文章编号:1007-5321(2018)06-0001-06 DOI:10.13190/j.jbupt.2018-017

一种基于 CFI 保护的 Android Native 代码保护框架

张 文[1], 刘文灵[2], 李 晖[2], 陈 泽[2], 牛少彰[1]
(1. 北京邮电大学 智能通信软件与多媒体北京市重点实验室, 北京 100876;
2. 北京邮电大学 网络空间安全学院, 北京 100876)

摘要: 针对 Android 应用的 native 代码面对的关键代码提取攻击和恶意代码植入攻击问题,提出了一个基于控制流完整性(CFI)保护的代码保护框架 DroidCFI. 该框架通过对被保护应用进行静态分析,提取其 native 代码的控制流特征,向开发者提供可视化策略配置视图设定关键函数,并根据策略配置生成对应的加固代码,与被保护应用的其他部分一起形成目标应用;目标应用在运行时,通过对关键函数进行动态 CFI 检查判定是否遭遇上述攻击,从而达到保护目的. 实验结果表明,DroidCFI 能够通过极小的性能开销实现对应用软件 native 代码的安全性保护.

关 键 词: native 代码; 关键代码提取攻击; 恶意代码植入攻击; 控制流完整性保护
中图分类号: TN929.53 **文献标志码**: A

A Protection Framework for Android Native Code Based on CFI

ZHANG Wen[1], LIU Wen-ling[2], LI Hui[2], CHEN Ze[2], NIU Shao-zhang[1]
(1. Beijing Key Laboratory of Intelligent Telecommunication Software and Multimedia, Beijing University of Posts and Telecommunications, Beijing 100876, China; 2. School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China)

Abstract: A native code reinforcement framework based on control-flow integrity (CFI), DroidCFI is proposed, to prevent native code of Android applications from core code extraction and malicious injection. This framework can extract the control-flow features of subroutine invocation process by static analysis, provide developers with a visual policy configuration view to set the reinforced points, generate the reinforcement code based on the CFI policy, and integrate the verification module into the target application. Then a CFI check is enforced during the run-time of the application to defend against the malicious attack. Experiments show that DroidCFI can provide secure protection to native code of applications by minimal performance overhead.

Key words: native code; core code extraction attack; malicious code injection attack; control-flow integrity protect

Android 提供了基于 Java 的 SDK(software development kit),以供开发者进行第三方应用的开发实现,与此同时,Google 还提供了一套标准完备的 JNI(Java native interface)接口,使得 Java 以尽量少的代码,尽可能相同的方式,通过 native 代码调用本地类库. 但是目前随着逆向工程的发展和攻击工具被不断提出,native 代码也面临各种威胁. 一方面,攻击者利用静态分析,通过反汇编工具提取可执行代码,或利用启发式分析重构代码;另一方面,攻击者可以在程序运行过程中,令 hook 指定 API(application programming interface),修改函数原有功能,或者通过内存漏洞构造缓冲区溢出或栈溢出,甚至直接

收稿日期: 2018-01-15
基金项目: 国家自然科学基金项目(61628202)
作者简介: 张 文(1981—), 男, 博士生, E-mail:roy.zhang@bupt.edu.cn; 李 晖(1970—), 女, 副教授.

修改内存中变量或寄存器的值,篡改代码执行逻辑.

传统的加固方案,如代码加密、代码混淆和防代码动态调试等,在一定程度上能够抵御上述攻击,但是这些方案的代码保护粒度不够,容易被攻击者绕过. 笔者从程序控制流(CF, control-flow)的角度出发,对 Android native 代码的各类攻击进行特征分析,并在此基础上提出了一个基于控制流完整性(CFI, control-flow integrity)保护的 native 代码保护框架 DroidCFI. 该框架针对 Android native 代码的 CF 实现细粒度的校验机制,能够有效抵御攻击者对 native 代码的复用攻击和植入攻击,为应用保护提供了一种新的思路.

