$

(a) 常用的指令替换

```
1. MOV R0, R2
2. BL   puts
3. SUB R0, R6, R5
4. ADD R6, R0, #3
5. SUB R0, R0, #3
6. ADD R6, R0, R5
7. B   loc_xx
```

(b) 反混淆前

```
1. MOV R0, R2
2. BL   puts
3. SUB R0, R6, R5
4. ADD R6, R0, #3
5. SUB R6, R6, #2
6. NOP
7. B   loc_xx
```

(c) 反混淆后

图 2 指令替换混淆

程序运行时的每一种运算操作, 都存在着多种功能等效的指令. 为达到干扰逆向分析的目的, 并在结果中实现指令膨胀, 指令替换机制会在众多的替代方案中随机选取一种, 然后进行混淆处理. 此外, 汇编级混淆后的指令往往与真实指令交织, 难以精准区分. 因此仅仅在指令替换混淆期间搜索特定操作码组合是不够的. CiANa 在该部分分析中沿用了 DiANa 的方法, 并在此基础上完成了 ARM64 指令的适配. 根据指令特征码匹配, 并结合污点分析来确定混淆指令的组合, 定位需要重写的指令, 实现指令替换的反混淆.

2.2 虚假控制流及其反混淆策略

虚假控制流的原理是对函数和基本块进行多层混淆, 通过向随机选择的基本块添加条件跳转来修改控制流图, 该跳转指向原始基本块或指回循环条件跳转这样的伪基本块. 其中“虚假”的核心逻辑发生在后续的混淆过程中, 即采用“不透明谓词”来修改所有永真式, 使其变得更加复杂, 并且移除所有基本块和指令的名称. 通过创建包含不透明谓词的新代码块, 来生成条件跳转, 跳转到真正的基本块或另一个包含垃圾指令的代码块. OLLVM 中使用的不透明谓词表达式是 $if(y < 10 \parallel x * (x + 1) \% 2 == 0)$. 表达式的后半部分是奇数乘以模 2 的偶数, 该值必然等于 0. 因此, 整个表达式为永真式, 并且混淆器将向代码引入一个死分支. 由于 x 的不确定性, 编译器并不会对分支进行正确地反汇编识别, 因此始终只会走向真实分支, 而虚假分支为永不可达的死分支. 此外, OLLVM 还同时使用更为复杂的不透明谓词, 对死分支进行隐藏, 例如“3X+1”猜想[24], 即对任意一个正整数, 如果为偶数则除以 2, 如果是奇数则乘以 3 再加 1, 如此迭代若干次后一定能收敛回常数 1, 这样的不透明谓词增加了迭代次数, 使用符号执行时, 极易产生路径爆炸问题.

CiANa 借助 Unicorn (跨平台的模拟执行框架, 支持在 ARM/ARM64 架构下模拟执行汇编指令)[25]的模拟执行, 识别虚假控制流中的真实分支与虚假分支. 因为模拟执行与程序真实运行所需时间相差并不大, 面对不透明谓词也可以直接计算出真实的分支流向. 可有效避免谓词块中迭代运算导致的符号执行路径爆炸问题. 使用 Unicorn 进行虚假控制流还原的主要挑战在于不透明谓词所在基本块的定位, 以及谓词块所需上下文的获取.

相比于指令替换, 虚假控制流更为复杂, 混淆前的相邻指令可能会被新加入的虚假分支分隔开, 从而落到不同的基本块中. 若不透明谓词包含垃圾代码的基本块返回其父节点块, 传统的符号执行反混淆方法将陷入死循环. 尽管虚假控制流混淆处理已经修改了控制流 (通过引入不可能分支), 但它仍然是指令级混淆处理, 最终依旧要回到

© 中国科学院软件研究所 http://www.jos.org.cn

对指令进行识别. 基于此, CiANa 采用汇编特征匹配, 检测虚假控制流混淆, 不同于传统单一指令识别方法, CiANa 将基本块内“字符串+操作数”进行拼接, 并记录决定跳转块的条件指令, 生成唯一的签名 (如: “movz12 movk12cmp11b.ne2”), 区分谓词块与无关块. CiANa 还建立两个字典 (无关块和谓词块对应的分支预测), 保证进入对应的跳转指令的更新操作, 有条件的跳转可以不采用并继续执行紧随条件跳转之后的代码的第 1 个分支, 也可以采用并跳转到程序存储器中代码的第 2 个分支所在的位置. 最后根据无关块和谓词块在匹配相关特征指令的过程中可以添加相应的映射, 从而定位判定永真分支和永假分支的谓词块. 需要说明的是, CiANa 内置的签名集合针对 OLLVM 框架生成, 对于其他混淆框架, 通过人工分析的方式总结谓词块汇编特征, 并在 CiANa 中编辑替换签名即可实现适配.

