

## 基于浮点数类型转换和运算的不透明谓词构造方法

王庆丰, 梁浩, 王亚文, 谢根琳, 何本伟

(信息工程大学信息技术研究所, 河南 郑州 450001)

**摘要:** 随着软件功能的日趋复杂和网络攻击技术的不断演进, 软件盗版、软件破解、数据泄露、软件恶意修改等恶意行为呈上升趋势, 软件安全问题逐渐成为行业领域普遍关注的焦点和研究方向。代码混淆是一种典型的对抗逆向工程的软件保护技术, 它能够在保持程序原有功能不变的条件下加大攻击者对程序进行分析和理解的难度, 被广泛应用和深入研究。现有的代码混淆技术大多由于追求混淆效果而普遍存在性能损耗偏高、隐蔽性差等问题。控制结构混淆是代码混淆技术中应用较广泛的一种, 它通过扰乱程序的控制流从而提高代码逆向工程难度, 不透明谓词混淆是其一大分支。为了弥补现有代码混淆技术的缺陷, 提出了基于浮点数类型转换和运算的不透明谓词构造方法, 利用计算机浮点数类型转换和运算过程中伴随的精度损失现象使特定条件下产生与常理相悖的运算结果, 通过选择若干个小数进行强制类型转换、加法运算和乘法运算, 基于其运算结果统计可以构造一系列不透明谓词, 实现代码混淆功能。相较于传统的不透明谓词, 该构造方法具有隐蔽性高、通用性好、可逆性、开销低等优点。实验验证表明, 该方法在大幅降低攻击者对软件进行逆向工程等工作速度的同时, 对于符号执行等动态分析技术具有良好的抵御性能。

**关键词:** 代码混淆; 虚假控制流; 不透明谓词; 浮点数运算

**中图分类号:** TP393

**文献标志码:** A

**DOI:** 10.11959/j.issn.2096-109x.2023068

## Constructing method of opaque predicate based on type conversion and operation of floating point numbers

WANG Qingfeng, LIANG Hao, WANG Yawen, XIE Genlin, HE Benwei

Information Technology Research Institute, Information Engineering University, Zhengzhou 450001, China

**Abstract:** With the increasing complexity of software functions and the evolving technologies of network attacks, malicious behaviors such as software piracy, software cracking, data leakage, and malicious software modification are on the rise. As a result, software security has become a focal point in industry research. Code obfuscation is a common software protection technique used to hinder reverse engineering. It aims to make program analyzing and

收稿日期: 2022-09-05; 修回日期: 2023-01-30

通讯作者: 梁浩, lhmailhappy@163.com

基金项目: 国家自然科学基金 (62002383)

**Foundation Item:** The National Natural Science Foundation of China (62002383)

**引用格式:** 王庆丰, 梁浩, 王亚文, 等. 基于浮点数类型转换和运算的不透明谓词构造方法[J]. 网络与信息安全学报, 2023, 9(5): 48-58.

**Citation Format:** WANG Q F, LIANG H, WANG Y W, et al. Constructing method of opaque predicate based on type conversion and operation of floating point numbers[J]. Chinese Journal of Network and Information Security, 2023, 9(5): 48-58.

understanding more difficult for attackers while preserving the original program functionality. However, many existing code obfuscation techniques suffer from performance loss and poor concealment in pursuit of obfuscation effectiveness. Control flow obfuscation, particularly opaque predicate obfuscation, is widely used to increase the difficulty of code reverse engineering by disrupting the program's control flow. A method was proposed to address the limitations of existing code obfuscation techniques. It utilized the phenomenon of precision loss that occurred during type conversion and floating-point number operations in computers. Under certain conditions, this method produced operation results that contradict common sense. By performing forced type conversion, addition, and multiplication with selected decimal numbers, a series of opaque predicates can be constructed based on the statistical analysis of their operation results. This approach achieved code obfuscation with high concealment, good generality, reversibility, and low overhead compared to traditional opaque predicates. Experimental verification demonstrates that this method significantly slows down attackers' reverse engineering efforts and exhibits good resistance to dynamic analysis techniques such as symbolic execution.

**Keywords:** code obfuscation, bogus control flow, opaque predicates, floating point operations

## 0 引言

随着信息技术的发展,软件的应用领域越来越广泛、功能越来越丰富、复杂性越来越高、市场规模越来越大。软件行业规模的扩大不可避免地引入更多的软件缺陷,根据奇安信发布的《2022 中国软件供应链安全分析报告》<sup>[1]</sup>,每1 000行代码就有超过6个安全缺陷。与此同时,信息技术的发展让更多的人接触到计算机与网络技术,这间接导致发起攻击的门槛降低。软件带来便利的同时,人们的工作与生活、社会的正常运转等对软件的依赖程度越来越高,相应地,攻击者造成的软件安全问题所产生的危害越来越大。以软件盗版为例,全球范围内未经授权软件约占个人计算机软件的37%,其总价值超过460亿美元<sup>[2]</sup>。