1 相关研究

Abadi 等[1-3]于2005年提出 CFI 的概念,给出了其理论基础及假设和证明公式,并提出了2个 CFI 执行的实用技术和 CFI 的2种具体实现. 这些早期方案虽然存在一些问题,但是为后续程序代码保护提供了一个加强安全策略的基础.

随着后续研究的不断深入,侧重实用性的粗粒度 CFI 方案不断被提出,重点在于提高执行 CFI 检验的效率,同时兼容多架构. Zhang 等[4]提出了将 CFI 与随机化结合的实现方案 CCFIR; Zhang 和 Sekar[5-6]提出了针对 COTS(commercial off-the-shelf)程序的 CFI 机制 binCFI 以及 CFCI,并发布开源软件. 粗粒度的 CFI 机制有效降低了 CFI 引入的开销,但是同时也引入了一些安全问题. Göktas 等[7]指出了粗粒度 CFI 的局限性,并提出了2种特殊的代码段,即入口点攻击和调用点攻击,绕过完整性校验.

为了解决这一问题,后续研究主要考虑细粒度的 CFI 方案. Mashtizadeh 等[8]提出了 CCFI 方案,对运行上下文信息进行加密保存,程序返回时解密并对比,以保证更细粒度和可行性. Niu 和 Tan[9]提出了 RockJIT,基于源码构造一个细粒度的控制流图(CFG, control flow graph),并随着加入新代码动态更新 CF 策略. Criswell 等[10]不使用重量级的完整内存安全,提出了一个针对商用操作系统的全面的 CFI 保护方案. van der Veen 等[11]提出了 PathArmor,基于当前商用硬件,利用上下文定义更高精度的 CF 边,实现更准确的完整性控制. Mohan 等[12]提出了一个二进制软件随机化与 CFI 结合的执行系统 O-CFI,隐藏了 CF 的边,阻止攻击者通过内存中代码细节和堆栈信息实现代码重用攻击. Wang 等[13]将二进制文件分为互斥的代码块,进一步分类间接跳转来执行严格的 CFI 限制. Ge 等[14]针对内核应用提出了一个基本自动化方案,利用软件特点执行全面、高效、细粒度的 CFI. 这些方案进一步提高了完整性校验的强度,但是总体说来开销太大,难以实际部署.

目前,也有一些针对移动平台或嵌入式系统的 CFI 方案. Davi 等[15]首次提出了针对移动设备的 CFI 通用方案 MoCFI,通过将基于 x86 平台的 CFI 移植到 ARM(advanced RISC machines)平台,实现了在 iOS 上的 CFI 保护. Pewny 和 Holz[16]提出了基于编译器的 CFI 方案,该方案对 LLVM(low level virtual machine)编译器进行扩展,在应用编译阶段添加 CFI 执行方案,增强了 CFI 的可用性,并且实现了性能的优化. Tice 等[17]考虑支持增量编译和动态库保护,将 CFI 集成到编译器,实现了 GCC(GNU compiler collection)和 LLVM 上的细粒度、前向 CFI 方案. Payer 等[18]针对嵌入式系统提出了 LockDown,一个模块化、细粒度的 CFI 策略,保护二进制应用和库文件. 但是这些方案大多需要从源码或汇编代码中提取静态 CF,再将完整性验证模块集成到编译器中,并不具有可应用性.

2 针对代码攻击的 CF 特征分析

Android native 代码主要面临2种攻击方式,即关键代码提取攻击(CEA, code extraction attack)和恶意代码植入攻击(CIA, code injection attack). 下面对2种攻击的 CF 特征进行详细分析.

2.1 CEA 攻击 CF 特征分析

CEA 攻击的目标主要有两大类,即代码分析和代码复用. 其中,代码复用攻击的攻击过程会破坏原始模块的 CFI. 在关键代码段中,包含了一个或多个基本块以及多条有向边,原始程序调用关键代码的 CF 边界为一组确定的输入边. 当攻击者复用该段代码后,关键代码段的首个基本块和末尾基本块的有向边将发生变化.

