计算机工程与应用
Computer Engineering and Applications
ISSN 1002-8331,CN 11-2127/TP

《计算机工程与应用》网络首发论文

题目： LLVM 中间语言的控制流混淆方案
作者： 李成扬，黄天波，陈夏润，文伟平
网络首发日期： 2022-05-24
引用格式： 李成扬，黄天波，陈夏润，文伟平. LLVM 中间语言的控制流混淆方案[J/OL]. 计算机工程与应用.
https://kns.cnki.net/kcms/detail/11.2127.tp.20220523.1631.006.html

LLVM 中间语言的控制流混淆方案

李成扬，黄天波，陈夏润，文伟平

北京大学 软件与微电子学院，北京 102600

摘 要：软件安全问题在后疫情时代越发突出，代码混淆作为一种成熟的保护方案，借助于 LLVM 提供了跨平台使用的可能性，但基于 LLVM 的控制流混淆算法在保护力度上有所局限，一方面是现有的算法模式固定，缺乏结构上的创新性，另一方面是混淆处理时，未考虑到攻击者可以根据基本块的入度进行虚假块的预先判断，存在容易被识别的风险，因此提出两种算法：首先是嵌套 switch 混淆，打破固有的扁平化处理模式，通过在内部重新构造 switch 结构，增强对跳转变量的隐藏；其次是入度混淆，在虚假控制流中添加防入度分析策略，通过改变虚假块的入度规避虚假块的识别问题。在 LLVM10 上实现了方案原型并进行实验，结果表明：混淆方法在 1.5 倍内的时空开销内，相较于已有的控制流混淆方案，可以进一步降低 58.67% 的程序基本块相似度，增加 64.44%的跳转指令。

关键词：软件保护；代码混淆；控制流混淆
文献标志码: A **中图分类号**:TP309 doi：10.3778/j.issn.1002-8331.2112-0035

Control flow obfuscation scheme for LLVM intermediate languages

LI Chengyang, HUANG Tianbo, CHEN Xiarun, WEN Weiping

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

Abstract：Software security issues are becoming more prominent in the post-epidemic era, and code obfuscation as a mature protection scheme provides the possibility of cross-platform use with the help of LLVM. However, LLVM-based control flow obfuscation algorithms are limited in terms of protection strength, on the one hand, the existing algorithm model is immutable and lacks structural innovation, On the other hand, the obfuscation processing does not take into account the fact that attackers can base on the basic block, Therefore, two algorithms are proposed: firstly, nested switch obfuscation, which breaks the inherent flat processing model and enhances the hiding of the hopping amount by reconstructing the switch structure internally; secondly, indegree obfuscation, which adds an anti-entry degree analysis strategy to the false control flow to circumvent the false block by changing the indegree of the false block. The results show that the obfuscation method can further reduce 58.67% of the basic block similarity and increase 64.44% of the jump instructions compared to the existing control-flow obfuscation scheme within 1.5 times the temporal overhead.

Key words：software protection; code obfuscation; control flow obfuscation

后疫情时代背景下，软件为生产、生活及医疗提供便利和高效率的同时，始终面临内部存储数据的隐私性[1]和知识产权有效保护[2]的问题。与此同时，代码混淆[3]作为一种成熟的低成本软件保护技术得到广泛

作者简介：李成扬(1996－),男,硕士研究生,研究方向为代码混淆,E-mail: lcymoon@pku.edu.cn;黄天波(1997－),男，硕士研究生，研究方向：网络空间安全、恶意代码检测、代码混淆；陈夏润（1997-），男，研究生，主要研究方向：漏洞挖掘、软件安全防护；文伟平(1976—)，男，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。

使用。随着 IoT 的飞速发展，对不同架构和语言混淆处理的需求日益突出，特别是跨平台的混淆方案在当下成为新的研究热点。目前混淆的跨平台实现方案主要基于 LLVM 进行实现，但现有成熟的实现框架一方面较少，另一方面现有的算法缺乏内容上的创新、混淆特征较为明显等问题，特别是基于 LLVM 的控制流混淆，少有研究针对算法结构上进行创新。因此，针对现有控制流混淆[4]，本文在原有混淆算法基础上，借助 LLVM[5]的通用性，提出新的混淆算法，以增强现有的保护力度。

