doi :10.3969/j.issn.1671-1122.2021.07.003

一种对抗符号执行的代码混淆系统

文伟平，方莹，叶何，陈夏润
（北京大学软件与微电子学院，北京 100080）

摘　要：符号执行在辅助挖掘软件漏洞和软件去混淆等领域取得了很大的进展，作为保护软件安全最有效手段之一的代码混淆系统如今几乎无法对抗符号执行的攻击。为解决上述问题，文章从混淆系统的功能扩展和符号执行工具的弱点利用出发，通过丰富 OLLVM 的同义指令替换集合和恒真谓词集合，设计不透明谓词对软件控制流进行混淆以及对分支条件进行加固，提出一种可以兼容多种编程语言、可扩展并且可以对抗符号执行攻击的代码混淆系统。实验表明，文章提出的混淆系统可以在不改变软件原有功能的前提下诱导符号执行工具分析不可达的路径或者保护软件的部分路径不被探索到，从而可以有效对抗符号执行攻击。相较传统混淆系统，文章提出的混淆系统在对抗符号执行攻击方面具有明显优势。

关键词：代码混淆；符号执行；不透明谓词；OLLVM

中图分类号：TP309　**文献标志码：**A　**文章编号：**1671-1122（2021）07-0017-10

中文引用格式：文伟平, 方莹, 叶何, 等 . 一种对抗符号执行的代码混淆系统 [J]. 信息网络安全, 2021, 21 (7): 17–26.

英文引用格式：WEN Weiping, FANG Ying, YE He, et al. A Code Obfuscation System against Symbolic Execution Attacks[J]. Netinfo Security, 2021, 21(7): 17–26.

A Code Obfuscation System against Symbolic Execution Attacks

WEN Weiping, FANG Ying, YE He, CHEN Xiarun
(*School of Software and Microelectronics*, *Peking University*, *Beijing* 100080, *China*)

Abstract: Symbolic execution technology has made great progress in the areas of assisting the discovery of software vulnerabilities and software de-obfuscation. As one of the most effective means to protect software security, the existing code obfuscation system can be hardly resilient against symbolic execution attacks. To solve the problem mentioned above, starting from the function extension of the obfuscation system and the weakness exploitation of the symbolic execution tool, this paper enriches the synonymous instruction replacement set and the identical truth predicate set, and designs opaque predicates to obfuscate the control flow and reinforce the branching conditions of the software, and proposes a code

收稿日期：2021-04-05
基金项目：国家自然科学基金 [61872011]
作者简介：文伟平（1976—），男，湖南，教授，博士，主要研究方向为网络攻击与防范、软件安全漏洞分析、恶意代码研究、信息系统逆向工程和可信计算技术；方莹（1995—），女，浙江，硕士研究生，主要研究方向为网络与系统安全、智能合约安全；叶何（1998—），男，安徽，硕士研究生，主要研究方向为网络与系统安全、云计算；陈夏润（1997—），男，江西，硕士研究生，主要研究方向为网络与系统安全、漏洞挖掘。
通信作者：文伟平 weipingwen@ss.pku.edu.cn

obfuscation system that is compatible with multiple programming languages, scalable, and can resist symbolic execution attacks. The experimental results show that this obfuscation system can induce symbolic execution tools to enter unreachable paths, or protect some right paths from being explored by symbolic execution tools without changing the original functions of the software, so that it can be resilient against the attacks of symbolic execution tools. Compared with traditional obfuscation systems, this system has obvious advantages in combating symbolic execution attacks.

Key words: code obfuscation; symbolic execution; opaque predicate; OLLVM

0 引言

信息时代，软件作为信息交换的载体提供着各类服务，已成为人们生活中必不可少的工具。在软件发布时，软件供应商往往以可执行程序的形式将软件发布到互联网上，供用户下载使用。然而，软件的可执行文件和执行环境是公开的，攻击者会在未经授权的情况下，通过逆向工程等手段获取软件的敏感数据与代码，挖掘软件漏洞，甚至直接篡改代码。面对这些攻击，未经保护的可执行程序不再安全，一方面影响软件供应商的经济效益，另一方面也可能威胁用户的隐私安全。

为了保护这些软件，研究人员研究了多种安全加固方法。其中，代码混淆被认为是最受欢迎的可以保护软件不受逆向工程攻击的技术之一，即在不改变原始程序语义的情况下，通过增加代码的复杂度提升攻击者破解的难度。学术界对代码混淆做了大量研究，包括控制流混淆、数据混淆、标识符混淆等[1]。但随着反汇编、符号执行、模式匹配等代码分析手段的发展，传统的混淆技术的保护效果被极大削弱，无法有效对抗现有代码分析技术的攻击，尤其是符号执行攻击[2]。究其原因，一方面是因为现有的开源混淆系统设计的不透明谓词过于简单，不能保证控制流的安全性；另一方面是由于混淆的特征明显，如OLLVM（Obfuscator-LLVM，OLLVM）的指令替换集合太小。

为了解决上述问题，本文丰富并优化了开源混淆系统OLLVM，包括丰富同义指令替换方案、增加恒真谓词、优化虚假控制流混淆方案。同时，利用符号执行工具的弱点，本文设计了一种可以诱导符号执行工具去分析软件实际运行时不可达路径的不透明谓词。此外，利用密码学和代数理论，本文还设计了符号执行工具无法求解的多种不透明谓词，保护程序的部分路径不被符号执行工具攻击。

