软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2022,33(5):1833−1848 [doi: 10.13328/j.cnki.jos.006197]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于回调函数的控制流深度模糊模型*

沙子涵[1], 舒 辉[1], 武成岗[2], 熊小兵[1], 康 绯[1]

[1](数学工程与先进计算国家重点实验室, 河南 郑州 450001)

[2](中国科学院 计算技术研究所, 北京 100190)

通信作者: 舒辉, E-mail: shuhui123@126.com

摘 要: 控制流是程序过程的抽象表现, 对控制流进行混淆, 可有效提高代码抗逆向能力. 提出了控制流深度模糊思想:针对循环结构, 利用回调函数构造等价循环模型, 将过程内基本块跳转变更为过程间函数调用, 对抗逆向技术. 综合应用控制流分析和数据流依赖性分析, 建立了基于回调函数的控制流深度模糊模型, 并给出功能一致性证明. 为进一步增大混淆强度, 设计并实现了函数调用融合算法, 构造更为复杂的函数调用过程. 最后, 使用 OpenSSL 和 SpecInt-2000 标准测试套件作为测试集, 验证了模型的可行性和有效性.

关键词: 深度模糊; 回调函数; 循环结构; 数据流依赖; 代码变换

中图法分类号: TP311

中文引用格式: 沙子涵, 舒辉, 武成岗, 熊小兵, 康绯. 基于回调函数的控制流深度模糊模型. 软件学报, 2022, 33(5): 1833–1848. http://www.jos.org.cn/1000-9825/6197.htm

英文引用格式: Sha ZH, Shu H, Wu CG, Xiong XB, Kang F. Deep Control Flow Obfuscation Model Based on Callback Function. Ruan Jian Xue Bao/Journal of Software, 2022, 33(5): 1833–1848 (in Chinese). http://www.jos.org.cn/1000-9825/6197.htm

Deep Control Flow Obfuscation Model Based on Callback Function

SHA Zi-Han[1], SHU Hui[1], WU Cheng-Gang[2], XIONG Xiao-Bing[1], KANG Fei[1]

[1](State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China)

[2](Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: Control flow is an abstract expression of the program process, and it is of critical significance to obfuscate the control flow to effectively reinforce the code's ability to resist reverse manners. This study proposes the idea of deep control flow: as for the loop structure, the callback function is utilized to construct an equivalent loop model, and the basic block in the program process is converted into inter-process function calling to counter reverse technology. This study comprehensively applies control flow analysis and data flow dependency analysis to establish a deep control flow obfuscation model based on callback function and gives proof of functional consistency. To further enhance obfuscation, the function calling fusion algorithm is designed and implemented pertinently to construct a more sophisticated function calling process. Finally, OpenSSL and SPECint-2000 benchmark suite is used as the test set to verify the feasibility and effectiveness of the proposed model.

Key words: deep control flow; callback function; loop structure; data flow dependency; code transformation

软件保护是网络安全的重要组成部分, 在软件保护领域, 一个关键性的研究方向在于构造有效的混淆算法. Collberg 等人[1]将代码混淆技术分为 4 类:布局混淆、数据流混淆、控制流混淆和预防性混淆. 其中, 对控制流混淆算法的研究相对较多. Blazy 等人[2]提出了平展控制流混淆算法, 将程序中原有的嵌套循环和条件转移语句平展开, 通过 switch 语句进行连接, 破坏原始控制流结构. 但其对流相关信息的隐藏不够彻底, 算法具有可逆性[3]. Sharif 等人[4]提出了条件混淆算法, 用哈希值等价替代路径分支条件中的常量, 使该常量从原程序中剥离, 并利用哈希值加密代码块以增加逆向工程的难度. 但对于特定程序, 条件取值集合有限, 因此该算法存在被暴力猜解攻破

* 基金项目: 国家重点研发计划 (2016YFB08011601)
收稿时间: 2020-04-07; 修改时间: 2020-06-04, 2020-10-15; 采用时间: 2020-10-28; jos 在线出版时间: 2021-11-24

的可能. 陈喆等人[5]提出了基于随机森林的路径分支混淆方法, 将逆向分析路径的难度等价于抽取随机森林规则的难度, 防止分支信息泄露. 但该算法引入大量冗余计算, 影响程序运行效率. Popov 等人[6]提出了一种基于 Linux 信号机制的二进制代码混淆技术, 使用各种异常信号代码替换程序中的跳转指令, 对抗静态逆向分析工具. 但该技术无法对二进制代码的路径分支进行混淆, 因此不能解决软件执行过程中的信息泄露问题.

循环结构包含大量算法逻辑相关的有效信息, 对循环结构进行混淆, 可有效提高代码保护能力. 本文在相关研究的基础上提出了控制流深度模糊思想, 即: 利用回调函数, 将显式的循环执行转为隐式的函数调用, 隐藏循环相关的路径分支信息. 传统循环通过过程内基本块跳转实现循环功能;而对于回调型循环, 循环相关代码处于不同函数内, 通过过程间函数调用关系实现循环功能. 本文将这种控制转移方式称为控制流的深度. 深度模糊就在于利用过程间函数调用, 灵活替换传统循环, 隐藏原始控制流, 对抗逆向分析. 围绕这一概念, 本文进一步提出了函数嵌套融合算法, 使循环调用由相邻过程间转移至随机过程间, 强化模型的深度特征.

本文的主要贡献在于:

(1) 从控制流和数据流的角度出发, 论证回调型循环与传统循环逻辑等价, 并给出了等价变换方式;

(2) 设计函数调用融合算法, 通过随机添加冗余函数调用, 构造复杂的控制流过程, 提高生成程序复杂度;

(3) 混淆系统实现: 针对两类循环结构进行局部优化, 确保模型应用前后功能一致, 并通过实验验证了系统的可行性与抗逆向分析的效果.

1 相关工作

控制流混淆的思路主要有两种: (1) 破坏原始程序控制流结构; (2) 采用加密等手段保护跳转信息. 前者以控制流扁平化算法为代表, 对分支结构进行等价变换. 在一定程度上隐藏路径信息, 但不能做到完全消除, 因此具有可逆性. 后者以条件混淆算法、分类器混淆算法等[7,8]为代表, 通过不可逆算法, 实现跳转相关信息的彻底消除. 但此类算法应用面相对狭窄, 如条件混淆算法仅对相等条件进行处理, 且要求程序和用户存在交互过程, 因此难以做到广泛应用.

总体而言, 当前代码混淆主要围绕过程内控制流展开, 代码等价执行过程由程序本身实现. 因此, 本文考虑在传统混淆算法的基础上实施改变, 使混淆过程由过程内扩展至过程间, 控制流转移变为程序与系统协同进行. 通过异常处理函数实施控制流混淆是这一思想的一种实现方式, Popov 等人[6], Lin 等人[9]和贾春福等人[10]在异常处理的基础上, 分别提出了自己的分支混淆方法. 通过异常触发, 将控制权转移到预置的异常处理函数, 由系统负责跳转执行, 隐藏了条件跳转指令, 实现对控制流的混淆.

然而, 异常处理机制的复杂性导致控制流转移过程不完全可控, 且容易引起程序安全问题. 而函数回调过程相对简单, 执行流可预测. 因此, 本文提出了控制流深度模糊思想, 通过系统回调函数, 将循环结构分离至不同控制流执行过程, 同时将控制流转移过程交付系统执行, 实现对程序路径信息的隐藏, 对抗逆向分析.

2 回调型循环结构

回调型循环通过回调函数“响应-执行”机制, 将基本块循环跳转变换为函数间的反复调用. 以 *EnumFonts* 函数为例, 该函数接受其他函数指针作为参数, 执行 Windows 字体库遍历操作. 对其中每个字体均调用回调函数处理, 字体库遍历过程即等价替代循环过程.

如图 1 所示, 多数系统回调函数均具有如下机制: 当返回值非 0 时, 回调过程继续;当返回值为 0 时, 回调过程结束, 程序继续运行. 该机制也是实现循环控制的重要基础.

显然, 传统循环转变为回调循环, 代码被剥离至不同函数体内, 将面临以下问题.

(1) 控制转移过程是否等价, 对复杂循环结构是否具有适用性;

(2) 如何维持变量作用域和数据依赖关系, 保证功能一致性.