图 3 为 CiANa 虚假控制流反混淆过程示例, 其中虚假控制流引入原有真实分支 Block F, 并增加虚假分支 Block E, 并且 Block E 中的代码为死循环. 当检测到不透明谓词操作时, CiANa 将块 D 设置为谓词块并加入队列, 并记录其中使用的寄存器. 完成谓词块 Block D 的定位后, CiANa 会收集谓词块需要的上下文信息. 由于不透明谓词的永真或永假性, 不透明谓词的上下文通常在谓词块内给出, 但由于 Android 编译过程中部分基本块存在对 ARM 指令的优化, 不透明谓词也会使用前驱基本块中使用的寄存器值, 对此, CiANa 从每个谓词块进行前驱基本块回溯, 直到获得谓词块所需的寄存器上下文信息, 并在回溯的终点基本块开始模拟执行. 模拟执行会执行多次, 如果遇到执行指令中含有未初始化的寄存器, 则停止, 并回溯到该寄存器所在基本块, 更新起点重新开始模拟执行. 对于跳转指令模拟执行会进入这些跳转, 考虑到 Android Native 层文件调用中存在跳转调用外部函数的可能性, CiANa 引入 AndroidNativeEmu[26]和 Frida[27]框架, 前者解决 JNI 函数调用、系统调用等问题, 后者通过 Hook 方式解决 Java 函数的调用问题, 这些情况覆盖了大部分外部函数调用, 对于无法覆盖的调用请求, CiANa 会尝试跳过该指令并记录指令地址, 以方便人工分析并确保排除可能产生的误差. 如果遇到上下文来自谓词块不同的前驱, 则从不同的前驱执行多次模拟执行, 直到获得所需寄存器值; 对于在不同分支中对上下文的赋值情况 (如 R5 值可能来自于 Block B 或 Block C), Unicorn 只需从初始块顺序执行即可获得准确的赋值结果. 通过模拟执行最终控制流将走向真实分支, 有效地解决了挑战 2 的问题.

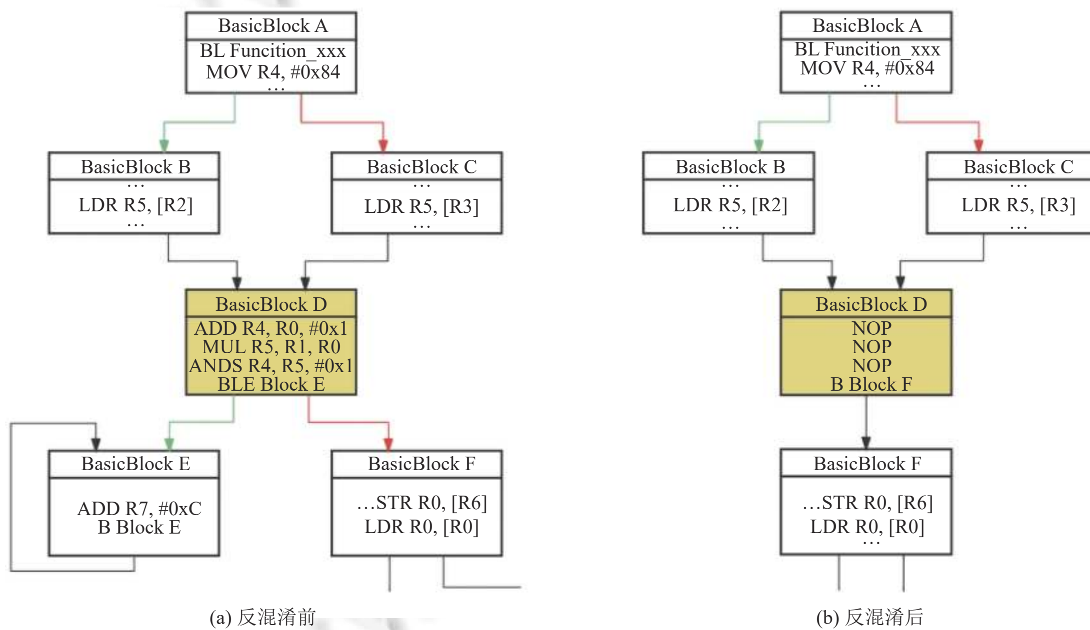


图 3 虚假控制流反混淆前后对比

后续所有需修改的指令反混淆信息将被传递给重建过程, 当二进制没有同时用后续提到的控制流程平坦化进行混淆处理时, 反混淆的结果将直接被重写为二进制文件, 否则将等待所有流程结束后再统一重写. 其中不透明谓

词部分无用代码使用 NOP 指令填充, 并且跳转逻辑会被修复, 如图 3(b) 所示.

2.3 控制流程平坦化及其反混淆策略

控制流程平坦化是目前最有效也最难破解的混淆方法, 基本思想是删除所有易于识别的条件跳转和循环结构, 并将控制流重建为大型 While+Switch 结构. 要去除相应混淆, 必须确定包含真实操作的真实块, 并重建整个控制流. 显然控制流平坦化分析是对数据流敏感的, 而符号执行[25]已经被证明是一种有效的程序分析技术, 可以尽可能多地探索执行路径, 因此 CiANa 将结合使用自动静态分析和流敏感混合执行来去除控制流平坦化混淆.

2.3.1 真实块识别

结合 ARM 上控制流程平坦化的特点, 我们将混淆控制流结构的基本块分为以下 6 类.

① 序言块 (prologue) 是待分析原始函数的起始地址块, 同时也包含关于此函数中几乎所有常量信息; ② 分发器 (dispatcher) 是一个条件跳转块, 其约束由 PC 寄存器和分发变量 (某种立即数) 决定. 序言的后继通常为主分发器 (序言指向的第一个块), 主分发器后继也可能继续承接次分发器, 这里统称为分发器; ③ 预分发器 (pre-dispatcher) 是入度大于 2 的承上启下的调度程序块. 一个混淆处理的函数可能有多个预分发器. 预分发器的后继为主分发器; ④ 返回块 (return) 是出度为 0 的基本块, 混淆处理的函数可能有多个返回块; ⑤ 相关块 (relevant) 是维持原始函数操作的基本块; ⑥ 无用块, 以上 5 类之外都是无用块.