对软件的攻击催生出众多软件保护技术,根据实现方式的不同可以分为基于硬件的保护技术和基于软件的保护技术<sup>[3]</sup>,基于软件的保护技术主要有软件水印、篡改防护、数据销毁、代码混淆。其中,代码混淆是一种典型的对抗逆向工程的软件保护技术,它可以通过重写代码中的部分逻辑或者修改标识符名称等实现,在保持程序原有功能不变的条件下对其进行修改,加大攻击者对程序进行分析和理解的难度<sup>[4]</sup>,已被广泛应用。围绕代码混淆的研究主要集中在控制结构混淆,许多研究成果处于理论阶段,相关的新技术往往忽视了实用性、通用性以及性能损耗而导致难以

推广。

针对已有的基于不透明谓词的代码混淆技术面临构造复杂、性能损失较大、不能抵抗符号执行代码分析技术等问题,本文提出基于浮点数类型转换和运算的不透明谓词构造方法,主要贡献如下。

1) 提出一种通过浮点数类型转换和运算的不透明谓词构造方法。

2) 以浮点数类型转换和运算为理论基础构造不透明谓词,以此为基础完成虚假控制流实验,并从强度、耐受性、开销、隐蔽性、通用性、可逆性,以及能否抵抗符号攻击7个方面进行评估。

## 1 相关研究

二进制代码是一种重要的计算机软件存在形式,也是非常普遍的软件分发形式。攻击者发起攻击在很大程度上依赖于对软件有充分的理解和分析,这要求攻击者必须对二进制代码进行逆向工程。Collberg等<sup>[5]</sup>阐述了代码混淆的概念,他们对代码混淆技术进行了系统性的总结分类,并通过强度、耐受性和开销3个方面对代码混淆质量进行评估。

根据处理目标的不同,代码混淆可以分为源代码混淆、中间代码混淆、字节码混淆、二进制代码混淆等。根据代码混淆原理的不同,代码混淆可以分为外形混淆、控制结构混淆、数据混淆、预防混淆等。其中,控制结构混淆是代码混淆技

术中应用最广泛的一种,其目的是使攻击者难以理解程序的控制流。常见的控制结构混淆技术包括虚假控制流<sup>[6]</sup>、控制流扁平化<sup>[7]</sup>、代码虚拟化<sup>[8-12]</sup>、不透明谓词<sup>[13-14]</sup>等。

不透明谓词是指在混淆时插入一些条件控制语句,其中,由软件开发者控制的条件表达式逻辑上恒真或者恒假,然而攻击者难以判断条件表达式的值。例如,对于任意整数  $x$ ,条件表达式“ $x(x+1)\%2==0$ ”恒成立,但是攻击者无法枚举  $x$  取值范围内的每一个值,无法简单得出该条件表达式恒成立的结论。不透明谓词可以与多项控制结构混淆技术结合,在控制结构混淆中占有重要地位。不透明谓词的质量直接决定了控制结构混淆的实际效果。

常用的不透明谓词往往基于简单的代理定理和逻辑规则,如“ $x(x+1)\%2==0$ ”恒成立,其构造简单,运行开销低,因此被广泛应用。然而,这类简单的不透明谓词有助于抵抗静态分析,但对于越来越多的反混淆手段却束手无策。因此,出现了更多的不透明谓词构造方法,大致可以分为基于困难问题的不透明谓词、基于密码学的不透明谓词和基于复杂代数运算的不透明谓词。

Collberg 等<sup>[13]</sup>提出基于别名分析问题以及并发分析问题的不透明谓词。其中,基于别名分析问题的不透明谓词首先构造一个复杂的动态数据结构,再构造一些指针指向这个数据结构中的元素,然后向程序代码中插入代码来编辑这个数据结构。在编辑时需要维护一些不变的性质,这些性质即不透明谓词的本质。后者原理与前者类似,区别在于编辑数据结构的代码位于并发执行的线程中。Moser 等<sup>[15]</sup>提出基于 3SAT (3-satisfiability) 问题的不透明谓词构造方法,该方法通过将程序中的  $n$  个布尔变量构造一个困难问题,从而构造出一个值恒定的表达式作为不透明谓词,假设攻击者必须先解决 3SAT 问题才能反混淆不透明谓词,则其反混淆难度为解决 NP 困难问题的难度。但是 Sheridan 等<sup>[4]</sup>指出 Moser 等的方法所依赖的假设并不成立,攻击者可以基于启发式检测策略来识别并反混淆此类不透明谓词。Tiella 等<sup>[16]</sup>基于

NP 困难问题中的分团问题实现构造不透明谓词,此类不透明谓词破解难度依赖于问题规模,而构造大规模问题将导致性能损耗严重,实用性差。此外,当 NP 问题不是最坏情况时,计算复杂度会降低,相应的安全性也会降低。

Sharif 等<sup>[17]</sup>提出一种基于密码学哈希函数的不透明谓词构造方法,把程序中常量与变量之间的相等判断转换为经过哈希函数之后常量与变量的比较,并与对称加密算法联合起来,实现了高强度的密码隐藏,该方法安全性高,但开销较大,且通用性较差。Zhu 等<sup>[18]</sup>提出基于余数系统的同态性来混淆代码中的整数运算。这两种方法安全性较高,但同样不具备通用性。

