计算机应用
Journal of Computer Applications
ISSN 1001-9081,CN 51-1307/TP

《计算机应用》网络首发论文

题目：　改进的基于底层虚拟机混淆器的指令混淆框架
作者：　王雅仪，刘琛，黄天波，文伟平
收稿日期：　2022-01-06
网络首发日期：　2022-06-15
引用格式：　王雅仪，刘琛，黄天波，文伟平．改进的基于底层虚拟机混淆器的指令混淆框架[J/OL]．计算机应用．
https://kns.cnki.net/kcms/detail/51.1307.TP.20220613.1756.006.html

Journal of Computer Applications ISSN 1001-9081
计算机应用 CODEN JYIIDU http://www.joca.cn

doi:10.11772/j.issn.1001-9081.2021122234

改进的基于底层虚拟机混淆器的指令混淆框架

王雅仪，刘琛，黄天波，文伟平*

(北京大学 软件与微电子学院，北京 102600)

(*通信作者电子邮箱 weipingwen@pku.edu.cn)

摘 要: 针对底层虚拟机混淆器（ollvm）在指令混淆层面只支持指令替换一种算法，且仅支持 5 种运算符和 13 种替换方案的问题，设计了一种改进版的指令混淆框架（InsObf），以加强底层虚拟机混淆器指令层面的混淆效果。InsObf 包含指令加花和指令替换，指令加花首先对基本块的指令进行依赖分析，然后插入叠加跳转和虚假循环两种花指令；指令替换在 ollvm 的基础上，拓展至 13 种运算符，共计 52 种指令替换方案。在底层虚拟机（LLVM）上实现了框架原型，通过实验表明，与 ollvm 指令替换功能相比，InsObf 在时间开销增长约 10 个百分点，空间开销增长约 20 个百分点的情况下，圈复杂度和抗逆向能力均可提高 4 倍；与同样基于 ollvm 改进的 Armariris 和 Hikari 相比，InsObf 在同一量级时空开销下，可以提供更高强度的代码复杂度。因此，InsObf 可提供指令层级的有效保护。

关键词: 软件保护；代码混淆；指令混淆；底层虚拟机混淆器；指令加花；指令替换

中图分类号: TP311 **文献标志码**: **A**

Improved instruction obfuscation framework based on Obfuscator Low Level Virtual Machine

WANG Yayi, LIU Chen, HUANG Tianbo, WEN Weiping*

(School of Software and Microelectronics, Peking University, Beijing 102600, China)

Abstract: Focused on the issue that only instruction substitution with 5 operators and 13 substitution schemes was supported in Obfuscator Low Level Virtual Machine(ollvm) at the instruction obfuscation level, an improved instruction obfuscation framework(InsObf) was proposed. InsObf, including junk insertion and instruction substitution, can enhance the obfuscation at the instruction level based on the ollvm. For junk insertion, firstly, the dependency of the instruction inside the basic block of the program was analyzed, and then two kinds of junk instructions: multiple jump instructions and bogus loop instructions, were inserted to disrupt the structure of the basic block. As for instruction substitution, it was expanded to 13 operators, with 52 instruction substitution schemes. The framework prototype was implemented on Low Level Virtual Machine (LLVM). The experimental results showed that cyclomatic complexity and resilience could be increased by four times, with a time cost of about 10 percentage points and a space cost of about 20 percentage points, compared to ollvm. Moreover, InsObf can provide higher intensity code complexity at the same amount of space and time cost than Armariris and Hikari, which are also based on improved ollvm. Therefore, InsObf can provide adequate protection at the instruction level.

Keywords: software protection; code obfuscation; instruction obfuscation; Obfuscator Low Level Virtual Machine (OLLVM); junk insertion; instruction substitution

0 引言

从早期的个人计算机发展至今，软件的攻击与保护之争从未停止过，在软件种类和平台呈现百花齐放的当下，如何有效的保护软件，始终是一个值得探究的问题。

作为软件保护技术之一的代码混淆因使用上的灵活性和较低成本带来了可观的安全性，在工业界和学术界得到不断关注和发展。软件攻击的目的在于理解或部分理解原生代码，从而添加或者移除代码逻辑，以满足攻击者自身的利益诉求。代码混淆预期的目标是增大原生程序的理解难度，包括阻止自动化工具的反编译，增加原生程序本身的复杂度等。Collberg[1]在 1997 年首次对代码混淆技术进行了明确的分类，

收稿日期: 2022-01-06; 修回日期:2022-05-20; 录用日期: 2022-05-24。
基金项目:北京大学横向课题(代码混淆保护研究)(2020001763)
作者简介: 王雅仪(1998—)，女，四川成都人，硕士研究生，主要研究方向：代码混淆、漏洞挖掘；刘琛(1999—)，男，河南新乡人，硕士研究生，主要研究方向：代码混淆、漏洞挖掘；黄天波(1997—)，男，河北邯郸人，硕士研究生，主要研究方向：网络空间安全、恶意代码检测、代码混淆；文伟平(1976—)，男，湖南益阳人，教授，博士，主要研究方向：系统与网络安全，大数据与云安全，智能计算安全。

包括布局混淆、数据混淆、控制流混淆和预防性混淆。此后，又出现了各种形式的代码混淆研究，涵盖了软件的不同阶段、代码的不同部分，如在实现[2]或编译时[3]，在源代码[4]、二进制代码[5,6]或中间代码[7]级别。

不同于文献[1]的分类方式，本文按照混淆的粒度，将代码混淆方法分为函数混淆、基本块混淆和指令混淆。指令作为高级程序的底层组织，面向指令层级的混淆可以做一些更细致精准的操作。如果攻击者能够破解程序，首先面对的便是破解后的程序指令，因此指令混淆对此阶段的代码保护具有重要作用。这也正是本文的工作聚焦于指令混淆的原因。

底层虚拟机混淆器[7]（Obfuscator Low Level Virtual Machine, ollvm）是一款经典的开源混淆项目，它支持指令替换、虚假控制流、控制流扁平化等功能。然而，在指令混淆层面，它只提供指令替换一种功能，并不支持指令加花。并且，指令替换只支持 5 种运算符，共计 13 种替换方案。