1 相关工作

1.1 符号执行

符号执行思想最早在1975年左右被提出，BOYER[3]等人实现了一个名为SELECT的系统，该系统通过求解路径约束来辅助调试程序。随后，CLARKE[4]和KING[5]也相继使用符号执行思想对程序路径上的约束条件求解以发现潜在的代码问题。

SALWAN[6]等人和YADEGARI[7]等人提出基于符号执行和污点分析对程序进行实验，结果证明该方法对软件去混淆有效，但仍然需要符号执行工具对符号变量的求解，当无法求解不透明谓词时，并不能保证去混淆的有效性。YADEGARI[8]等人提出了一种反代码混淆的通用方法。他们没有对所使用的混淆的本质做任何假设，而是通过保留语义转换程序来简化混淆代码。该方法有一定的去混淆效果，但对不能被污点传播的路径去混淆效果有限。BRUMLEY[9]等人提出利用动态符号执行的方法对软件进行去混淆。该方法可以绕过复杂的混淆保护（如打包、运行时自修改），找到触发程序分支的输入用例，甚至能触发程序隐藏行为，从而最终可以在受控环境中分析触发的行为。MING[10]等人将不透明谓词分为恒定不透明谓词、上下文不透明谓词和动态不透明谓词三类，并提出了检测三种不透明谓词的方法，该方法在面对简单的不透明谓词时有很好的效果，但当面对无法被符号执行工具解析的复杂不透明谓词时，可能会出错。文献[11]

提出了一种名为Dose的基于对等语义的新型去混淆方法。该方法通过Dose来消除被混淆代码中的静态混淆转换，改善基于动态符号执行的去混淆技术，提高代码覆盖率。Dose与传统的二进制分析工具的最大区别是，Dose引入了语义相等性检测来消除不透明谓词，该方法同样是基于不透明谓词能够被符号执行工具正确求解。

EYROLLES[12]等人提出了一种处理混合布尔算术表达式（MBA）的方法，即通过建立一组语义等同的MBA表达式库，使用等价MBA表达式对原始表达式进行替换，直到无法替换为止，这种方案可以很快简化已有的表达式。该去混淆方案主要对程序中已知的MBA表达式进行简化，对包含未定义的MBA表达式的程序去混淆效果有限。GABRIEL[13]提出了一种Miasm框架，通过识别OLLVM混淆后的特征谓词，对OLLVM的控制流扁平化、虚假控制流和指令替换等去混淆，取得了很好的效果。该去混淆方案主要基于OLLVM混淆后的谓词特征比较明显，如分支上的恒真谓词单一等。

SCHRITTWIESER[14]等人虽然没有提出新的攻击思路，但展示了面对符号执行攻击时，传统的混淆方法是十分脆弱的。他们通过使用两个代码混淆工具和两个符号执行工具分别对5000个C语言程序进行混淆和反混淆，研究不同混淆对抗动态符号执行攻击的强度，研究结果表明，很多混淆方法无法有效抵抗动态符号执行的攻击。

综上可知，目前符号执行对软件的攻击主要基于两个方面，一方面是现有的软件中谓词过于脆弱，会很容易地被符号执行工具求解，如文献[6]、文献[7]、文献[8]、文献[9]、文献[10]、文献[11]中的研究方案；另一方面是现有的软件混淆后特征明显，如文献[12]、文献[13]中的研究方案。同时，从文献[9]、文献[11]、文献[13]、文献[14]中的实验效果来看，不仅大多数软件的不透明谓词都比较简单，而且现有的代码混淆系统设计的不透明谓词同样单一，难以对抗符号执行的攻击。

1.2 代码混淆

代码混淆是指在不改变程序语义的情况下，对程序采取复杂化处理或者加固等手段，来增加程序的复杂度，加大攻击人员分析程序的难度，保护程序安全。

2010年，瑞士西北应用科技大学安全实验室基于LLVM编译工具，开发了一个代码混淆项目OLLVM[15]，该项目实现了指令替换、控制流扁平化、虚假控制流等基本混淆功能。指令替换功能主要是将常见的二元运算符，如加法、减法、与、或、异或等，用同义但更为复杂的指令序列进行代替。然而，替换后的指令特征比较明显，且不够复杂，很容易被符号执行工具反混淆。控制流扁平化是通过将if-else语句拆分成基本的代码块，再用do-while语句组合起来，以增加逆向的复杂度，但扁平化模式也较为单一，容易被反混淆。虚假控制流是指在一些简单运算外层嵌套多层if-else判断语句，嵌入的判断条件是恒真的，但由于判断条件很容易被符号执行工具解析，因此这种模式也容易被攻击者反混淆。

上海交通大学密码与计算机安全实验室在OLLVM的基础上，开发了Armariris混淆框架[16]，该框架扩展了字符串混淆的功能。NAVILLE[17]基于OLLVM混淆系统开发的Hikari框架扩展了一些新的功能，如直接跳转转间接跳转、函数调用混淆等。但这两个框架均没有针对符号执行的弱点设计安全的谓词，从而很难对抗符号执行工具的攻击。

ZHOU[18]等人提出了一种混合布尔和算术运算（MBA）的等价谓词替换方式，来增加谓词的复杂度，使得符号执行的约束求解器无法解析或者在较短的时间内无法求解出正确的约束，以对抗符号执行的攻击。这是一种很好的混淆思路，然而，随着符号执行的发展，该方法已经很难对抗符号执行的攻击了。