原始操作存在于序言、返回和相关块中, 三者统一构成了真实块. 由于存在优化, 相关块可以部分合并. 在识别真实块过程中, CiANa 采用符号执行以探索每个可能的路径. 找出真实块和序言和返回块之间的调用关系, 同时保存执行到当前基本块时的路径约束、寄存器、内存等状态信息. 路径信息包括执行路径上的历史节点、调用栈等, 随后就以当前路径为基础创建后继路径. CiANa 规定, 初始化符号执行阶段一直寻找存在后继块且未超出预先设定循环阈值的相关块, 其特征是后继都是预分发器.

CiANa 遍历分析序言开始的所有块, 建立块地址及其对应分支后继地址的映射. 当一个块存在分支时, 后继之一为预处理器的块为真实块, 换言之, 真实块的后继通常都会连接预处理器. 同时, 验证当前分析的块不是序言块且不是主分发器、虚假块以及当前块的分支不是自身 (不加入指向自身的基本块), 以上这些均为实际真实块的特征. 为尽可能多地找到真实块, CiANa 增大筛选的粒度, 例如, 若父块是虚假块, 且第 1 个分支不是虚假块, 且当前块的指令长度大于 1, 则也算作真实块. 同时 CiANa 向上溯源父块的指令长度以及当前父块是否处于之前寻找到的虚假块列表中, 若在则从真实块列表中删除, 只保留指令长度为 1 的父块. 从而避免真实指令以一个块的形式呈现, 并由此产生误报的情况.

ARM64 平台上大多使用操作码“CSEL/CSET”作为特征, 检测是否存在多个分支. 由于 ARM 指令集的简单性, 编译优化后分支也可能以其他简单指令组合的形式存在, 因此 CiANa 使用在划分基本块操作中建立好的分发器字典作为主要指示符. 如果赋值操作中源操作数存在于分发器字典中, 则将这条指令分配的寄存器 (即存放该条 ARM 指令第 1 个操作数的寄存器) 设置为“受污染寄存器”. 如果在这个块返回到预分发器的路径上有一个污染寄存器相关的条件移动指令, 则认为是真实块且在原始控制流中有多个后继者. 当被正常的指令修改时, 受污染的寄存器将被释放.

CiANa 通过静态分析反向排除虚假块. 针对具有内联汇编混淆条件传送指令, 使用污点分析技术追踪寄存器数据流, 定位影响存储分发变量的寄存器, 从而判断真实块中的指令, 增加真实块的筛选粒度, 有效解决挑战 1 所面临问题.

2.3.2 流敏感的混合执行

流敏感体现在反混淆过程需要保持基本块间的依赖关系. 在对基本块进行符号执行时, 若均使用空符号状态, 则会导致上下文缺失、后续块匹配乱序或路径爆炸. 对此, CiANa 动态保存当前每一步符号执行状态, 并更改关键的影响条件分支临时变量, 完成状态更新以保证由具体值指导的状态继承流.

DiANa 采用双指针动态遍历交换基本块的方式来确保大多数基本块在执行前继承先前分析的状态, 其潜在问题是带来大量重复、耗时高的交换. 与之不同, CiANa 可在实时执行过程中维护程序的真实块执行流和“状态-块”字典, 根据状态在分支时刻输入具体值[25]指导符号执行更快速地运行. 具体流程如下.

(1) 初始化. 构建真实块列表, 存储为真实块识别过程中静态分析产生的相关块.

(2) 预处理. 针对列表中每一个真实块扩展出中间语言块, 每执行到块首地址时以键值对的方式记录当前执行状态.

(3) 判断是否有两个分支, 是则说明存在条件分支, 并进入步骤 (4). 否则进入步骤 (5), 同时记录当前块键值.

(4) 寻找关键的分支变量. 中间语言块中的表达式中寻找能引起条件分支变化的临时变量, 并使用具体的位向量 (0 或 1) 来指导其符号执行求解环节, 如果当前块不是返回块则进入步骤 (5), 否则进入步骤 (6). 同时记录当前块为步骤 (3) 键值对, 形成真实块的“块-状态”字典.

(5) 确定状态和块之间的关系. 对当前块进行符号执行, 并寻找到下一个块后, 返回当前状态对应块的首地址, 同时更新“块-状态”字典, 并记录当前块为步骤 (3) 键值对. 并返回步骤 (2).

(6) 若当前块为返回块, 说明程序执行完毕, 结束符号执行.

在真实块流分析过程中, CiANa 只关心当前块及其子块, 每次混合执行前均会恢复字典中对每个父块的状态, 执行完后就会得出正确的子块首地址及其状态, 再次加入字典中以供下次经过该子块时搜索使用保存过的状态. CiANa 通过“块-状态”字典实时维护块和状态间的关系, 并根据字典为符号执行提供真实块的轻量级状态查询, 避免了 DiANa 的不足.