针对 ollvm 的不足，本文设计了一种改进的指令混淆框架，命名为 InsObf，旨在对其中的指令混淆进行补充。已实现的优化工作包括：拓展指令替换功能至 13 种运算符，共计 52 种指令替换方案；增加指令加花的功能，包含叠加跳转和虚假循环两种指令加花方案。通过实验证明，本文提出的指令混淆框架的效果相较于 ollvm，在时间开销增长约 10 个百分点，空间开销增长约 20 个百分点的情况下，圈复杂度和抗逆向能力均提升了 4 倍；在同基于 ollvm 改进的 Armariris 和 Hikari 相比，本文提出的框架在同一量级时空开销下，圈复杂度增长最高，达到了混淆前的 6 倍，并将指令相似度降低到 23.5%，可以提供更高强度的代码复杂度。

1 背景知识

1.1 指令混淆

指令混淆是针对程序中的指令进行添加、修改等操作的一种混淆方法，包括指令加花[8-10]、指令替换[7,11]、指令加密[12,13]和指令重组[14]。向指令流[15]中添加死代码[16]、冗余操作数[17]、新段[18]等，都属于这类转换。相较于指令加密和指令重组带来的时空开销高、程序健壮性差等问题，指令加花和指令替换相对更为成熟，且使用场景更为广泛，特别地，在 ollvm 仅实现了指令替换，存在指令加花的技术空白，因此本文研究一方面聚焦于填补 ollvm 指令加花的空白，另一方面着眼于进一步加强目前已有的指令替换方案。下文将介绍指令加花和指令替换的相关研究工作。

1.1.1 指令加花

指令加花的目的是向程序中添加一些冗余信息，这些信息不会改变程序预期的行为，但可增大指令的理解难度，扰乱指令的原生特征。

常见的方式是插入一些不会被执行或执行后不影响程序预期执行结果的指令。Zhang 等[19]通过引入动态不透明谓词在两个基本块之间插入一个冗余的基本块，从而扰乱原始基本块间的依赖关系；Cho 等[17]提出插入无关的循环变量和冗余的操作数，复杂化原有的计算表达式；文献[20]通过插入虚假的调用，并在后续指令中使用调用结果来干扰程序的分析；Peng 等[21]提出在循环条件前利用不透明谓词插入指令，复杂化程序结构。也有一些研究[22-24]采用插入 NOP 指令的方式。尽管 NOP 指令不执行任何操作，但其随机性的存在可以使代码片段在内存中位置的预测变得更加困难。

另一类是在指令的特定位置插入垃圾指令来阻止反汇编工具的处理。Linn 等[25]通过在指令流的特定位置插入不完整的字节来引入反汇编错误；文献[26]针对 ARM 指令进行混淆处理，通过添加无效指令和构造随机数据，实现对线性扫描反汇编同步的延迟，以阻止自动化反编译器对指令的约简。

1.1.2 指令替换

指令替换是指用功能相同但更复杂的指令序列替换源程序中的指令运算符或操作数，使得程序中的指令更难理解。例如，指令中的操作数可以用一段代码替换[20]，拆分为多个变量[17,27]，反之，也可以把多个变量合并在一起。Rajba 等[28]使用按位与、左移和异或运算符对加法运算符进行混淆。对布尔变量进行分割[29]、将常量替换为一串运算表达式[30]或者利用代码动态生成[29]也是常见的替换方案。Kanzaki 等[31,32]提出首先通过代码自修改机制将原始指令替换成另一套伪指令集，然后在代码运行时自动恢复成原始代码片段，使得逆向人员只能拿到替换后的伪指令集。Darwish 等[33]针对汇编语言，指出系统中任何操作都可以通过不同方式实现，因此可以从原指令的可能替换组中随机生成等价的代码替换。

1.2 混淆工具

1.2.1 ollvm

底层虚拟机（Low Level Virtual Machine, LLVM）[34]为代码混淆技术跨平台使用提供了可能。ollvm 是第一个基于 LLVM 的开源混淆项目，借助于 LLVM 的跨平台性，ollvm 适用于任何 LLVM 支持的编程语言，例如 C/C++、Objective-C、Fortran 等。

在指令替换部分，ollvm 支持整数 ADD、SUB 和布尔运算符 AND、OR 和 XOR。表 1 是具体支持的替换方案。

1.2.2 Armariris

Armariris[35]是由上海交通大学密码与计算机安全实验室维护的 LLVM 混淆框架，目前提供了字符串加密、控制流扁平化和指令替换三个功能。项目本身相较于 ollvm 的改进在于字符串加密方案的补充，虽然仅针对源代码中的常量字

符串提供异或加密的方式，但是为基于 LLVM 的字符串加密提供了新的可能性。

表1 ollvm 指令替换方案

Tab. 1 Instruction substitution in ollvm

运算符	等价指令序列
a = b + c	a = b – (-c)
	a = -(-b + (-c))
	r = rand(), a = b + r, a = a + c, a = a - r
	r = rand(), a = b – r, a = a + c, a = a + r
a = b - c	a = b + (-c)
	r = rand(), a = b + r, a = a – c, a = a - r
	r = rand(), a = b – r, a = a – c, a = a + r
a = b & c	a = (b ^ !c) & b
a = b && c	a = !(!b | !c) && (r | !r)
a = b | c	a = (b & c) | (b ^ c)
a = b || c	a = [(!b && r) || (c && !r) ^ (!c && r) || (c && !r)] || [!(!b || !c) && (r ||!r)]
a = b ^ c	a = (!b & c) | (b & !c)
	a = (!b && r || b && !r) ^ (!c && r || c && !r)

1.2.3 Hikari

Hikari[36]是一个基于 ollvm 进行优化的二进制加固工具，主要改进了控制流混淆和字符串混淆等功能，未涉及指令层级的混淆优化。其中，控制流混淆的改进，如直接跳转间接化，通过将程序中跳转指令的目标地址由明文改为数组匹配的方式，实现间接跳转。在字符串层面，不同于 Armariris 的字符串处理方式，加密是在函数内部完成而非全局处理，且只有运行过的函数才会将解密过的字符串存放在内存中。

2 指令混淆框架

InsObf 指令混淆框架的工作原理如图 1 所示，可以被分为三层：源代码层、LLVM 中间表示（以下简称 LLVM-IR）层和目标平台层。源代码层包含待混淆的源程序。在 LLVM-IR 层中，源程序通过 LLVM 前端编译器（例如 Clang[37]）转换为 LLVM-IR 文件，然后由指令混淆生成混淆后的 LLVM-IR 文件。根据预期的混淆强度，可以选择继续混淆或使用 LLVM 后端生成基于特定目标平台的可执行程序。

