小 型 微 型 计 算 机 系 统
Journal of Chinese Computer Systems
DOI:10.20009/j.cnki.21-1106/TP.2024-0554
2026 年 2 月 第 2 期
Vol.47 No.2 2026

多函数混合的程序控制流执行逻辑与混淆方法

唐成华[1,2],林 和[1,2],张 靖[1,3],强保华[2,3]

[1](桂林电子科技大学 广西可信软件重点实验室,广西 桂林 541004)
[2](广西密码学与信息安全重点实验室,广西 桂林 541004)
[3](广西云计算与大数据协同创新中心,广西 桂林 541004)
E-mail:tch@guet.edu.cn

摘 要: 针对目前基于 LLVM 的控制流混淆局限于函数内部执行程序流程控制的问题,提出一种多函数混合的程序控制流混淆方法.基于汇编实现函数外部基本块跳转逻辑,以汇编文件中的所有函数的基本块为单位,将其杂乱混合在一个函数中,并采用两种改进混淆算法,首先是实施多函数混合的基本块虚假控制流混淆,在正常的控制流中插入函数内或外的虚假跳转,其次是执行多函数混合的基本块控制流扁平化混淆,由一个变量和一个分发器控制所有的跳转,最终达到多函数混合的控制流有效混淆目的.该方法能实现隐藏基本块所属函数,使得只能从一个大函数对执行逻辑进行逆向分析.此外,优化了代码分发器实现,时间复杂度从原来的 O(n)降为了 O(1).实验结果表明,多函数混合的控制流混淆方案相对于已有的控制流混淆可进一步降低代码相似度,其中多函数混合虚假控制流下降达 63.48%,多函数混合控制流扁平化方法在较高复杂度程序上运行速度提升至其他扁平化混淆的 2~3 倍.

关 键 词: 软件安全;代码混淆;虚假控制流;控制流扁平化;执行逻辑
中图分类号: TP301 **文献标识码:** A **文 章 编 号:** 1000-1220(2026)02-0468-09

Execution Logic and Obfuscation of Multi-function Mixed Program Control Flow

TANG Chenghua[1,2],LIN He[1,2],ZHANG Jing[1,3],QIANG Baohua[2,3]

[1](Guangxi Key Laboratory of Trusted Software,Guilin University of Electronic Technology,Guilin 541004,China)
[2](Guangxi Key Laboratory of Cryptography and Information Security,Guilin 541004,China)
[3](Guangxi Cloud Computing and Big Data Collaborative Innovation Center,Guilin 541004,China)

Abstract: In order to solve the problem of control flow obfuscation limited to the internal execution of program flow control within functions based on LLVM,a multi-function mixed program control flow obfuscation method is proposed. Based on the assembly implementation of function external basic block jump logic,the basic blocks of all functions in the assembly file are randomly mixed into one function,and two improved obfuscation algorithms are adopted. Firstly,the implementation of basic block bogus control flow obfuscation for multi-function mixing involves inserting false jumps inside or outside the function into the normal control flow. Secondly,the implementation of basic block control flow flattening obfuscation for multi-function mixing involves controlling all jumps with a variable and a distributor,ultimately achieving effective control flow obfuscation for multi-function mixing. This method can hide the functions to which the basic blocks belong,so that only one large function can perform reverse analysis on the execution logic. In addition,the code distributor implementation has been optimized,reducing the time complexity from O(n) to O(1). The experimental results show that the control flow obfuscation scheme with multi-function mixing can further reduce code similarity compared to existing control flow obfuscation. Among them,the bogus control flow of multi-function mixing can be reduced by 63.48%,and the flattening method of multi-function mixing control flow can run 2 to 3 times faster on higher complexity programs than other flattened obfuscations.
Keywords: software security;code obfuscation;bogus control flow;control flow flattening;execution logic

0 引 言

代码混淆技术的核心思想是对软件源代码或字节码的转换,旨在使其在功能不变的前提下变得难以理解和分析.随着逆向工程技术的发展,特别是在移动应用和云计算的兴起,代码混淆逐渐演变为软件开发和发布过程中的标准实践之一,其技术应用已覆盖移动应用[1]、Web 应用[2]、嵌入式系统[3]等领域,成为保护软件知识产权和敏感信息的重要手段,通常采用标识符名称混淆[4]、控制流结构混淆[5]、基于加密或存储方式数据混淆[6]、虚假代码插入混淆[7]、动态执行混淆[8]等方式实现.Collberg 等人[9]于 1997 年开创性地提出了代码混淆技术的理论框架,将混淆技术划分为专注于布局结构的

收稿日期:2024-12-20 收修改稿日期:2025-01-18 基金项目:国家自然科学基金项目(62062028)资助;广西可信软件重点实验室基金项目(KX202320)资助;广西研究生教育创新计划项目(YCSW2024328)资助. 作者简介:**唐成华**,男,1974 年生,博士,教授,CCF 会员,研究方向为网络信息安全、恶意代码分析;**林 和**,男,1998 年生,硕士研究生,研究方向为网络信息安全、代码保护;**张 靖**,女,1999 年生,硕士研究生,研究方向为网络信息安全;**强保华**,男,1972 年生,博士,教授,CCF 会员,研究方向为智能信息处理、大数据安全.

布局混淆、影响程序执行流程的控制流混淆,以及针对数据表示的数据混淆等类别,并给出了衡量混淆效果的系列标准. Barak 等人[10]从密码学角度探讨了混淆方法的安全性,认为尽管尚未存在一种绝对安全的、免疫任何逆向工程手段的混淆器,但代码混淆技术是一项极具研究价值的技术领域,且其在保护软件免受未授权访问、复制或篡改等威胁方面极具重要性和不可或缺性.

随着逆向工程技术的进步,传统的代码保护技术如代码加密和数据混淆等方法的有效性逐渐减弱. 而控制流混淆是通过改变程序的控制流结构,使得逆向工程和代码分析变得更加困难,能够防止恶意攻击者提取核心算法. 控制流图能够将复杂的代码逻辑简化为直观的图形,Luo 等人[11]通过控制流图构建更适合描述指针引用关系的程序模型,遍历其分支路径,识别指向内存资源的指针引用路径,同时剔除与这些指针对象无关的控制流信息,从而减少非关键路径的分析开销,提升程序分析的精度. 诸多研究者们深入探讨了程序分析领域内控制流图的关键作用,通过不断优化混淆算法,提升其复杂性和隐蔽性,以增强其抵抗逆向工程和分析的能力[12-14]. 目前,常用的控制流混淆方式包括虚假控制流、控制流扁平化、控制流中指令替换等,已在保护软件免受逆向工程攻击方面展现出卓越潜力.