Arboit 等<sup>[19]</sup>提出利用二次剩余构造不透明谓词簇,该方法没有考虑到不透明谓词的弹性,且密码安全性较差。袁征等<sup>[14]</sup>基于同余方程生成不透明谓词,同二次剩余构造方法一样,这种不透明谓词容易受到符号执行攻击,且运算构造复杂。Wang 等<sup>[20]</sup>提出基于考拉兹猜想的不透明谓词构造方法,利用符号执行容易引起路径爆炸的缺陷,降低符号执行反混淆的效率,该方法同样灵活性差,容易受到特征匹配攻击。付蒙<sup>[21]</sup>提出基于置换函数的单射性的不透明谓词构造方法,通过使用比特向量上的置换函数,再结合函数复合的性质,可以构造关于置换函数单射性的恒等式,即不透明谓词,该方法构造简单,运行高效,可以作为备选的不透明谓词构造方法。

符号执行<sup>[22]</sup>是一种软件分析技术,也是较为流行的研究方向,在软件测试等领域具有较好的应用场景。Ming 等<sup>[23]</sup>提出符号执行与逻辑推理结合起来反混淆,采用不透明谓词的代码混淆后的二进制程序。该方案通过动态二进制插桩平台获取程序执行时的汇编指令并用符号执行框架对其进行分析,对程序中的每个谓词进行反向切片,生成执行到谓词所在路径的约束条件,最终通过约束求解和逻辑推理即可正确识别不透明谓词。针对只含有简单路径约束的不透明谓词,该方法能实现零误报率的不透明谓词检测,且运行效率高。Yadegari 等<sup>[24]</sup>提出基于污点分析和符号执行的反混淆技术。Bardin 等<sup>[25]</sup>提出有限后向动态符号执行来反混淆不透明谓词混淆。以上基于符号执行

的代码反混淆受限于约束求解器, 无法应对含有复杂路径约束的代码混淆。随着以符号执行的动态分析技术应用于代码反混淆, 能抵抗符号执行攻击的代码混淆技术研究逐渐增多。例如, Banescu 等<sup>[26]</sup>提出抵抗符号执行攻击的代码混淆方法能够将符号执行攻击的速度降低 4 个数量级, 但这种方法只能用于混淆循环语句, 实用性较低。

综上, 基于困难问题的不透明谓词往往混淆程序高、隐蔽性好, 但是构造复杂、开销较大、难以与原生代码融合。基于密码学的不透明谓词安全性较高, 但不具备通用性。基于复杂代数运算的不透明谓词往往灵活性较差, 容易受到软件动态分析技术攻击以及特征匹配攻击。

## 2 本文方法

## 2.1 精度损失现象

常用的数字包括整数和分数，两者的集合称为有理数。其中整数可以看作分母为 1 的分数，因此可以说有理数由分数组成。有理数既可以用分数表示，又可以用小数表示，其中小数分为有限小数和无限循环小数。任意分数一定可以转换为有限小数或无限循环小数，反之亦然<sup>[27]</sup>。全体有理数的集合为有理数集，它是无限集合。将有理数按照大小进行排序后，任意两个有理数之间必然存在其他有理数，即有理数具有稠密性。借助数轴来理解这一概念，数轴上任意两个点相连形成一个线段，无论这条线段有多短，在其上一定能找到一个新的有理数。

浮点类型是编程语言中一种基本数据类型，高级语言中浮点类型的存储方式源自 1985 年出现的 IEEE-754 标准。根据该标准，一个二进制浮点数由符号位、指数位以及尾数部分组成。

高级编程语言在设计基本数据类型时往往需要平衡取值范围和所占用的存储空间。在 C 语言中, float 数据类型在内存中的存储共占用 4 byte, 其中指数位占用 8 bit, 尾数部分占用 23 bit。double 数据类型在内存中的存储共占用 8 byte, 其中指数部分占用 11 bit, 尾数部分占用 52 bit。因此 float 数据类型理论上最多能表达 232 个数

字，而 `double` 数据类型理论上最多能表达 264 个数字，即无论是 `float` 还是 `double` 数据类型都只能表示有限个数字，且精度是有限的。`float` 和 `double` 数据类型有限的精度导致其在运算过程中可能出现严重的错误。

float 和 double 数据类型表示数据的范围是有限的, 而有理数是无限集合, 每一个 float 或 double 类型变量  $v$  都对应无数个有理数, 它们的分布范围可以表示为  $(v-e1, v+e2)$ , 其中  $e1$  和  $e2$  都是非常小的值。任意两个不等的  $(v-e1, v+e2)$  范围内的变量  $v1$  和  $v2$  使用 float 数据类型表达后是相等的。如果其中一个数字位于边界附近, 那么一点小小的扰动就可能导致其落在  $(v-e1, v+e2)$  范围外。以十进制小数举例, 假设一个数据类型只有 2 个有效的小数位, 所有在  $[1.665, 1.674)$  范围内的数字经过四舍五入都等于 1.67, 但是 0.001 的变量就可以让 1.674 四舍五入变成 1.68。