早期的软件保护关注的是盗版软件问题，企业会为了占据更多的市场份额而允许盗版的存在[6]，但不同于早期多是单机版的状态，现在的软件多是网络交互，涉及众多用户的隐私数据，因此现有的软件保护在考虑维护企业利益的同时，必须要为用户提供足够的安全保障。从攻击场景分析，使用文献[7]中的前提假设：软件运行在恶意的宿主机上，受到盗版、恶意逆向攻击和篡改的威胁。一般恶意的逆行工程是破解者根据逆向调试获取到必要的信息，然后根据自身的需求对程序进行篡改或者绕过某个或某些权限检查。在获取信息的途径上，根据是否运行程序，逆向调试可以分为静态和动态两种分析方式。静态分析工具如 IDA[1]，可以获取程序的控制流图，数据流图和类 C 伪码；动态调试工具如 Ollydbg[2]，可获取程序运行时具体的堆栈信息，获取实际的执行路径。从攻击者层面出发，给予足够的时间和精力，软件一定会被破解，但若付出的时间和精力成本大于破解获取到的收益，理性上推导认为攻击者没有动机进行这种破解行为；从保护者层面出发，软件安全是一个多面体的范畴，任何一个层面上的缺失都会导致安全性的降低，所以对于软件保护，追求的是全方位多层次的保护，而非单一层次的“绝对保护”。

为了实现软件安全，学术界和工业界提出了包括软件防篡改[8,9]、代码混淆[3,4]，软件多样性[10,11]、白盒加密[12,13]和硬件层级的可信平台[14,15]，以及软件水印[7,16]等技术。代码混淆增大了代码的理解难度，功能上和静态的软件防篡改、白盒加密有部分交叠，作为阻止破解者逆向分析软件的第一道防线，自上世纪 90 年代提出，发展至今，其效果从理论和实际生产环境中均得到了有效性的校验。本文针对代码混淆中的控制流混淆提出新的算法设计，基于 LLVM 平台实现跨平台的控制流混淆方案。

本文的贡献在于：1）在原有扁平化算法的基础上，实现 switch 结构的嵌套，进一步加强路径的复杂度的同时隐藏扁平化处理的原有痕迹；2）增加虚假块的入度混淆，抵抗针对虚假控制流的入度分析。经实验证明本文提出的混淆方法可以在较小的时空开销内获得较高的安全性。

1 相关工作

1.1 代码混淆

上世纪 90 年代，开始出现关于代码混淆的相关研究：1993 年 Cohen[17]明确提出指令替换和添加垃圾指令用于保护软件，但并没有提出代码混淆的概念范畴；Collberg 等[4]在 1997 年给出代码混淆的明确概念、分类和评价指标，从混淆所针对的不同属性，将混淆方法分为布局混淆、控制流混淆、数据混淆，从软件工程的角度提出了<potency, resilience, cost, stealth>的四元组评估指标[18]，其中 potency 指代的是混淆方法对程序复杂度的影响，resilience 指代的是针对混淆方法进行反混淆的难度，包括反混淆器的构造难度和反混淆器执行时所需的时空开销，cost 指代的是混淆方法对程序进行处理时，所需的时空开销，stealth 指代的是混淆方法添加的混淆代码与原程序代码的相似性，后期文献[19]从程序安全分析的角度提出了新的混淆评估指标，给出了结合<code, control flow, data, data flow>的四元组评估框架；2001 年，Barak[20]从密码学的角度对混淆方法的安全性进行求证，证明目前并不存在绝对安全的混淆器，混淆后的程序除输入和输出信息外，还会暴露其他的一些相关信息，进而提出了虚拟黑盒（virtual black box）的概念，，引出了代码混淆新的研究分支，从可证明的安全性角度出发，发展出面向模型的混淆方法，可以从理论上证明其提供的安全性，但其可用性是进一步发展中需要解决的难点之一。同时文献[21]讨论了面向模型混淆方法中提出的常见模型在安全性上的优劣；Chow[13]在 2002 年提出白盒加密的策略，为混淆中结合密码学的研究提供了新的思路。

跨平台的混淆方案在 Junod 等[22]提出基于 LLVM 的混淆框架 ollvm 开始得到关注。ollvm 中实现了控制流混淆中的虚假控制流和扁平化，数据混淆中的指令替换等，但因为其本身是基于 LLVM 4.0，对现有的一些平台特性无法提供良好的支持，另一方面是提供的混淆方法本身的混淆特征易被识别，在缺少创新性发展的情况下，无法有效的对现有的反混淆器产生良好