在虚假控制流(Bogus control flow)方面,主要是通过在程序中引入一些不实际执行的控制流结构,例如虚假的条件分支、死代码插入、循环混淆等[15],从而干扰逆向工程和代码分析,但不会改变程序的实际执行路径. 程序中的控制流是由条件分支形成的,加入虚假的分支,甚至设定决定分支结果的布尔表达式的形式都能够影响对路径的分析与理解,获得各种混淆的相互作用[16]. Ullah 等人[17]认为程序中的缺陷往往偏离正确执行的控制流路径,提出使用条分支计数的方式监控多线程程序执行行为,但获取程序执行的有价值数据是困难的. Huidobro 等人[18]提出在不同签名中插入死代码以实现恶意软件的代码混淆,但当文件的十六进制结构中有多个由防病毒系统标识的签名时,它将无法运行,此时需要执行另一个过程来插入死代码. 循环混淆也是代码自身保护的手段. Pan 等人[19]提出一程适用于 XSS 攻击检测的分类方法,通过循环解码和代码拼接来减少攻击者使用混淆技术掩盖恶意代码意图的干扰. 实际上,虚假控制流方法的核心在于虚假块的构造和不透明谓词的处理. Li 等人[20]针对基于 LLVM 框架的控制流混淆算法在保护力度上的局限问题,提出入度混淆的概念,在虚假控制流中加入防入度分析策略,以规避虚假块被攻击者的预先判断,但不足之处在于算法测试仅以单文件为主,对特定路径的分析效果有限. 不透明谓词则是软件混淆的公认的基本构建块,它实现了一个在编译时其值已知,但在运行时对于攻击者来说难以预测的表达式. 该表达式提供恒定的布尔输出,但在静态分析中似乎具有动态行为[21]. 利用不透明谓词,可以构造出复杂的条件表达式,引导控制流进入虚假的路径. Jens 等人[22]具体化了不透明谓词的概念,能重用软件中已有的数据结构及其 API 来实现软件保护,证明了灵活的不透明谓词能提供很强的灵活性和隐蔽性,但其部署成本很高.

在控制流扁平化(Control flow flattening)方面,主要是通过将不同的控制流结构(如条件分支、循环等)转化为统一的控制结构,使得程序的执行路径变得不直观和不易预测,从而增加理解代码逻辑的难度. 通常,控制流扁平化过程将执行逻辑打散,将程序所有的基本块放入一个大的调度循环中,并使用一个调度器块来确定随机化块的执行顺序,通过隐藏程序基本块之间的边来抵抗逆向工程的静态分析. Blazy 等人[23]验证了 C 程序代码混淆中控制流图扁平化的安全优势,涉及语义状态之间的非平凡匹配关系,其语义保持依赖于一个模拟证明,以确保混淆过程保留了程序的语义. Busi 等人[24]针对代码混淆处理中程序的安全属性是否得以保留问题,提出了一种基于时间恒定的控制流扁平化策略,并证明了每个满足策略的程序在混淆转换后仍然保持不变的执行时间. Johansson 等人[25]提出了一种控制流扁平化结构和轻量级调度器,对程序性能的影响较小,对控制流的静态分析具有良好的保护作用,同时,只要调度器状态不泄露,就可以实现攻击者目标的高度复杂性. Dong 等人[26]认为由于所依赖的结构特征发生了变化,一些控制流扁平化反混淆工具编译优化二进制文件的成功率较低,因此提出并实现了一个编译器无关的扁平化反混淆器,通过状态变量的数据流分析来识别有用块,并通过选择性符号执行来恢复有用块的控制流图. 该方法在面对调度块被修补,或者基本块内含多种不同属性的子块时容易失败. 为了提高软件白盒安全性,Liang 等人[27]利用控制流扁平化将程序源代码分解为多分支 While-Switch 循环结构,并使用不透明谓词来混淆扁平代码,即实现二次混淆,证明了以扁平化控制流系统为核心对代码进行模糊处理的可行性,同时也指出增强的代码混淆是以增加代码时空开销为代价的,以及如何制定最佳的混淆策略来实现操作效率和安全性的最佳平衡是未来的方向.

在指令替换(Instruction substitution)方面,主要是使用复杂的指令代替控制流中的一些二元运算,改变原始指令的逻辑,但不改变程序的结果[28]. 通常,指令替换可以用在虚假控制流的上下文中,例如在虚假条件分支中插入替代指令,从而使得逻辑更加晦涩;也可以在控制流扁平化的每个基本块中使用替代指令,使得每个块的真实意图更加隐蔽. Yang 等人[29]基于 Android 应用程序的 Dalvik 指令集字节码级提出一种将调用指令的被调用方法替换为另一个方法,以实现重定向其控制流传输,并将转换后的代码中的控制传输动态恢复到原始控制传输目标,该方法增加了应用程序控制流的不可预测性和隐蔽性,使逆向工程变得困难. Wang 等人[30]提出了一种基于指令替换的二进制混淆转换方法,生成的代码仍然是一个正确的程序,但具有更复杂的指令执行序列和复杂的控制流图,不过该转换方法减缓了程序的执行速度,因此替换模板需要改进. Mumtaz 等人[31]基于指令替换、垃圾代码插入等技术设计了混淆变形引擎,通过实验将处理五组不同的算术或逻辑指令进行了替换,增加了被检测的难度,该方法局限于仅适用于整数. Zhao 等人[32]指出通过使用对手不熟悉的虚拟指令替换程序指令的代码混淆是保护 Android 应用程序免受逆向工程攻击的一种有前景的方法,提出将 DEX 字节码转换为常见的 LLVM 中间表示的自动化方法,具有较强的安全强度,并以适中的成本实现了良好的隐身性. 当然,该方法只对所提取的 C 代码中的 20% 进行了虚拟化,本身并不利

于软件维护,而且虚拟化功能最终将与 Android 运行时交互,可能会被监听泄密. Yu 等人[33]针对云游戏软件保护,以迭代方式用等效的汇编指令替换原始汇编代码,通过混淆控制流以迷惑攻击者,同时保持程序产生相同的输出,不足在于通过动态混淆和自我监控增加了受保护软件的执行时间,影响游戏性能. 在技术实践中,指令替换往往是存在于虚假控制流或控制流扁平化的过程中,形成多样化的代码控制流混淆.

通常,控制流混淆是为了使得程序的逻辑更加复杂,但混淆可能会影响程序性能,因此又需要在安全性和性能之间找到平衡. Junod 等人[34]提出基于 LLVM 的混淆框架 OLLVM,实现了虚假控制流和控制流扁平化,以及数据混淆中的指令替换等技术,使得跨平台的混淆开始受到关注. 然而,OLLVM 的混淆特征明显,导致混淆过程易被识别,因此在应对现有反混淆器时抵御效果有限. Li 等人[20]通过在内部重构 Switch 结构,进一步增强对控制流扁平化跳转变量的隐藏效果,通过增加虚假块的调用,降低了虚假块被识别的风险. Hikari[35]通过将跳转地址加载入寄存器中,再利用寄存器的值实现跳转,达到了隐藏控制流的效果.

尽管代码控制流混淆技术在增强软件安全性方面发挥了重要作用,但也面临诸多挑战,主要涉及混淆过程可能会影响软件的运行速度或资源消耗、混淆代码的复杂性使得软件的调试和维护变得困难、不断发展的逆向工程技术对混淆代码产生破解风险等. 因此,在实施代码混淆时,必须综合考虑安全性、性能和可维护性等之间的平衡.