InsObf 指令混淆框架包含两种混淆方法：指令加花和指令替换。在混淆的过程中，InsObf 首先通过指令加花方法来混淆 LLVM-IR 文件，插入一些冗余指令组成的基本块。然后用指令替换方法处理操作符和操作数，进一步加强安全性。

指令加花通过结合指令的数据依赖分析，随机选择花指令插入的位置，保证混淆结果的多样性和随机性。插入的叠加跳转指令和原始基本块内的指令高度相似但不同，虚假循环指令使用原始基本块中的变量作为循环控制条件，两种形式的花指令在充分利用源程序中指令信息的同时，能够有效干扰逆向人员的分析，加大分析的难度。指令替换可以从生成的多个等价表达式中随机选择替换方案，来实现对原有语义的隐藏，给程序带来多样化。当指令替换同指令加花进行效果叠加后，这种隐藏能力被最大程度的发挥，既作用于源程序的真实指令语义，破坏反混淆器既定的模式识别，又作用于添加的花指令，进一步隐藏真实指令和源程序指令之间的相似度，使得通过利用程序地址空间的攻击更难实现。

2.1 指令加花

本文提出了两种指令加花方法。一种是叠加跳转，另一种是虚假循环。文献[19,38]将垃圾指令添加到基本块之间，然而基本块内部指令在语义上更有连贯性，相应地，直接破坏基本块内部指令结构将给程序的理解上带来更大的难度。因此，本文在原有基本块内部添加花指令，同时考虑到程序的三种基本结构中，相较于顺序结构，分支和循环结构包含更多程序的控制流信息，往往是攻击者关注的重点，大量分支和循环结构的存在会加大分析的难度。因此，在花指令的设计形式上，构造叠加跳转和虚假循环的形式。为了构造形式的完整性，本文将构造的花指令以新的基本块的形式进行添加，即首先对原有的基本块进行拆分，然后在拆分后的基本块间插入构造的花指令基本块，并通过不透明谓词进行有效的整合。

2.1.1 叠加跳转

定义 1 叠加跳转。记程序的基本块集合为 *originBBs*，对" $b\hat{\imath}$ *originBBs*，将其分割为 b_1 和 b_2，并根据分割后的基本块生成新的虚假块集合 *bogusBBs*，在保证程序语义的前提下，构造 *bogusBBs* 内虚假块与 b_1、b_2 的跳转关系，形成具有层级的基本块关系。

添加叠加跳转指令的过程中有如下两点需要讨论说明：

（1）基本块内的指令之间往往存在数据依赖关系，如何在不破坏数据依赖关系的情况下对基本块进行分割？

（2）插入基本块给程序的运行带来了新的开销，如何尽可能地降低增加的时空开销？

为了解决问题(1)，本文设计了一种数据依赖分析算法，详细的步骤如算法 1 所示。在对基本块进行分割之前，需要先分析基本块指令间的数据依赖关系，进而将原本单一基本块内的所有指令划分为几个独立的指令集合。每个集合内的指令是高度内聚的，即具有强相关性；不同集合中的指令无数据依赖关系。极端情况下，整个基本块内的指令均存在强相关性，此时，一个基本块即是一个集合，将不对基本块做任何处理。

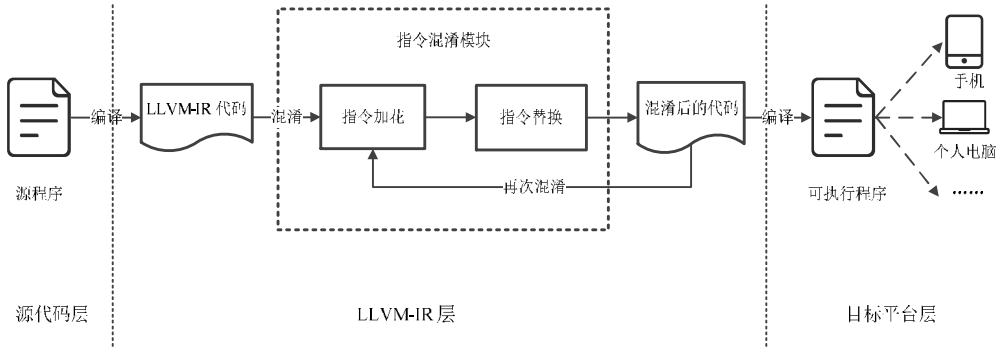

图1 InsObf 系统整体架构

Fig. 1 System architecture diagram of InsObf

算法 1：Dependency Analysis Algorithm

输入：*instList*←instruction set in the basic block

输出：*resStack*←instruction dependency analysis results

```
while instList.size > 0 do
    inst ← instList.getLast()
    stack.add(inst); queue.add(inst)
    while queue.size > 0 do
        toAnalyzeInst ← queue.pop()
        depInst← searching forwards for instructions that toAnalyzeInst depends on in the instList
        stack.add(depInst); queue.add(depInst)
        instList.remove(depInst)
    end while
    resStack.add(stack)
end while
```

算法的输入为基本块内所有的 LLVM-IR 指令，输出为存储指令依赖分析结果的嵌套堆栈集合，集合内的每一个子集也是一个栈，用于存储一组相互依赖的指令。算法实现的中间过程需要借助栈和队列来完成，前者用于存储相关依赖指令，最后作为子集存储到依赖分析结果集中；后者用于存储当前指令搜索过程中涉及到的直接依赖指令和间接依赖指令，并不断向外扩展，直到没有新的依赖指令。针对每一条新加入的指令，均需要备份并对其依赖关系进行重新检查，通过队列实现每一条加入的新指令，其依赖的指令也会同步加入指令堆栈中。在指令依赖关系分析过程中，若出现所依赖的指令已经添加到其他的指令集合中了，则生成原生指令的备份存储到新的集合中，以避免数据流的断裂问题。在基本块分割时，则从指令依赖分析结果集之间随机选择一个位置，如原始的基本块根据算法 1 的分析结果被分割成 *startBB* 和 *mainBB*，效果如图 2 所示。

问题（2），因受限于算法本身花指令的设计，在空间开销上未作出较大优化。对于时间开销，本文使用 Palsberg[39] 引入的动态不透明谓词，将花指令添加到运行时不可达的位置来尽可能降低时间上的开销。如在图 2 的基础上将 *mainBB* 用作原型副本，复制得到新的基本块，并随机替换其中一些指令（不需要等价），从而获得 $bogusBB_1$，$bogusBB_2$ 等。这些新被创建的基本块和 *mainBB* 高度相似但不同，在充分利用程序信息的同时起到混淆视听的效果。然后，这些创建的新基本块将通过不透明谓词控制，插入到 *startBB* 和 *mainBB* 之间。图 3 为插入叠加跳转指令后的控制流示意图。另外，创建的新基本块的数量可以由用户指定，针对不同规模的程序提供定制化的功能，从而提供更高的灵活性，默认数值为 2。