2.3.3 自动化修复及跳板指令构建

自动化修复及构建跳板指令的任务是在前面真实块识别及其关系间探索的基础上, 进一步将这些有关系的真实块连接起来. 每条 ARM 指令都包含 4 位条件码, 位于指令的最高 4 位 [31:28]. 共有 16 种条件码, 每种条件可用两个字符表示, 并可添加在指令助记符后和指令同时使用. 如 BEQ 是跳转指令 B 加上后缀 BEQ 表示“相等则跳转”. CiANa 在各真实块中遍历指令, 若遇到 MOVXX 或 CSEL/CSET 等条件移动指令, 就在静态分析中记录其指令地址及控制条件 (EQ、NE、LE、GE、LT、GT、AL 等), 并填充 NOP 消除后面的 CMP 指令. 此阶段, CiANa 会利用在真实块中符号执行时提前保存好的 MOVXX 或 CSEL/CSET 指令位置, 自动修改其机器码使其替换为 BXX 指令. 由于真实块会在末尾通过 B 指令跳回主分发器或次分发器, 因此在控制流还原时无需关心跳回分发器的 B 指令, 只需在 BXX 指令地址+4 处直接插入一条跳向 False 后继块的 B 指令覆盖原有跳向分发器的操作. 具体流程如下.

(1) 处理虚假块. 由于已经找到了真实块, 那么将剩余标记成虚假块的块填充 NOP 指令.

(2) 分析真实块执行流和“状态-块”字典, 若存在两个分支, 则直接进入步骤 (4), 否则进入步骤 (3).

(3) 替换真实块的末尾指令是否为跳转到真实块执行流下一个真实块的指令. 返回步骤 (2).

(4) 根据静态分析记录的条件传送指令操作码中提取出条件部分, 并结合 B 指令形成带有对应条件的跳转指令, 修复到真实块的末尾, 形成真实的条件分支结构. 返回步骤 (2).

在实际修复程序时, 若 ARM64 架构中基本块的结尾并非 B 系列跳转, 则会导致真实指令被消除的错误情况. 为此, CiANa 在处理 64 位程序时构建跳板指令, 先跳转到无用代码区域 (通常是含有多个 NOP 指令的基本块, 并保证这个基本块可容纳构建跳转指令的长度). 如图 4 所示, 从左侧“状态-块”字典可知, 0x820 对应 0x94C, 由此选择紧邻 Loc_820 块的 Loc_83C 块, 插入一条跳转到 0x94C 的 B 指令. 若临近位置处没有无用代码区域, CiANa 则从之前寻找相关块时过滤掉的虚假代码块中获取.

此外还存在可利用空间有限的问题. ARM 中存在基本块代码复用的情况 (如利用 PUSH 和 POP 进行寄存器值传递等), 被复用的基本块具有多个前驱和后继, 且这些执行路径在逻辑上并无相关性, 此时反混淆处理需要对基本块代码进行拷贝, 上述操作会造成一定程度的代码膨胀, 对于单个函数的修复利用相邻地址空间即可, 但 CiANa 的目标是修复整个可执行文件而非单个函数, 这对程序地址空间的利用提出了新的要求.

综合考虑 ARM64 程序中末尾指令的特殊情况, CiANa 填充 NOP 指令的虚假块所处的空间并构建跳向各种指令地址的跳板. 进一步考虑 NOP 指令替换存在空间不足的情况, CiANa 会在寻找不到合适无用地址空间时在段末尾新增一段空白地址空间, 并在全部控制流还原完成后修复可执行文件结构. 最终, CiANa 结合真实块识别和流敏感混合执行的结果, 将有关系的真实块连接起来, 通过自动化修复跳转的方式还原回原始控制流, 有效解决了挑战 3 所面临的问题.