2.2 CIA 攻击 CF 特征分析

在 CIA 攻击中,hook 攻击是典型的动态恶意代码植入攻击方式. 图1给出了基于导入表/导出表这2种 hook 攻击下的 CF 特征.

在基于导入表/导出表的 hook 攻击过程中,攻击者会篡改 ELF(executable and linking format)文件在内存映像中的数据段,将目标函数地址 ADDR2

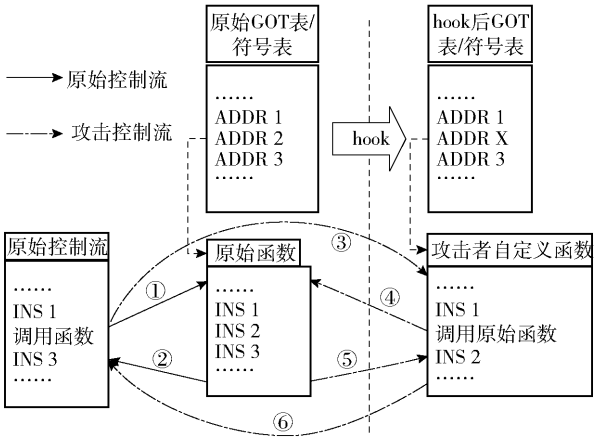

图 1　导入表/导出表 hook 攻击 CF 特征

修改为攻击者的恶意函数地址 ADDRX. 程序执行过程中,会根据篡改后的地址,在执行目标函数前后,分别调用攻击者自定义的代码. 可以明显看出,调用函数和被调用函数的执行 CF 发生了改变.

3　CFI 方案

下面提出一个基于 CFI 的 native 代码加固框架 DroidCFI,该框架以函数边界的 CF 转移特征为加固基本点,相对于传统加固方案更加细粒度,对常见的 native 代码攻击,能够提供有效的防御.

3.1　系统架构

DroidCFI 的系统架构由两部分组成,即加固系统和运行时环境. 其中,加固系统完成对 Android 应用程序的 native 层关键代码的加固处理,包括预处理模块、策略配置模块和 CF 加固模块 3 个子模块;运行时环境完成应用程序 native 代码在运行时的安全保护,主要由 CFI 执行模块构成. DroidCFI 的总体架构如图 2 所示.

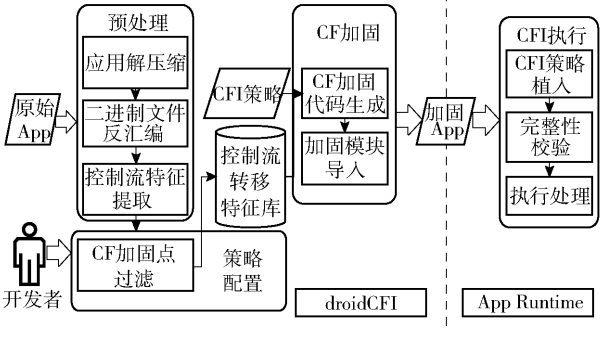

图 2　DroidCFI 总体框架

1）预处理模块. 原始应用进入加固系统后,首先经过预处理模块进行二进制文件提取和反汇编分析,然后进行 CF 特征提取,以收集 CFI 加固需要的信息. 预处理模块会记录程序中所有的函数调用指令以及函数返回指令,同时还会提取这些指令中与 CF 相关的特征,以便后续进行 CF 加固代码的嵌入.

2）策略配置模块. 策略配置模块提供加固过程中面向应用开发者的加固策略配置,允许开发者针对所开发的代码选择关键代码段进行加固. 策略配置模块提供的函数视图基于 IDA(interactive disassembler)的函数交叉参考生成.