XU[19]等人提出了一种基于符号的不透明谓词，实现的主要思路是在每个不透明谓词中引入动态符号执行难以处理的问题。例如，利用符号执行工具对指针

分析的弱点，将变量从数组中读出，阻断符号执行对变量的分析，从而可以有效对抗动态符号执行的攻击。但是不透明谓词的安全性取决于符号执行工具的能力，如果符号执行工具加入分析符号内存的机制，则该混淆方案会受到威胁，但该混淆方案可以作为本文设计的混淆方案的补充。SHARIF[20]等人提出了使用Hash函数来构造不透明谓词，以对程序的分支条件进行加固，但是没有考虑存在Hash碰撞的可能，此外，该方案未开源。WANG[21]等人使用一种收敛数列的算法设计不透明谓词，符号执行工具无法求解这种谓词，虽然数列收敛的次数远小于符号执行求解的次数，但在程序正常运行时，仍需要执行循环直到数列收敛，该方案也未开源。

STEPHENS[22]等人提出使用隐蔽通道来隐藏信息流，欺骗符号执行工具，使其无法分析程序中变量之间的关系，从而达到混淆的效果，但存在较大的系统开销，且不够准确。

综上所述，现有的开源代码混淆系统并不能提供足够强大的不透明谓词，本文在研究已有方案的优缺点后，对已有的代码混淆系统进行优化，设计多种对抗符号执行攻击的代码混淆方案，保护程序有效对抗符号执行工具的攻击。

2 系统总体架构

本文设计并实现了一款基于LLVM的代码混淆系统，系统的直接作用对象为LLVM编译产生的中间代码（IR）。每个混淆模块都作用于IR文件，并在混淆后输出新的IR文件。各混淆模块可以作用于前一个混淆模块生成的IR文件，可以组合或多次使用，并且可以实现多次混淆加固。各模块彼此独立，具有良好的扩展性和低耦合性。

2.1 模块设计目标

本文设计的混淆模块，从设计目标上可以分为复杂化和路径偏移两个部分，每个部分设计的混淆模块如图1所示。

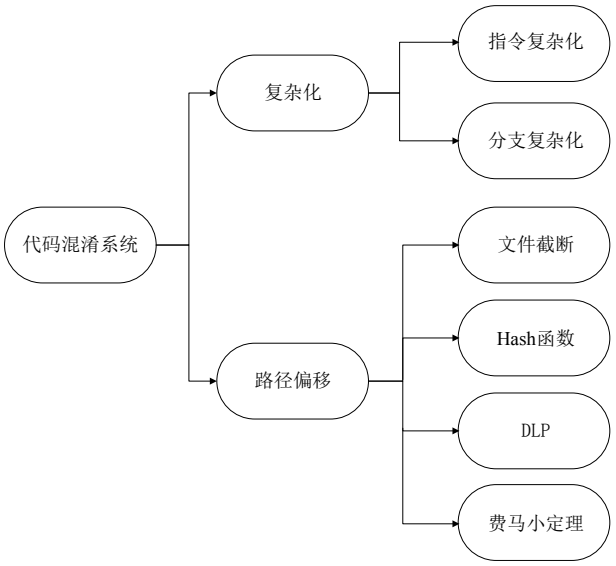

图 1 混淆模块示意图

2.1.1 复杂化

复杂化混淆模块的设计目的是在不改变程序语义的情况下，通过指令替换、控制流重构等方式，提高程序的复杂性指标，如指令数量、圈复杂度等。相较其他混淆工具，混淆后程序无明显混淆特征，从而可对抗基于特征的去混淆攻击。

该模块包括指令复杂化和分支复杂化两个模块。指令复杂化模块是通过同义指令替换，将简单的二元操作指令替换为一组同义指令集合；分支复杂化模块是通过构造恒真谓词，在不改变源程序语义的前提下，重构源程序的控制流结构，使其更加复杂。

2.1.2 路径偏移

路径偏移混淆模块的设计目的是诱导符号执行工具去执行实际执行时不可达的路径分支，或者保护某条路径不被符号执行工具执行，从而对抗符号执行工具对程序的攻击。

该模块包括文件截断混淆模块、Hash函数混淆模块、DLP函数混淆模块和费马小定理混淆模块。文件截断混淆模块利用符号执行工具无法正确处理外部函数调用的特点，将程序路径上的变量通过存取文件阻断符号执行工具对此变量的分析，诱使符号执行工具去执行构造的不可达分支。Hash函数混淆模块是利用

Hash函数的单向性，用Hash函数对分支条件变量进行加密，使符号执行工具无法求解满足约束的具体值。DLP函数混淆模块是基于离散对数分解难题，对程序分支条件的变量进行加密，使得符号执行工具求解满足分支约束的具体值在计算上不可行。费马小定理混淆模块是利用费马小定理的性质，对分支条件上的具体值和变量值进行等效变换，在不改变程序语义的前提下，对抗符号执行工具对分支约束条件的求解。

2.2 代码混淆流程

代码混淆流程包括前端编译、IR分析、IR转换、后端编译四个阶段，本文混淆系统各个混淆模块的主要工作在于IR分析和IR转换两个阶段。

IR分析是基于IR文件对程序进行分析，获取模块信息（模块上下文环境、模块中函数信息等），函数信息（函数上下文环境、函数名、函数所属模块信息、函数中基本块信息等），基本块信息（基本块上下文环境、基本块名、基本块所属函数信息、基本块中指令信息、基本块间位置关系等），指令信息（指令名称、指令操作数、指令种类、指令所属基本块信息、指令间位置关系等）。

IR转换是基于IR文件对程序进行转换，包括在源IR文件的模块中创建函数，在函数中创建基本块、分割基本块、删除基本块、新建指令、替换指令、删除指令、更改指令位置关系、重构程序的控制流等。