浮点数运算出现精度损失主要有两个原因：

①强制类型转换，`double` 类型变量转 `float` 类型会损失精度；② 浮点数数学运算，如 0.1 加 0.6 得到的实际计算结果不等于 0.7（不完全相等）。由于浮点数表示一个范围内的有理数，所以出现精度损失并不意味着计算结果一定出现错误。只有经过计算才可以确定某个或某几个浮点数经过类型转换或运算后的计算结果是否与逻辑上的计算结果相等。因此，可以将浮点数的类型转换或运算作为一种不透明谓词引入代码混淆。接下来用两个例子说明。

如图 1 所示，定义 `double` 类型变量 `double1` 并给 `double1` 赋值 0.1，然后定义 `float` 类型变量 `float1`，随后将变量 `double1` 赋值给变量 `float1`，最后将 `float1` 赋值给 `double` 类型变量 `double2`。此时的变量 `double1` 和 `double2` 的值在逻辑上是相等的，而且输出到控制台的结果都是 0.10000。然而以二进制格式打印的 `double1` 结果是 0, 01111111011, 100110011001100110011001100110，以二进制格式打印的 `double2` 结果是 0, 01111111011, 10011001100110011010000000000000000000000000000000000，所以变量 `double1` 的值小于变量 `double2`。通过仔细观察整个过程，可以发现在将 `double1` 赋值给



|               |               |                                 |
|---------------|---------------|---------------------------------|
| 0.1+0.6 计算过程: |               |                                 |
| 0.1           | 0, 01111011,  | 10011001100110011001101         |
| 0.6           | +0, 01111110, | 00110011001100110011010         |
| 0.1+0.6 计算结果: | 0, 01111110,  | 01100110011001100110 <u>100</u> |
| 0.7           | 0, 01111110,  | 01100110011001100110 <u>011</u> |

图 3 浮点数加法计算结果偏差原理示意

Figure 3 Illustration of the principle of deviation of the result of floating-point addition calculation

## 2.2 不透明谓词构造方法

根据上述计算过程中观察到的现象, 可以对若干浮点数对进行加法和乘法运算, 并统计其结果用作后续浮点数运算。如表 1 和表 2 所示, 分别是 0.1 到 0.9 这 9 个数字进行加法和乘法运算后实际结果和逻辑结果比较。例如, 表 1 中 0.1 对应的行和 0.6 对应的列 (第一行第六列的) “>” 符号表示在计算机浮点数运算中, 0.1+0.6 的计算结果大于其理论值 0.7。

表 1 浮点数加法运算结果示例  
Table 1 Example of floating point addition result

| 数字  | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0.1 | =   | =   | =   | =   | =   | >   | =   | >   | =   |
| 0.2 | =   | =   | =   | =   | =   | =   | =   | =   | =   |
| 0.3 | =   | =   | =   | >   | =   | >   | =   | =   | =   |
| 0.4 | =   | =   | >   | =   | =   | =   | =   | =   | =   |
| 0.5 | =   | =   | =   | =   | =   | =   | =   | =   | =   |
| 0.6 | >   | =   | >   | =   | =   | =   | =   | >   | =   |
| 0.7 | =   | =   | =   | =   | =   | =   | =   | =   | <   |
| 0.8 | >   | =   | =   | =   | =   | >   | =   | =   | =   |
| 0.9 | =   | =   | =   | =   | =   | =   | <   | =   | =   |

表 2 浮点数乘法运算结果示例  
Table 2 Example of floating point multiplication result

| 数字  | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0.1 | >   | >   | >   | >   | =   | >   | =   | >   | <   |
| 0.2 | >   | >   | >   | >   | =   | >   | =   | >   | <   |
| 0.3 | >   | >   | =   | >   | =   | =   | >   | >   | =   |
| 0.4 | >   | >   | >   | >   | =   | >   | =   | >   | <   |
| 0.5 | =   | =   | =   | =   | =   | =   | =   | =   | =   |
| 0.6 | >   | >   | =   | >   | =   | =   | >   | >   | =   |
| 0.7 | =   | =   | >   | =   | =   | >   | <   | =   | =   |
| 0.8 | >   | >   | >   | >   | =   | >   | =   | >   | <   |
| 0.9 | <   | <   | =   | <   | =   | =   | =   | <   | <   |

在验证浮点数类型转换与运算构造不透明谓词构造可行后, 可以通过算法使其工程化。首先

构造浮点数加法运算、乘法运算的浮点数对, 这些浮点数对满足计算结果与逻辑值不等。此外, 需要收集满足浮点数类型转换后精度损失明显的浮点数。在完成准备工作后, 即可开始不透明谓词混淆。首先收集可以进行不透明谓词混淆的点, 然后决定采用哪一种浮点数混淆方式, 并进行混淆。具体代码混淆原理如图 4 所示的虚假控制流示意, 每一个顺序执行的基本块都可以作为不透明谓词混淆的点, 在它前面插入一个 if 语句, 浮点数类型转换与运算不透明谓词作为 if 语句的判断条件, 使原顺序执行的基本块作为条件为真的跳转分支, 再构造一个虚假的基本块作为条件恒假的跳转分支。经过虚假控制流处理后的基本块控制流结构明显比原基本块复杂, 这将极大增加对代码进行逆向分析的难度。