要解决上述问题, 首先应选取适当的回调函数, 本文选取的回调函数均具有以下特点.

(1) 即时响应性: 执行调用者函数后, 立即触发响应事件, 调用回调函数;

(2) 反复回调性: 程序运行时, 循环次数不可控. 故在不触发退出条件的情况下, 调用者反复调用回调函数;

(3) 特定退出条件: 对应循环边界条件, 提供相应的退出机制, 以结束回调过程, 继续执行程序的其他功能.

Windows 操作系统中存在大量回调函数, 包括消息传递、字符处理等. 按照上述规则, 本文初步选取 100 个符合条件的回调函数作为模型的基础. 下文将围绕控制转移过程的等价性和数据依赖关系的一致性展开论述, 证明模型的可行性.

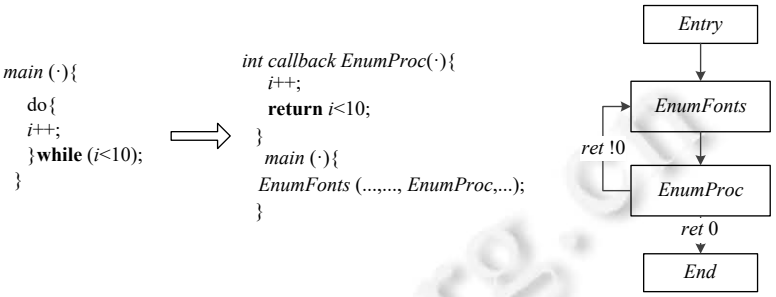


图 1 回调型循环代码示例与执行过程

3 两类循环等价性证明

控制流和数据流是进行程序分析的主要依据, 本文首先通过支配关系分析准确定义循环结构, 说明两类循环控制流等价; 其次, 利用跨过程别名分析, 证明代码变换前后数据流一致. 并给出相应的代码变换策略, 证明两类循环具有等价性.

3.1 传统循环结构分析

对循环结构的分析一般以基本块为单位, 构造控制流程图 $G=\{N,E\}$, 其中, N 为程序中基本块节点的集合, E 为基本块间边的集合. 在此基础上, 给出有关形式化定义如下.

定义 3.1. 若从程序入口点到 N_j 的任意路径一定经过 N_i, 则 N_i 对 N_j 构成支配关系, 记为 N_i DOM N_j.

定义 3.2. 若边的目标节点对起始节点构成支配关系, 则称该边为回向边.

定义 3.3. 若控制流图中存在具有回向边的子图, 则该子图对应循环结构. 称回向边的目标节点为循环头, 回向边的起始节点为循环尾, 循环头到循环尾的执行路径为循环体.

为进一步分析循环结构, 分别建立支配关系矩阵 A 和节点边矩阵 B. 若 N_i DOM N_j, 则 A_{ij}=1;若存在边 $N_i \rightarrow N_j$, 则 B_{ij}=1; 否则均为 0. 由于回向边路径方向与支配关系对称, 因此对于回向边 $N_i \rightarrow N_j$, 一定有 $A_{ij}^{\mathrm{T}} = B_{ij}$. 该式可用于循环的发现与判定.

定义 3.4. 以 N_i 为目标节点的边的数量称为节点 N_i 的入度, 记作 $N_{i\text{-}in}$. 以 N_i 为目标节点的边的数量称为节点 N_i 的出度, 记作 $N_{i\text{-}out}$, 其中, $N_{i\text{-}out} \leqslant 2$ 恒成立.

结合边关系矩阵, 有关基本块的入度与出度有如下计算公式:

$$N_{i\text{-}in} = \sum_{j=1}^{n} B_{ji} \tag{1}$$

$$N_{i\text{-}out} = \sum_{j=1}^{n} B_{ij} \tag{2}$$

基本块的出度可以作为标识循环类型的重要依据, 参照 C 语言代码风格, 循环分为执行优先循环 (do-while), 和判断优先循环 (while-do, for). 根据出度的不同, 判定方式如下.

推论 3.1(循环类型由循环头的出度判定). 执行优先循环出度为 1, 判断优先循环出度为 2.

针对不同的循环类型, 代码变换策略有所区别, 具体处理方式将在第 3 节中介绍. 下文将以执行优先循环为对象, 证明两类循环间存在控制流等价关系, 并对特殊循环结构和复杂控制流进行讨论.

3.2 控制流等价性证明

回调型循环以函数调用的方式实现, 传统循环以基本块间跳转的方式实现, 两者不完全相同, 但在可见行为上一致, 符合代码混淆的要求. 因此, 对传统循环结构的分析方法同样适用回调型循环.

回向边是标识循环结构的主要依据, 参照图 1 示例, 回调函数 F_{cb} 必须经由调用者 F_c 调用执行, 因此有 F_c DOM F_{cb}. 当返回值非 0 时, 存在跳转路径 $F_{cb}\rightarrow F_c$, 显然构成回向边, 满足循环定义. 此时, 调用者函数成为循环头, 回调函数成为循环尾. 等价变换模式如图 2 所示.

在此基础上, 针对循环结构中标识控制流转移关系的控制流关键字 continue 和 break 进行分析: continue 使控制流由循环体内跳转至循环尾, 属于循环内跳转, 不需额外处理; break 使控制流由循环体跳转至循环出口点, 当回调函数返回值为 0 时, 回调过程结束, 程序继续运行. 两类行为逻辑相同, 可以进行等价替换.

最后, 对实际编程中存在的特殊循环结构进行分类讨论, 判定代码变换的可行性及应用的变换方式, 如图 3 所示.

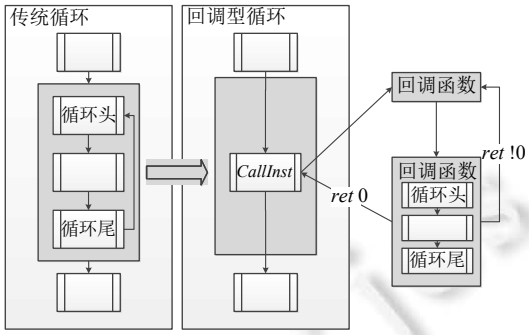


图 2 两类循环等价执行流程

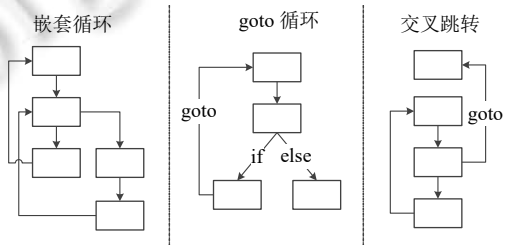


图 3 特殊循环结构

本文共总结 3 类特殊循环结构.

(1) 嵌套循环, 是指循环头属于其他循环循环体的控制结构. 对此类循环进行代码变换, 应按照支配关系依次进行. 若存在回向边 $N_i\rightarrow N_j$ 和 $N_p\rightarrow N_q$ 且 N_j DOM N_q, 先对循环头 N_q 进行代码变换, 该循环头标识的循环结构被隐藏, 嵌套循环退化为简单循环结构, 继续对循环头 N_j 进行处理可完成变换.

(2) goto 循环, 指通过跳转关键字 goto 等价形成的循环功能, 该类循环具有回向边, 满足循环定义, 在控制流图层面与传统循环结构完全相同, 因此可等价处理.

(3) 交叉跳转, 指通过跳转关键字 goto 在循环内外进行任意跳转, 这种跳转行为只存在于函数内部, 不适用于回调型循环跨函数间跳转过程, 且违反程序可规约性, 因此对该种结构不进行处理, 维持原程序.

综上所述, 利用回调相关机制形成的循环结构与传统循环在控制流层面等价, 且对特殊循环结构具有处理能力. 下文将对数据流进行分析, 证明两类循环具有数据流等价性.

3.3 数据流依赖的发现与重建

回调型循环将循环代码块由原函数转移至回调函数, 代码的分割将会破坏数据依赖关系. 因此需对程序中数据流进行分析, 发现有关依赖关系并完成重建.