3）CF 加固模块. CF 加固模块会将加固代码以动态库的形式集成到原程序中. CF 加固代码基于 CFI 策略和 CF 转移特征库生成,作为预加载模块在原始 native 代码加载之前完成加载. 加固代码总体分为验证代码和载入代码. 验证代码分别针对不同的 CF 转移进行 CFI 验证. 载入代码完成 native 代码和验证代码中的代码动态修改,实现待加固函数的指令改写和验证代码的地址重定位.

4）CFI 执行模块. CFI 执行模块主要负责加固后应用的加固策略执行,包括策略载入和策略执行两部分. 在应用软件启动时,该模块预先加载加固代码,然后在原始 native 代码加载完成后,通过加固代码中的载入代码完成所有 CFI 加固策略的加载. 而在应用软件执行过程中,该模块会通过关键代码的跳转指令,跳转执行加固代码中的 CFI 验证代码,并校验程序的实际跳转地址是否合法.

3.2　关键流程

1）CF 转移表达式

DroidCFI 的核心思想是针对 native 代码关键函数的控制流转移(CFT, control flow transfer)进行控制. 一条 CFT 是由跳转指令所在基本块的出口点与跳转目标的入口点组成的,其通用表达定义为

$$\text{Feature}_{\text{CFT}} = \langle \text{Caller}, \text{Callee}, \text{Type}, \text{Addr}_{\text{caller}}, \text{Addr}_{\text{callee}}, \text{Addr}_{\text{ret}} \rangle$$

其中:Caller 为调用函数名称,Callee 为被调用函数名称. 这 2 个特征均通过解析 ELF 文件的 TEXT 段中的函数符号获得;Type 为调用函数向被调用函数进行跳转时的跳转指令类型;$\text{Addr}_{\text{caller}}$为调用函数在跳转发生时的指令地址;$\text{Addr}_{\text{callee}}$为调用函数跳转至被调用函数的目标地址;$\text{Addr}_{\text{ret}}$为调用过程完成后返回调用函数的指令地址,即 $\text{Addr}_{\text{caller}}$的下一条指令地址.

2）CF 特征提取

DroidCFI 对 ELF 文件的 CF 特征提取分为以下

3步,具体过程如下:

① 函数信息提取. DroidCFI首先会根据ELF文件中的符号表提取所有引用的符号信息,并且结合哈希表和字符串表查询获得所需函数的符号信息. 基于符号表项的st_value字段及st_size字段,获得函数起始地址及长度,进而提取到函数所有机器指令.

② 二进制反汇编. DroidCFI利用IDA对提取函数的机器指令进行反汇编.

③ 指令特征提取. DroidCFI根据函数的首条指令提取LR(link register)寄存器内容,计算调用函数调用时的执行指令地址$Addr_{caller}$;从调用函数执行的地址处提取函数调用指令,获得跳转地址$Addr_{callee}$;根据符号表得到调用函数和被调用函数的名称Caller和Callee;基于寄存器提取返回地址$Addr_{ret}$. 最终获得每一个函数调用时的CFT特征集合〈Caller,Callee,Type,$Addr_{caller}$,$Addr_{callee}$,$Addr_{ret}$〉.

3) CFI加固策略

CFI加固策略定义了执行不同CFT时采取的加固机制,具体分为函数调用和函数返回2种情况.

① 函数调用. ARM指令集中通过跳转指令B(branch)跳转到立即数所指向的地址或者寄存器中数据指向的地址. 进入调用的函数后,程序首先对LR寄存器、FP(frame pointer)寄存器及其他可能必要的通用寄存器进行保存,通过PUSH指令入栈. 在执行被调用函数的功能指令之前,DroidCFI会完成对调用者的验证,以保证函数调用过程的CFI.