[1] https://hex-rays.com/ida-free/
[2] https://www.ollydbg.de/

的抵抗力。孤挺花项目[3]提出基于 LLVM 的字符串加密方案，在混淆处理时先识别出程序中的字符串常量，并与随机生成的 key 值进行异或处理得到加密后的字符串，而在程序真正执行时，先进行全局异或解密再执行。虽然加密算法的安全性存在进一步发展的空间，不过拓展了基于 LLVM 进行混淆的方法门类。Hou 其后期出现了 Hikari[4]对 LLVM IR 层字符串加密的安全性进行了增强。

控制流混淆使得程序的控制流图尽可能复杂，文献[23-25]阐述了控制流图在程序分析中的重要性。对于控制流混淆的研究主要集中在不透明谓词与扁平化的研究上：前者包括不同类别的不透明谓词构造方式[26,27]、不透明谓词用于移动代理的保护[28]，不透明谓词的安全性[29]以及文献[30]采用具体的真人实验展示了标识符重命名和不透明谓词对于软件保护的作用；后者包括对扁平化安全性的证明[31]，对扁平化的隐蔽性（stealth）[32]的论证以及通过使用轻量级的加密算法对跳转变量进行隐藏[33]从而在保证一定安全性的同时实现轻量级的扁平化。但现有的研究更多的针对扁平化的跳转变量进行处理，针对扁平化结构未做出新的设计。因此本文主要关注于控制流混淆中的虚假控制流和扁平化处理，因此对两者做进一步说明。

1.1.1 虚假控制流

定义 1（虚假控制流） 程序中所有的执行路径记为集合 A，在保证程序功能不变的前提下，构造路径 b，通过在原程序中嵌入不透明谓词 p，将路径 b 和集合 A 组成集合 A' 作为混淆后程序的执行路径集合。

针对概念做两点补充说明：

1. 集合 A 和 A' 的集合关系不固定。如果混淆并未改变程序中的基本块层级关系，移除路径 b 后，执行路径和未混淆前相同，则 A 是 A' 的真子集；若混淆打乱了基本块的层级关系，在语义等价的前提下，移除路径 b，执行路径不同于原生的执行路径集合，则 A 不是 A' 的真子集，甚至相交的程度依混淆算法设计所影响；

2. 不透明谓词 p 类型不做限定，可以是静态不透明谓词[18]或动态不透明谓词[27]。

虚假控制流添加的本质目的在于隐藏原生的控制流，使得程序在面对静态分析时，也可以有效的隐藏必要的真实信息，给分析者提供较多的冗余信息。方法本身的核心在于虚假块的构造和不透明谓词的处理。前者决定了添加的内容和原生内容融合的程度，后者决定了破解者手动或者通过自动化工具进行破解的难度。一般而言，虚假块依据原生程序的虚假块进行改造生成，例如添加、移除指令或将不同基本块内的指令进行组合。后者构造的必要条件是：构造的不透明谓词始终存在恒定关系，若是静态不透明谓词，则其值是恒真或者恒假，若是动态不透明谓词，则存在关联的谓词间，其数值上的对应关系是恒定的，同时谓词变量的定义域空间要足够大。

虚假控制流因为需要添加基本块，所以混淆后程序的体积增加较为明显。同时现有的虚假控制流构造中，为了进一步复杂化虚假块同原生程序的数据流关系，虚假块的入度一般低于真实块，若攻击者可判断出程序使用了虚假控制流处理，则完全有可能根据基本块的入度，对基本块本身是否为真实块做出一个概率性的预估，所以针对这种攻击方法，本文提出了入度混淆的概念。

1.1.2 扁平化

定义 2（扁平化） 程序的执行路径所构成的路径树记为 T，树的深度记为 D，在保证程序功能不变的前提下，对程序的基本块进行重组，使得重组后的路径树为 T'，其深度 D' 恒小于 D。

扁平化的核心在于原本存在上下层级的基本块变为横向的同一层级的基本块之间的调度，可简单理解为多 if-else 分支同 switch-case 的对比。因此，现有的扁平化实现思路多以循环结构内部嵌套 switch 实现原生的程序语义。