定义 3.5. 基本块由指令组成, 设 Ins_i 为基本块中的一条指令:

$$Ref(Ins)=\{Q|Q\text{为执行}Ins\text{时引用的变量}V\text{的集合}\}$$

$$Def(Ins)=\{V|V\text{为执行}Ins\text{时作为定义性出现的变量}\}$$

需要说明的是: 本文实施静态程序分析的对象是 LLVM 中间过程, 由于 LLVM 中间过程采用静态单赋值形式, 因此对任意变量 V, 有且仅有唯一的 $Def(Ins)=V$.

定义 3.6. 若 $Def(Ins_i)=V$ 且 $V\in Ref(Ins_j)$, 则有 Ins_j 依赖于 Ins_i, 记作:

$$Dep(Ins_i,Ins_j)=1.$$

推论 3.2. 数据依赖性具有传递性:

$$Dep(Ins_i, Ins_j) = 1 Dep(Ins_j, Ins_k) = 1 Dep(Ins_i, Ins_k) = 1.$$

推论 3.3. 若 *Ref*(*Ins*)=Ø, 称 *Ins* 为该数据流的起始顶点.

为实现数据依赖关系的发现, 综合上述定义与推论, 本文实现了基于深度优先搜索的数据依赖发现算法. 首先对循环内指令的依赖数据进行迭代扫描, 如果发现数据流起始顶点, 则返回上层继续扫描; 如果发现存在对循环外部指令的依赖, 则记录该依赖数据并返回. 算法描述如算法 1.

算法 1. 数据依赖发现算法.

输入: 指令 *Ins*;

输出: 指令循环外依赖数据集合 *DepSet*.

```
begin
1 Array<Ins *>DepSet
2 procedure GetDepData(Ins)
3     if Ins not in Loop then do
4         Add(DepSet,Def(Ins)) /*记录循环外依赖数据*/
5         Return
6     if Ref(Ins)==null then do
7         Return /*递归边界条件*/
8     end if
9     foreach Ins in Ref(Ins)
10        GetDepData(Ins)
11    end foreach
12 end procedure
13 GetDepData(Ins)
end
```

应用该算法, 可有效挖掘原程序中存在的数据依赖关系, 并为依赖关系重建提供基础. 本文采用的重建方法是通过构建依赖数据的别名指针, 实现对原始变量地址的读取和写入, 构建过程如下.

首先, 将依赖数据地址依次入栈, 使栈中数据与依赖数据构成别名; 再按照对应关系, 完成对循环内引用变量的替换, 实现数据传递. 下文将进一步对该方法进行分析, 证明应用前后数据流等价.

3.4 数据流等价性证明

通过指针传参, 程序可以在回调函数中访问原始变量, 此时, 对同一存储地址产生多条访问路径. 按照程序分析相关理论, 如果两个访问路径在指令 Ins_i 每次执行时都指向相同的存储地址, 则称这些访问路径在 Ins_i 处是必然别名. 如果访问路径在指令 Ins_i 的某些次执行中指向相同的存储地址, 则构成可能别名.

在本文模型中, 如果栈指针与原始变量在执行过程中构成必然别名, 则证明程序变换前后数据流等价, 具有数据一致性. 为完成这一证明, 首先给出别名分析的相关形式化定义.

定义 3.7. 若变量*p 和 q 引用同样的存储地址, 则互为别名, 记作<*p,q>.

定义 3.8. 若指令 Ins_i 在执行时产生新的别名关系, 记为集合 Gen_i;若破坏已有别名关系, 记为集合 $Kill_i$;Ins_i 在执行前由上条指令继承的别名关系, 记为集合 In_i;Ins_i 执行后, 程序中有效的别名关系记为 Out_i.

推论 3.4. 指令执行时别名集合具有以下关系:

$$Out_i = Gen_i(In_i Kill_i).$$

回调函数在执行时, 由调用者函数接受参数指针, 此时, 栈指针 s 与原始变量 v 指向同一存储区域. 因此, 在函

数入口处有别名关系<*s,v>. 即: 对于入口点指令 Ins_0, 有 In_0={<*s,v>}. 程序运行时不会主动破坏该别名关系, 因此对于所有指令, 均有 In={<*s,v>}.

应用推论 3.4 对回调函数中所有指令执行时别名状态集进行分析, 根据不动点理论, 对任意指令可解出确定的状态集合, 且集合中始终包含{<*s,v>}. 因此, 栈指针 s 与原始变量 v 在回调函数中构成必然别名, 程序变换前后数据流等价.

综上所述, 基于回调函数的循环模型在控制流和数据流上均可与传统循环形成等价, 因此, 两类循环间存在可替代关系.

4 混淆系统的构建与评估

LLVM 框架是可扩展的程序优化平台, 提供大量 API 用于分析和修改中间语言代码[11]. 本文在 LLVM 的基础上实现混淆系统, 进行自动化代码变换. 在系统实际构建过程中, 存在两点问题: (1) 回调事件触发时, 首先调用回调函数执行, 再进行边界判定, 因此不适用于判断优先型循环; (2) 仅对循环代码块进行替换, 转换模式存在被识别的可能. 针对第 (1) 个问题, 本文通过引入冗余代码块, 解决系统适用性问题; 对于第 (2) 个问题, 本文设计了函数调用融合算法, 实现混淆强度的提升.

混淆系统执行过程共分 4 个阶段: 第 1 阶段, 以 C/C++编写的源程序作为输入, Clang 将源程序转换为 LLVM 的中间表示; 第 2 阶段, 对中间语言进行分析, 挖掘循环代码块与数据依赖关系; 第 3 阶段, 引入回调函数并进行代码重组, 完成功能一致性处理与混淆强度提升; 第 4 阶段, 生成目标平台可执行文件.

本节将对第 3 阶段功能一致性处理与混淆强度提升进行具体讨论, 并对最终建立的系统进行科学评估.

4.1 两类循环重组与边界设定

循环入口点的出度是标识循环类型的主要依据, 对于不同类型的循环, 由于判断和执行的优先级不同, 标识跳转条件的比较指令位置也有所差异: 判断优先型循环中, 跳转条件位于循环头; 执行优先型循环中, 跳转条件位于循环尾. LLVM 通过比较指令 *CmpInst* 完成判断功能, 并作为循环结构中的跳转条件出现. 前文已知, 回调函数以返回值是否为 0 作为回调边界条件, 因此在定位 *CmpInst* 后, 对该指令进行布尔值转化, 在回调函数末尾添加 *RetInst* 返回该值, 完成边界条件设定.

然而, 在具体应用中, 两类循环的等价转换方式有所区别: 执行优先型在程序运行后立即进入循环结构, 与函数调用逻辑相符; 判断优先型先进行循环边界检测再确定是否进入循环结构. 因此, 针对后者额外添加跳转条件代码块, 在回调函数首次触发时, 执行该基本块进行入口判断, 通过代码冗余保持原程序执行逻辑不变. 具体形式如图 4 所示.

4.2 基于函数调用融合的可伸缩回调模型

控制流深度模糊模型利用回调函数, 将循环逻辑隐藏于不同函数的调用过程中, 从而对抗逆向分析. 为了进一步强调深度化的模型特征, 本文在代码变换的基础上构造了一种具有深度可伸缩的回调模型, 即按照用户需求, 随机融入相应量级的冗余回调函数, 形成更为复杂的调用关系, 提高混淆强度.

函数调用过程在形式上类似于基于深度优先的多叉树节点遍历, 因此, 实施多叉树融合将是函数调用融合的抽象解决办法: 假设对于任意函数 $Func_i$, 调用了 $Func_j$ 和 $Func_k$. 对程序进行完整扫描, 生成描述函数调用关系的多叉树, 假设该树叶子节点为 n, 对所有叶子节点进行码长为 n 的编码, 有 $Func_j$=01,$Func_k$=10. 父亲节点的码值为该节点所有子节点的或运算结果, 即 $Func_i$=$Func_j$|$Func_k$=11, 生成编码树 A 如图 5 所示.