3 系统功能实现

3.1 指令复杂化模块

本模块在OLLVM设计的13种对等语义指令替换的基础上，丰富了对等语义指令替换的种类，针对简单的二元指令，设计了多种复杂的同义指令集合，如表1所示。表中的*bh*表示操作数*b*的高位，*bl*表示操作数*b*的低位，*zero*表示值恒为0的表达式，如(*x*+1)**x*%2。

3.2 分支复杂化模块

本模块的设计思想是随机在程序的控制流任意一处进行分割，并在分割后的上下文之间插入分支条件，当分支条件为真时，跳转至分割后的部分，分支条件为假时，则跳转至人为构造的代码块，人为构造的代码块随机执行程序的其他位置。

表 1 指令替换表

原指令	替换后
$a=b+c$	$a=(b+d)+(c-d)$
	$a=(b-r+r)+(c+r-r)$
	$a=(bh+ch)+(bl+cl)$
	$a=b+c+zero$
$a=b-c$	$a=b+d;a=a-c;a=a-d$
	$a=b-d;a=a-c;a=a+d$
	$a=(bh-ch)+(bl-cl)$
	$a=(b+r-r)-(c-r+r)$
	$a=b-c-zero$
$a=b*c$	$a=(b-r+r)*c$
	$a=(b-r+r)*(c-r+r)$
$a=b*c$	$a=(b+zero)*(c-zero)$
$a=b<<c$	$a=(b+r-r)<<c$
	$a=(bh<<c)\%1<<16+(bl<<c)\%1<<16$
$a=b>>c$	$a=(b+r-r)<<c$
$a=b\&c$	$a=(b+r-r)\&(c+r-r)$
	$a=(b+zero)\&(c-zero)$
	$a=bl\&cl+bh\&ch$
$a=b\|c$	$a=(b+r-r)\|(c+r-r)$
	$a=(b+zero)\|(c-zero)$
	$a=bl\|cl+bh\|ch$
$a=b\^{}c$	$a=(b+r-r)\^{}(c+r-r)$
	$a=(b+zero)\^{}(c-zero)$
	$a=bl\^{}cl+bh\^{}ch$

本文提出的分支复杂化较OLLVM中的Bogus Control Flow混淆，添加了5种恒真谓词。OLLVM中构造的恒真条件为$x*(x-1)\%2==0$ or $y<10$，该条件特征较为明显，故本文通过设计多种恒真式来加强混淆后的隐蔽性。如果基本块分割上一条指令为二元操作指令，形如$z=x$ op y，且x与y为整型变量，则构造如下5种恒真式，其中，等号左边为关于x的恒真等式，右边为关于x和y的等式。由于左边恒为真，所以下述5个条件恒为真。

$x*(x+1)\%2==0$ or $\exp(x,y)$ （1）

$x*(x+1)*(x+2)\%6==0$ or $\exp(x,y)$ （2）

$x*(x+1)*(x+2)*(x+3)\%24==0$ or $\exp(x,y)$ （3）

$x*(x+1)*(x+2)*(x+3)*(x+4)\%120==0$ or $\exp(x,y)$ （4）

$x*(x+1)*(x+2)*(x+3)*(x+4)*(x+5)\%720==0$ or $\exp(x,y)$ （5）

此外，不同于OLLVM混淆后控制流的固定格式，

本文设计的不可达分支会随机跳转到当前函数的一个基本块以增加控制流的复杂度。

具体算法流程如代码1所示。本模块的混淆目标是程序中的函数，首先将函数的全部基本块存入向量，然后遍历基本块中的指令。当指令为二元操作指令时，提取指令中的两个操作数，并在控制流中插入判断分支，分支条件为使用两个操作数构造的上述5种恒真式之一。如果恒真式为真，则跳转到正确的路径上；否则，跳转到不可达分支。

代码1 分支复杂化伪代码

```
输入：IR 文件，混淆次数 Times，混淆概率 P
输出：分支复杂化混淆后的 IR 文件
1 将函数 F 的所有基本块存入 vector;
2 For block in vector:
3    inst = block.getFirstNonPHIOrDbgOrLifetime();
4    If not inst.operator→isBinary() or random(1) > P:
5        continue;
6    If Times ≤ 0:
7        return;
8    Times−−;
9    originBlock = block.splitAt(inst);
10   fakeBlock = createFakeBlock();
11   condition = 使用 inst 左右操作数创建的恒真等式;
12   BranchInst(true:originBlock, false:fakeBlock, condition, from:
block);
13   BranchInst(to: vector[1 + random % vector.size], from:
fakeBlock);
```

3.3 文件截断模块

本混淆模块设计思路是，选择程序路径上变量v，将其存入文件，并从文件中取出赋值给v'，在路径中插入恒真分支条件$v == v'$。由于符号执行工具无法准确分析外部函数行为，故其无法准确分析出v'的值，会去分析$v!=v'$这条路径上的不可达代码块，而该代码块是无限循环的，符号执行工具会陷入循环直到超时，由此增加符号执行工具对程序分析的时间开销，提升其对程序路径分析求解的代价。文件截断混淆示意图如图2所示。

3.4 Hash 函数模块

本模块的设计思路是在程序分支路径中，如果分支条件形如$x==a$，其中a是立即数，则混淆时首先计算b=Hash(a)，然后将分支条件替换为Hash(x)==b。由于符号执行工具不能由b反推出满足条件的x的值，使得Hash(x)==b，因此符号执行工具无法求解出满足分支条件的测试用例，以对抗符号执行的攻击。