以上及现有诸多研究主要针对跳转变量进行处理,实际上未有新的控制流结构设计,并且都局限在单个函数内进行跳转混淆. 鉴于当前存在的相关问题亟待解决,本研究聚焦于探索一种全新的代码混淆方案. 具体而言,该方案旨在汇编代码层面达成控制流混淆的目标,其核心在于将所有函数的代码整合于一个函数体中,通过巧妙的设计与优化,在最大程度降低运行开销的同时,有效隐匿相邻基本块之间的跳转关系,从而为提升代码的安全性与保密性提供创新思路和有效手段.

1 研究动机

本文提出的控制流混淆方法是基于汇编代码的多函数之间程序执行流程的分析过程,本质上是将汇编代码中多个函数的代码实施混淆算法,在并不改变函数功能的情况下,将程序代码混淆在一起,改变程序执行流程. 该方法联合多个函数的结构和逻辑,模糊化函数之间的关联,增加针对函数之间调用关系和数据流动的理解难度,打破函数之间的独立性,将多个函数的执行逻辑交织在一起,使得单独追踪和分析单个函数的执行逻辑时,无法只通过局部分析一个函数来理解整体逻辑. 反编译工具也难以在全局范围内恢复函数原本的边界或原始调用关系. 这种全局性的混淆比单一函数的混淆复杂得多,从而增加了代码的理解难度. 本文主要贡献如下:

1)提出了基于汇编代码实现控制流混淆模型,使得可将单个汇编文件的所有函数代码块混淆在一个函数中.

2)实现了不同函数间的基本块和由不透明谓词构成的虚假跳转混淆,达到干扰攻击者还源程序的目的.

3)基于单独基本块来控制不同函数间基本块,采用状态变量实现逻辑顺序的控制流扁平化混淆,隐藏执行路径. 同时,优化控制流扁平化实现逻辑控制,解决了原有控制流扁平化在函数基本块过多时,代码运行效率低的问题.

2 混合控制流执行逻辑与混淆

要实现多个函数控制流的混淆,需要通过函数外部的基本块跳转,但这在高级语言中无法做到,因此需要在汇编语言中实现混淆. 从源代码文件到生成混淆后的可执行程序的多函数混合控制流混淆总体模型如图 1 所示. 首先在预处理阶段使用 LLVM 的 clang 工具,通过预处理器执行以#开头的预处理指令,例如,将#include 所指定的头文件内容插入到源代码中,以及进行宏定义的替换等操作. 经过预处理后,得到的仍然是 C 语言代码的文本文件. 继而在编译阶段首先进一步使用 clang 工具进行词法和语法分析与检测,生成 bc 文件,之后使用 opt 工具进行代码优化,比如,通过死代码消除去除程序中不会被执行的代码,以减少代码体积和提高运行效率. 循环展开可以减少循环的开销,从而提高循环的执行速度. 最后使用 llc 工具转换为特定目标平台的汇编代码,同时也会进一步优化生成的代码. 在混淆阶段使用本文实现的工具进行混淆,能读取原汇编的内容,并根据混淆算法将汇编内容重构并生成混淆后的汇编文件. 在汇编阶段使用 clang 工具进行汇编,汇编器对每条汇编指令进行解析,确定其操作码和操作数,处理汇编代码中的符号引用,确定它们在内存中的地址,并根据解析后的指令和操作数,生成对应的机器码. 在链接阶段使用 lld 工具进行链接,链接器将所需的库文件与目标文件进行链接,将库中的函数代码和数据合并到可执行文件中.

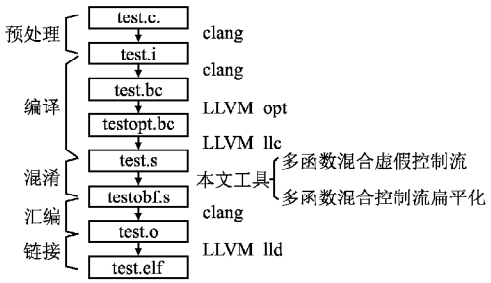


图 1 混合控制流混淆总体模型

Fig. 1 Overall model of mixed control flow obfuscation

对于混淆阶段采用两种控制流混淆算法实现:1)是不同函数间的基本块和由不透明谓词构成的虚假控制流混淆算法,主要用来在程序中插入看似合理但实际上没有实际功能的控制流路径来迷惑攻击者;2)是不同函数间的基本块用同一个调度基本块去处理控制,用一个状态变量来实现逻辑顺序的控制流扁平化混淆算法,目的是将所有函数的基本块使用一个调度基本块来决定执行哪个基本块,将原本清晰明了的控制流程隐藏起来,使程序变得复杂和难以理解. 另外,两种混淆算法还会打乱并合并所有函数的基本块,模糊化函数之间的界限,形成复杂的互相依赖,使得单独分析某个函数并不能揭示其完整的逻辑.

2.1 多函数混合虚假控制流

定义 1. 设程序 P 中包含函数集合 $f=\{f_i \mid i \in n\}$,其对应

的控制流图集合记为 $G=\{G_i \mid G_i=(V_i,E_i), i\in n\}$,其中,$V_i$ 表示函数 f_i 的基本块集合;E_i 表示基本块之间的控制流关系. 将所有函数 f_i 的基本块集合 V_i 提取,打乱其顺序后形成一个新的基本块集合 $V'=\cup_{i\in n}V_i$,基于维持原程序语义的基本块间跳转不变的情况下,在 V' 中重新定义边集合 E',形成新的控制流图 $G'=(V',E')$,G'即为多个函数混合而成对应的控制流图,该过程称为多函数混合.

定义 2. 在程序 P 的执行路径中,令所有有效路径为集合 *A*. 在确保程序功能不发生改变的条件下,构造一组虚假路径,记为集合 *B*. 通过在 P 中引入不透明谓词 p,将集合 *B* 与集合 *A* 结合,形成新的执行路径集合 *A'*,则集合 *A'*表示经过虚假控制流混淆后的程序 P 的执行路径集合.

现有的基于 LLVM 利用 IR 指令实现的虚假控制流,主要是通过向正常控制流中插入若干不可达基本块和由不透明谓词造成的虚假跳转以产生大量垃圾代码,干扰攻击者分析. 其不可达基本块来源于克隆原始基本块,在整个混淆过程中只被使用一次,即代码中只有一次使用跳转指令跳转到不可达基本块,而正常代码块通常会有多处使用跳转指令到达的情况. 虚假块所具有的这个显著特征,可被逆向分析用来识别基本块的真假,从而得到真实执行路径. 因此,在原有的虚假控制流混淆上,本文随机使用汇编文件中所有函数的真实基本块作为跳转分支. 这既避免了虚假块被识别,导致暴露程序真实执行路径,又减少了混淆后程序的体积,还可使整个汇编文件中的所有基本块杂乱混合在一起,极大增加了程序复杂度.

汇编文件的多函数混合虚假控制流混淆处理过程如算法 1 所示. 由于虚假控制流添加越多,执行的指令也越多,程序运行效率就会越低. 定义参数 p 用于控制每个基本块被混淆的概率. 它决定了在编译过程中,有多少个基本块会被选中添加虚假控制流. 通过调整 p 的值,可以间接影响整个程序的混淆强度. 在需要高度保护的关键代码段中,可设置较高的 p 值,反之,在对性能要求较高或对混淆效果并不严格的代码中,可适当降低 p 值. 该算法将所有函数的代码块混淆在一个函数中,并插入不透明谓词分支混淆,误导分析者分析错误的程序执行分支.