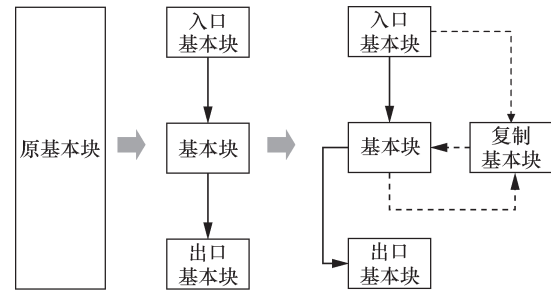


图 4 虚假控制流示意

Figure 4 Illustration of bogus control flow

基于浮点数类型转换和运算的不透明谓词混淆算法如算法 1 所示。

**算法 1** 基于浮点数类型转换和运算的不透明谓词混淆算法

输入 待混淆的 LLVM 中间代码 target\_code

输出 混淆后的 LLVM 中间代码 result\_code

开始

1) Add\_Array={<a1, a2>, <a3, a4>, ...}; // 构造浮点数加法参数

2) Multi\_Array={<b1, b2>, <b3, b4>, ...}; // 构造浮点数乘法参数

3) Cast\_Array={c1, c2, c3, c4, ...}; // 构造浮点数类型转换参数

4) ...

5) EntryPoint\_Array=Get\_EP(target\_code); // 找出所有的不透明谓词插入点

6) for each entriypoint in Entrypoint // 遍历

所有的插入点

```
7)   if Execute_Insert(rand_number) // 根据概率决定是否执行插入
8)       if Select_Add() // 选择浮点数加法作为不透明谓词
9)           Execute_Add(Add_Array);
10)      else if Select_Multi() // 选择浮点数乘法作为不透明谓词
11)          Execute_Multi(Multi_Array);
12)      else // 选择浮点数类型转换作为不透明谓词
13)          Execute_Cast(Cast_Array);
14)      end if
15)  end if
16) end for each
17) end
```

### 3 实验与评估

本节对提出的基于浮点数类型转换和运算的不透明谓词构造方法进行实验和评估。实验围绕6个基准测试程序进行基准测试，对每一个基准测试程序源代码，将其源代码编译成4个版本的可执行文件。基准测试包括程序执行时间、程序大小、程序相似度比较。完成基准测试后，根据基准测试结果从安全性、高效性、隐蔽性、通用性4个方面对提出的基于浮点数类型转换和运算的不透明谓词构造方法进行评估。

#### 3.1 实验过程

在实验展开之前，需要完成实验准备工作，包括实验环境构建和基准测试程序修改。实验环境包括程序编译执行环境和程序相似度比较环境，两个环境均通过虚拟机实现。程序编译执行环境安装ubuntu20.04操作系统，为其分配4 GB内存和20 GB磁盘。最后在该环境中安装LLVM12.0.1，用它编译基准测试程序。LLVM项目是模块化、可重用编译器和工具链技术的集合。开发人员可以借助LLVM将C++代码编译成LLVM中间代码，并在中间代码的基础上进行代码优化，最后生成二进制文件。程序相似度比较环境安装Windows10操作系统，为其分配4 GB内存和60 GB磁盘，最后在该环境中安装IDA pro7.5

和BinDiff5.0用于测量程序相似度。基准测试程序的源代码默认采用gcc进行编译，因此需要给它们重新编写采用LLVM编译的Makefile来生成实验需要的可执行文件。

IDA pro（简称IDA）是Hex-Rays公司出品的一款静态反编译软件，在程序分析过程中发挥着举足轻重的作用。IDA pro是交互式的、可编程的、可扩展的、多处理器的，支持Windows/Linux/macOS等平台文件格式。IDA pro已经成为分析恶意代码的标准和攻击研究领域的重要工具。BinDiff是一个二进制文件比较工具，可以协助安全研究人员和工程师快速定位反汇编代码中的差异和相似之处。BinDiff最早由Zynamics公司开发，该公司2011年被谷歌公司收购，随后谷歌将BinDiff融入许多内部文件分析系统中，利用其二进制对比技术追踪恶意程序家族。BinDiff还可以用于识别供应商提供的修补程序中的漏洞，分析多个版本相同的二进制数据，鉴别代码被盗以及专利侵权的证据收集等。

基准测试程序包括bzip2、mcf、gobmk、sjeng、lbm、sphinx3。对于每一个基准测试程序源代码，采用LLVM编译环境编译得到4个版本的可执行文件：原始可执行文件、虚假控制流混淆处理后的可执行文件、采用浮点数运算作为不透明谓词的虚假控制流混淆处理后的可执行文件、采用浮点数类型转换和运算作为不透明谓词的虚假控制流混淆处理后的可执行文件。在完成编译的基础上，统计这些程序的文件大小、执行时间，以及两种虚假控制流程序与原始程序的相似度比较。

##### 3.1.1 基准测试程序的运行时间

本节采用time命令测量基准测试程序运行时间，为了减小误差，将每一个可执行文件执行10次取其平均值作为其运行时间。表3为基准测试程序的运行时间，其中每一列包含指定基准测试程序运行时间。4行数据分别表示origin、origin+bcf1、origin+bcf2、origin+bcf3这4个版本的基准测试程序数据，其中origin表示直接编译基准测试程序得到的可执行文件，origin+bcf1表示采用基准测试程序经过示例虚假控制流混淆处理后的可执行文件，origin+bcf2表示基准测试程序经过