因为原本的执行流变为 switch 分发执行，扁平化在执行时间上的增加较为明显。同时，现有的扁平化研究中，主要是针对 switch 的跳转变量通过加解密或跳转表等形式进行隐藏，但针对与扁平化本身的结构创新研究较少，因此本文提出嵌套 switch 从扁平化的结构创新出发进一步强化现有扁平化的效果。

1.2 LLVM

Chris[5]在提出并实现 LLVM 时，将其定位为一个编译器框架，用于对任意程序提供透明的、持久化的程序分析和转换。透明指代的是使用者无需关注内部的实现细节；持久化指代的是分析和转换等优化工作不再局限于编译和链接时的优化，而是将优化推广到软件的全生命周期，包括编译、链接、装载、运行、离线优化，借助于自身的中间表示（Intermediate Representation, IR）为不同阶段的优化提供了可能性。

[3] https://github.com/GoSSIP-SJTU/Armariris
[4] https://github.com/HikariObfuscator/Hikari

LLVM 相较于传统的 GCC 等编译器，在代码的优化、可移植性和模块化设计上有其自身的优势。发展至今，LLVM 已经变为一个大型项目的统称，包含众多模块化和可重用的编译工具链相关的技术实现，如针对 C/C++的编译器 Clang，面向符号执行的 KLEE 等。

因为 LLVM 本身的三段式设计，将前端编译层、中间优化层、后端生成层进行了模块化的拆分，加上单一静态变量设计的 IR，使得针对于中间表示的优化工作只需关注于 IR 本身，可以不依附于具体的平台和语言，从而实现和前后端的抽离。因此，基于 LLVM IR 层进行混淆处理，为适用于不同语言和平台提供了可能性。本文基于 LLVM10 实现原型系统，通过实验印证方法本身的效果。

2 控制流混淆模型

针对现有的控制流混淆提出两种算法模式，分别从扁平化跳转变量的保护和基本块的入度进行设计。前者关注的是现有扁平化功能的增强，通过在一般扁平化后的结构中再次构造 switch 结构，实现在结构上创新的同时对跳转变量实现一定的保护，算法也可以拓展至虚假控制流的构造，如虚假块的构造方式；后者关注的是相邻基本块间的控制流保护，在基本块间加入新的数据流虚假块从而实现控制流分析中数据的复杂化，同时通过算法设计使得虚假块的入度高于真实块，从而避免针对虚假块的入度分析，方法可以归属于对现有虚假控制流的增强。

2.1 嵌套 switch 混淆

现有的扁平化处理中，将原生的具有上下层级关系的基本块通过 switch 分发转变为同一层级的形式，所有的基本块均汇聚到统一的分发点后，依据跳转变量 case 进行跳转的调度。case 的安全性是扁平化设计中的核心，保护思路包括对其拆分或者加解密。同时对于逆向攻击者而言，判断出程序是否进行过扁平化或虚假控制流混淆处理，这本身也是一种利于程序分析的有效信息。因此在扁平化的保护上，提出在内部再次添加 switch-case 结构，将扁平化后的层级再次拉伸成多层关系，同时借鉴虚假控制流中基本块的构造方式，为了尽可能的减少混淆后程序的体积，在虚假块中仅对程序中的跳转变量 case 进行拆分处理。通过在构造的基本块中对 case 进行混淆处理，从而隐藏真实的跳转 case 构造。算法伪码如表 1 所示。

表 1 嵌套 switch 混淆算法

Table 1 Nested switch obfuscation

Algorithm 1: Nested switch obfuscation

Data: *IR files*, *caseNum*
1. Iterating through the basic blocks in programs, noted as the set *originBBs*
2. getting *entryBlock* and *exit-Block*, $BBs = \{x \mid x \in originBBs, x \neq entryBlock 且 x \neq endBlock\}$
3. generating loop structure, embedded switch-case in it
4. for (*BB* : *BBs*){ // attach case value
5. mapping case value to *BB*
6. }
7. for (*BB* : *BBs*){ // reset jump case
8. *succBBs* ← successor blocks to *BB*
9. for (*succBB* : *succBBs*){
10. *iCase* ← case value attached to *succBB*
11. storing *iCase* in *BB* and perform basic block jumping according to *iCase*
12. }
13. }
14. for (*BB* : *BBs*){ // split case value and generate new switch
15. continue if BB is the default block for switch
16. *iCase* ← case value to the successor block
17 *low*, *high* = split *iCase* according to bits
18 generating new switch containing *caseNum* cases
19 in the basic block corresponding to each case, random generating arithmetic and logical operations for *low* and *high*
20 reorganize low and high into *iCase*
21 }