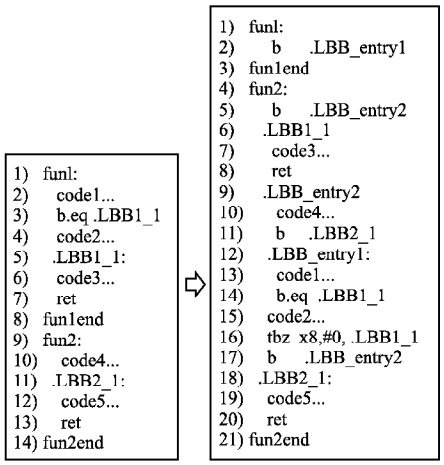
算法 1. 多函数混合虚假控制流混淆.

输入:汇编代码文件 file;混淆概率 p;

输出:控制流混淆后的汇编代码文件 file_ofu.

```
1. parseFile(file);
2. for fun in funList do
3.    clearAllIns(fun);/*清空所有指令*/
4.    addJumpEntryBasicBlockIns(fun);/*添加跳转至入口指令*/
5. end for
6. lastFun = getLastFun();
7. for BB in BBList do
8.    int probnum = randValue();
9.    Ifprobnum > p then
10.       continue;
11. end if
12. BBLable = randSelectBB(BBList);
13. op = selectOpaquePredicate();
14. code = genBCFCode(BBLable,op);
15. BB = modifyCode(BB,code);/*代码块中添加虚假控制流*/
16. insertBBCode(lastFun,BB);/*插入代码块指令到函数中*/
17. end for
18. write(file_ofu,funs)/*写入混淆汇编文件*/
```

算法 1 的第 1 行解析汇编文件,将所有函数和代码块保存在 funs 与 BBs 中,为后续的混淆处理做准备. 第 2 行 ~ 第 4 行是将每个函数的指令清空,并添加一条跳转指令,令其直接跳转到该函数入口的基本块,使得可将原来函数的代码放入其他函数中而不影响程序原来的执行. 第 7 行 ~ 第 16 行是遍历汇编文件中的所有基本块,每次遍历时生成一个范围在 0 ~ 100 之间的随机数. 如果随机数小于设定的混淆概率 p,则在该基本块中添加一个虚假控制流,以此控制混淆的强度. 此过程中,所有基本块的代码会被放入到最后一个函数中,达到将所有代码块混乱在一起到的目的. 算法 1 的时间复杂度为 O(C),其中 C 为 BBList 中的代码块数,即单个汇编文件中所有函数的代码块数.



(a)未混淆　　(b)混淆后

图 2　多函数混合虚假控制流混淆的过程

Fig. 2　Process of mixing multi-function with bogus control flow obfuscation

一段未混淆的汇编源程序代码如图 2(a)所示,其中含有两个函数,每个函数有两个代码块,代码块中的 code 是省略的非控制流代码. 针对该汇编代码经过多函数混合虚假控制流混淆后的汇编代码如图 2(b)所示. 对每个函数的第一个基本块前添加标签,以便跳转到函数入口处,例如 LBB_entry1 正是为了跳转到 fun1 入口基本块所添加的标签. 所有函数的代码块会打乱顺序再写入到其中一个函数中,然后在每个函数中额外添加一个跳转指令,跳转到它原来的函数入口. 这样使得逆向分析无法确定哪个代码块属于哪个函数. 在第 15 行会插入一个恒为真的不透明谓词,不透明谓词的结果保存在寄存器中. 第 16 行使用这个寄存器的值做判断分支执行后续代码,不透明谓词的值恒为真,程序将跳转到第 6 行执行 code3 的代码,保证了程序的功能和原来一致. 不透明谓词使用的寄存器优先使用后续代码不需要用到的寄存器,这样就不需要保存恢复原来的值,如果所有的寄存器都需要用到原来的值,插入不透明谓词可将使用的寄存器的值保存在栈中,在用完后恢复原来的值,保证程序执行逻辑不变. 当然,程序的代码运行在用户态中,无法保存状态寄存器,因此在后序代

码中需要使用到当时的状态寄存器的值时,插入的控制流不能使用会改变状态寄存器值的指令.例如使用 cmp 指令比较两个数的大小,然后根据值是否大于、小与或等于判断跳转执行哪里的代码,这种控制方式会改变状态寄存器的值,解决方法是将两个数相减的结果,用 cbz 指令判断是否为 0 来替换等于判断跳转,用 tbz 指令判断最高位的符号位是否为 1 来替换大于小于判断跳转.

2.2 多函数混合控制流扁平化

定义 3. 令程序的执行路径构成的路径树记为 T,程序的代码块构成树的节点 N_1、N_2、N_3…,其深度记为 $D(N_1)$、$D(N_2)$、$D(N_3)$….在确保程序功能不变的前提下,对程序中的基本块进行重组,使得重组后的路径树为 T',使得除根节点以外的其他节点 N 的深度相同,即满足条件:$|D(N_i)-D(N_j)|=0$,节点 $N_i,N_j\in T'$,则树 T' 表示控制流扁平化混淆后的程序执行路径树.

在已有的扁平化处理过程中,原本嵌套在单个函数内部、带有层级结构的基本块,通过主分发器的作用被重新组织到同一层级上.这实现了基本块之间的层级扁平化,使得单个函数的所有基本块都汇聚到一个统一的分发节点.而多函数混合控制流扁平化则会将整个汇编文件中所有函数的基本块汇聚到一个统一的分发节点.本质上,扁平化处理跳转实现是通过为每个基本块赋予一个值,依据跳转变量进行跳转调度,即主分发器寻找与跳转变量值具有相等值的基本块,若成功则跳转到相应基本块.实际上,当一个函数中的基本块数量过多时,主分发器将需要消耗大量时间资源进行比较,这严重影响了程序的运行效率.

为解决这一问题,本文将整个汇编文件中所有函数的基本块的起始地址存入数组中,跳转变量的值改为要跳转基本块的起始地址在数组中的下标,主分发器可直接用此下标快速在数组中找到要跳转的基本块.汇编文件多函数混合控制流扁平化混淆处理过程如算法 2 所示.该算法将所有函数的代码块混淆在一个函数中,整个程序执行逻辑由代码分发块实现,达到隐藏程序执行路径的目的,提高程序的复杂度.

算法 2. 多函数混合控制流扁平化混淆.