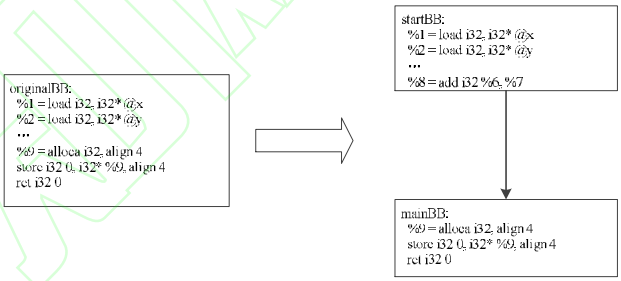

图2 拆分基本块示意图

Fig. 2 Diagram of splitting basic blocks

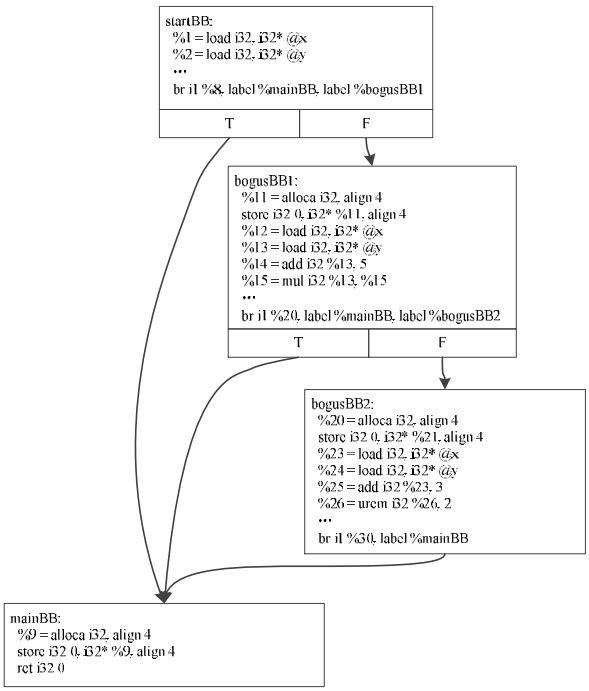

图3 插入叠加跳转指令后的控制流示意图

Fig. 3 Control flow diagram after inserting the multiple jump instruction

2.1.2 虚假循环

定义 2 虚假循环。构造循环对应的基本块集合 *loopBBs*，记程序的基本块集合为 *originBBs*，对 "$b \in originBBs$，将其分割为基本块 b_1 和 b_2，在保证程序语义的前提下，通过构造跳转关系，在 b_1 和 b_2 中嵌入 *loopBBs* 等虚假块。

为了增加混淆效果的多样性，可以由用户指定混淆概率，对程序中的所有基本块以设定的概率依次决定是否要添加虚假循环，默认混淆概率为 80%。

以一段用 C 语言编写的循环程序为例，为了叙述上的简便性，这里的 *loopBBs* 为程序对应的基本块集合，*loopBBs* = {*entryBB*, *loopCondition*, *loopBody*}。*entryBB* 对应于第 1-2 行循环语句的前序声明对应的基本块，*loopCondition* 为第 3 行循环条件判断对应的基本块，*loopBody* 为第 4 行内部循环体对应的基本块。

```
1. int i = 0;
2. int bound = 10;
3. while(i < bound){
4.       i += 1;
5. }
```

图 4 为插入虚假循环指令后的控制流示意图。在 *startBB* 和 *bogusBB* 之间插入 *loopBBs*，如图 4 中深色区域所示。值得注意的是，循环指令的设计具有高度灵活性，在实际应用时可以使用与源程序有着数据相关性、更加复杂的形式来代替。例如，在循环体内添加源程序中相关变量的运算操作，在循环条件中引入源程序中的变量等，从而进一步和源程序产生数据关联。

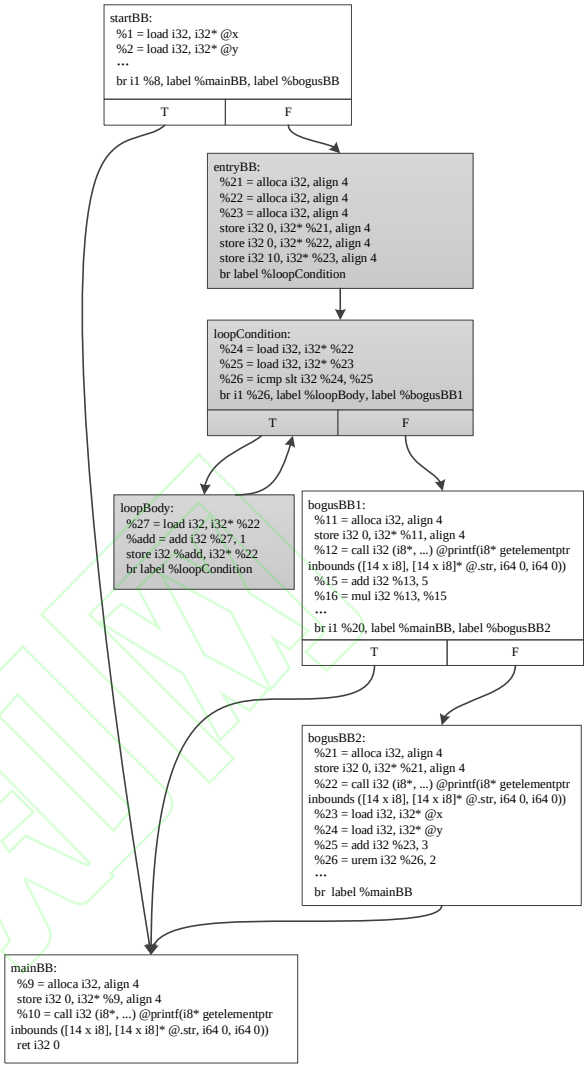

图4 插入虚假循环指令后的控制流示意图

Fig. 4 Control flow diagram after inserting bogus loop instruction

2.2 指令替换

指令替换混淆是指令的等效替换，即用语义相同但形式不同的一条或多条指令来替换。在 ollvm 中，指令替换仅支持 5 种运算符，共计 13 种替换方案。本文基于 ollvm，在支持的指令操作符数量和替换方案上做了补充，共计 13 种运算符，52 种替换方案，具体的运算符和替换方案如表 2 所示。在实际应用时，指令替换可以随机选择替换方案，给程序带来多样性。