记该树中所有节点的集合为 $Nodes_A$={$Node_{A\_i}$,$Node_{A\_j}$,$Node_{A\_k}$}, 节点的子节点数为该节点的度, 节点中, 度的最大值为 *MaxNode*. 在编码树 A 的基础上, 构造目标融合树 B, 该树需满足 4 个条件: (1) 完备多叉树; (2) 各节点度=*MaxNode*; 3) 叶子节点数 m>n; 4) 树 B 层数大于数 A. 此处取 m=4, 生成目标融合树 B, 树 B 中的节点由冗余函数调用组成, 并同时对该树进行如图 6 所示的编码.

显然, 将树 A 与树 B 融合, 有 $C_m^n = C_4^2 = 6$ 种不同的融合路径. 设存在融合函数集合:

$$F = F_1, F_2, F_3, F_4, F_5, F_6.$$

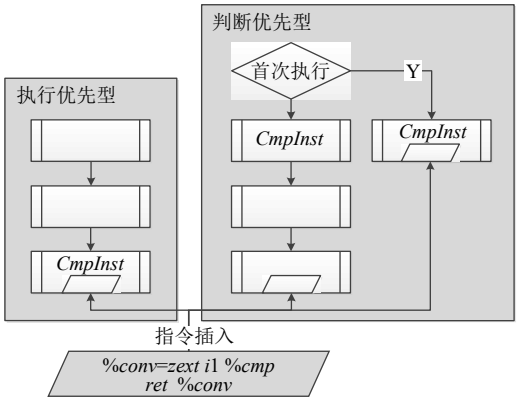


图 4 两类循环回调边界设定

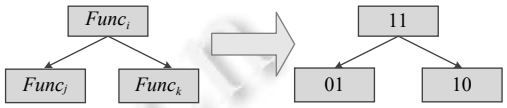


图 5 编码树 *A*

F_i 的功能为在长度 m 的编码中固定选取 n 位, 若有 $Node_{A\_i}=F_1(Node_{B\_i})$, 则编码为 $Node_{A\_i}$ 的函数节点可融合至 $Node_{B\_i}$ 中执行. 显然, 节点融合可能为一对一或一对多关系. 若 $Node_{A\_i}=F_1(Node_{B\_i})=F_1(Node_{B\_j})$, 以上文编码树 *A* 和树 *B* 为例, 取 F_1 为 $Node_{A\_i}$[3:4], 则有如图 7 所示的融合方式.

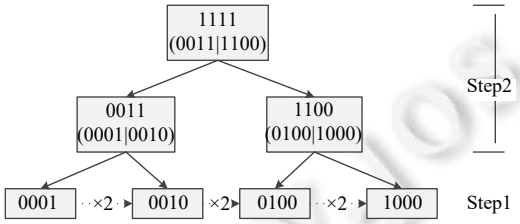


图 6 编码树 *B*

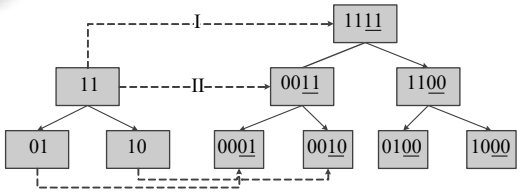


图 7 编码树随机融合

值得注意的是: 图 7 中的节点 11 有多个融合选项, 若选择根节点作为目标融合节点, 将改变原程序左右子树关系, 进而破坏算法逻辑. 产生这个问题的原因在于: 若取 F_1 为 $Node_{A\_i}$[3:4], 则根节点编号值由其子节点传递产生, 并非由左右节点或运算产生, 因此造成树结构关系被破坏. 为解决这一问题, 本文模型中对于一对多的对应关系, 选择下层节点作为目标融合节点, 避免因节点值传递而造成逻辑混乱. 除此之外, 目标融合函数的函数类型应与原函数一致, 且包含原函数参数列表, 避免因函数融合导致数据传递与返回值的偏差.

函数融合算法的主要思想在于利用函数深度优先的执行方式构造符合函数执行序列的树结构, 并对树结构进行编码解析, 将函数融合问题抽象为树融合问题, 基于一定的随机选择, 最终给出融合策略. 算法设计如算法 2.

算法 2. 树融合算法.

输入: 树 *A* 节点集合 $Nodes_A$, 树 *B* 节点集合 $Nodes_B$;
输出: 融合节点对应关系.

begin

1 *Map<Node *,Node *>Correspond*
2 *Array<Node *>*$Nodes_A$
3 *Array<Node *>*$Nodes_B$
4 procedure *GetMapRelationship*(*Temp*)
5 i←0
6 j←0
7 **foreach** $Node_B$ in $Nodes_B$:

```
        foreach Node_A in Nodes_A:
          if Temp^Nodes_B==Node_A:
              Correspond[i].left←Node_A
              Correspond[i].right←Node_B
            i←i+1
        if notmatch
          return Null
        Index=ChooseLow(j,i) //在满足条件的对应关系中选择下层节点
        Correspond[j].left←Correspond[Index].left
        Correspond[j].right←Correspond[Index].right
        j←j+1
        i←j
      return Correspond
    end procedure
  foreach Temp in ComBinRandom(length(Nodes_B),length(Nodes_A))
    output=[·]
    Output.append(GetMapRelationship(Temp)) //输出符合条件的融合序列
  RandomChoose(Output)
end
```

函数融合完成后, 回调循环过程将不再局限于相邻过程间调用, 而转为跨随机过程调用. 相比融合前, 函数调用过程更为复杂, 关键代码处于更深层的调用函数中, 有效提高抗逆向能力.

4.3 混淆系统开销分析

模型开销是评估抗逆向方法好坏的重要组成部分, 该指标主要包括程序变换后的代码膨胀率和执行效率. 本文在系统建立的前提下, 针对系统开销进行了理论分析, 进一步说明实际可用性.

定义 4.1. 记程序变换前大小为 $Size_{\text{old}}$, 程序变换后大小为 $Size_{\text{new}}$, 程序膨胀率可表示为:

$$Expand = \frac{Size_{\text{new}} - Size_{\text{old}}}{Size_{\text{new}}} \tag{3}$$

由于代码变换不涉及数据段, 因此可用指令条数近似标识程序大小. 本文记程序中循环数为 $LoopNum$, 记所有循环代码块集合为 $Loop$, 记函数 $Num(Basic_i)$ 标识 $Basic_i$ 中的指令数目, 并给出程序增量计算方式:

$$Size_{\text{new}}Size_{\text{old}} = 2Num(Loop) + 2LoopNum + LoopNum[Num(Basic_i) + 2] \tag{4}$$

其中, α 为循环中需要重构数据依赖的指令占比. 因此, $2\alpha Num(Loop)$ 表示新增栈操作指令, $2\cdot LoopNum$ 表示新增边界设定指令, β 表示判断优先型循环在所有循环中的占比. 此类循环额外增添了判断代码块和首次执行的跳转指令.

设 γ 表示循环代码块在程序代码段中的占比, 设 δ 表示程序代码段在程序中的占比, 推导出增长率计算公式:

$$Expand = \gamma \cdot \delta \cdot \frac{Size_{\text{new}} - Size_{\text{old}}}{Num(Loop)} \tag{5}$$

显然, $Num(Loop) >> 2\cdot LoopNum$, $Num(Loop) >> \beta LoopNum[Num(Basic_i)+2]$. 因此, 当程序增大, 循环相关指令增多时, 有:

$$Expand = \lim_{Num(Loop)\to\infty} 2\cdot\alpha\cdot\gamma\cdot\delta \tag{6}$$

由公式 (6) 可知: 判断优先型循环的冗余代码是导致开销增加的主要原因, 执行优先型循环的代码替换基本不引发代码膨胀. 除此之外, 与传统抗逆向方法对复杂程序敏感不同, 本模型的代码膨胀率随程序的增大逐步趋于稳定, 可以有效适用大规模开发场景.