© 中国科学院软件研究所 http://www.jos.org.cn

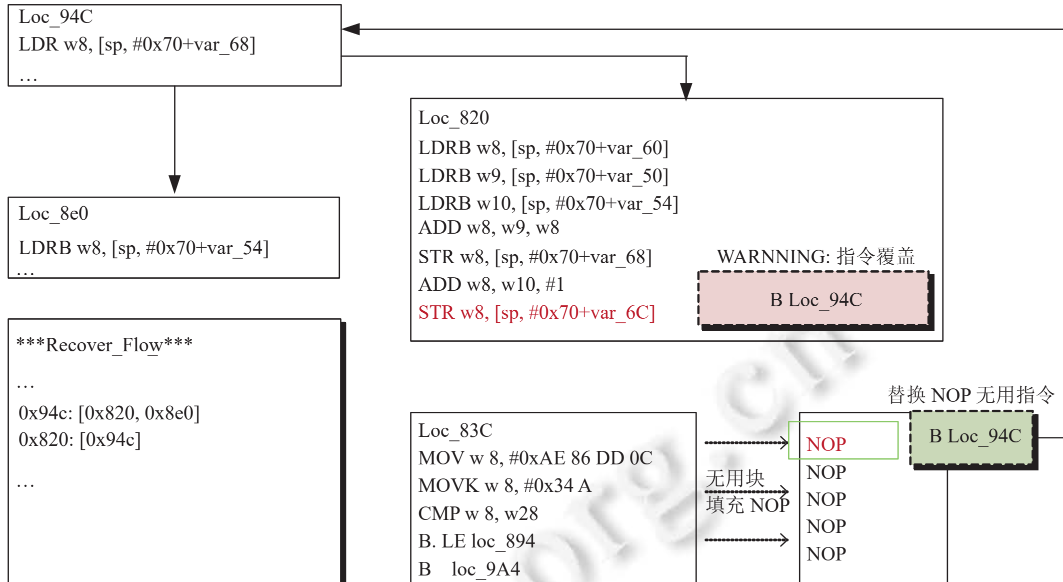


图 4　NOP 无用指令替换

3 实　验

我们设置了相关测试, 对 CiANa 做进一步的实验验证, 测试环境安装部署了最新版 LLVM-4.0 混淆框架、Angr 9.0.6852、Unicorn 1.0.3、Frida 12.8.0 以及 CiANa. 运行 CiANa 的操作系统为 Ubuntu 18.04 AMD64, 详细配置为 Intel(R)Core(T)i7-4790@3.60 GHz, 内存 12 GB; 运行 Frida 模块的操作系统为 AOSP Android-8.0.0_r2, 设备型号为 Google Pixel 1, 内存 4 GB.

对于无源码目标程序反混淆效果无法精确验证, 为此我们采用 Python 的 NetworkX 包对 CiANa 的节点型数据结果进行可视化处理, 生成控制流图 CFG, 同时从图特征向量相似性计算 (观察相似性度量 *Similar* 的值), 以及反混淆结果运行正确性判定两个方面进行验证. 相似性计算公式如下:

$$Sp_i = laplacian_spectrum(G_i) \tag{1}$$

$$k_j \leftarrow \frac{\sum_{i=1}^{k++} Sp_j[i]}{\sum_{i=1}^{len(Sp)} Sp_j[i]} \geqslant 90\% \tag{2}$$

$$k = \min(k_1, k_2) \tag{3}$$

$$Similar = \sum\nolimits_{i=1}^{k} (Sp_1^2[:k] - Sp_2^2[:k]) \tag{4}$$

实验场景设置如下, CiANa 为需要比较的两个图 (反混淆后的 CFG 图和原始开源程序的 CFG 图) 的邻接矩阵计算拉普拉斯特征值并输出 *Sp* 列表. 对于每个图的 *Sp* 列表进行遍历并找到下标 *k*, 使这最大 *k* 个特征值的总和至少占所有特征值之和的 90%. 如果两个图之间的 *k* 值不同, 则使用较小的一个. 然后, 相似性度量是图形之间最大 *k* 个特征值之间的平方差之和. 这将产生 [0, ∞] 范围内的相似性度量值, 结果中 *Similar* 更接近 0, 则说明值更相似. 在结果一致性测试方面, 使用 Frida Hook 获取可执行文件基址与函数列表, 通过主动调用的方式传入输入值并调用目标函数, 并对每个目标函数构造 2000 组随机输入, 检验输出结果是否一致.

如表 1 所示, 本文选取了从 SPECint-2000 (https://www.spec.org/cpu2000/CINT2000/) 下载的 10 组二进制开源程序进行了不同程度的 OLLVM 混淆加固 (下载地址在每项条目的完整描述中), 通过设置-mllvm -sub、-bcf、-fla

© 中国科学院软件研究所 http://www.jos.org.cn

这 3 个参数控制指令替换、虚假控制流、控制流平坦化的开启与否. 实验选择的 10 个典型测试样例均开启了控制流程平坦化, 且每个程序通过-mllvm -fla_loop 来进行配置, 拥有不同程度的 Loop 层数 (最大设置成了对函数应用 3 层平坦化机制). 与其他工具不同, CiANa 可以自动化修复为恢复的可执行二进制文件, 且能保证结果一致性基本为 100%, 说明 2000 组随机输入都产生了一致的结果, 即在使用相同输入的情况下, 反混淆后的程序能输出和源程序一样的结果, 运行结果一致性为 100%. 表 1 中 *Similar*-1 代表原程序与混淆后程序的相似度值, *Similar*-2 代表原程序与经过反混淆还原的相似度值. 其中 *Similar*-2 结果一直维持在较低水平, 未曾超过 15, 说明还原后的程序在图的相似度上满足相似性. 在实验过程中, 我们也一直人工观察反混淆后的结果, 均可清楚地看到原始程序的核心算法过程. 综合各实验数据可知, 反混淆相似性 *Similar* 与分支循环个数 Fla-Loop 呈正相关, 分支循环个数越多, 越难以完美还原原始的控制流, 但最后基本上都维持在一个可以接受的范围之内.

表 2 依次列出了实验所用的 10 个混淆后的未开源程序样本及其大小信息. 由于未开源程序无法和原始内容做对比验证, 只能观察运行结果是否一致. 从运行结果不难看出, CiANa 对未开源程序也是具有高度的还原率, 能保证反混淆与文件开源与否呈无关联系. 由于不清楚具体开了多少层混淆, 只能观察到反混淆后大小是和源文件相同. 其原因是反混淆并不会对文件进行裁剪, 而是对相应代码无关位置填充 NOP 指令 (并不是空, 也有对应的机器码), 抑或借用无关指令区域填充跳板指令, CiANa 在无法使用无关指令区域时添加新地址空间的机制并未被频繁使用, 因此最终呈现的是反混淆后文件大小基本保持不变.