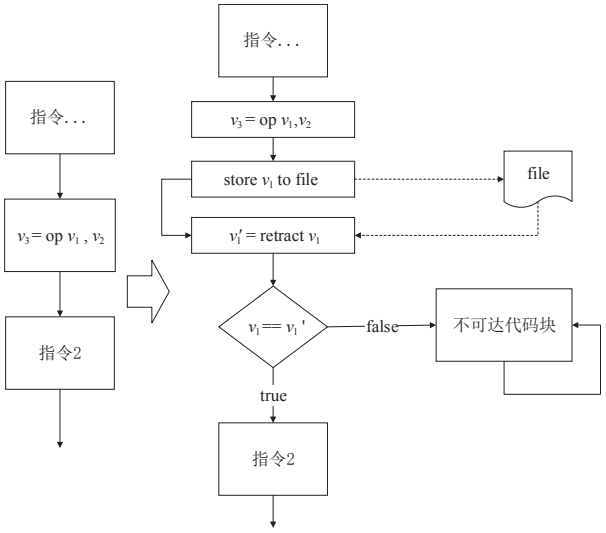

图 2 文件截断混淆示意图

Hash函数混淆模块包括3个阶段：分支条件选择、Hash函数构造和分支条件重构。

1）分支条件选择：本模块的目标是分支条件形如$x==a$和$x!=a$，其中x为变量，a为立即数，并按照设置的混淆概率和混淆次数决定是否对此分支进行混淆。

2）Hash函数构造：构造单向Hash函数y=Hash(x)，使得Hash函数可以完成立即数到立即数的映射，并且满足由x计算y容易，由y计算x困难。

3）分支条件重构：选定符合混淆标准的分支条件后，通过构造Hash函数，对原分支条件进行替换。由于Hash函数可能会存在哈希碰撞，即Hash(x) == Hash(z)时，x不一定等于z，但Hash(x)!=Hash(z)时，x一定不等于z，因此，在替换分支条件后，仍然需要在特定路径上添加指定判断。针对分支条件中两种比较指令（恒等和不等），Hash函数混淆示意图分别如图3和图4所示。

3.5 DLP 函数模块

由于Hash函数混淆模块会在额外的约束中暴露程序初始的分支条件，存在被未来符号执行工具混淆的风

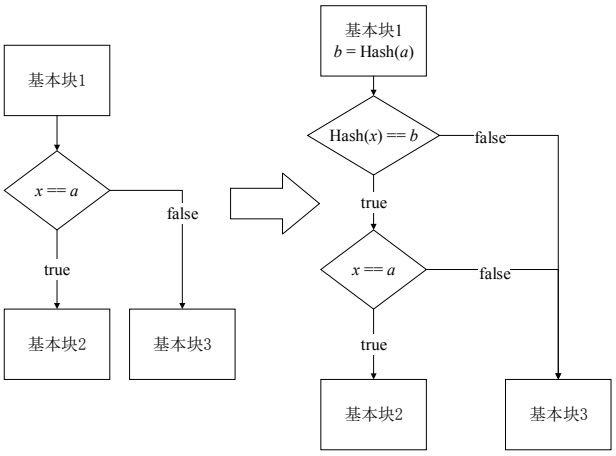

图 3 Hash 函数恒等指令混淆示意图

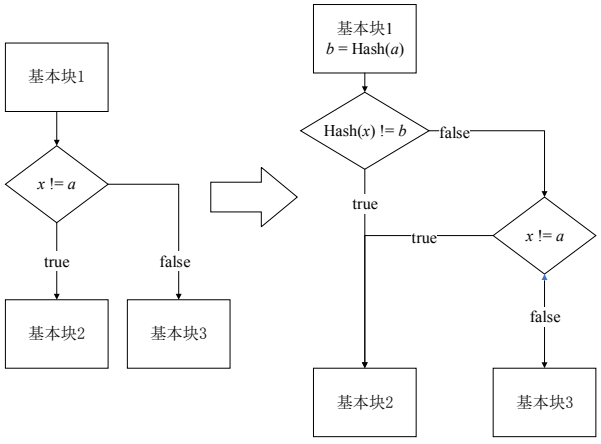

图 4 Hash 函数不等指令混淆示意图

险，故本模块利用离散对数分解难的性质进行设计。离散对数难题（Discrete Logarithm Problem，DLP）有如下性质：

性质1 已知素数p及原根r，在x为正整数且$x<p$时，对每个x，$r^x \bmod p$值均不同。

利用性质1，本模块设计了一种函数f，使得对于$y=\mathrm{f}(x)$，x与y一一映射，并且程序在执行时通过x来计算y容易，反推困难。以此函数代替Hash函数，可有效保护程序路径不被符号执行工具攻击，且不暴露原始的分支条件，降低被未来符号执行工具去混淆的风险。

该模块包括4个阶段：分支条件选择、原根选择、核心函数构造和分支条件重构。

1）分支条件选择可参考Hash函数模块。

2）原根选择是选择一个大素数p，并计算此大素数的一个原根r。根据性质1，可以保证在变量x小于p时，x取不同的值，$r^x \bmod p$的值也不相同。

3）核心函数构造是使用大素数p和原根r，构造一个快速计算幂次的函数，形如$y=\mathrm{f}(x)=r^x \bmod p$，计算$b=\mathrm{f}(a)$，将分支条件中的$x==a$替换为$\mathrm{f}(x)==b$，将分支条件$x!=a$替换为$\mathrm{f}(x)!=b$。根据性质1，如果$r$为大素数$p$的原根，则在$x<p$时，所有$\mathrm{f}(x)$都是不同的。