用户首先指定待创建的 switch 中 case 的数量，即 *caseNum* 的数值，考虑到空间开销的问题，默认值为当前程序中基本块的数量和 10 的最小值；然后构造出扁平化的框架，即外部的无限循环加内部的 switch 分发跳转；再针对基本块的跳转变量 *iCase* 进行处理，将其按照高低位拆分为 *low* 和 *high*；之后生成指定的 *caseNum* 的 switch 结构，在每个 case 中针对 *low* 和 *high* 随机生成算数和布尔运算，最后再合并 *low* 和 *high*，重新赋值为 *iCase*。算法的创新点在于 switch 结构的嵌套构造，通过对跳转变量进行拆分，添加随机化的算数或布尔运算从而进一步模糊化 *iCase* 的值，即下一个后继块对应的 case 值。而程序的原本待执行的后继块的 *iCase* 值存放于 default 分支中。当然也可以在重组 *iCase* 后，再对后继块的 case 值进行刷新或者先获取 *iCase* 值，在对应的基本块内使用算数或布尔运算重组出对应的值避免更改后续的 *iCase*，无论是这里的“后更改”还是“前更改”均可以避免真正的后继块地址始终存放在 default 分支中，从而提供更高的安全性，但相较于改变 *iCase* 的值，本文试图探讨如同 default 分支和 switch 的逻辑跳转块这样的相邻局部基本块间关系的隐藏，即基本块的出入度算法试图解决的问题。

2.2 入度混淆

当针对程序中相邻的少量基本块的保护时，在不过多添加空间资源消耗的情况下，隐藏基本块间的跳转关系，避免基本块间的入度分析是混淆算法需要考虑的一个方面。本文通过在构造虚假块时，收集前后基本块中的数据，在不影响原生基本块数值的前提下，

在虚假块中构造新的相关变量的处理，从而加强虚假块同原生块的数据依赖关系，如图 1 所示，通过收集基本块 A、B、C 中的数据，在虚假块 blockOne 和 blockTwo 中构造相应的逻辑处理，并通过在虚假块 blockOne 的末尾嵌入不透明谓词，使程序执行时，只会执行左侧 B、C 的路径即可。进而能否移除 blockOne 和 blockTwo，取决于不透明谓词的隐藏性和两个新增块内对原生数据流的处理逻辑。

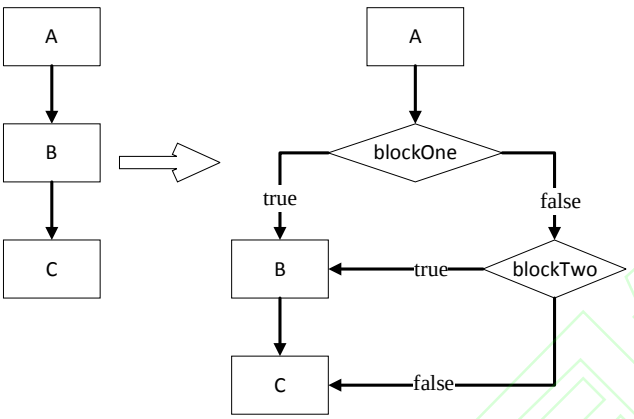

图 1　虚假块的添加

Fig.1　Addition of bogus blocks

但这样的设计，在面对程序的出入度分析时可能极容易被识破。当破解者有能力识别出程序已经进行了虚假控制流混淆的操作时，针对图 1 所示的模型分析，会发现基本块 blockOne 的后继块 B 和 blockTwo 中，明显块 B 的入度多于块 blockTwo，即在嵌入虚假块时，一般会产生指向原生基本块的边，换言之，入度可以作为一种指标评判是否是真实块还是虚假块，从而跳过不透明谓词或者数据流分析的问题。