定义 4.2. 记程序变换前执行耗时为 $Time_{old}$, 程序变换后执行耗时为 $Time_{new}$, 程序执行效率变化可表示为:

$$Efficient = \frac{Time_{new} - Time_{old}}{Time_{new}} \tag{7}$$

此处, 本文用程序运行时执行指令数近似表示运行耗时. 显然, 除数据依赖和边界设定等冗余指令外, 程序需额外执行回调响应事件 E 中的相关指令,ε 表示回调响应事件指令在循环指令中的占比, 即:

$$Time_{new}Time_{old} = N(Size_{new}Size_{old} + Num(Loop)) \tag{8}$$

其中, N 表示程序中平均循环次数. 根据代码膨胀率中的有关推论, 当程序增大, 循环相关指令同步增多时, 近似解出效率计算公式:

$$Efficient = \lim_{Num(Loop)\to\infty}(2\cdot\alpha+\varepsilon)\cdot\gamma\cdot\delta \tag{9}$$

显然, 程序执行效率与回调响应事件的执行效率紧密相关. 因此, 回调函数的选取是影响模型开销的主要因素. 一般来说, 本文选取的回调函数满足 $\varepsilon\to0.4$ 与程序膨胀率相同, 本模型的执行效率对复杂程序敏感度低, 满足大规模开发需求.

5 实验评估

回调函数是本文模型的基础, 表 1 列举了系统中具有代表性的回调函数, 为读者提供一定的研究基础.

表 1 回调函数信息

函数名	主要功能	函数名	主要功能
bsearch	对排序后的数组进行搜索	*EnumFonts*	遍历Windows字体模块
_lfind	对指定对象进行线性搜索	*EnumProps*	枚举窗体属性列表
_lsearch	对指定对象进行线性搜索	*EnumWindows*	枚举顶级窗体
qsort	执行快速排序	*EnumChildWindows*	枚举指定父窗体的子窗体

可以看出: 我们选取的回调函数以数组和模块的遍历为主, 主要进行读操作, 保证混淆算法不影响程序原始功能. 另外, 为使回调函数满足反复回调性, 我们在逆向分析的基础上对遍历指针进行动态复位, 如 *_lfind* 函数, 需在回调函数中进行如下操作.

代码 1. 遍历指针复位.

```
1 //Compile:vs2015-Debug
2 int *p
3 _asm
4 {
5 mov eax, esi
6 sub eax,0x24
7 mov p eax
8 }
9 *p=*p−4
```

分析表明: 当程序复杂度增大时, 混淆后的程序开销与原程序开销的比值趋于固定. 为展现模型特征, 本文选取 OpenSSL 中的 3 种常用算法和基准测试套件 SpecInt-2000 中的 3 类复杂程序作为测试代码, 实施模型评估. Parser 和 Twoif 的循环数量相对较多, 我们认为: 为避免代码混淆特征过于明显, 同一回调函数不应被多次使用. 一般情况下, 我们以 5 为阈值, 首先按照循环相关基本块的数量对循环进行排序, 优先处理基本块数量较多的循环, 当存在同数量级循环无法完全处理时, 优先选择字符处理回调函数进行处理.

在上述测试样例之外, 本文选用简单排序算法 Bubble Sort 作为部分实验环节的演示示例. 测试集基本信息见表 2.

表 2 测试用例基本信息

测试源码程序	代码行数	循环数量	处理循环数量
Bubble Sort	28	2	2
SHA256	183	7	7
AES256	634	25	25
RSA2048	1 375	38	38
Bzip	4 625	171	171
Parser	11 421	562	421
Twoif	20 500	906	526

首先对模型进行功能一致性检测: 对于加解密算法和压缩算法, 利用脚本随机填充数据, 目标类型包括字符串和二进制文件, 生成大小不同 (1 B–1 000 MB) 的测试用例; 对于 Parser 和 Twoif 算法, 直接应用 SpecInt-2000 标准测试样例, 比较模型应用前后输出结果. 实验表明: 回调型循环与传统循环算法逻辑一致、功能一致, 可以进行进一步测试.

本文设计的控制流深度模糊模型旨在增强代码的保护能力, 目前学术界主要采用 Collberg 提出的多维定性描述法来分析保护技术是否有效[1]. 本文从该方法出发, 基于开销、抗逆性、抗同源性这 3 个维度对控制流深度模糊模型进行有效性分析.

5.1 模型开销分析

模型开销主要针对程序变化后的代码膨胀率和执行效率, 前文已经从理论上证明: 本模型对于复杂程序敏感度低, 开销随着程序增大趋向一个稳定值. 实验部分主要确定该值的具体大小并与传统逆向方法进行比较, 证明本模型具有开销低和适用大型程序的优势.

对测试集应用回调函数模型, 图 8 说明了测试算法在模型应用下的代码膨胀情况. 可以看到: 随着程序复杂度的增加, 代码膨胀率逐步趋向稳定值.

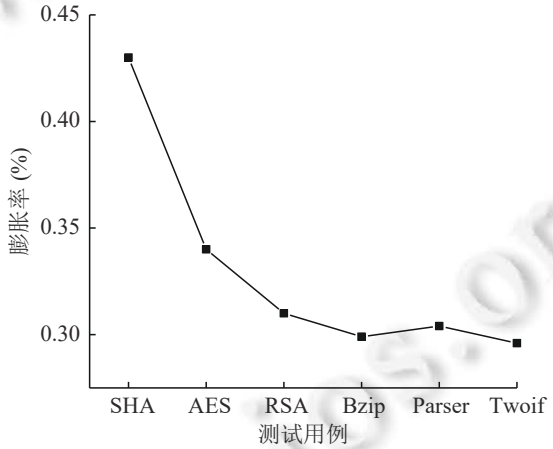


图 8 模型应用后代码膨胀率

在公式 (4.6) 中, 本文用执行指令数近似分析了模型应用后执行效率的变化, 但在实际测试中, 需要用更准确的耗时指标测定程序执行效率. 此外, 为了检验回调函数选取对于模型开销的影响, 本文分别选取 $\varepsilon\to0.4$ 和 $\varepsilon\to0.7$ 这两类回调函数作为基础变换样本.

如图 9 所示, 回调触发事件对执行效率具有显著影响, 但从根本上来讲不影响效率的变化趋势:当程序复杂度提高时, 执行效率变化趋向于一个稳定值.

因此, 实验检测结果与本文先期理论计算相符: 当程序复杂度增大时, 程序开销与程序规模仍呈线性关系. 学界认为: 线性相关的混淆算法可以支持大型程序的混淆, 因此本文模型具备良好的实用性.

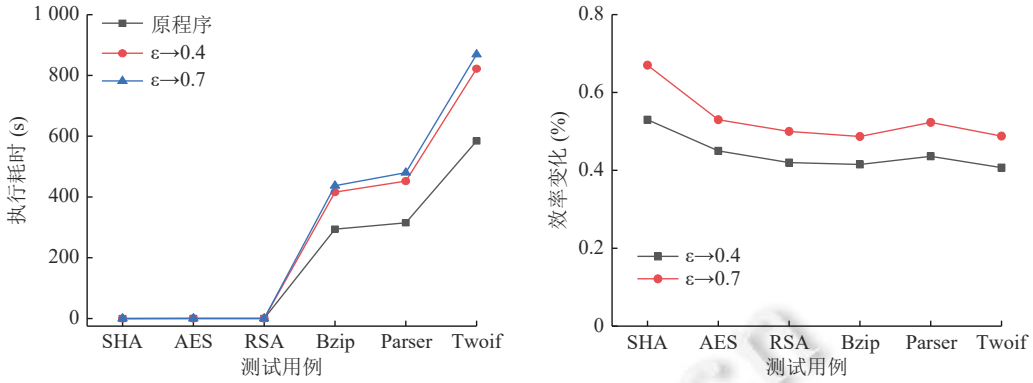


图 9 模型应用后程序执行效率

OLLVM 是著名的开源控制流混淆模型[12], 为了进一步验证模型在开销方面的优势, 本文选取 OLLVM 中的指令替换算法、分支合并算法、虚假控制流算法、控制流扁平化算法作为对比样例, 实验结果如图 10 所示.