4）分支条件重构是使用上一阶段选择的p和r及构造的函数f，通过计算$b=r^a \bmod p$，用构造的新分支$(r^x \bmod p)==b\&\&x<p$和$(r^x \bmod p)!=b\&\&x<p$替换原始分支条件$x==a$和$x!=a$。针对恒等指令和不等指令，本模块混淆方案如图5和图6所示。

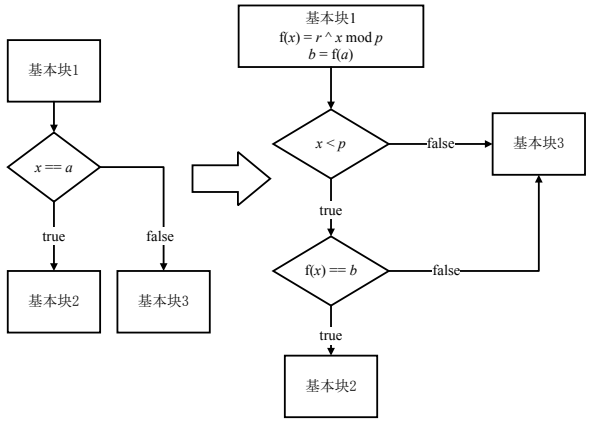

图 5 DLP 函数恒等指令混淆示意图

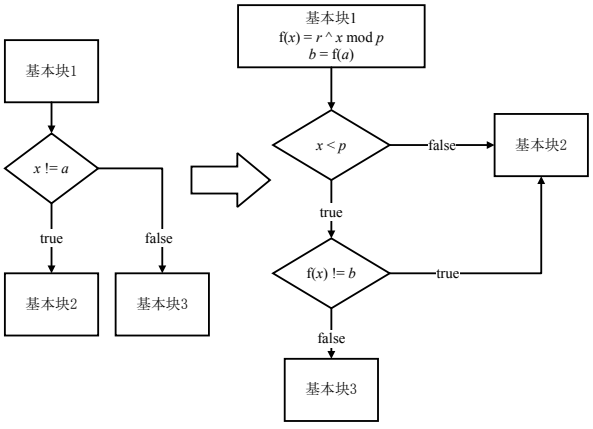

图 6 DLP 函数不等指令混淆示意图

3.6 费马小定理模块

基于DLP和Hash函数的混淆方案只能支持对形如

$x==a$ 或者 $x!=a$ 的分支条件进行混淆，本模块的混淆方案利用了费马小定理的部分性质对形如 $x>a$ 或者 $x<a$ 等的分支条件进行混淆。

性质2 如果 p 是一个素数，x 是一个小于 p 的正整数，那么 x^p-x 是 p 的倍数，可以表示为 $x^p == x \pmod p$。

$x^p == x \pmod p$ 也可改写为 $x^p \bmod p == x$，将形如 x op a 的分支条件转换为 $x^p \bmod p$ op a。本混淆模块的核心是选定大素数 p，构造一个快速计算幂次的函数，函数形如 $y=f(x)=x^p \bmod p$，使得 $f(x)$ 是 x 的一个等效替换。这种混淆不会改变程序的语义，且可以在 $O(\log(p))$ 的时间复杂度内计算出 $x^p \bmod p$。因为该性质对现有的符号执行工具（如KLEE）是透明的，故符号执行工具求解此分支条件需要 $O(p)$ 的时间复杂度。因此，本混淆模块可以对形如 $x>a$，$x<a$，$x \geqslant a$，$x \leqslant a$ 的分支条件进行混淆，有效对抗符号执行的攻击。

4 实验及分析

本文设计的实验主要评估可用性、弹性和开销。可用性即本混淆系统混淆后是否改变软件的功能；弹性即相较于其他开源混淆系统，本混淆系统是否具有更好的对抗符号执行工具攻击的能力；开销即各个混淆模块混淆所产生的额外时间和空间开销。

本文实验在一台使用2.4GHz主频、16GB内存的英特尔8核i7处理器的主机上进行，所使用的软件环境如表2所示。

表 2 软件运行环境

类型	版本
操作系统	Ubuntu 20.04.1 LTS
LLVM	LLVM version 10.0.0
KLEE	tags/v2.1
angr	tags/v9.0.6488
Python	Python 3.7.0
CMake	CMake version 3.16.3
IDA pro	version 7.0.170914

4.1 可用性评估

本文采用多种类型软件的源代码作为测试集进行测试，期望全面地评估本混淆系统的可用性。

首先选择多种类型的小型开源软件，因为比较容易判断混淆后软件功能是否改变。此处选用coreutils中随机4个Linux命令的程序源代码、gzip源代码、diffutils源代码作为数据集。另外还选择了一些标准库和加密算法在GitHub的开源实现，这些程序具有较为完备的测试用例集合且具有足够的复杂度，可以更可靠地比较混淆前后代码功能是否改变。此处选用glibc中随机4个函数的源代码、MD5开源实现代码、SHA1开源实现代码作为数据集。

可用性测试结果如表3所示，实验结果表明本混淆系统不会破坏软件的原始功能，具有可用性。

表 3 可用性测试结果表

测试项目	测试结果
coreutils-8.32 源代码随机 4 个文件	正常执行，测试通过
gzip1.6 源代码	正常执行，测试通过
diffutils3.7 源代码	正常执行，测试通过
glibc2.33 源代码随机 4 个文件	正常执行，测试通过
MD5 开源实现代码 [23]	正常执行，测试通过
SHA1 开源实现代码 [24]	正常执行，测试通过