输入:汇编代码文件 file;

输出:控制流混淆后的汇编代码文件 file_ofu.

```
1. parseFile(file);
2. addBBAddress(BBList);
3. for fun in funList do
4.    clearAllIns(fun);
5.    addEntryBBAddressIndexToRegIns();/*保存 x30 寄存器的值入
      栈后,新值修改为入口基本块地址在数组中的索引*/
6.    addJumpSwitchIns(fun);  /*跳转至代码分发块*/
7. end for
8. lastFun = getLastFun();
9. addSwitchCode(lastFun);/*添加分发块代码*/
10. for BB in BBList do
11.    for ins in BB do
12.       if isJumpIns(ins) then
13.          int index = getJumpBBArrayIndex(ins);
14.          modifyJumpIns(index);/*修改 x30 寄存器的值并跳转至
             代码分发块*/
15.       end if
16.       if isRetIns(ins) then
17.          addStoreRegisterIns(BB);  /*恢复 x30 寄存器的值*/
18.       end if
19.    end for
20.    insertBBCode(lastFun,BB);  /*将代码块插入函数中*/
21. end for
22. write(file_ofu,funs);
```

算法 2 的第 1 行解析汇编文件,将所有函数和代码块保存在 funs 与 BBs 中,为后续的混淆处理做准备.解析出来的同一个函数的代码块标签是挨着的,第 2 行会打乱代码块标签顺序并添加到数组中,避免逆向分析人员从数组中分析出代码块的所属函数,第 3 行~第 7 行会把所有函数指令清空,之后插入存储 x30 寄存器的值到栈中的指令以及跳转到代码分发器的指令,以此达到跳转至所有混合在某一个函数中的原来的函数入口基本块.第 10 行~第 20 行会遍历每一条指令,如果是跳转指令先根据要跳转的标签在数组中的位置设置 x30 的值,之后将所有的跳转指令都替换成跳转到代码分发块的指令,这样通过跳转到代码分发块再跳转到真正要执行的代码块,达到隐藏程序执行路径.如果是返回指令则需要将之前在函数入口处保存在栈中 x30 寄存器的值恢复,避免影响程序的运行.第 19 行会将所有代码块的指令混合在一个函数中,使得逆向分析人员只能在一个庞大的函数中分析整个程序的运行情况.算法 2 的时间复杂度为 O(CN),其中 C 为 BBList 中的代码块数,即单个汇编文件中所有函数的代码块数,N 为代码块的指令数.

```
1)  thn1:                          25) ret
2)    sub  sp,sp,#16               26)   .LBB_entry2:
3)    stur  x30,[sp,#8]            27)    code4…
4)    mov  x30,#24                 28)   mov x30,#16
5)    b      .Ltmpfla              29)   b    .Ltmpfla
6)  fun1end                        30)   .LBB2_1:
7)  fun2:                          31)   code5…
8)    sub   sp,sp,#16              32)   ldur x30,[sp,#8]
9)    stur  x30,[sp,#8]            33)   add sp,sp,#16
10)    mov  x30,#8                 34)   ret
11)    b    .Ltmp fla              35)  .LBB_entryl:
12)  .Ltmpfla:                     36)   code1…
13)    sub  sp,sp,#16              37)   mov x30,#0
14)    stur  x9,[sp,#8]            38)   b.eq  .Ltmpfla
15)    adrp x9,.LJTIfla0           39)    code2…
16)    add  x30,x9,:lo12:.LJTIfla0 40)   mov x30,#0
17)    ldur x30,[x9,x30]           41)   b  .Ltmpfla
18)    ldur x9,[sp,#8]             42)  fun2end
19)    add sp,sp,#16               43)   .data
20)    br  x30                     44)  .LJTIfla0:
21)  .LBBl_1:                      45)   .xword .LBB1_1
22)    code3…                      46)   .xword .LBB_entry2
23)    ldur x30,[sp,#8]            47)   .xword .LBB2_1
24)    add sp,sp,#16               48)   .xword .LBB_entry1
```

图 3 多函数混合控制流扁平化混淆的汇编程序

Fig. 3 Assembly program with multi-function mixed control flowflattening obfuscation

针对图 2(a)代码经过多函数混合控制流扁平化混淆后的汇编代码如图 3 所示.在代码结构中,在每个函数的首个基本块之前需添加一个标签,此标签用于实现跳转到相应函数入口处的功能.所有代码块标签将以打乱的顺序存入名为“LJTIfla0”的数组.在编译为机器指令的过程中,该数组会被转换为相应代码块的起始地址.所有的代码块会以打乱的顺序写入到一个函数之中.随后,在每个函数入口处以及所有代

码块内的每一个跳转位置,都会设置 x30 寄存器的值,之后程序将跳转到“Ltmpfla”代码分发块. x30 寄存器作为保存函数返回地址的寄存器,在函数执行完成后的返回操作中被使用. 因此,利用 x30 寄存器存储跳转变量,仅需在函数入口处保存其原有值,并在函数返回时恢复该值. 相较于使用其他寄存器,此方式节省了大量用于保存和恢复操作的执行时间. 在“Ltmpfla”代码分发块中,会加载存储着数组的地址,随后依据 x30 的偏移值将数组中存储的基本块地址加载至 x30 寄存器,最终依据 x30 寄存器的值进行跳转,从而实现代码的分发. 在此过程中,需要使用额外的寄存器 x9,且在其使用前后也需进行保存和恢复操作,以避免对程序原有执行流程产生影响.

3　实验过程与分析

实验环境为在 12th Gen Intel(R) Core(TM) i5-12400F 的 CPU,32GB 内存的 Windows10 操作系统主机上. 实验混淆是基于 LLVM14 编译生成的汇编代码进行实现,样本运行环境为在天玑 8200Ultra 的 CPU,12GB 内存的 Android13 手机上. 聚焦于控制流混淆,将本文方法与 OLLVM[34] 及文献[20]混淆方案进行对比. 实验数据集 1 来源于文献[20],链接地址为 https://gitee.com/lcy20/paper_data/tree/master/src,实验数据集 2 来源于 github 上 c 语言开源的程序,包括 MD5、SHA256、DES、QuickSort、SelectSort 等项目,链接地址为 https://github.com/fswy600/program. 数据集 1 相对于数据集 2 程序运行时间较短,程序的时间复杂度较低. 实验使用 IDA Pro7.7 分析混淆后的程序控制流程,使用 BinDiff6.0 分析混淆程序与源程序的代码相似度,以此评估混淆效果.

3.1　运行效率影响分析

代码混淆使程序转换添加了额外的指令,必然要付出一定的性能代价. 使用 Android 的 time 命令计算程序运行的 3 个相关时间 real、user 和 sys. 其中,real 时间是实际运行的时间,包括了进程被调度等待的时间等;user 时间是进程在用户态运行的时间;sys 时间是进程在内核态运行的时间. 试验采用 real 时间进行计算,以源程序运行时间与混淆程序的运行时间的比,作为混淆程序应用混淆方法后的运行效率. 进行多函数混合虚假控制流混淆时,p 的值取 100. 针对数据集 1 的不同混淆方法的运行效率如表 1 所示,其中前 3 种混淆方法的运行效率源于文献[20]的数据进行计算. 可以看出,在低时间复杂度程序上,本文混淆方法对应的运行效率在 99.1% 之上,混淆程序的运行效率受影响很小.