采用浮点数运算作为不透明谓词的虚假控制流混淆处理后的可执行文件，`origin+bcf3` 表示基准测试程序经过采用浮点数类型转换和运算作为不透明谓词的虚假控制流混淆处理后的程序。每一个版本的基准测试程序运行时间信息包括 `real time`、`user time` 和 `sys time`，三者分别表示程序实际运行时间、程序用户态运行时间、程序系统态运行时间。通过表 3 的数据可知：用户态运行时间加上系统态运行时间略小于程序实际运行时间；基准测试程序 4 个版本的可执行文件的系统态运行时间基本一致；所有基准测试程序的 4 个可执行文件运行时间大小均满足 `origin<origin+bcf3<origin+bcf2<origin+bcf1`。

表 3 基准测试程序运行时间  
Table 3 Benchmark run time

| 文件类型        | 基准测试程序  | 程序运行时间/s |          |       |
|-------------|---------|----------|----------|-------|
|             |         | 实际       | 用户态      | 系统    |
| origin      | bzip2   | 2.699    | 2.687    | 0.002 |
|             | mcf     | 470.588  | 470.255  | 0.246 |
|             | gobmk   | 0.153    | 0.136    | 0.011 |
|             | sjeng   | 5.042    | 4.954    | 0.078 |
|             | lbm     | 2.351    | 2.245    | 0.098 |
|             | sphinx3 | 0.352    | 0.332    | 0.015 |
| origin+bcf1 | bzip2   | 14.213   | 14.199   | 0.002 |
|             | mcf     | 1400.885 | 1400.324 | 0.273 |
|             | gobmk   | 0.941    | 0.929    | 0.008 |
|             | sjeng   | 44.066   | 43.973   | 0.077 |
|             | lbm     | 3.919    | 3.828    | 0.082 |
|             | sphinx3 | 1.022    | 0.992    | 0.023 |
| origin+bcf2 | bzip2   | 13.721   | 13.707   | 0.002 |
|             | mcf     | 1427.152 | 1426.625 | 0.231 |
|             | gobmk   | 0.855    | 0.846    | 0.005 |
|             | sjeng   | 34.516   | 34.428   | 0.07  |
|             | lbm     | 4.024    | 3.943    | 0.076 |
|             | sphinx3 | 0.98     | 0.957    | 0.018 |
| origin+bcf3 | bzip2   | 13.51    | 13.494   | 0.002 |
|             | mcf     | 1160.316 | 1159.864 | 0.239 |
|             | gobmk   | 0.517    | 0.506    | 0.004 |
|             | sjeng   | 22.4     | 22.322   | 0.067 |
|             | lbm     | 3.616    | 3.526    | 0.087 |
|             | sphinx3 | 0.719    | 0.696    | 0.018 |

### 3.1.2 基准测试程序的可执行文件大小

表 4 为基准测试程序的可执行文件大小。通过表中数据可知，所有基准测试程序的 4 个可执行文件大小满足 `origin<origin+bcf3<origin+bcf2<origin+bcf1`。

### 3.1.3 基准测试程序文件相似度比较

表 5 为基准测试程序的原始可执行文件与经虚假控制流混淆的可执行文件相似度比较。通过表中数据可知，除了 `sphinx3`，另外 5 个基准测试程序采用不同的虚假控制流混淆方法处理后的可执行文件与原始可执行文件相似度基本一致。

## 3.2 实验分析与评估

对不透明谓词的混淆一般从强度、耐受性、开销、隐蔽性、通用性方面进行评价，这里再增加一个指标：可逆性。此外，近年来符号执行技术开始被用于攻击不透明谓词混淆，且具有较好的效果，因此将它也作为一个评估方法。

强度是指混淆给原始程序增加了多少复杂度。软件复杂度评估有多种方法，McCabe QA 是其中一个比较常用的方法，McCabe 复杂度包括圈复杂度、基本复杂度、模块设计复杂度等。圈复杂度是评估软件复杂度的一个重要标准，可以用于衡量软件的复杂度和质量。不透明谓词方法只引入 3 行代码，基本上不会改变软件复杂度，即对强度基本没有影响。

耐受性是指变换后的程序对使用自动去混淆工具进行攻击的抵抗度。本文方法最大的弱点就是攻击者可以通过实际执行得到不透明谓词的真实值，这是大部分不透明谓词构造方法的固有缺陷。如果将该方法结合其他代码混淆方法，能较大程度上避免这一缺陷，同时只引入非常小的开销。

开销是指经过混淆变换后的程序相较原来的程序所产生的开销，如程序执行时间、程序所需存储空间等。如表 3 所示，本文方法在运行时间上明显优于 LLVM 提供的不透明谓词示例。如表 4 所示，本文方法在可执行文件所需存储空间方面明显优于 LLVM 提供的不透明谓词示例。实际上，这是因为本文方法本身产生 3 条指令，而 LLVM 提供的不透明谓词示例中间代码产生 17 条指令，因此无论是执行时间还是所需存储空间，本文方法都有较大的优势。