4.2 弹性评估

本文选用开源的coreutils-8.32中Linux命令cat、who、pwd和uname，以及glibc2.33中的cmp.c、qsort.c和MD5及SHA1的源码作为测试数据集，设计了3个评估指标，分别是T、FN以及FP。T表示混淆后的程序的全部路径均可被符号执行工具探索，无法有效对抗符号执行工具的攻击；FN表示混淆系统会创建不可达分支，并诱使符号执行工具去分析此路径，可有效对抗符号执行工具的攻击；FP表示混淆系统有效保护了程序的部分路径不被符号执行工具探索，可有效对抗符号执行工具的攻击。

表4将本文设计的6个混淆模块与其他3个开源混淆系统进行对比，针对KLEE和angr分别进行实验。实验表明，OLLVM、Armariris和Hikari并不能有效对抗KLEE和angr的攻击。另外，本文混淆系统分支复杂化模块设计的5种恒真谓词一定程度上降低了混淆特征，但是并不能对抗符号执行工具的求解；指令复杂化模块将简单的二元指令替换为复杂的同义指令集合，没有改变程序的控制流结构，也没有引入约束条

件求解的难题。因此这两个模块无法有效对抗符号执行工具攻击。其他4个模块表现良好。

表 4 弹性评估结果

混淆模块	KLEE	angr
未混淆	T	T
OLLVM	T	T
Armariris	T	T
Hikari	T	T
分支复杂化模块	T	T
指令复杂化模块	T	T
文件截断模块	FN	T
Hash 函数模块	FP	FP
DLP 函数模块	FP	FP
费马小定理模块	FP	FP

4.3 开销评估

为评估程序因为混淆而额外增加的开销，本文选用coreutils-8.32中Linux命令cat、who、pwd和uname的源码作为测试数据集合。

图7展示了本文混淆系统的各个混淆模块对程序进行混淆前后，程序所占用磁盘空间字节数。分支复杂化模块因为需要构造虚假代码块，所以存在较高的磁盘空间开销。但总体来看，本文混淆系统各个混淆模块混淆造成的额外磁盘空间开销并不显著。

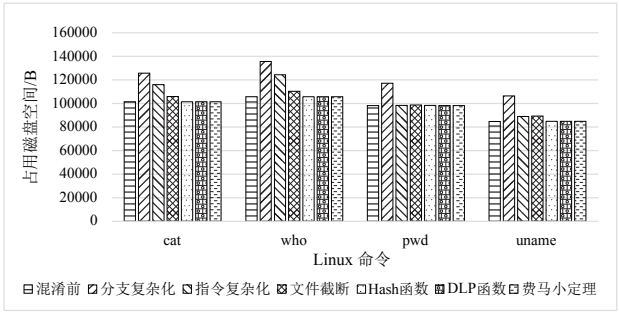

图 7 混淆前后磁盘空间对比

图8展示了本文混淆系统各个混淆模块对程序进行混淆前后，程序运行所需的时间。文件截断模块混淆的时间代价比较大，因为该模块需要创建文件，并存在读写文件操作，其他混淆模块并不会产生较大的时间开销。

4.4 测试总结

实验结果表明，本文混淆系统不会影响软件功能，并且相比另外3种开源混淆系统，本文混淆系统混淆后的软件可以有效对抗符号执行工具（KLEE和angr）

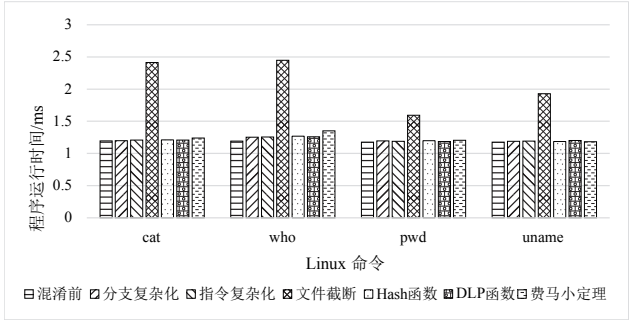

图 8 混淆前后运行时间对比

的攻击。此外，本文混淆系统带来的额外时间和空间开销并不显著。

5 结束语

本文设计并实现了可以提高程序复杂性、降低程序混淆特征，同时可对抗符号执行工具攻击的代码混淆系统。系统首先将不同编程语言通过编译器前端生成统一格式的IR文件，继而获取IR文件中的函数、基本块、指令等相关信息，根据混淆的目标，设计了一组混淆模块。这些混淆模块可以对源程序进行指令替换、控制流重构、插入外部函数调用、构造不可达路径、对分支条件的操作数进行加密等混淆操作，从而可以在目前符号执行技术逐渐成熟的情况下，有效保护软件的安全性，抵抗未授权攻击者对软件的攻击。但本文系统仍存在不足之处，如分支复杂化设计的恒真谓词集合不足、路径偏移相关的混淆方案的作用范围较窄等，未来工作将在此方面进行更深入的研究。

参考文献：

[1] COLLBERG C, THOMBORSON C, LOW D. Breaking Abstractions and Unstructuring Data Structures[C]// IEEE. The 1998 International Conference on Computer Languages, May 14–16, 1998, Chicago, IL, USA. Los Alamitos: IEEE Comp Soc, 1998: 28–38.