在 ollvm 的指令替换方案中，主要关注的是指令中操作符的替换，在操作数方面仅提供了增加冗余操作数的替换方案，考虑到混淆优化处于 LLVM-IR 层，在后端编译时，如果使用高级别的优化选项，例如-O2 和-O3，ollvm 中提出的类似于 $a = b + c$ 替换为 $a = b - (-c)$的方案可以被编译器优化恢复。因此，除 ollvm 中提供的替换思路，本文还额外增加了两种更高强度的方法：数据拆分替换和循环替换，来对指令中的操作数进行破坏，消除后端编译器优化的影响，目前这两种方法只支持整数类型的变换。

表2 ollvm 和 InsObf 支持的运算符和替换方案数

Tab. 2 The number of operators and substitution schemes in ollvm and InsObf

运算符	ollvm 支持的替换方案数	InsObf 支持的替换方案数
ADD	4	5
FADD	不支持	4
SUB	3	5
FSUB	不支持	4
AND	2	5
OR	2	5
XOR	2	5
MUL	不支持	3
FMUL	不支持	3
REM	不支持	3
SHL	不支持	4
LSHR	不支持	3
ASHR	不支持	3

定义 3　数据拆分替换。程序的运算操作集记为 O，O 所涉及的常量和变量集合记为 D，Δx 记为运算中产生的进位或借位信息。对"$N \in D$，将 N 拆分为高位数据 N_{high} 和低位数据 N_{low}，"$o \in O$ 转变为针对 N_{high} 和 N_{low} 的操作，并同步更新 Δx 的数值，默认为 0。

以 32 位整数的 ADD 指令为例，数据拆分替换会将该整数拆分为高 16 位和低 16 位，具体的方案如图 5 所示。

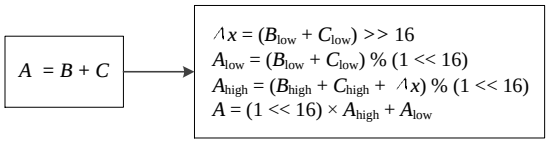

图5　ADD 指令数据拆分替换示例

Fig. 5　Example of data split substitution for ADD instruction

在数据拆分替换中，因为在 LLVM-IR 层 8 位整数不能被拆分为更低位的数字，因此只关注 16 位、32 位、64 位和 128 位的整数。

考虑到数据拆分替换后的混淆特征较为明显，因此进一步提出循环替换算法，对操作符进行降级处理，例如将左移指令转变为乘法指令、乘法指令转变为加法指令等。

定义 4　循环替换。记操作的阈值为 N，操作 $Op1$ 为将程序中的左移指令转变为循环形式的乘法指令，操作 $Op2$ 为将乘法指令转变为循环形式的加法指令。使用 $Op1$ 或 $Op2$ 对"$i \in \{inst \mid inst \in \{MUL, SHL\}\}$ 进行处理，若循环的处理次数超过阈值 N，阈值内的循环逻辑保留，超出的部分仍使用之前的指令形式。

以操作 $Op2$（乘法指令转变为循环形式的加法指令）为例，循环替换的过程如图 6 所示。需要注意的是，为了保证程序运行的效率，阈值 N 可由用户设定。如用户未设定，在程序首次使用时，统计程序中所有乘法指令转变为加法指令的频次分布，取中位数作为默认值。

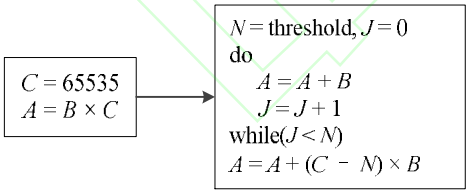

图6　乘法指令循环替换示例

Fig. 6　Example of loop substitution for MUL instruction

3　实验评估

本文选择了 20 个典型的 C++程序[40]为实验数据集，包括数组、树、图等数据结构和搜索、排序、加密等算法。实验环境为 8G 内存的 Ubuntu 18.04 64 位机，实验基于 LLVM 10 release 版进行。为了验证本文提出的指令混淆框架 InsObf 的混淆效果，分别从指令混淆的优化效果和不同混淆方法的对比上进行实验分析。其中，InsObf 指令混淆框架的指令加花子模块实现，记为 InsObf-junk；InsObf 指令混淆框架的指令替换子模块实现，记为 InsObf-sub；InsObf 指令混淆框架整体实现，记为 InsObf-mix；ollvm 的指令替换模块，记为 ollvm-sub。

3.1　基于 ollvm 的指令混淆优化效果

指令混淆优化效果从混淆效果分析和性能分析两方面进行评估。对于混淆效果分析，采用 Collberg[1]提出的定量的混淆指标：混淆强度[14]（potency）和抗逆向[14]（resilience）；对于性能分析，采用时空开销作为测试指标。

3.1.1　混淆强度

圈复杂度（Cyclomatic Complexity，简称 CC），由 Thomas J. McCabe[41]1976 年提出，用来描述程序的复杂性。混淆强度用来衡量自然人理解程序的难度[1]，常用 CC 进行评估。在数学上，程序控制流图 G 的圈复杂度 $V(G)$可以定义为：

$$V(G) = E - N + 2$$

其中 E 是 G 中边的数量，N 是节点的数量。

目前公开的圈复杂度工具需依据源码进行数值处理，因此本文使用 llvm-cbe[42]从混淆前后的可执行文件中获取到对应的 C 源代码，然后使用 lizard[43]得到混淆前后的圈复杂度。结果如表 3 所示。

表3　InsObf 和 ollvm 圈复杂度分析

Tab. 3　Cyclomatic complexity of InsObf and ollvm

混淆方法	圈复杂度增长率/%
ollvm -sub	128.8
InsObf -sub	231.7
InsObf -junk	300.6
InsObf -mix	607.2

从表 3 的结果可以看出，同 ollvm-sub 相比，InsObf-sub 的圈复杂度提升近 1 倍，印证了新增的替换模式能够有效增大程序的复杂性。InsObf-junk 通过构造基本块和跳转关系，对源程序的控制流进行了破坏，从而实现圈复杂度相较于 ollvm-sub 提升近 2 倍。而指令混淆框架的整体实现 InsObf-mix 的圈复杂度相较于 ollvm-sub 更是提升了近 4 倍，有效加大了攻击者分析和理解程序的难度。