表 1 开源文件反混淆相似性度量和运行结果

组数	指令替换	虚假控制流	控制流平坦化	Fla-Loop	*Similar*-1	*Similar*-2	结果一致性 (%)
1	√	×	√	3	34.1517	6.2534	100
2	×	√	√	2	127.4912	5.7342	100
3	×	√	√	1	47.1471	1.5788	100
4	×	×	√	1	30.4871	0.6355	100
5	√	√	√	2	103.1498	9.7464	100
6	√	√	√	3	91.1477	11.4494	100
7	√	×	√	3	57.2454	7.8513	100
8	×	×	√	2	31.7418	6.1794	100
9	√	√	√	3	75.7845	7.1465	100
10	×	√	√	3	50.5677	8.1275	100

表 2 未开源文件大小和反混淆运行结果

程序样本	混淆后大小 (KB)	反混淆后大小 (KB)	结果一致性 (%)
1	13.8	13.8	100
2	24.5	24.5	100
3	26.2	26.2	100
4	29.8	29.8	100
5	36.4	36.4	100
6	52.6	52.6	100
7	59.7	59.7	100
8	68.8	68.8	100
9	81.7	81.7	100
10	106.1	106.1	100

表 3 中列出了当前较为流行的 OLLVM 自动化反混淆框架的横向对比情况, CiANa 使用的 Capstone、Unicorn 框架支持 ARM、ARM64、MIPS、X86 等架构, 可移植性强, 我们也研究了 X86 架构下实现自动化反混淆的策略, 仅需要在 CiANa 中添加关于 X86 条件指令的机器码, 并在相应位置添加 IF 分支跳转处理即可. DiANa 中涉及的技术和 CiANa 有一些相同点, 但其由于使用和 BARF 相同的二进制框架, 而 ARM 与 ARM64 使用不同的寄存器, 指令集也存在区别, BARF 仅适配 ARM32 位指令, 因此这两个工具在 ARM64 架构上完全无法起作用 (Miasm 也同样不支持 ARM64 架构). DiANa 使用的 Angr 及其依赖的其他包的版本明显过时 (依赖包数目多), 它还使用 Python 2 启动 Angr, 而最新维护的 Angr 框架仅支持 Python 3. 本框架使用的是目前为止 Angr 的最新版本, 且不受 ARM 多架构影响. 此外, DiANa 在处理 Release 版本的 OLLVM 之时, 对 MOVW 条件传送指令的判断有错误点. CiANa 还可以和 BARF 框架一样自动化地修复指令, 完成输出可执行反混淆结果的任务, 但 BARF 仅在 X86 指令集上完成, 而 ARM 指令集中的机器码中地址偏移计算是需要探索相应的转换公式. DiANa 和基于 Miasm 的反混淆框架一样仅仅能最后输出一张节点控制流图, 其中 Miasm 生成的中间语言图晦涩难懂, 不易理解.

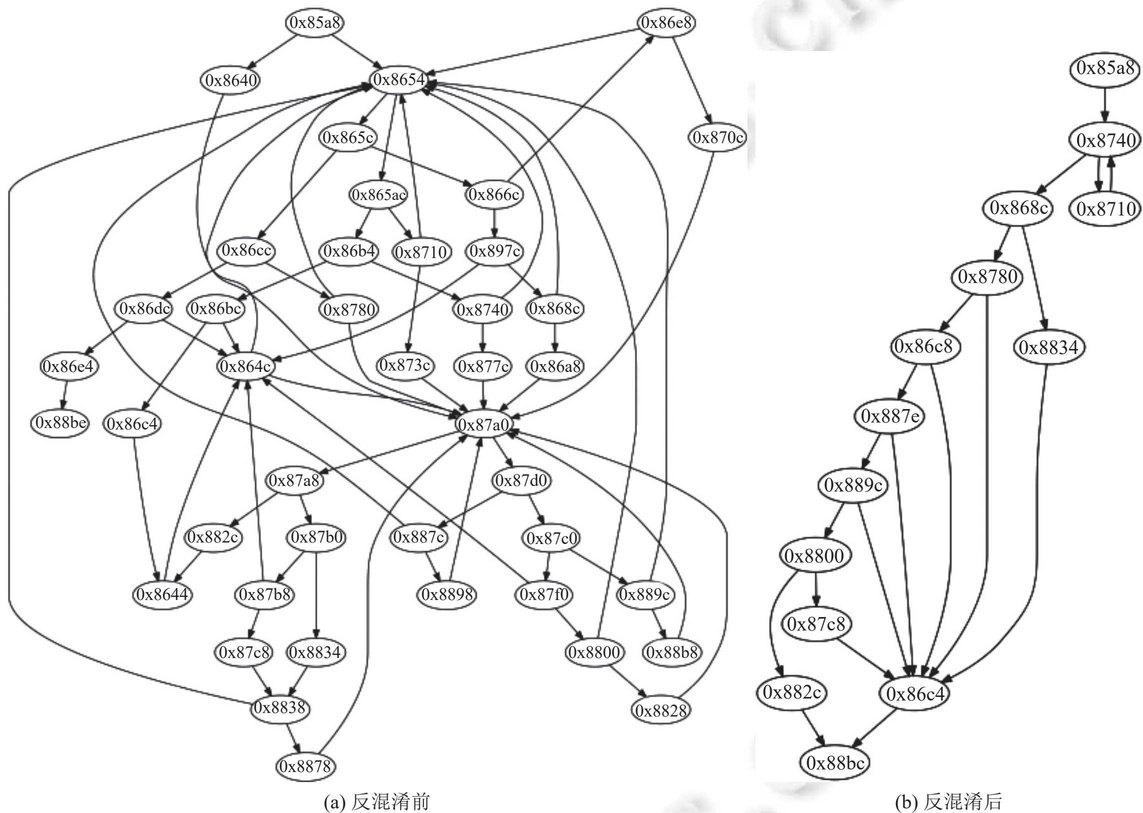
CiANa 最后输出的控制流图同 DiANa, 使用 NetworkX 绘制. 我们展示了 gzip 的其中一组前后控制流图对比, 程序添加了虚假控制流和控制流平坦化, 对应表 1 的第 3 组, 结果如图 5 所示. 对比图省去了代码部分, 仅输出地址信息及其相应的块间关系, 可以看出函数的控制流逻辑被极大简化. 从左图可以看到混淆后前驱大于 5 的基本块有 3 个, 分别为 0x8654、0x864c 和 0x87a0, 这是控制流混淆中支配节点的一大特征, 而右图中 CiANa 识别并删