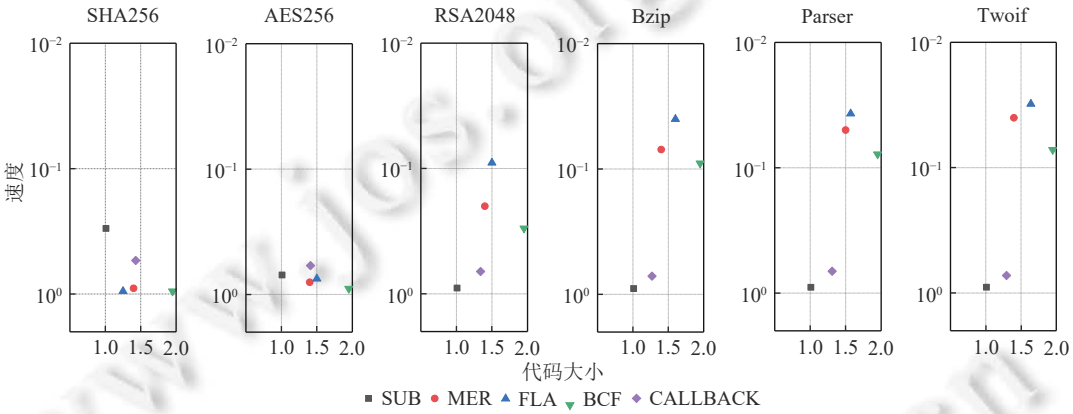


图 10 混淆算法开销对比

显然, 与 OLLVM 中的抗逆向算法相比, 本文模型在程序大小和运行效率上有显著区别, 稳定和低开销是本文模型的特点.

值得注意的是: 在实验中, 指令替换算法表现出了相似的开销特征, 然而指令替换算法保护效果相对较差. 因此, 在后续实验中, 本文将进一步说明本模型在抗逆向分析方面的优势.

5.2 抗逆性分析

动态行为调试和静态结构分析, 是逆向分析过程中的两个重要阶段. 本模型将同一函数内的循环跳转切分为多个函数的循环调用, 程序运行时, 循环跳转行为被隐藏, 调试分析的难度增大. 同时, 由于循环结构被重组, 循环跳转由系统自动化回调来实现, 程序控制流在静态层面将表现为顺序结构, 隐藏原本的控制流逻辑, 对抗静态分析.

5.2.1 抗动态分析能力

获取程序的跳转信息有助于分析人员理解程序执行过程, 同时也是进行逆向分析的基础. 因此, 混淆技术的目的之一就在于隐藏跳转信息, 添加冗余信息, 增大程序的复杂性. 为描述混淆模型的抗逆向能力, 本文提出了程序执行熵的概念, 定量描述混淆前后程序的信息量. 模型针对循环结构进行处理, 有效跳转信息的多少, 成为衡量模型抗逆性的重要指标.

本文给出程序执行熵的具体计算方法如下.

首先记录程序运行过程中出现的条件跳转指令集合:

$$JmpSet = J_1, J_2, J_3, J_4, \ldots, J_n.$$

记程序中共有 n 条指令, 跳转指令 J_i 的执行次数为 J_{i_num}, 执行频率为 $p(A_i)$:

$$p(A_i)=\frac{J_{i\_num}}{n} \tag{10}$$

记程序中共有非跳转指令 m 条, 作为干扰信息以 $1/n$ 的频率纳入熵值计算. 设程序执行熵为 *entropy*, 计算公式为

$$entropy=-\left(\frac{m}{n}\cdot\log_2\frac{1}{n}+\sum_{i=1}^{n}p(A_i)\cdot\log_2 p(A_i)\right) \tag{11}$$

显然, 熵值越大, 程序中包含的跳转信息越少, 算法逆向难度越大. 而为了进一步增大模型应用后的程序执行熵, 本文建立了函数嵌套融合算法, 通过更深层次的函数调用减少有效信息含量. 根据前文所述, 设程序变换前函数调用树的叶子节点数为 n, 节点中, 度的最大值为 p, 分别取目标融合树的叶子节点数 $m=p^1n$,$m=p^2n$,$m=p^3n$, 测试该算法对程序执行熵的影响程度. 首先对所有算法应用回调模型, 然后通过动态二进制插桩记录变换前后的基本块跳转指令, 最后生成程序执行熵.

贾春福等人[10]提出的基于异常处理的分支混淆方法通过对条件跳转指令进行替换, 将控制流转移过程交由系统完成, 隐藏控制流逻辑, 与本文模型的设计思路具有一定的相似性. 因此在评估抗动态分析方面, 选取该算法作为对照模型, 探索本模型的优势与不足.

需要说明的是: 该项目并未开放源码, 本文参照其技术原理进行了算法重构, 一定程度上反映了该工作所起到的混淆功效.

首先对所有算法应用两类模型, 然后通过动态二进制插桩平台 Dynamorio, 在基本块的末尾指令 (通常为分支跳转指令) 处, 记录当前指令地址和跳转的目标地址, 最后生成程序执行熵, 结果见表 3.

执行熵是程序有效跳转信息的评价指标, 本文模型与对照模型都通过改变程序跳转方式隐藏控制流信息, 增大程序熵值. 对照模型针对条件跳转指令进行处理, 既适用于分支结构, 也适用于循环结构. 因此, 相比本文模型覆盖面更广, 取得了更好的跳转信息隐藏效果. 然而, 本文模型的优势在于良好的伸缩性, 通过随机嵌入冗余函数, 构造复杂的控制流过程, 可以进一步压缩有效跳转信息. 如表 3 中所示: 程序执行熵随融合树的增大而增大, 并取得了比对照模型更好的混淆效果. 当然, 冗余函数的引入同样会降低程序执行效率, 增大代码尺寸, 因此对模型伸缩性需要做进一步分析.

表 3 模型应用前后程序执行熵

测试程序	变化前熵值	对照模型	$m=p^1n$	$m=p^2n$	$m=p^3n$
SHA256	13.43	28.76	19.84	40.52	116.50
AES256-CBC	25.31	42.84	36.13	44.15	136.79
RSA2048	26.24	58.64	40.41	76.53	208.91
Bzip	40.46	80.43	74.15	151.94	342.84
Parser	38.18	83.71	71.24	139.64	318.42
Twoif	40.94	78.57	72.88	145.87	329.11

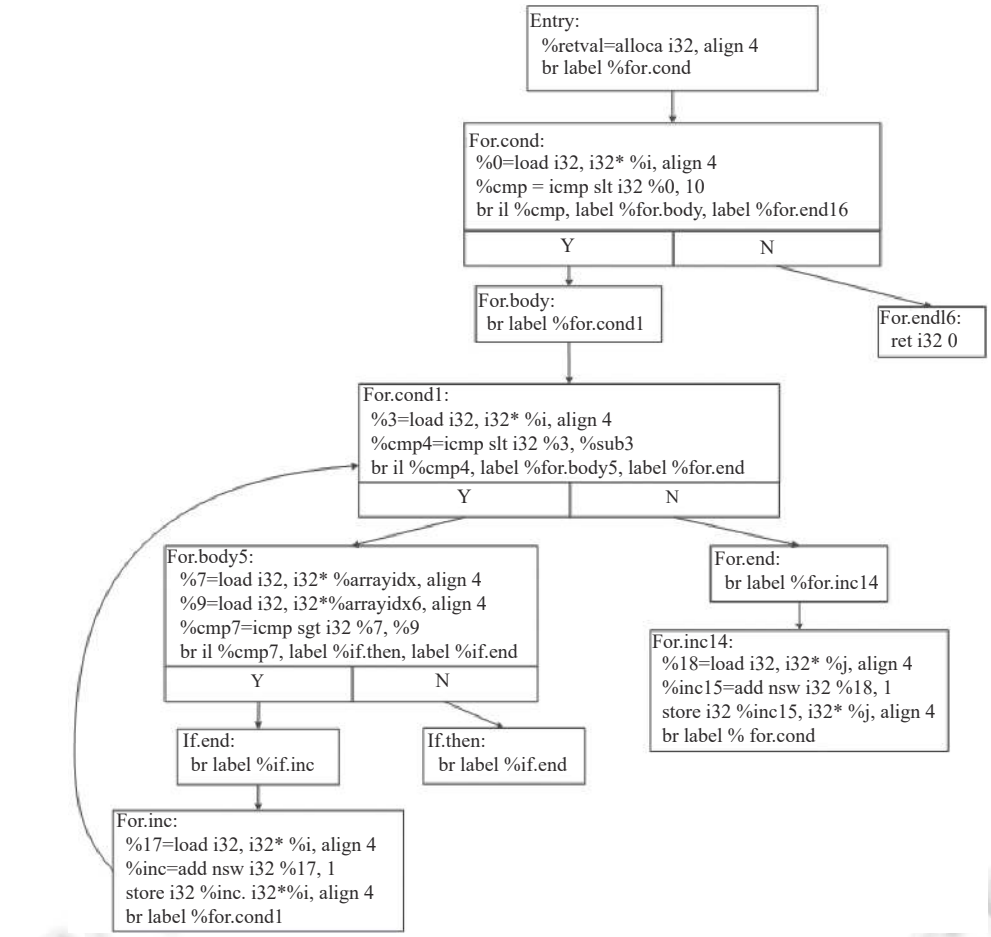
伸缩性由融合树的深度决定, 深度越深, 抗调试效果越好, 相应的模型开销也越大. 然而函数嵌套仅在程序中加入函数调用, 且该调用过程在循环体外, 不对程序本身执行效率构成较大影响. 且程序执行熵的增长与目标融合树深度的增加近似于指数关系, 因此, 该算法将以较小的效率损失快速提高程序执行熵.