3.1.2　抗逆向

混淆后的抗逆向能力，表示混淆后的程序抵御自动去混淆攻击的能力[1]，为了有效的量化该指标，本文进一步做了如下定义：

定义 5　抗逆向。针对混淆添加的程序代码，使用去混淆工具无法移除的代码比重。比重越大，即不可移除的混淆代码越多，则混淆方法的抗逆向能力越强，反之，抗逆向能力越弱。抗逆向能力计算方式为：

$$resilience = \frac{L_{deobf} - L_{origin}}{L_{obfus} - L_{origin}}$$

其中，L_{origin} 是混淆前文件的代码行数，L_{obfus} 是混淆后文件的代码行数，L_{deobf} 是去混淆处理后文件的代码行数。

本文使用 IDA Pro[44]获取混淆前后文件的代码行数，然后使用插件 D-810[45]对混淆后的文件进行去混淆处理，得到对应的去混淆文件代码。分别统计针对 ollvm-sub、InsObf-sub、InsObf-junk 和 InsObf-mix 抗逆向的效果，并以 ollvm-sub 作为对比基准，结果如表 4 所示。

表4 InsObf 和 ollvm 抗逆向分析

Tab. 4 Resilience of InsObf and ollvm

混淆方法	抗逆向较 ollvm-sub 增长率/%
ollvm -sub	0.0
InsObf -sub	163.3
InsObf -junk	301.9
InsObf -mix	386.7

从表 4 可以看出，InsObf-sub 的抗逆向能力在 ollvm-sub 基础上提升近 2 倍，InsObf-junk 提升近 3 倍，可印证两种方法在防反混淆工具处理层面是有明显的增强效果的，同时考虑到指令加花添加了更多的指令，因此数值上会高于指令替换。指令混淆框架的整体实现 InsObf-mix 更是提升近 4 倍，可有效抵抗反混淆工具的攻击。

3.1.3 时空开销

为了获取混淆前后增长的时空开销，首先，针对测试数据集中的每一个文件进行混淆处理，然后使用脚本获取具体的混淆前后的时空开销：时间统计上取多次运行的平均值，空间上取可执行文件的大小。针对数据集，分别使用 ollvm-sub、InsObf-sub、InsObf-junk 与 InsObf-mix 进行如上处理，数值记录如表 5 所示。

表5 InsObf 和 ollvm 时空开销分析

Tab. 5 Time and space cost of InsObf and ollvm

混淆方法	时间开销增长率/%	空间开销增长率/%
ollvm-sub	21.6	20.6
InsObf-sub	23.3	25.1
InsObf-junk	15.0	36.4
InsObf-mix	31.0	42.5

从表 5 可以看出，InsObf-sub 的时空开销和 ollvm-sub 差距不大，均在 5 个百分点以内。同时 InsObf-junk 相较于 ollvm-sub 在时间开销上降低约 6 个百分点，空间开销增加约 16 个百分点。在时间开销上，虽然 InsObf-junk 使用到的不透明谓词有所增加，但相较于 ollvm-sub 还是有明显的优势。另一方面，算法本身添加的大量的花指令使其空间开销高于 ollvm-sub。而整体的指令混淆框架，因为指令加花和指令替换的叠加效果，相较于 ollvm-sub 时间上增加约 10 个百分点，空间上约 20 个百分点。

综合表 3、表 4 和表 5，可以得到如下结论：

（1）InsObf-sub 在 ollvm 的基础上，以增加 5 个百分点以内的时空开销为代价，可有效增加混淆后的程序圈复杂度和抗逆向的能力；

（2）InsObf-junk，相较于 InsObf-sub，在增加 10 个百分点的空间开销的情况下，圈复杂度可以提升近一半，抗逆向能力可以提升近一倍；

（3）相较于 ollvm，InsObf 在时间开销增加约 10 个百分点，空间开销增加约 20 个百分点的情况下，可提高 4 倍的圈复杂度和抗逆向能力。

3.2 基于 ollvm 改进的不同混淆方法的对比

不同的混淆方法在实现时侧重于程序不同的安全属性，如指令混淆侧重于指令的替换和复杂化，控制流混淆侧重于隐藏或扰乱程序的原生路径，字符串混淆侧重于隐藏或保护程序中的字符串等常量信息，所以目前没有统一的指标可以在混淆所带来的安全性角度，对不同混淆方法给出一个定性或定量、足够完备或公平的对比说明。但从广义上的程序复杂性角度，可以使用时空开销和圈复杂度进行效果的说明。因此，本文将从时空开销和圈复杂度的角度比较 InsObf 和目前业界针对 ollvm 的两大改进版：Armariris 和 Hikari 的效果。分别统计混淆前后可执行文件在时空和圈复杂度上的增长，表 6 是不同混淆方法对比的效果，Armariris-string 为 Armariris 改进的字符串加密（StringObfuscation）的实现；Hikari-string 为 Hikari 改进的字符串加密（StringEncryption）的实现；Hikari-cf 为 Hikari 改进的控制流混淆方法的叠加使用，包括 FunctionCallObfuscate、FunctionWrapper 和 IndirectBranching 三种。

表6 不同混淆方法混淆效果

Tab. 6 Obfuscation effects of different methods

混淆方法	开销增长率		圈复杂度增长倍数/%
	时间/%	空间/%	
InsObf-mix	31.0	42.5	607.2
Armariris-string	18.4	0	102.8
Hikari-string	36.1	23.8	215.0
Hikari-cf	40.1	36.6	—

时间开销上，Armariris 的字符串加密混淆，使用了异或的加解密方式，所以时间增长上最少，Hikari 的控制流混淆涉及控制流的多级处理，在时间开销上尤甚。空间开销上，本文提出的指令混淆框架增加最多，Armariris 的字符串加密混淆增长最少，几乎为 0，考虑到本文方法添加了较多的花指令，而 Armariris 使用异或进行加解密，未在空间上引入额外开销，结果在预料内。圈复杂度数值处理上，需要使用 llvm-cbe 获取混淆前后的源文件，但 Hikari 混淆后的文件无法使用 llvm-cbe 有效生成源文件，因此表 6 中 Hikari-cf 一栏置空。但从已有的数据中，InsObf-mix 混淆前后的圈复杂度增长最多，达到了 6 倍。

为了同 Hikari 改进的控制流混淆方法进行有效对比，使用 BinDiff[46]比较 Hikari 和 InsObf 混淆前后程序的指令相似度，比较结果如表 7 所示。