© 中国科学院软件研究所 http://www.jos.org.cn

除了 0x8654 与 0x87a0, 保留了 0x86c4, 经过人工验证 0x86c4 为原程序代码, 说明 CiANa 正确区分了真实代码与混淆添加的虚假代码.

表 3　与现有 O-LLVM 反混淆框架的对比

框架	针对ARM平台多架构支持程度	反混淆结果可执行	自动化修复指令	直观输出程序控制流图
Miasm	不支持ARM64	×	×	√
BARF	不支持ARM/ARM64	√	√	×
DiANa	不支持ARM64	×	×	√
CiANa	二者均支持	√	√	√



(a) 反混淆前　　(b) 反混淆后

图 5　反混淆前后控制流图对比

4　总结和展望

为了解决 Android Native 层 OLLVM 混淆加固, 本文提出了 CiANa 自动化反混淆框架. CiANa 框架能有效地解决 ARM/ARM64 平台 Release/Debug 版本的 OLLVM 混淆. 实验数据表明, CiANa 还原的控制流与原始控制流的相似度 Similar 值接近 0 (由于是方差计算, 因此越接近 0 说明越相似), 且执行效果一致度达 100%. 这基本上可以认为 CiANa 在某种程度上恢复了原始控制流, 且大大增加对 So 文件的逆向分析效率, 可为分析正常 App 的安全性提供反混淆后的 So 文件输入.

CiANa 目前只对最多开启 3 层混淆的程序进行了研究, 在实验的过程中我们也尝试过开 3 层以上的混淆, 但面对多层混淆会令真实块中的代码越来越发生“代码膨胀”的现象, CiANa 尚未能把真实块中冗余的一些指令全都

© 中国科学院软件研究所 http://www.jos.org.cn

精准地识别出来; 模拟执行的一个关键点是需要对谓词块进行精准的定位, 在实验中我们发现谓词块可能在 ARM 的优化机制下被分割到不同的基本块中, 甚至进入真实代码的基本块中, CiANa 此时会错误地识别并获取上下文; 同样的, 基本块内指令混合的问题也影响了 CiANa 识别真实块的精度, AndroidNativeEmu 与 Frida 主动调用并不能覆盖所有外部函数调用的情况, 这些问题将在未来加以改进.

另外, CiANa 对于谓词块的识别具有局限性, 内置的签名只适合用于 OLLVM 框架, 对于其他类型的或是经过修改的类 OLLVM 混淆框架, 需要人工提取特征再修改签名方可适配. 要完成对各类框架的普适化, 我们认为通过机器学习对特征进行识别是值得研究的方向, 这将是我们下一步工作的方向.

References:

[1] IDC Corporation. Smartphone market share. 2020. https://www.idc.com/promo/smartphone-market-share/os

[2] Qihoo Corporation. Special report for Android malwares in 2019. 2019. https://blogs.360.cn/post/review_Android_malware_of_2019.html

[3] Wang W, Gao ZZ, Zhao MC, Li YD, Liu JQ, Zhang XL. DroidEnsemble: Detecting Android malicious applications with ensemble of string and structural static features. IEEE Access, 2018, 6: 31798–31807. [doi: 10.1109/ACCESS.2018.2835654]

[4] Google. Android NDK—Android developers. 2021. https://developer.Android.com/ndk/

[5] Afonso VM, de Geus PL, Bianchi A, Fratantonio Y, Kruegel C, Vigna G, Doupé A, Polino M. Going native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy. In: Proc. of the 23rd Annual Network and Distributed System Security Symp. San Diego, 2016. 1–15.

[6] Alam S, Qu ZY, Riley R, Chen Y, Rastogi V. DroidNative: Semantic-based detection of Android native code malware. arXiv:1602.04693, 2016.