表 1　不同混淆方法的运行效率

Table 1　Execution efficiency of different obfuscation

混淆方法	运行效率
OLLVM 控制流扁平化	99%
嵌套 switch 混淆	97.08%
嵌套 switch 混淆 + 人度混淆	96.15%
多函数混合虚假控制流(本文)	99.3%
多函数混合控制流扁平化(本文)	99.1%

因文献[20]方案源码未公开,实验只使用本文与 OLLVM 混淆方案,针对数据集 2 进行相对于源程序未混淆时的运行效率对比,如图 4 所示. 可以看出,在较高时间复杂度程序上,多函数混合虚假控制流方法自身的运行效率下降在 50% 以内,但相对于 OLLVM 虚假控制流仅略有下降,主要原因在于多函数混合虚假控制流在使用不透明谓词时使用

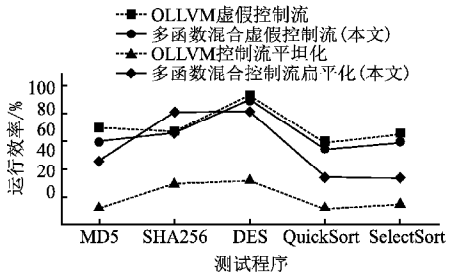


图 4　混淆程序运行效率的对比

Fig. 4　Execution efficiency of obfuscated programs

到的寄存器需要保存和恢复,每使用多一个寄存器将多执行两条汇编指令. 多函数混合控制流扁平化方法自身的运行效率下降在 66% 以内,但相对于 OLLVM 控制流扁平化方法的运行时间,仅处于其 1/2 ~ 1/3 的水平. 由于 OLLVM 控制流扁平化的代码分发器是用跳转变量逐一对比基本块代表的值进行选择下一个执行的基本块,它的时间复杂度为 O(n),而多函数混合控制流扁平化的代码分发器其跳转变量为对应基本块起始地址数组的索引,可直接从数组中得到下一个执行基本块的位置,时间复杂度为 O(1). 另外,每个基本块执行完后都会执行代码分发块,代码分发块是会被频繁执行的. 因此,本文的多函数混合控制流扁平化相对于其他扁平化混淆方法的运行效率有得到较大的提高.

3.2　混淆效果分析

使用逆向工具 IDA Pro 进行静态分析. 针对图 2(a)程序代码经过各种虚假控制流混淆后的流程如图 5 和图 6 所示.

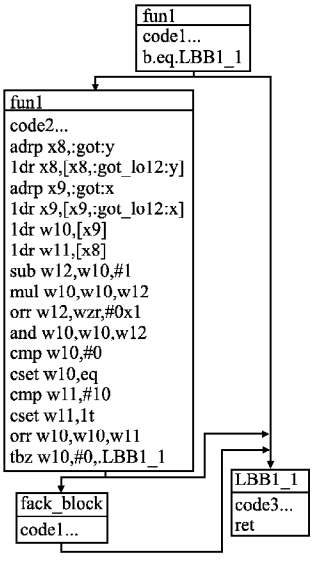


图 5　OLLVM 虚假控制流混淆程序流程

Fig. 5　OLLVM boguscontrol flow obfuscation

其中,图 5 展示了函数 fun1 被 OLLVM 虚假控制流复制 code1 的代码构造了一个虚假代码块 fake_block,并在代码块

fun1_1 处添加了一个不透明谓词,从而形成跳转至虚假代码块的虚假跳转,以此迷惑攻击者. 在这种混淆方式中,生成的虚假代码块只有虚假跳转指向它,这表明在流程图中虚假代码块只有一个前驱代码块,这是一个可用于判断虚假代码块的特征. 尽管真实块可能只有一个前驱代码块,但这也增加了虚假块被识别的风险. 虚假块被识别虚假跳转也将被识别,从而被攻击者知道程序的真实执行路径. 图 6 展示了函数 fun1 被多函数混合虚假控制流在 LBB_entry1_2 中添加了一个虚假跳转,先间接跳转到 LBB_entry1_3 恢复寄存器的值再跳转到函数 fun2 中的 LBB_entry2 真实块. 这种方式用真实块作为虚假跳转块,降低了虚假跳转被识别的风险,并且把多个函数的代码混淆在了一起.

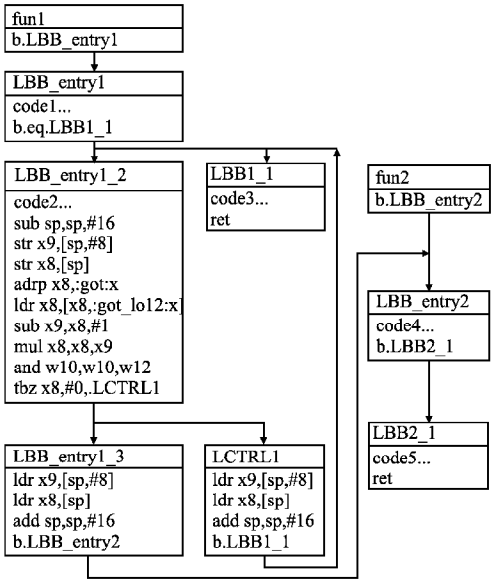


图 6 多函数混合虚假控制流混淆程序流程

Fig. 6 Multi-function mixed bogus control flow obfuscation

针对图 2(a)程序代码经过各种扁平化混淆后的流程如图 7 和图 8 所示. 其中,图 7 展示了函数 fun1 除入口基本块的每个真实块被 OLLVM 控制流扁平化添加了一个对应的 cb1、cb2、cb3 代码分发块. 调用这些真实块,需要通过设置一个状态变量,之后经过这些分发器逐一比对,确定跳转位置. 在这种混淆方式中,一个函数的真实块越多,分发块也越多,从一个真实块跳转到另一个真实块需要经过的分发器也越多,程序运行效率也就越低. 图 8 展示了函数 fun1、fun2 被多函数混合控制流扁平化在代码中添加了一个分发块 Ltmpfla. 该分发块通过 X30 寄存器为索引,从存储所有函数代码块地址的数组 Ljitfla0 中加载下一个需要执行的代码块地址. 在这种扁平化混淆方式,分发器的运行效率更高,并且所有函数共用一个分发器,还将函数的基本块杂乱混合在了一起. 另外,Ltmpfla 使用的 br 指令跳转,其目标地址是寄存器的值,这个信息在编译阶段是无法确定的,而 IDA Pro 使用的是递归下降反汇编算法,递归下降算法在进行语法分析时,主要依据的是静态的语法规则和输入的字符流,它没有办法获取程序运行时的动态信息. Ltmpfla 可跳转至每一个真实块,但流程图中却没有指向任意一个代码块,这就是导致 LBB2_1 与 LBB1_1 没有被任意代码块调用的原因. 因此,这种混淆也能有效地干扰软件的分析.