[2] BANESCU S, COLLBERG C, GANESH V, et al. Code Obfuscation Against Symbolic Execution Attacks[C]//ACSA. 32nd Annual Computer Security Applications Conference, December 5–9, 2016, Los Angeles, CA, USA. New York: Association for Computing Machinery, 2016: 189–200.

[3] BOYER R S, ELSPAS B, LEVITT K N. SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution[J]. ACM SigPlan Notices, 1975, 10(6): 234–245.

[4] CLARKE L A. A Program Testing System[C]//ACM. 1976 ACM Annual Conference, October 20–22, 1976, Houston, TX, USA. New

York: Association for Computing Machinery, 1976: 488–491.
[5] KING J C. Symbolic Execution and Program Testing[J]. Communications of the ACM, 1976, 19(7): 385–394.
[6] SALWAN J, BARDIN S, POTET M L. Symbolic Deobfuscation: From Virtualized Code Back to the Original[C]//SIDAR. 15th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA 2018, June 28–29, 2018, Saclay, France. New York: Springer Verlag, 2018: 372–392.
[7] YADEGARI B, DEBRAY S. Symbolic Execution of Obfuscated Code[C]//ACM SIGSAC. 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, October 12–16, 2015, Denver, CO, USA. New York: Association for Computing Machinery, 2015: 732–744.
[8] YADEGARI B, JOHANNESMEYER B, WHITELY B, et al. A Generic Approach to Automatic Deobfuscation of Executable Code[C]// IEEE Computer Society's Technical Committee (TC). 36th IEEE Symposium on Security and Privacy, May 18–20, 2015, San Jose, CA, USA. Piscataway: Institute of Electrical and Electronics Engineers Inc., 2015: 674–691.
[9] BRUMLEY D, HARTWIG C, LIANG Zhenkai, et al. Automatically Identifying Trigger–based Behavior in Malware[M]. New York: Springer, 2008.
[10] MING Jiang, XU Dongpeng, WANG Li, et al. Loop: Logic–oriented Opaque Predicate Detection in Obfuscated Binary Code[C]// ACM SIGSAC. 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, October 12–16, 2015, Denver, CO, USA. New York: Association for Computing Machinery, 2015: 757–768.
[11] TOFIGHI S R, CHRISTOFI M, ELBAZ–VINCENT P, et al. Dose: Deobfuscation Based on Semantic Equivalence[C]//ACM. 8th Software Security, Protection, and Reverse Engineering Workshop, SSPREW 2018, December 3–4, 2018, San Juan, PR, USA. New York: Association for Computing Machinery, 2018: 1–12.
[12] EYROLLES N, GOUBIN L, VIDEAU M. Defeating Mba–based Obfuscation[C]//ACM SIGSAC. 2nd International Workshop on Software PROtection, SPRO 2016, October 28, 2016, Vienna, Austria. New York: Association for Computing Machinery, 2016–27–37.
[13] GABRIEL F. Deobfuscation: Recovering An OLLVM–protected Program[EB/OL]. https://blog.quarkslab.com/deobfuscation–recovering–an–ollvm–protected–program.html, 2014–12–04.
[14] SCHRITTWIESER S, KATZENBEISSER S, KINDER J, et al. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?[J]. ACM Computing Surveys (CSUR), 2016, 49(1): 1–37.
[15] R1KK3R. Obfuscator–LLVM[EB/OL]. https://github.com/obfuscator–LLVM/obfuscator, 2017–06–29.
[16] WJQ. Armariris[EB/OL]. https://github.com/GoSSIP–SJTU/Armariris, 2019–06–03.
[17] NAVILLE. HikariObfuscator[EB/OL]. https://github.com/HikariObfuscator/Hikari, 2020–01–30.
[18] ZHOU Yongxin, MAIN Alec, GU Yuanxiang, et al. Information Hiding in Software with Mixed Boolean–arithmetic Transforms[EB/OL]. https://link.springer.com/chapter/10.1007/978–3–540–77535–5_5, 2007–08–27.
[19] XU Hui, ZHOU Yangfan, KANG Yu, et al. Manufacturing Resilient Bi–opaque Predicates against Symbolic Execution[C]//IEEE. 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, June 25–28, 2018, Luxembourg City, Luxembourg. Piscataway: Institute of Electrical and Electronics Engineers Inc., 2018: 666–677.
[20] SHARIF M I, LANZI A, GIFFIN J T, et al. Impeding Malware Analysis Using Conditional Code Obfuscation[C]//NDSS. 15th Annual Network and Distributed System Security Symposium, February 10–13, 2008, San Diego, CA, USA. Reston: Internet Society, 2008: 321–333.
[21] WANG Zhi, MING Jiang, JIA Chunfu, et al. Linear Obfuscation to Combat Symbolic Execution[C]//ESORICS. 16th European Symposium on Research in Computer Security, September 12–14, 2011, Leuven, Belgium. Berlin: Springer Verlag, 2011: 210–226.
[22] STEPHENS J, YADEGARI B, COLLBERG C, et al. Probabilistic Obfuscation through Covert Channels[C]//IEEE. 3rd IEEE European Symposium on Security and Privacy, EURO S and P 2018, April 24–26, 2018, London, UK. Piscataway: Institute of Electrical and Electronics Engineers Inc., 2018: 243–257.
[23] JIEWEIWEI. MD5[EB/OL]. https://github.com/JieweiWei/md5, 2014–10–24.
[24] RSWINDELL. SHA1[EB/OL]. https://github.com/clibs/sha1, 2021–03–24.