先设计一种入度混淆算法，类比于图 1，在添加虚假块时，修改原有基本块的跳转关系，增大虚假块的入度。如图 2 所示，虚假块 blockOne、blockTwo 和 blockThree 的构造亦是通过收集基本块 A、B、C 中的数据，并添加随机的算数和逻辑运算；在基本块 A、B 和 blockOne 中添加不透明谓词，修改真实块 A、B 的跳转关系实现虚假块的入度多于真实块，同时在添加基本块时考虑到 blockOne 在程序执行时是一定会执行的，在 blockOne 中仅可以使用原生的数据，而在 blockTwo 和 blockThree 中除了使用，还可以改变原生的数据，因为在真正执行时 blockTwo 和 blockThree 所在的路径是不会被执行的。

同时因为涉及到原生跳转关系的修改，如果基本块 A 和 B 均含有两个后继块时，可以考虑将其转变为 switch-case 的情况从而去除跳转关系的限制。因此对于扁平化设计中提出的问题，可以使用该方法对新创建的 switch-case 结构中的基本块再次处理，从而进一步隐藏 default 块和其他 case 块的差异，进一步隐藏数据。

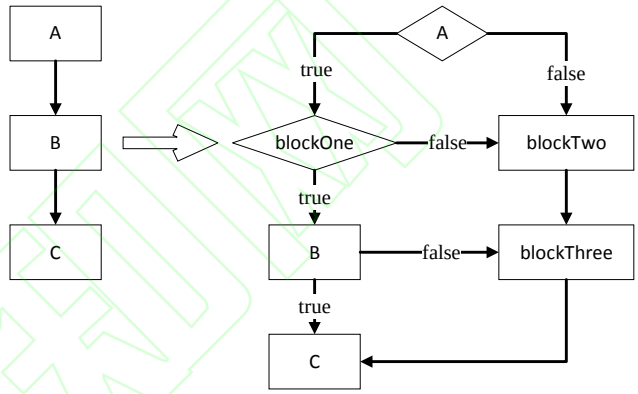

图 2　虚假控制流入度构造

Fig.2　Indegree of bogus control flow

3.　实验分析

OLLVM[22]作为 LLVM 平台上实现混淆算法的先行者，提供了控制流扁平化、虚假控制流和指令替换的混淆方法。孤挺花项目，基于 OLLVM 添加了针对于字符串的加解密混淆，虽然使用的加密方式较为简单，但是在字符串混淆方面提供了可供借鉴的新思路。Hikari 提供了基于寄存器的间接跳转和不同于孤挺花的字符串混淆策略。因为本文关注的是控制流混淆，因此同原生的 OLLVM 进行对比，分别从算法的正确性、混淆效果、时空开销进行实验说明。

算法的正确性指代可以成功生成可执行文件且混淆后的可执行文件可正常执行；混淆效果，使用 BinDiff[5]分析混淆后可执行文件，从基本块、跳转指令、函数和整体相似度上对比以展示效果；时空开销，为了尽可能的减少实验误差，使用脚本文件统计对应的数值，对于同一文件，连续执行 10 次取执行时间的平均值作为运行时间，取可执行文件的大小作为空间大小。

3.1　实验数据及环境

基于 LLVM 的混淆方案的通用性，在文献[34]中已经得到证明。本文实验数据选择 github 上 c 语言实

[5] https://www.zynamics.com/bindiff.html

现的算法库[6]；实验环境为在 Inter(R) Core(TM) i5-4210U、4G 内存的 Ubuntu18.04 64 位机上，基于 LLVM10 进行实现。

同时为了方便脚本的自动执行，对算法中涉及到输入的部分，由用户输入转为固定值的输入，且去除了部分输入复杂的样例，最后仅保留 124 个源文件。相关的测试文件和后文实验分析所依据的数据，可以从 gitee 库[7]中获取。

3.2 结果及分析

从表 2 中可以看出，针对 124 个源文件，OLLVM 和嵌套 switch 混淆均可以生成可执行文件，且均可以正常执行。在使用入度混淆处理时，因为修改了原生程序中的跳转关系，对于其中涉及到的 phi 指令需要在混淆完成后，修正对应的关系。在具体实现时先标识出程序中的 phi 指令，然后将其转变为对应的堆栈指令，从而避免 phi 指令引发的错误。从目前的测试样本中可以看出，本文提出的嵌套 switch 混淆和入度混淆是完全可以混淆所有文件的。