但是, 深度的增加与冗余代码的生成也成指数关系, 容易造成代码快速膨胀. 经过实验测算, 假设 $m=p^\alpha n$, 一般情况下取 $\alpha<6$, 可使模型开销控制在可接受范围内, 同时显著提升程序抗动态调试能力.

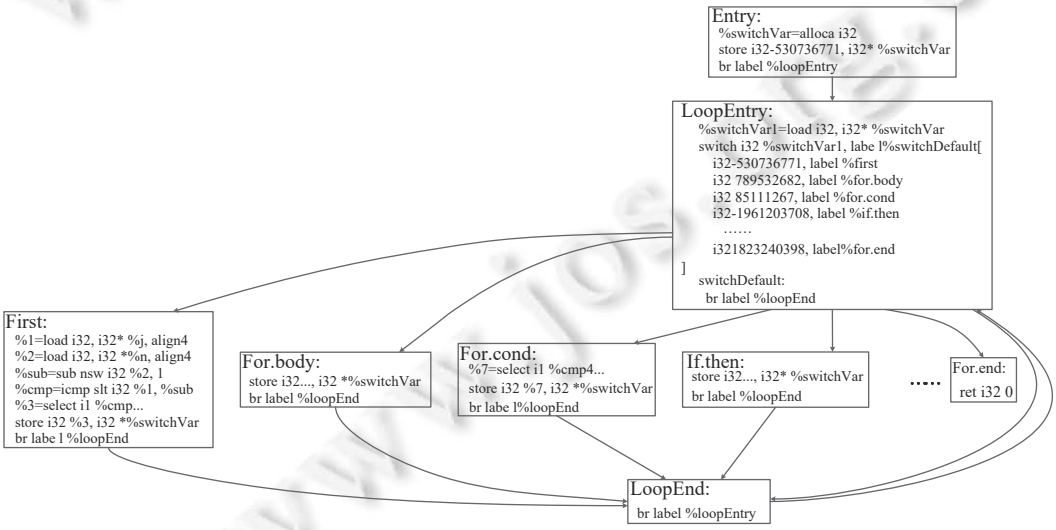
5.2.2 抗静态分析能力

抗静态分析是应用本模型的另一个优势, 此处以冒泡排序为测试样例, 同时选取控制流扁平化混淆算法作为对照模型, 展示本模型在抗静态分析方面的能力.

IDA 是实行静态分析的主要软件, 应用模型前的冒泡排序算法在 IDA 中完全暴露了自身的控制流逻辑, 分析者可以根据程序控制流图轻易掌握程序的跳转逻辑, 并进一步解析出算法思路. 因此, 隐藏原程序的跳转关系、破坏控制流图, 对于逆向分析具有重要意义. IDA 中以汇编语言表述程序功能, 为便于理解, 本文图 11 和图 12 中以 LLVM 中间语言代替反汇编语言并省略部分过程指令, 以更直观地描述基本块间的跳转关系.



(a) 混淆前



(b) 混淆后

图 11 控制流扁平化混淆算法

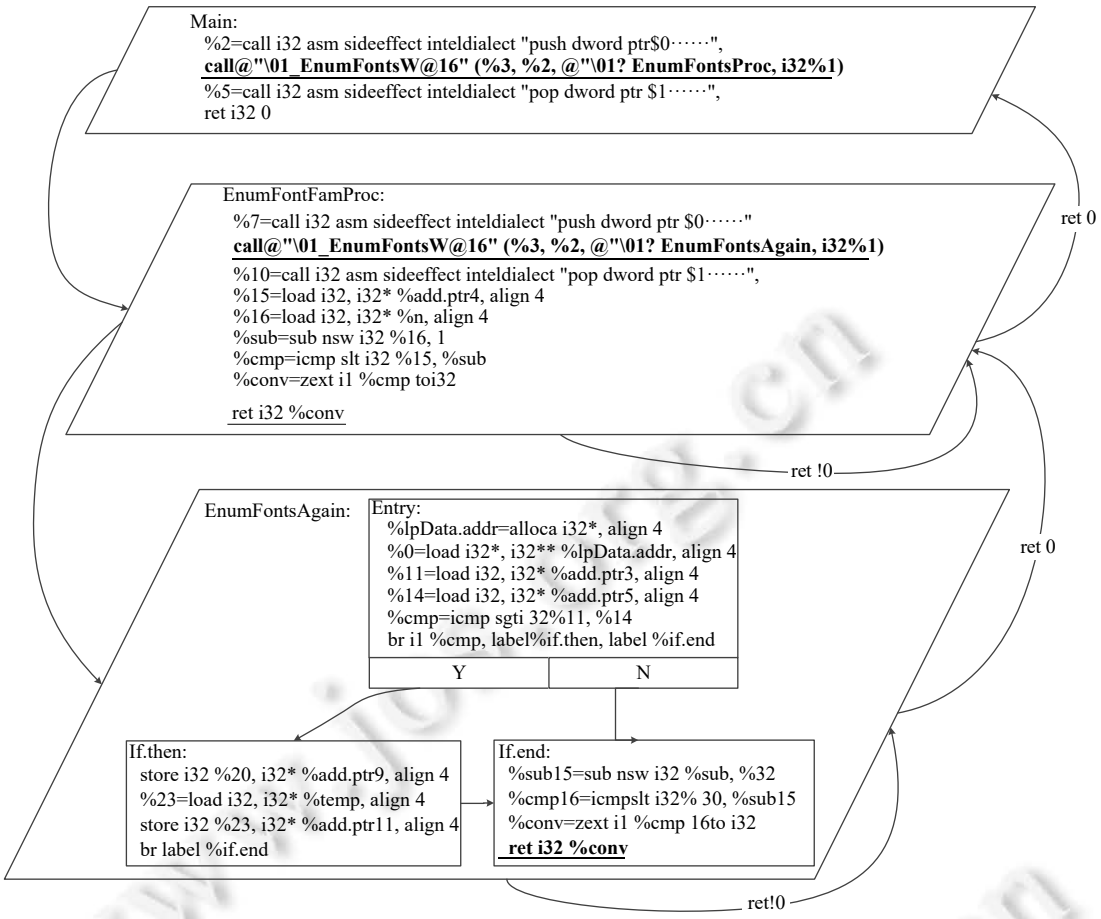


图 12 回调型循环控制流图

图 11 展示了控制流扁平化算法对程序静态结构的混淆, 从图中可以看出:该算法将基本块拆分, 通过 switch 语句进行执行流控制, 实现代码结构的破坏. 值得注意的是, 在 *LoopEntry* 基本块中记录了控制转移相关信息. 因此, 分析人员可以快速识别出该混淆算法, 通过解析控制转移表来还原执行逻辑, 算法隐蔽性较差.

图 12 展现了冒泡排序算法在应用回调模型后的静态结构. 回调函数作为常用的编程技巧, 在实际代码中大量存在. 图中也可以看出, 常规的函数调用不会表现出明显的混淆特征. 因此, 与控制流扁平化算法相比, 本文模型更具隐蔽性.

综上所述, 本文模型既增强了程序执行过程的复杂性, 又显著改变了程序的静态组成结构, 起到了良好的代码保护效果.