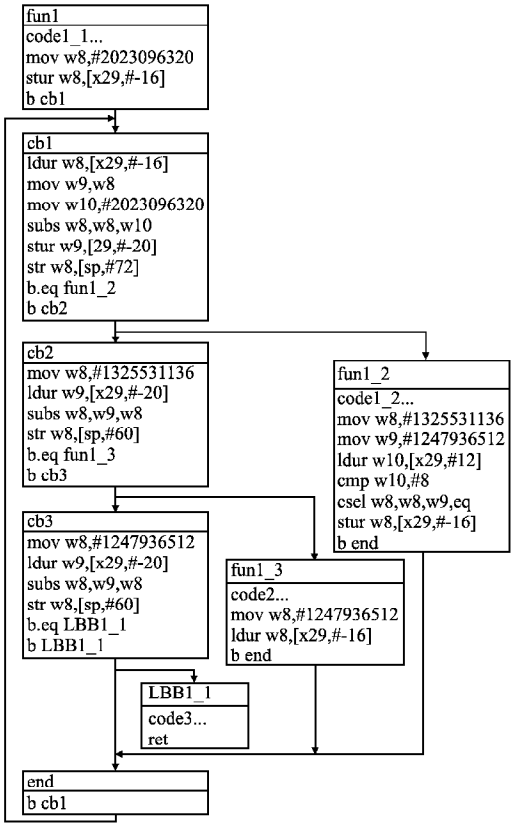


图 7 OLLVM 控制流扁平化混淆程序流程

Fig. 7 OLLVM control flow flatteningobfuscation

将汇编指令转换为高级语言时,OLLVM 控制流实现的混淆都能直接转换为高级语言. 而转换多函数混合虚假控制流混淆过的汇编指令,还原的高级语言显示所有函数的代码混合在了一起,这对于逆向分析显然是灾难性的. 转换多函数

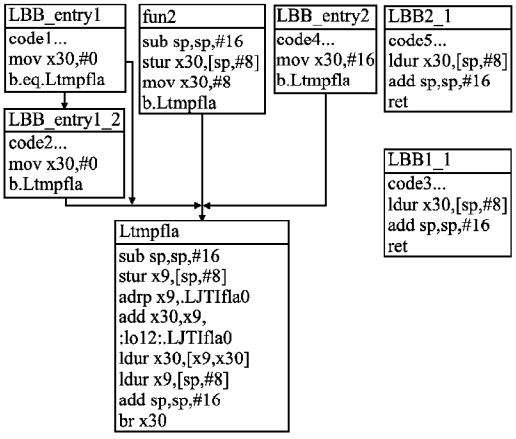


图 8 多函数混合控制流扁平化混淆程序流程

Fig. 8 Multi-function mixed control flowflattening obfuscation

混合控制流扁平化混淆过的汇编指令时,因为分发器使用的 br 指令无法识别出跳转目标,所以无法转换成高级语言. 因此,两种多函数混合控制流混淆都会将多个函数的基本块混淆在一起,攻击者若希望还原多函数混合控制流混淆的某个函数,须得先从大量的基本块中识别出哪些基本块是属于哪

个函数,并且还无法转换成高级语言,代码的可读性更差,混淆效果更好.

3.3 相似性对比分析

采用代码相似度衡量两段或多段代码在结构、逻辑、语法等方面的相似程度.针对混淆后的代码与原始代码,从他们的控制流结构(如循环、条件判断等结构的相似性)、数据结构的使用(如数组、链表等数据结构在代码中的布局和操作的相似性)、操作符和函数调用的相似性等多个维度进行评估,代码相似度越低,在结构与逻辑方面的混淆效果越好,逆向工程难度越高.

表 2　不同混淆方法下的与源程序跳转相似度

Table 2　Jumps similarity of source program under different obfuscation

混淆方法	平均值	标准差
OLLVM 控制流扁平化	35.65	21.87
嵌套 switch 混淆	20.58	23.17
嵌套 switch 混淆 + 入度混淆	14.43	19.2
多函数混合虚假控制流(本文)	5.27	4.63
多函数混合控制流扁平化(本文)	12.95	12.41

BinDiff 是由谷歌开发的一款二进制文件分析与对比工具.其工作原理是比较两个程序的控制流图(CFG),对于每个函数,其控制流图可表示为 $G=(V,E)$,这里的 V 表示基本块集合,而 E 表示控制流边集合.该工具采用图匹配算法来对两个控制流图进行比较,通过计算找出基本块和控制流边之间的匹配关系.其中,跳转相似度的核心在于比较两个控制流图中跳转边的匹配程度.具体而言,对于来自第一个控制流图中的每一个跳转边 $e_1 \in E_1$,都会在 E_2 中寻找与之最为接近的匹配边 e_2.最终,相似度的计算是依据匹配的基本块和控制流边在各自集合中所占的比例来得出的.使用 BinDiff 在图形化界面选择未混淆的源程序文件,以及对应混淆后的程序文件,能获得基本块相似度、指令相似度、跳转相似度等.实验关注于控制流混淆,因此选取各种混淆方法实施后的程序与源程序的跳转相似度进行对比.实验对象是 3.1 节混淆后的程序文件,实验中针对数据集 1 的不同混淆方法下的与源程序跳转相似度如表 2 所示.可以看出,对于混淆低时间复杂度程序,多函数混合控制流扁平化的跳转指令相似度相对于嵌套 switch + 入度混淆略有下降,而多函数混合虚假控制流有明显的提高,进一步约下降 63.48%.

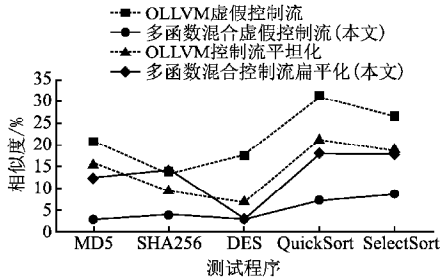


图 9　混淆程序与源程序相似度对比

Fig. 9　Comparison of similarity between obfuscated program and original program

针对数据集 2 的混淆程序与源程序的相似度情况如图 9 所示.可以看出,混淆较高时间复杂度程序时,多函数混合虚假控制流对比 OLLVM 虚假控制流与源程序的控制流相似度也具有明显的效果.而多函数混合控制流扁平化与 OLLVM 控制流扁平化相比相似度总体要低一些,但也存在样本效果一般,表明其在混淆较高时间复杂度程序时,对于部分样本还有待改进.实验结果表明,基于汇编实现的控制流混淆相比当前现有的混淆,拥有更低的代码相似度,在隐藏原始代码结构和逻辑方面效果更好.

4　结　论

传统的控制流混淆实现方式存在仅能在单个函数内混淆的问题.本文基于汇编实现的混淆方案,因其语法的灵活性,可以拥有更好的混淆方式,提出了多函数混合虚假控制流混淆和多函数混合控制流扁平化混淆方法,实现了源程序的控制流程混淆,隐藏了基本块所属的函数,增加了程序流程的复杂度,提高了攻击者还源程序的难度.实验结果表明了该方法混淆的有效性,在虚假控制流算法中显著降低代码相似度提升混淆效果,并在控制流扁平化混淆方面相对于其它同类混淆具有很好的速度优势.不足之处在于,使用汇编只能局限在单个汇编文件所有函数混淆在一起,下一步研究可在可执行程序文件中进行混淆,达到可将整个可执行文件的函数混淆在一起的效果.

References:

[1] Conti M, Vinod P, Vitella A. Obfuscation detection in android applications using deep learning[J]. Journal of Information Security and Applications, 2022, 70: 103311, doi: 10.1016/j.jisa.2022.103311.

[2] Skolka P, Staicu C A, Pradel M. Anything to hide? studying minified and obfuscated code in the web[C]//Proceedings of the World Wide Web Conference, 2019: 1735-1746.