表7 InsObf 和 Hikari 相似度分析

Tab. 7 Similarity of InsObf and ollvm

混淆方法	指令相似度/%	CALL 指令相似度/%	JUMP 指令相似度/%
Hikari -cf	36.3	16.1	39.1
InsObf -mix	23.5	25.7	19.1

表 7 的结果显示，InsObf 混淆前后在指令相似度的降低上优于 Hikari，这是因为一方面 InsObf-junk 通过不透明谓词引入了大量的跳转指令，这也就导致 JUMP 指令的相似度也随之降低，另一方面由于 InsObf-sub 对指令中的运算符和操作数进行了大量的替换工作，最终使得指令相似度大大降低。而 Hikari 因为 FunctionWrapper 针对函数调用做了较多混淆处理，所以其 CALL 指令的相似度低于 InsObf。

综合表 6 和表 7，可以得到如下结论：本文提出的指令混淆框架 InsObf 在空间开销上和 Hikari 的控制流混淆处于同一量级；在时间开销上，高于 Armariris 的字符串混淆，低于 Hikari 的字符串混淆和控制流混淆，处于中等水平；在圈复杂度和指令相似度方面，也可以提供明显的增强。在同基于 ollvm 优化的多种混淆方法的对比中，可以证明本框架可对程序指令进行完备的混淆处理，提供指令层级的有效保护。

4 结语

本文基于 ollvm 的指令混淆模块进行了改进，拓展了指令替换功能支持的运算符数和替换方案数，并额外增加了指令加花的功能。其中，指令替换支持 13 种运算符共计 52 种方案，可以从生成的多个等价表达式中随机选择替换方案，使得通过利用程序地址空间的攻击更难实现。指令加花通过插入叠加跳转指令和虚假循环指令，在充分利用源程序中指令信息的同时，能够有效破坏程序的结构，加大攻击者分析、理解程序的难度。通过实验证明，与 ollvm 的指令替换功能相比，本文提出的框架在时间开销约 10 个百分点，空间开销约 20 个百分点的情况下，圈复杂度和抗逆向能力均可提升 4 倍；在同 Armariris 和 Hikari 的对比中，通过时空和圈复杂度、指令相似度等指标，论证本文提出的指令层级的混淆方案，在同一量级的时空开销下可以提供更高强度的代码复杂度。

未来工作中可以针对时空开销的降低和增加浮点运算符的指令替换进行处理，且在指令替换中结合加密算法，如密码学中的同态加密，可以进一步提高代码的安全性。

参考文献

[1] COLLBERG C, THOMBORSON C, LOW D. A taxonomy of obfuscating transformations[R]. Department of Computer Science, The University of Auckland, New Zealand, 1997.

[2] KUANG K, TANG Z, GONG X, et al. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling[J]. Comput. Secur., 2018, 74: 202-220.

[3] ZHAO Y, TANG Z, YE G, et al. Compile-time code virtualization for android applications[J]. Comput. Secur., 2020, 94: 101821.

[4] ZHANG X, HE F, ZUO W. An inter-classes obfuscation method for java program [C]// ISA '08: Proceedings of 2008 International Conference on Information Security and Assurance. Piscataway, NJ: IEEE, 2008: 360-365.

[5] SHOSHITAISHVILI Y, WANG R, SALLS C, et al. SOK: (State of) the art of war: offensive techniques in binary analysis [C]// S&P 2016: Proceedings of 2016 IEEE Symposium on Security and Privacy. San Piscataway, NJ: IEEE, 2016: 138-157.

[6] MU D, GUO J, DING W, et al. ROPOB: obfuscating binary code via return oriented programming [C]// SecureComm 2017: Proceedings of 13th EAI International Conference on Security and Privacy in Communication Networks. Niagara Falls: Springer, 2017: 721-737.

[7] JUNOD P, RINALDINI J, WEHRLI J, et al. Obfuscator-LLVM--software protection for the masses [C]// SPRO '15: Processing of 2015 IEEE/ACM 1st International Workshop on Software Protection. Washington: IEEE, 2015: 3-9.

[8] CAO L, SUN G, WANG H, et al. Logic invariability study of junk code transformation[J]. Computer Engineering, 2006, 32(20): 135.

[9] YANG Y, HUANG W, LI Z, et al. Code obfuscation algorithm to resist static disassembly[J]. Transactions of Beijing Institute of Technology, 2015, 35(3): 272-278.

[10] 孙国梓, 陈丹伟, 蔡强. 子程序花指令模糊变换逻辑一致性研究[J].计算机科学, 2009, 36(8): 89-91. (SUN G Z, CHEN D W , CAI Q. Research on logic consistency of junk code transformation within sub-function[J]. Computer Science, 2009, 36(8): 89-91.)

[11] BANESCU S, COLLBERG C S, GANESH V, et al. Code obfuscation against symbolic execution attacks [C]// ACSAC '16: Proceedings of the 32nd Annual Conference on Computer Security Applications. New York: ACM, 2016: 189–200.

[12] HALEVI S, HALEVI T, SHOUP V, et al. Implementing BP-obfuscation using graph-induced encoding [C]// CCS '17: Proceedings of 24th ACM-SIGSAC Conference on Computer and Communications Security. New York: ACM, 2017: 783-798.

[13] ISMANTO R N, SALMAN M. Improving security level through obfuscation technique for source code protection using AES algorithm [C]// ICCNS 2017: Proceedings of the 2017 the 7th International Conference on Communication and Network Security. New York: ACM, 2017: 18–22.

[14] 潘雁, 祝跃飞, 林伟. 基于指令交换的代码混淆方法[J]. 软件学报, 2019, 30(06): 1778-1792. (PAN Y, ZHU Y F, LIN W. Code obfuscation based on instructions swapping[J]. Journal of Software, 2019, 30(6): 1778-1792.)

[15] CECCATO M, TONELLA P. CodeBender: remote software protection using orthogonal replacement [C]// Proceedings of IEEE Software. Piscataway, NJ: IEEE, 2011, 28 (2): 28-34.

[16] ARMOOGUM S, CAULLY A. Obfuscation techniques for mobile agent code confidentiality[J]. Journal of Information and Systems Management, 2021, 1.

[17] CHO S, CHANG H, CHO Y. Implementation of an obfuscation tool for C/C plus plus source code protection on the XScale architecture [C]// SEUS '08: Proceedings of 6th International Workshop on Software Technologies for Embedded and Ubiquitous Systems. Berlin: Springer, 2008: 406-416.