② 函数返回. 有别于x86架构,ARM架构下并没有提供专门的返回指令,程序通过跳转指令和修改PC(program counter)寄存器等2种方式实现程序的返回. 为了保证每个函数返回合法调用者的调用点,DroidCFI会验证LR寄存器或堆栈中的返回地址是否为合法地址.

4) CFI策略植入

在加固阶段,DroidCFI会根据提取到的CFT指令集及其特征库,基于CFI加固策略,生成CFI验证代码,每一条CFT对应的验证代码被称为一个Trampoline. 在Trampoline中,会完成4个任务:①备份寄存器,将当前程序状态进行保存;②调用验证函数,为一条函数调用指令,根据不同类型的CFT跳转到不同的验证策略函数,进行CFI验证;③还原寄存器,验证完成后,对初始程序状态进行还原;④跳转指令,执行原始跳转指令,恢复程序运行.

在native代码加载阶段,预处理代码会首先执行,利用上述Trampoline对内存中的二进制文件映像进行重写. 原始二进制文件中每个需要加固的函数入口与出口位置的指令会被替换为一条跳转到Trampoline的指令,从而将验证代码以插桩的形式植入到二进制内存映像中. 具体策略植入参见图3. 发生函数调用CFT时(见图3中的实线),程序通过BLX(branch with link exchange)指令跳转到子程序,子程序首先通过PUSH指令将需要保存的寄存器值入栈,入栈完成后,下一条MOV指令会被一个Trampoline替换,进行函数的入口验证. 发生函数返回CFT时(见图3中的虚线),程序将通过BX(branch exchange)指令跳转到LR寄存器保存的返回地址位置,此条BX指令被一个Trampoline替换,进行函数的出口验证.

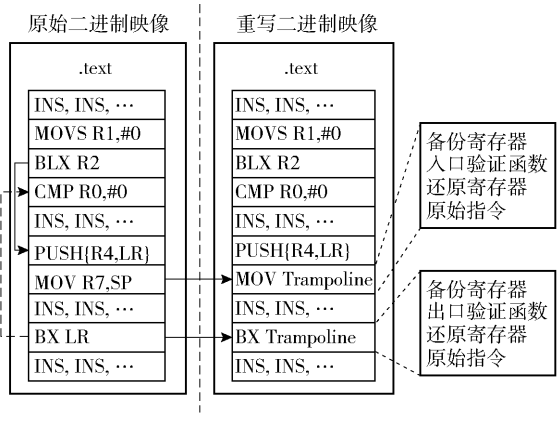

图3 控制流转移

5) CFI加固执行

在程序运行过程中,DroidCFI会调用CFI执行模块完成对程序CFT的监控及完整性校验. 程序执行到CF的加固点时,会通过插桩的跳转指令进入Trampoline代码段,由验证函数执行CFT完整性的校验,具体的验证过程如下:验证函数首先基于当前指令上下文确定验证类型. 对于函数调用的验证,DroidCFI会对比预生成的CFT地址白名单,判断该地址对是否在白名单中;而对于函数返回,DroidCFI则以地址对〈$Addr_{caller}$,$Addr_{callee}$〉为键值查询CFT白名单中的返回值,并与返回值特征$Addr_{ret}$对比. 对比后的验证结果会传递给执行处理子模块,以判断继续执行还是终止运行程序.

4 实验与评估

下面对DroidCFI的安全性和性能进行评估分

析，评估方法主要通过开源项目 VirtualApp 建立虚拟运行环境，加载目标应用、DroidCFI 加固模块以及攻击代码. 对于 ELF 文件的分析和特征提取，PC (personal computer)的验证环境配置是 CPU Intel i7-6500u，内存为 8 GB，操作系统为 Microsoft Windows 10(64)，对于应用软件和加固模块的执行，手机的验证环境是 CPU 骁龙 800(2.3 GHz，32-bit)，内存为 2 GB，操作系统为 Android-5.1.1_r1 和 Android-7.1.1.