[3] Fyrbiak M, Rokicki S, Bissantz N, et al. Hybrid obfuscation to protect against disclosure attacks on embedded microprocessors[J]. IEEE Transactions on Computers, 2018, 67(3): 307-321.

[4] Cao Shangtong, He Ningyu, Guo Yao, et al. WASMixer: binary obfuscation for WebAssembly[C]//Proceedings of the 29th European Symposium on Research in Computer Security, 2024: 88-109.

[5] SHA Z H, SHU H, WU C G, et al. Deep control flow obfuscation model based on callback function[J]. Journal of Software, 2022, 33(5): 1833-1848.

[6] Yoshinaka Y, Kita K, Takemasa J, et al. Programmable name obfuscation framework for controlling privacy and performance on CCN [J]. IEEE Transactions on Network and Service Management, 2023, 20(3): 2460-2474.

[7] Huang Weihao, Meng Guozhu, Lin Chaoyang, et al. Are our clone detectors good enough? An empirical study of code effects by obfuscation [J]. Cybersecurity, 2023, 6(1): 1-19.

[8] LU H, GUO R S, JIN C J, et al. Automated anti-obfuscation technology based on capstone and flow-sensitive concolic execution [J]. Journal of Software, 2023, 34(8): 3745-3756.

[9] Collberg C, Thomborson C, Low D. A taxonomy of obfuscating transformations[R]. New Zealand: Department of Computer Science, the University of Auckland, 1997.

[10] Barak B, Goldreich O, Impagliazzo R, et al. On the (Im) possibili-

ty of obfuscating programs[J]. Journal of the ACM,2012,59(2):1-48.

[11] LUO K,JIN D H,GONG Y Z. Research on static detection method of fortran memory leak[J]. Journal of Chinese Computer Systems,2024,45(7):1778-1786.

[12] Ahmed H,Hyder M F,Haque M F,et al. Exploring compiler optimization space for control flow obfuscation[J]. Computers and Security,2024,139:103704.

[13] Tran N P,Nguyen D T,Le H H,et al. An efficient algorithm to extract control flow-based features for IoT malware detection[J]. Computer Journal,2021,64(4):599-609.

[14] Fass A,Backes M,Stock B S. HidenoSeek:camouflaging malicious Javascript in benign asts[C]//Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security,2019:1899-1913.

[15] Golovko I,Sacenko O,Vizhevskyi P,et al. Obfuscation technologies of high-level source code using artificial intelligence[C]//Proceedings of the 1st International Workshop on Intelligent and CyberPhysical Systems,2024:324-338.

[16] Baron A,Granot I,Yosef R,et al. Understanding logical expressions with negations:Its complicated[C]//Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering,2024:303-312.

[17] Ullah F,Gross T R. Detecting anomalies in concurrent programs based on dynamic control flow changes[C]//Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium Workshops,2016:308-317.

[18] Huidobro C B,Cordero D,Cubillos C,et al. Obfuscation procedure based on the insertion of the dead code in the crypter by binary search[C]//Proceedings of the 7th International Conference on Computers Communications and Control,2018:183-192.

[19] Pan Hongyu,Fang Yong,Guo Wenbo,et al. Few-shot graph classification on cross-site scripting attacks detection[J]. Computers and Security,2024,140:103749,doi:10.1016/j.cose.2024.103749.

[20] LI C Y,HUANG T B,CHEN X R,et al. Control flow obfuscation scheme for LLVM intermediate languages[J]. Computer Engineering and Applications,2023,59(8):263-269.

[21] Hoffmann M,Paar C. Stealthy opaque predicates in hardware obfuscating constant expressions at negligible overhead[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems,2018,2018(2):277-297.

[22] Jens V d B,Bart C,Bjorn D S. Flexible software protection[J]. Computers and Security,2022,116:102636,doi:10.48550/arXiv.2012.12603.

[23] Blazy S,Trieu A. Formal verification of control-flow graph flattening[C]//Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs,2016:176-187.

[24] Busi M,Degano P,Galletta L. Control-flow flattening preserves the constant-time policy[C]//Proceedings of the 4th Italian Conference on Cyber Security,2020:82-92.

[25] Johansson B,Lantz P,Liljenstam M. Lightweight dispatcher constructions for control flow flattening[C]//Proceedings of the 7th Software Security,Protection,and Reverse Engineering Workshop,2017:1-12.

[26] Dong Weiyu,Lin Jian,Chang Rui,et al. CaDeCFF:compiler-agnostic deobfuscator of control flow flattening[C]//Proceedings of the 13th Asia-Pacific Symposium on Internetware,2022:282-291.

[27] Liang Zheheng,Li Wenlin,Guo Jing,et al. A parameterized flattening control flow based obfuscation algorithm with opaque predicate for reduplicate obfuscation[C]//Proceedings of the 5th International Conference on Progress in Informatics and Computing,2017:372-378.

[28] Lü Di,Zhao Liang,Chen Bin. Research based on LLVM code obfuscation technology[C]//Proceedings of the 3rd International Conference on Industrial IoT,Big Data and Supply Chain,2022:163-167.

[29] Yang Xueyi,Zhang Lingchen,Ma Cunqing,et al. Android control flow obfuscation based on dynamic entry points modification[C]//Proceedings of the 22nd International Conference on Control Systems and Computer Science,2019:296-303.

[30] Wang Chang,Zhang Zhaolong,Jia Xiaoqi,et al. Binary obfuscation based reassemble[C]//Proceedings of the 13th International Conference on Malicious and Unwanted Software,2018:153-160.

[31] Mumtaz Z,Afzal M,Iqbal W,et al. Enhanced metamorphic techniques-a case study against havex malware[J]. IEEE Access,2021,9:112069-112080.

[32] Zhao Yujie,Tang Zhangyong,Ye Guixin,et al. Compile-time code virtualization for android applications[J]. Computers and Security,2020,94:101821,doi:10.1016/j.cose.2020.101821.

[33] Yu Lei,Duan Yucong. A smart obfuscation approach to protect software in cloud[J]. Computers,Materials and Continua,2023,76(3):3949-3965.

[34] Junod P,Rinaldini J,Wehrli J,et al. Obfuscator-LLVM--software protection for the masses[C]//Proceedings of the IEEE/ACM 1st International Workshop on Software Protection,2015:3-9.

[35] HikariObfuscator. Hikari[EB/OL]. https://github.com/Hikari-Obfuscator/Hikari,2023-09-25.

附中文参考文献:

[5] 沙子涵,舒 辉,武成岗,等. 基于回调函数的控制流深度模糊模型[J]. 软件学报,2022,33(5):1833-1848.

[8] 鲁 辉,郭润生,金成杰,等. 基于 Capstone 和流敏感混合执行的自动化反混淆技术[J]. 软件学报,2023,34(8):3745-3756.

[11] 罗 坤,金大海,宫云战. Fortran 内存泄漏静态检测方法研究[J]. 小型微型计算机系统,2024,45(7):1778-1786.

[20] 李成扬,黄天波,陈夏润,等. LLVM 中间语言的控制流混淆方案[J]. 计算机工程与应用,2023,59(8):263-269.