[18] BALACHANDRAN V, EMMANUEL S, KEONG N W, et al. Obfuscation by code fragmentation to evade reverse engineering [C]// IEEE Int. Conf. Syst. Man Cybern.: Proceedings of IEEE International Conference on Systems, Man, and Cybernetics. Piscataway, NJ: IEEE, 2014: 463-469.

[19] ZHANG Y, PANG J. A new compile-time obfuscation scheme for software protection [C]// CyberC 2016: Proceedings of 8th International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery. Washington, DC: IEEE Computer Society, 2016: 1-5.

[20] COHEN, FREDERICK B. Operating system protection through program evolution[J]. Computers and Security, 1993, 12(6): 565-584.
[21] PENG Y, SU G, TIAN B, et al. Control flow obfuscation based protection method for Android applications[J]. China Communications, 2017, 14(11): 247-259.
[22] JACKSON T, HOMESCU A, CRANE S, et al. Diversifying the software stack using randomized NOP insertion [C]// IWSEC 2013: Proceedings of 2013 Advances in Information Security. Berlin: Springer, 2013: 151-173.
[23] JANGDA A, MISHRA M, SUTTER B D. Adaptive just-in-time code diversification [C]// MTD '15: Proceedings of the Second ACM Workshop on Moving Target Defense. New York: ACM, 2015: 49-53.
[24] LU K J, XIONG S Y, GAO DB. RopSteg: program steganography with return oriented programming [C]// CODASPY '14: Proceedings of the 4th ACM conference on Data and application security and privacy. New York: ACM, 2014: 265-272.
[25] LINN C, DEBRAY S. Obfuscation of executable code to improve resistance to static disassembly [C]// CCS '03: Proceedings of the ACM Conference on Computer and Communications Security. New York: ACM, 2003: 290-299.
[26] 乐德广, 赵杰, 龚声蓉.基于模式切换的 ARM 汇编代码混淆算法[J].计算机工程与应用,2021, 57(18): 122-129. (LE D G, ZHAO J, GONG S R. ARM Assembly code obfuscation algorithm based on mode switch[J]. CEA, 2021, 57(18): 122-129.)
[27] WU J H, GAO X C, TIAN C H, et al. The study for protecting mobile agents based on time checking technology [C]// ROBIO 2007: Proceedings of 2007 IEEE International Conference on Robotics and Biomimetics, Washington, DC: IEEE Computer Society, 2007: 2013-2017.
[28] RAJBA P, MAZURCZYK W. Data hiding using code obfuscation [C]// ARES 2021: Proceedings of the 16th International Conference on Availability, Reliability and Security. New York: ACM, 2021:1-10.
[29] SU Q, WANG Z Y, WU W M, et al. Technique of source code obfuscation based on data flow and control flow transformations [C]// ICCSE 2012: Proceedings of 2012 7th International Conference on Computer Science & Education. Piscataway, NJ: IEEE, 2012: 1093-1097.
[30] PICHETA D. Code obfuscation for the C/C++ language[J]. CoRR, 2020, abs/2003.03449.
[31] KANZAKI Y, MONDEN A, NAKAMURA M, et al. Exploiting self-modification mechanism for program protection [C]// COMPAC 2003: Proceedings of 27th Annual International Computer Software and Applications Conference, Piscataway, NJ: IEEE, 2003: 170-179.
[32] KANZAKI Y, MONDEN A, NAKAMURA M, et al. A software protection method based on instruction camouflage[J]. Electronics and Communications in Japan Part Ⅲ-Fundamental Electronic Science, 2006, 89 (1): 47-59.
[33] DARWISH S M, Guirguis S K, Zalat M S. Stealthy code obfuscation technique for software security [C]// ICCEA '10: Proceedings of the 2010 International Conference on Computer Engineering & Systems, Washington, DC: IEEE Computer Society, 2010: 93-99.
[34] LATTNER C, ADVE V. LLVM: A compilation framework for lifelong program analysis & transformation [C]// CGO04: Proceedings of International Symposium on Code Generation and Optimization. Piscataway, NJ: IEEE, 2004: 75-86.
[35] GoSSIP-SJTU/Armariris [EB/OL]. (2019-06-09) [2021-12-29] https://github.com/GoSSIP-SJTU/Armariris.
[36] HikariObfuscator/Hikari: LLVM Obfuscator [EB/OL]. (2020-01-30) [2021-12-29] https://github.com/HikariObfuscator/Hikari.
[37] Clang: a C language family frontend for LLVM [EB/OL]. (2021-10-04) [2021-12-29] https://clang.llvm.org/.
[38] CHEN C, PETSIOS T, POMONIS M. CONFUSE: LLVM-based Code Obfuscation[J]. 2013.
[39] PALSBERG J, KRISHNASWAMY S, Kwon M, et al. Experience with software watermarking[J]. Proceedings 16th Annual Computer Security Applications Conference, 2000: 308-316.
[40] GeeksforGeeks. A computer science portal for geeks [EB/OL]. [2021-12-29] https://www.geeksforgeeks.org/.
[41] MCCABE T. A complexity measure[J]. IEEE Transactions on Software Engineering, 1976, SE-2: 308-320.
[42] JuliaComputingOSS/llvm-cbe [EB/OL]. (2021-07-19) [2021-12-29] https://github.com/JuliaComputingOSS/llvm-cbe.
[43] terryyin/lizard: A simple code complexity analyser without caring about the C/C++ header files or Java imports, supports most of the popular languages. [EB/OL]. (2021-12-23) [2021-12-29]. https://github.com/terryyin/lizard.
[44] IDA Pro – Hex Rays [EB/OL]. [2021-12-29]. https://hex-rays.com/IDA-pro/.
[45] eshard/D810 [EB/OL]. (2021-10-08) [2021-12-29]. https://gitlab.com/eshard/d810.
[46] Zynamics.com-BinDiff [EB/OL]. (2021-07-04) [2021-12-29]. https://www.zynamics.com/bindiff.html

This work is partially supported by Peking University horizontal research project(code obfuscation protection research) (2020001763).

WANG Yayi, born in 1998, M. S. Her research interests include code obfuscation and vulnerability mining.
LIU Chen, born in 1999, M. S. His research interests include code obfuscation and vulnerability mining.
HUANG Tianbo, born in 1997, M. S. His research interests include cyberspace security, malicious code detection and code obfuscation.
WEN Weiping, born in 1976, Ph. D. professor. His research interests include system and network security, big data and cloud security, intelligent computing security.