4.1　安全性评估

安全性评估重点评估 DroidCFI 对于 CEA 攻击和 CIA 攻击的防御能力. 首先，定义这两类攻击的 4 种典型的攻击场景：

1) 场景Ⅰ. 模拟 CEA 攻击，将攻击代码注入被攻击应用的进程，并通过攻击代码调用执行被攻击应用 native 代码中的目标函数.

2) 场景Ⅱ. 模拟 CIA 攻击中的导入表 hook 攻击，通过攻击程序修改被攻击应用 native 代码 GOT (global offset table)表中的目标函数地址，实现恶意代码注入.

3) 场景Ⅲ. 模拟 CIA 攻击中的导出表 hook 攻击，通过攻击程序修改被攻击应用的 native 代码进行链接时的目标函数地址，使用恶意代码注入.

4) 场景Ⅳ. 模拟 CIA 攻击中的 inline hook 攻击，通过攻击程序替换被攻击应用的 native 代码的目标函数指令，实现恶意代码注入.

测试应用从应用宝应用市场上下载，包括高德地图、酷狗音乐、美图秀秀、OFO、Quik，并从这些 App 中随机选择 native 代码的动态库，针对上述 4 种场景，让应用程序在普通环境和 DroidCFI 环境下分别进行实验. 实验结果显示，5 个目标应用运行在普通环境中，4 种攻击场景均能完成攻击，而在 DroidCFI 的安全环境下，4 种攻击都被成功防御. 通过实验可以看出：① 应用软件对于 CEA 攻击和 CIA 攻击基本不具备防御能力；② DroidCFI 基本达到了方案的设计目标，能够有效抵御 CEA 攻击和 CIA 攻击的 4 种典型攻击场景.

4.2　性能评估

性能评估主要针对应用程序在 DroidCFI 保护情况下的运行效率进行评估，包括分析应用程序的 CPU/内存的性能开销以及应用程序在加固模块加载过程和执行过程两阶段的时间性能开销.

在 CPU 和内存的性能评估中，从应用宝应用市场分别选择了涉及通信社交、影音试听、新闻阅读、生活休闲、地图旅游这五大类别总计 16 个应用 App，利用 Emmagee 分别在 Android-5.1.1_r1 系统下的普通环境和 DroidCFI 中测试其 CPU 和内存占用情况，并计算差值，部分测试结果如表 1、表 2 所示.

表 1　CPU 占用差值　　%

应用	运行时间/s					
	1	2	3	4	5	6
抖音	0	5	24	-3	29	16
大众点评	-13	15	-5	-30	-12	-27
花椒	0	-15	-29	-47	-5	12
网易云音乐	0	16	32	10	-6	8
酷狗音乐	0	-1	-18	-2	0	-13
高德地图	0	6	13	15	1	2
OFO	0	0	1	6	8	5
美图秀秀	0	-9	-20	-26	-9	4

表 2　内存占用差值　　MB

应用	运行时间/s						
	1	2	3	4	5	6	7
抖音	-41	-42	-38	-8	-17	-9	-1
大众点评	-6	32	47	53	51	48	46
花椒	7	7	-17	-116	-10	36	45
网易云音乐	5	-1	-8	6	-14	1	-37
酷狗音乐	79	-2	31	-38	-20	-6	21
高德地图	-11	58	59	67	16	17	16
OFO	21	22	23	23	31	31	31
美图秀秀	52	21	-13	-54	-12	-13	-1

从测试结果可以看出，被保护的应用 App 在 DroidCFI 保护下的 CPU/内存的性能开销与其在普通环境下执行过程的性能开销，差别并不明显，由于环境因素的影响，呈上下抖动的形态. 从实现原理来说，DroidCFI 对 CPU 的占用，主要来自于加固点的策略执行，呈现随机性. DroidCFI 对内存的占用，与加固点的数量呈线性增长关系. 但是单位加固点的缓存信息的内存占有量并不大，因此也很难造成显著的内存增益.