表4 可执行文件大小  
Table 4 Executable file size

| 文件类型        | 可执行文件大小/bit |         |            |           |         |           |
|-------------|-------------|---------|------------|-----------|---------|-----------|
|             | bzip2       | mcf     | gobmk      | sjeng     | lbm     | sphinx3   |
| origin      | 134 080     | 30 560  | 4 228 400  | 206 328   | 38 616  | 280 272   |
| origin+bcf1 | 1 915 344   | 305 744 | 20 450 280 | 3 149 064 | 173 984 | 3 703 776 |
| origin+bcf2 | 1 231 808   | 206 688 | 14 157 104 | 1 971 704 | 132 824 | 2 397 904 |
| origin+bcf3 | 936 896     | 161 632 | 11 552 048 | 1 471 992 | 116 440 | 1 840 848 |

表5 虚假控制流混淆前后可执行文件相似度比较  
Table 5 Comparison of executable file similarity before and after bogus control flow obfuscation

| 文件类型                  | 可执行文件相似度 |      |       |       |      |         |
|-----------------------|----------|------|-------|-------|------|---------|
|                       | bzip2    | mcf  | gobmk | sjeng | lbm  | sphinx3 |
| origin VS origin+bcf1 | 0.18     | 0.21 | 0.17  | 0.12  | 0.31 | 0.18    |
| origin VS origin+bcf2 | 0.15     | 0.22 | 0.14  | 0.12  | 0.32 | 0.13    |
| origin VS origin+bcf3 | 0.16     | 0.22 | 0.14  | 0.10  | 0.32 | 0.12    |

隐蔽性是指不透明谓词不具备明显的特征，可以用于衡量采用不透明谓词的混淆抵抗攻击者人工分析和特征匹配分析的能力。现有的大部分不透明谓词构造技术往往具备明显的特征，如基于 3SAT 问题的不透明谓词涉及较多的布尔操作，文献[20]提出的启发式检测策略能有效识别该不透明谓词构造技术。浮点数是 C/C++、Java、Python、PHP、JavaScript 等主流高级编程语言的基本数据类型，对其进行类型转换或运算不需要额外的支持，因此高级编程语言原生支持浮点数及浮点数类型转换或运算。并且，本文不透明谓词方法具有非常简短的代码，因此具有良好的隐蔽性。

通用性是指不透明谓词能用于多种编程语言，能运行于多种操作系统等。浮点数是 C/C++、Java、Python、PHP、JavaScript 等主流高级编程语言的基本数据类型，因此无论在任何操作系统上，本文不透明谓词方法都支持主流的高级编程语言。从程序生成过程的角度来看，本文方法不仅仅可以应用于源代码，还可以应用于以 LLVM IR 为代表的中间代码以及可执行文件。综上，本文方法具有良好的通用性。

可逆性是指使用软件所有者能否对经过混淆的软件进行反混淆。任何一个软件都有缺陷，经

过混淆后的软件同样如此。如果直接对混淆后的软件进行分析，则安全分析人员同样要面临软件逆向等方面的障碍。本文不透明谓词方法具有可逆性，只要提前构造好浮点数计算范围，就可以通过技术手段快速对混淆后的程序进行反混淆。

近年来，基于符号执行的程序分析技术被用于自动化反混淆不透明谓词混淆。与此同时，大部分不透明谓词，如条件表达式“ $x(x+1)\%2==0$ ”，往往具有固定的值，无法抵抗符号执行攻击。本文的不透明谓词方法并不具备固定的值，能够抵抗符号执行等软件动态分析手段。

## 4 结束语

本文提出一种基于浮点数类型转换和运算的不透明谓词混淆方法，利用计算机进行浮点数运算精度有限的原理构造不透明谓词。该方法可以在权衡软件安全性要求和软件性能，在产生较低开销的情况下增加对软件的保护；具有可逆性，在特殊情况下可以将混淆后的程序恢复；不仅能抵抗静态分析，还能在一定程度上抵抗动态分析，如所提方法生成的不透明谓词并不具备固定的值，能抵抗符号执行攻击。所提方法的可逆性是优点，也是容易被破解的缺点。因此后续需要构