[7] Sun MT, Tan G. NativeGuard: Protecting Android applications from third-party native libraries. In: Proc. of the 2014 ACM Conf. on Security and Privacy in Wireless & Mobile Networks. Oxford: ACM, 2014. 165–176. [doi: 10.1145/2627393.2627396]

[8] Guardsquare. Android obfuscation and runtime-self protection (RASP)—DexGuard. 2018. https://www.guardsquare.com/en/products/dexguard

[9] Dexprotector. DexProtector by Licel: Cutting edge obfuscator for Android APPs. 2018. https://dexprotector.com

[10] Guardsquare. ProGuard. 2018. https://www.guardsquare.com/en/products/proguard

[11] Zhang XH, Zhang Y, Chi XJ, Yang M. Protecting Android native code based on instruction virtualization. Journal of Electronics & Information Technology, 2020, 42(9): 2108–2116 (in Chinese with English abstract). [doi: 10.11999/JEIT191036]

[12] Junod P, Rinaldini J, Wehrli J, Michielin J. Obfuscator-LLVM—Software protection for the masses. In: Proc. of the 1st IEEE/ACM Int'l Workshop on Software Protection. Florence: IEEE, 2015. 3–9. [doi: 10.1109/SPRO.2015.10]

[13] Lu K. Deep analysis of Android rootnik malware using advanced anti-debug and anti-hook, Part I: Debugging in the scope of native layer. 2017. https://www.fortinet.com/blog/threat-research/deep-analysis-of-android-rootnik-malware-using-advanced-anti-debug-and-anti-hook-part-i-debugging-in-the-scope-of-native-layer

[14] Gu J, Zhang V, Shen S. ZNIU: First Android malware to exploit dirty COW. 2017. https://www.trendmicro.com/en_us/research/17/i/zniu-first-android-malware-exploit-dirty-cow-vulnerability.html

[15] Allatori. Allatori Java Obfuscator. 2021. https://www.allatori.com

[16] Bichsel B, Raychev V, Tsankov P, Vechev M. Statistical deobfuscation of Android applications. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. Vienna: ACM, 2016. 343–355. [doi: 10.1145/2976749.2978422]

[17] He JX, Ivanov P, Tsankov P, Raychev V, Vechev M. Debin: Predicting debug information in stripped binaries. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. Toronto: ACM, 2018. 1667–1680. [doi: 10.1145/3243734.3243866]

[18] Desclaux, Fabrice. Miasm: Framework de reverse engineering. Actes du Symposium sur la Sécurité des Technologies de l'Information et des Communications. SSTIC. 2012.

[19] Xiao ST, Zhou AM, Liu L, Jia P, Liu LP. Obfuscator low level virtual machine deobfuscation framework based on symbolic execution. Journal of Computer Applications, 2018, 38(6): 1745–1750 (in Chinese with English abstract). [doi: 10.11772/j.issn.1001-9081.2017122892]

[20] Kan ZL, Wang HY, Wu L, Guo Y, Luo DX. Automated deobfuscation of Android native binary code. arXiv:1907.06828, 2019.

[21] Programa-stic. BARF: Binary analysis and reverse engineering framework. 2021. https://github.com/programa-stic/barf-project

[22] Aquynh. Capstone engine: A disassembly framework with the target of becoming the ultimate disasm engine for binary analysis and

© 中国科学院软件研究所 http://www.jos.org.cn

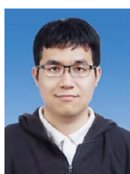
reversing in the security community. 2021. http://www.capstone-engine.org/
[23] Aquynh. Unicorn is a lightweight multi-platform, multi-architecture CPU emulator framework. 2021. https://www.unicorn-engine.org/
[24] Cheng R. New direction of code obfuscation research. 2021 (in Chinese). https://bbs.pediy.com/thread-270019.htm
[25] Sun S. Research on directed fuzzing technology based on keypoint coverage and concolic testing [MS. Thesis]. Harbin: Harbin Institute of Technology, 2020 (in Chinese with English abstract). [doi: 10.27061/d.cnki.ghgdu.2020.002285]
[26] AeonLucid. AndroidNativeEmu. 2021. https://github.com/AeonLucid/AndroidNativeEmu
[27] Oleavr. Frida. 2021. https://frida.re

附中文参考文献:

[11] 张晓寒, 张源, 池信坚, 杨珉. 基于指令虚拟化的安卓本地代码加固方法. 电子与信息学报, 2020, 42(9): 2108–2116. [doi: 10.11999/JEIT191036]
[19] 肖顺陶, 周安民, 刘亮, 贾鹏, 刘露平. 基于符号执行的底层虚拟机混淆器反混淆框架. 计算机应用, 2018, 38(6): 1745–1750. [doi: 10.11772/j.issn.1001-9081.2017122892]
[24] 程瑞. 代码混淆研究的新方向. 2021. https://bbs.pediy.com/thread-270019.htm
[25] 孙帅. 基于关键点覆盖和混合执行的定向模糊测试技术研究 [硕士学位论文]. 哈尔滨: 哈尔滨工业大学, 2020. [doi: 10.27061/d.cnki.ghgdu.2020.002285]

鲁辉(1981－), 男, 博士, 教授, 主要研究领域为网络攻防对抗, 智能化漏洞挖掘, 移动端脱壳, 反混淆技术.

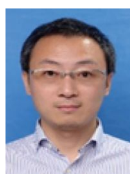
何陆潇涵(1995－), 女, 硕士生, 主要研究领域为网络攻防对抗, 移动端脱壳, 反混淆技术.



郭润生(1997－), 男, 硕士生, 主要研究领域为网络攻防对抗, 移动端脱壳, 反混淆技术.

王兴伟(1968－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为未来互联网, 云计算和网络安全.

金成杰(1994－), 男, 硕士生, 主要研究领域为网络攻防对抗, 移动端脱壳, 反混淆技术.



田志宏(1978－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为网络攻防对抗, APT 检测与溯源, 工控安全.