此外，针对上述 16 个应用程序在 Android-7.1.1 进行了加固模块时间性能开销的测试. 每个应用随机选取了 1 ~ 10 个函数进行加固，然后分别进行了 10 次测试并计算时间开销的平均值，测试结果如图 4 所示. 可以看到，单位函数加固处理的绝对时间的开销都在 10 ms 以内，因此对于 App 运行的时间性能影响是微乎其微的.

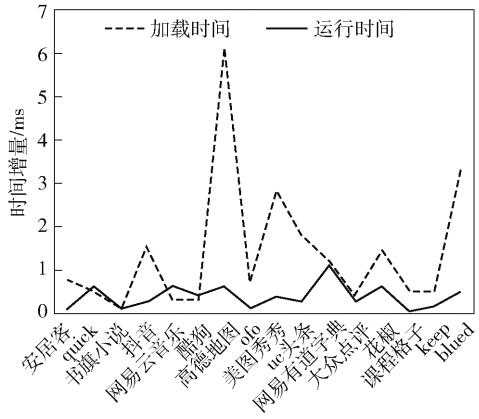

图4 加固模块时间性能开销

4.3 方案对比

在目前已有的针对移动平台的CFI方案中,大部分都是基于编译器的,因此需要被保护的应用提供源代码支持,仅有DroidCFI和MoCFI是针对二进制可执行文件直接提供保护. 2种方案都是针对ARM架构,其存在的差异包括:1) MoCFI主要针对iOS系统的应用程序;而DroidCFI主要针对Android系统的应用程序. 2) MoCFI主要考虑的攻击场景是针对应用软件的ROP攻击方式;而DroidCFI主要考虑的攻击场景是针对Android应用软件native代码的CEA攻击和CIA攻击. 3) MoCFI主要采用了基于程序基本块的CFI方案,面临严重的性能损耗;而DroidCFI将防御粒度上升到关键函数级进行防御,对应用软件的性能影响较小. 4) MoCFI的原型系统在部署时,需要较为苛刻的条件,而DroidCFI能够通过虚拟环境加载执行、软件重打包等方式,灵活植入CFI加固模块.

5 结束语

通过研究CEA攻击和CIA攻击对Android应用软件native代码CF的影响,提出了通过CFI对native代码的保护方案,并且实现了原型系统DroidCFI. 目前,DroidCFI可以通过重打包、虚拟环境加载等形式部署到目标App中,但还面临人工干预环节较多的问题. 为此,后续工作将以DroidCFI为基础,设计针对App的安全容器,为App提供能够动态部署安全策略、并实施安全代码防护的运行环境.

参考文献:

[1] Abadi M, Budiu M, Erlingsson U, et al. A theory of secure control flow[C]// International Conference on Formal Engineering Methods and Software Engineering. Berlin: Springer, 2005: 111-124.

[2] Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity[C]// Proceedings of the 12th ACM Conference on Computer and Communications Security. New York: ACM, 2005: 340-353.

[3] Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity principles, implementations, and applications[J]. ACM Transactions on Information and System Security, 2009, 13(1): 4.

[4] Zhang Chao, Wei Tao, Chen Zhaofeng, et al. Practical control flow integrity and randomization for binary executables[C]// IEEE Symposium on Security and Privacy (SP). New York: IEEE Press, 2013: 559-573.

[5] Zhang Mingwei, Sekar R. Control flow integrity for COTS binaries[C]// Proceedings of USENIX Conference on Security. Berkeley: USENIX Association, 2013: 337-352.

[6] Zhang Mingwei, Sekar R. Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks[C]// Proceedings of the 31st Annual Computer Security Applications Conference. New York: ACM, 2015: 91-100.

[7] Göktas E, Athanasopoulos E, Bos H, et al. Out of control: Overcoming control-flow integrity[C]// IEEE Symposium on Security and Privacy (SP). New York: IEEE Press, 2014: 575-589.