5.3 抗同源性分析

抗同源性强调程序变换前后的差异性, 增强程序的抗同源性可有效应对代码复用等攻击手段. 结构同源性与指令序列同源性是进行同源性分析的两个重要标准. 抗静态分析一节中, 本文已表明:应用回调模型混淆后的程序, 在不产生明显静态混淆特征的前提下, 显著改变了程序的控制流结构, 因此可有效对抗基于结构相似的同源性分析手段. 本节将主要针对指令序列相似性, 对模型进行分析.

取原程序中某函数起始地址为零地址, 设 $Index\_old(Ins_i)$ 表示该函数中指令 Ins_i 相对零地址的偏移, 设 $Index\_new(Ins_i)$ 表示指令 Ins_i 在变换后的程序中相对零地址的偏移. 则程序变换前后指令偏移可用 $Index\_new(Ins_i)-Index\_old(Ins_i)$ 表示. 本文以 Bubble Sort 算法为例, 随机抽取了 50 条非跳转指令进行分析, 同时选取潘

雁等人[13]基于指令交换的混淆技术进行对照, 得出程序执行前后指令偏移如图 13 所示: 应用本模型后, 程序相较变换前产生了一定的差异性; 与基本块间指令交换算法相比, 由于循环代码块迁移至不同函数中, 因此指令偏移尺度更大. 然而, 本文的代码变换主要以基本块和函数为单位, 相比指令级的代码混淆方法, 变换粒度较大. 如图 13 中所示, 变换后代码块中指令序列仍相对固定, 因此在抗指令序列同源性方面不具显著优势.

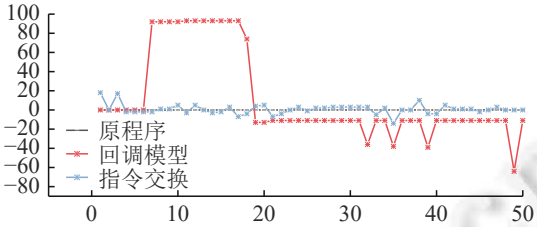


图 13 模型应用前后指令偏移

6 总结与展望

本文提出了控制流深度模糊的概念, 利用回调函数的执行特征, 将循环逻辑隐藏于函数调用中. 同时, 以代码变换为基础, 抽象出函数调用过程, 实施冗余函数的随机融合, 进一步增强抗逆向能力. 在实验部分, 本文选取了 3 类混淆算法作为对照模型, 分别从开销、抗逆性、抗同源性角度出发进行了深入分析. 结果表明: 本文模型在开销和抗逆性方面都取得了较好的混淆效果, 同时具备一定抗同源性分析能力, 具有良好的适用性.

本文目前对变量作用域的处理主要采用数据流依赖分析, 事实上, 准确抽取代码数据是一大难点, 特别是编程中动态特性的使用, 使得准确抽取这些数据难度更为加大[14,15], 这些动态特性会对软件代码分析造成显著影响. 针对这一问题, 我们通过暴力搜索算法获取所有直接和间接相关的变量指针 (无论该变量实际上是否被引用). 该算法虽然可以保证程序运行的正确性, 但同时引入了冗余变量, 在一定程度上对程序执行效率造成影响. 未来我们将在静态分析的基础上, 进一步优化数据处理方式, 降低模型开销. 而在提升混淆强度方面, 我们认为: 函数调用融合算法可与其他混淆算法有机结合, 如在函数调用节点添加不透明谓词, 增强模型复杂度. 控制流深度模糊的思想具有较好的实用性, 对软件保护产业的发展将具有积极意义.

References:

[1] Collberg C, Thomborson C, Low D. A taxonomy of obfuscating transformations. Technical Reports, #148, Department of Computer Science the University of Auckland New Zealand, 1997.

[2] Blazy S, Trieu A. Formal verification of control-flow graph flattening. In: Proc. of the 5th ACM SIGPLAN Conf. on Certified Programs and Proofs. St. Petersburg: ACM, 2016. 176−187. [doi: 10.1145/2854065.2854082]

[3] Yakdan K, Eschweiler S, Gerhards-Padilla E, Smith M. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. In: Proc. of the 22nd Annual Network and Distributed System Security Symp. San Diego, 2015. 286−293.

[4] Sharif MI, Lanzi A, Giffin JT, Lee W. Impeding malware analysis using conditional code obfuscation. In: Proc. of the Network and Distributed System Security Symp. San Diego, 2008. 321−333.

[5] Chen Z, Jia CF, Zong N, Zhen WT. Branch obfuscation using random forest. Acta Electronica Sinica, 2018, 46(10): 2458−2466 (in Chinese with English abstract). [doi: 10.3969/j.issn.0372-2112.2018.10.020]

[6] Popov IV, Debray SK, Andrews GR. Binary obfuscation using signals. In: Proc. of the 16th USENIX Security Symp. Boston, 2007. 275−290.

[7] Zong N, Jia CF. Branch obfuscation using “black boxes”. In: Proc. of the 2014 Theoretical Aspects of Software Engineering Conf. Changsha: IEEE, 2014. 114−121. [doi: 10.1109/TASE.2014.19]

[8] Ma HY, Ma XJ, Liu WJ, Huang ZP, Gao DB, Jia CF. Control flow obfuscation using neural network to fight concolic testing. In: Proc. of the 10th Conf. on Security and Privacy in Communication Networks. Beijing: Springer, 2014. 156−163. [doi: 10.1007/978-3-319-23829-6_21]

[9] Lin H, Zhang XH, Yong M, Wang BH. Branch obfuscation using binary code side effects. In: Proc. of the Int'l Conf. on Computer, Networks and Communication Engineering (ICCNCE 2013). Atlantis Press, 2013. 151−162. [doi: 10.2991/iccnce.2013.37]

[10] Jia CF, Wang Z, Liu X, Liu XH. Branch obfuscation: An efficient binary code obfuscation to impede symbolic execution. Journal of Computer Research and Development, 2011, 48(11): 2111–2119 (in Chinese with English abstract). [doi: 10.1007/s00466-010-0527-8]

[11] Zhao JZ, Nagarakatte S, Martin MMK, Zdancewic S. Formalizing the LLVM intermediate representation for verified program transformations. In: Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Philadelphia: ACM, 2012. 427−440.

[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] Pan Y, Zhu YF, Lin W. Code obfuscation based on instructions swapping. Ruan Jian Xue Bao/Journal of Software, 2019, 30(6): 1788−1792 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5429.htm [doi: 10.13328/j.cnki.jos.005429]

[14] Richards G, Lebresne S, Burg B, Vitek J. An analysis of the dynamic behavior of JavaScript programs. In: Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation. Toronto: ACM, 2020. 1−12.

[15] Jin WX, Cai YF, Kazman R, Zheng QH, Cui D, Liu T. ENRE: A tool framework for extensible eNtity relation extraction. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering: Companion Proceedings. Montreal: IEEE, 2019. 67−70. [doi: 10.1109/ICSE-Companion.2019.00040]

附中文参考文献:

[5] 陈喆, 贾春福, 宗楠, 郑万通. 随机森林在程序分支混淆中的应用. 电子学报, 2018, 46(10): 2458–2466. [doi: 10.3969/j.issn.0372-2112.2018.10.020]

[10] 贾春福, 王志, 刘昕, 刘昕海. 路径模糊: 一种有效抵抗符号执行的二进制混淆技术. 计算机研究与发展, 2011, 48(11): 2111–2119. [doi: 10.1007/s00466-010-0527-8]

[13] 潘雁, 祝跃飞, 林伟. 基于指令交换的代码混淆方法. 软件学报, 2019, 30(6): 1778−1792. http://www.jos.org.cn/1000-9825/5429.htm [doi: 10.13328/j.cnki.jos.005429]

沙子涵(1997−), 男, 博士生, 主要研究领域为网络安全.

熊小兵(1985−), 男, 博士, 副教授, 主要研究领域为网络信息安全.

舒辉(1974−), 男, 博士, 教授, 博士生导师,主要研究领域为网络安全.

康绯(1972−), 女, 教授, 主要研究领域为网络安全机制分析.

武成岗(1969−), 男, 博士, 正高级工程师, 博士生导师, CCF 高级会员, 主要研究领域为计算机系统安全.