造不透明谓词混淆框架, 所提方法与众多不透明谓词方法组合并在混淆时随机选择使用, 扩大不透明谓词的熵, 增大攻击者破解的枚举难度。

### 参考文献:

- [1] 奇安信代码安全实验室. 2022 中国软件供应链安全分析报告[EB]. Qi An Xin Code Security Lab. 2022 China software supply chain security analysis report[EB].
- [2] BSA Global Software Survey. Software management: security imperative, business opportunity[EB].
- [3] 张汉宁. 基于精简指令集的软件保护虚拟机技术研究[D]. 西安: 西北大学, 2010.  
ZHANG H N. Research on software protection virtual machine technology based on reduced instruction set[D]. Xi'an: Northwest University, 2010.
- [4] SHERIDAN B, SHERR M. On manufacturing resilient opaque constructs against static analysis[J]. Springer International Publishing, 2016.
- [5] COLLBERG C, THOMBORSON C, LOW D. A taxonomy of obfuscating transformations[J]. Technical Report, 1997.
- [6] 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, (4): 1-8.  
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, (4): 1-8.
- [7] LÁSZLÓ T, KISS A. Obfuscating C++ programs via control flow flattening[R]. 2009.
- [8] ROLLES R. Unpacking virtualization obfuscators[C]//Proceedings of the 3rd USENIX Conference on Offensive Technologies. 2009.
- [9] KUANG K, TANG Z, GONG X, et al. Exploiting dynamic scheduling for VM-based code obfuscation[C]//2017 IEEE Trustcom/BigDataSE/I SPA. 2017.
- [10] 陈耀阳. 基于代码混淆的软件保护技术与实现[D]. 南京: 南京邮电大学, 2021.  
CHEN Y Y. Research and implementation of software protection technology based on code obfuscation[D]. Nanjing: Nanjing University of Posts and Telecommunications, 2021.
- [11] 房鼎益, 张恒, 汤战勇, 等. 一种抗语义攻击的虚拟化软件保护方法[J]. 工程科学与技术, 2017, 49(1): 159-168.  
FANG D Y, ZHANG H, TANG Z Y, et al. DAS-VMP: a virtual machine-based software protection method for defending against semantic attacks[J]. Advanced Engineering Sciences, 2017, 49(1): 159-168.
- [12] 汤战勇, 李光辉, 房鼎益, 等. 一种具有指令集随机化的代码虚拟化保护系统[J]. 华中科技大学学报(自然科学版), 2016, 44(3): 28-33.  
TANG Z Y, LI G H, FANG D Y, et al. Code virtualized protection system with instruction set randomization[J]. Journal of Huazhong University of Science and Technology(Natural Science Edition), 2016, 44(3): 28-33.
- [13] COLLBERG C, THOMBORSON C, LOW D. Manufacturing cheap, resilient, and stealthy opaque constructs[C]//Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98). 1998: 184-196.
- [14] 袁征, 冯雁, 温巧燕, 等. 构造一种新的混淆 Java 程序的不透明谓词[J]. 北京邮电大学学报, 2007, 30(6): 103-106.  
YUAN Z, FENG Y, WEN Q Y, et al. Manufacture of a new opaque predicate for java programs[J]. Journal of Beijing University of Posts and Telecom, 2007, 30(6): 103-106.
- [15] MOSER A, KRUEGEL C, KIRDA E. Limits of static analysis for malware detection[C]//Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007). 2007.
- [16] TIELLA R, CECCATO M. Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation[C]//2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2017: 182-192.
- [17] SHARIF M, LANZI A, GIFFIN J, et al. Impeding malware analysis using conditional code obfuscation[C]//Proceedings of the Network and Distributed System Security Symposium (NDSS 2008). 2008.
- [18] ZHU W, THOMBORSON C. A provable scheme for homomorphic obfuscations in software security[R]. ACTA Press, 2005.
- [19] ARBOIT G. A method for watermarking Java programs via opaque predicates[C]//Proc Int Conf Electronic Commerce Research, 2002.
- [20] WANG Z, MING J, JIA C, et al. Linear obfuscation to combat symbolic execution[C]//European Symposium on Research in Computer Security. 2011.
- [21] 付蒙. 抵抗符号执行的不透明谓词混淆技术研究[D]. 西安: 西安电子科技大学, 2019.  
FU M. Research on opaque predicate-based obfuscations against symbolic execution[D]. Xi'an: Xidian University, 2019.
- [22] KING J C. Symbolic execution and program testing[J]. Communications of the ACM, 1976, 19(7): 385-394.
- [23] MING J, XU D, WANG L, et al. LOOP: logic-oriented opaque predicate detection in obfuscated binary code[C]//Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 2015: 757-768.
- [24] YADEGARI B, JOHANNESMEYER B, WHITELY B, et al. A generic approach to automatic deobfuscation of executable code[C]//2015 IEEE Symposium on Security and Privacy. 2015: 674-691.
- [25] BARDIN S, DAVID R, MARION J Y. Backward-bounded DSE: targeting infeasibility questions on obfuscated codes[C]//2017 IEEE Symposium on Security and Privacy (SP). 2017: 633-651.
- [26] BANESCU S, COLLBERG C, GANESH V, et al. Code obfuscation against symbolic execution attacks[C]//Proceedings of the 32nd

Annual Conference on Computer Security Applications. 2016: 189-200.

[27] 王国俊. 有理数、无理数与实数[J]. 数学的实践与认识, 1998(3): 258-266.

WANG G J . Rational, irrational and numbers [J]. Mathematics in Practice and Theory, 1998 (3): 258-266.



王亚文(1990-), 男, 河南郑州人, 信息工程大学助理研究员, 主要研究方向为拟态防御和云计算。

#### [作者简介]



王庆丰(1995-), 男, 河南周口人, 信息工程大学助理研究员, 主要研究方向为拟态防御和软件多样化。



谢根琳(1999-), 男, 河北辛集人, 信息工程大学硕士生, 主要研究方向为网络空间安全和软件多样化。



梁浩(1987-), 男, 河南郑州人, 信息工程大学副研究员, 主要研究方向为网络空间主动防御、内生安全。



何本伟(1998-), 男, 安徽滁州人, 信息工程大学硕士生, 主要研究方向为网络空间安全和软件多样化。