[8] Mashtizadeh A J, Bittau A, Boneh D, et al. CCFI: cryptographically enforced control flow integrity[C]// 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS). New York: ACM, 2015: 941-951.

[9] Niu Ben, Tan Gang. RockJIT: securing just-in-time compilation using modular control-flow integrity[C]// Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2014: 1317-1328.

[10] Criswell J, Dautenhahn N, Adve V. KCoFI: complete control-flow integrity for commodity operating system kernels[C]// IEEE Symposium on Security and Privacy (SP). New York: IEEE Press, 2014: 292-307.

[11] van der Veen V, Andriesse D, Göktaş E, et al. Practical context-sensitive CFI[C]// Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2015: 927-940.

[12] Mohan V, Larsen P, Brunthaler S, et al. Opaque control-flow integrity[C]// Proc of the 22nd Network and Distributed System Security Symposium. Washington, DC: Internet Society, 2015.

(下转第13页)

[13] Sermanet P, Eigen D, LeCun Y, et al. Overfeat: integrated recognition, localization and detection using convolutional networks[C]//International Conference on Learning Representations. Banff: arXiv preprint, 2013: 1312, 6229.

[14] Hochreiter S, Schmidhuber J. Long short-term memory [J]. Neural Computation, 1997, 9(8): 1735-1780.

[15] Oriol V, Alexander T, Samy B, et al. Show and tell: a neural image caption generator[C]//Computer Vision and Pattern Recognition. Boston: IEEE press, 2015: 3156-3164.

[16] Kelvin X, Jimmy L B, Ryan K, et al. Show, attend and tell: neural image caption generation with visual attention[C]//Proceeding of the 32nd International Conference on Machine Learning. France: ACM press, 2015: 2048-2057.

[17] Wu Q, Shen C S H, Liu L Q, et al. What value do explicit high level concepts have in vision to language problems[C]//Computer Vision and Pattern Recognition. Lasa Vegas: IEEE press, 2016: 203-212.

[18] Andrei K, Li F F. Deep visual-semantic alignments for generating image descriptions[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2017, 39(4): 664-676.

[19] Rashtchian C, Young P, Hodosh M, et al. Collecting image annotations using amazon's mechanical turk[C]// In NAACL HLT Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk. California: ACM press, 2010: 139-147.

[20] Kishore P, Salim R, Todd W, et al. Bleu: a method for automatic evaluation of machine translation[C]//Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. Philadelphia: ACM press, 2002: 311-318.

(上接第 6 页)

[13] Wang Minghua, Yin Heng, Bhaskar A V, et al. Binary code continent: finer-grained control flow integrity for stripped binaries[C]//Proceedings of the 31st Annual Computer Security Applications Conference. New York: ACM, 2015: 331-340.

[14] Ge Xinyang, Talele N, Payer M, et al. Fine-grained control-flow integrity for kernel software[C]//IEEE European Symposium on Security and Privacy (EuroS&P). New York: IEEE Press, 2016: 179-194.

[15] Davi L, Dmitrienko A, Egele M, et al. MoCFI: a framework to mitigate control-flow attacks on smartphones[C]//Annual Network and Distributed System Security Symposium, San Diego, February 2012.

[16] Pewny J, Holz T. Control-flow restrictor: compiler-based CFI for iOS[C]//Proceedings of the 29th Annual Computer Security Applications Conference. New York: ACM, 2013: 309-318.

[17] Tice C, Roeder T, Collingbourne P, et al. Enforcing forward-edge control-flow integrity in GCC & LLVM[C]//Proceedings of USENIX Conference on Security. Berkeley: USENIX Association, 2014, 26: 27-40.

[18] Payer M, Barresi A, Gross T R. Fine-grained control-flow integrity through binary hardening[C]//International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Berlin: Springer, 2015: 144-164.