表 2 算法的正确性

Table 2 Correctness of algorithms

混淆方法	生成的可执行文件	可正常执行的文件
OLLVM 扁平化	124	124
嵌套 switch 混淆	124	124
入度混淆	124	124

因为本文关注的是控制流混淆，因此混淆后的基本块、跳转指令、函数的相似度和程序混淆前后的相似度可以做为衡量混淆效果的指标之一。因此，使用 BinDiff 在算法样例中，对上述四者进行统计。表 3 是 OLLVM 混淆后的指标统计结果，表 4 是嵌套 switch 混淆后的指标统计结果，表 5 是使用嵌套 switch 和入度混淆后的指标统计结果。因为入度混淆的作用在于对真实块和虚假块的跳转关系进行转变，虚假块的构造方法上同 ollvm 一致，因此不再单独对其与源文件的相似度进行统计。

表 3 OLLVM 混淆文件与源文件的相似度

Table 3 Similarity of OLLVM obfuscated files to source files

OLLVM 扁平化	平均值	最小值	最大值	标准差
基本块相似度	68.10	40.60	100.00	13.17
跳转指令相似度	35.65	12.40	100.00	21.84
函数相似度	72.65	52.40	90.80	9.18
整体相似度	0.74	0.55	0.99	0.10

表 4 嵌套 switch 与源文件的相似度

Table 4 Similarity of nested switch to source files

本文方法	平均值	最小值	最大值	标准差
基本块相似度	40.48	15.10	100	20.81
跳转指令相似度	20.58	4.00	100	23.17
函数相似度	72.63	52.40	90.80	9.20
整体相似度	0.58	0.40	0.99	0.14

表 5 嵌套 switch+入度混淆与源文件的相似度

Table 5 Similarity of nested switch, entry degree obfuscation to source files

本文方法	平均值	最小值	最大值	标准差
基本块相似度	31.31	11.1	100	18.81
跳转指令相似度	14.43	2.6	100	19.20
函数相似度	72.63	2.60	90.80	9.20
整体相似度	0.51	0.36	0.99	0.13

首先，从表 3 和表 4 对 OLLVM 和嵌套 switch 的对比分析中可以看出，本文提出的嵌套 switch 相较于 OLLVM：1）可以有效的降低基本块的相似度。因为添加内嵌的 switch 结构，相较于 OLLVM，基本块的相似度下降 40.56%，但对应的标准差明显高于 OLLVM，证明嵌套 switch 的效果因程序不同而有较大差异，方法本身的适用性还有待进一步提高；2）跳转指令上略有下降。因为目前的测试文件以算法文件为主，对于单文件的基本块数量有限，所以构造的跳转指令有限；3）函数相似度上基本无差异，因为混淆算法本身未涉及到相关的函数。

其次，从表 4 和表 5 的数据分析中可以看出：1）入度分析结合嵌套 switch 处理后在基本块的相似度和指令的相似度上均有明显的强化作用，基本块的相似度相较于 OLLVM 下降 54.02%，跳转指令的相似度上下降 59.52%；2）在函数相似度上亦无差异。

最后，需要做补充说明的是，相似度是混淆效果的指标之一，但如果只是单纯的增加基本块和跳转指令的数量，这样的处理方式并不会对实质的逆向破解起到对应的抵抗作用，因为本文在基本块和跳转指令的构造上，结合原程序本身的控制流和数据流进行处理，因此在混淆效果上是有一定保障的。

表 6 为自动化脚本统计的混淆后文件相较于未混淆的文件，在时空开销上的增长倍数：OLLVM 几乎无增加，本文方法在空间上的开销约在 1.6 倍内，时间几乎无增加。排除一部分不满足混淆条件的程序的无效干扰，可以论证本文方法总体上开销较小。当然算法的测试集中以单文件的算法为主，在大型项目

[6] https://github.com/mandliya/algorithms_and_data_structures

[7] https://gitee.com/lcy20/paper_data

中使用时，可能会出现开销增大的情况，因此在具体的项目使用时，可以针对具体的算法文件进行混淆处理，因为未改变函数名称，所以混淆后的文件在原项目中的编译链接应当是不存在影响的。

表 6　混淆方法的时空开销

Table 6　Temporal overhead of obfuscation methods

混淆方法	时间开销	空间开销
OLLVM 扁平化	1.01	1.05
嵌套 switch 混淆	1.03	1.35
嵌套 switch+入度混淆	1.04	1.62

针对测试数据集，使用 angr[35]对算法文件的最终执行路径进行求解，统计其对应的时间，并将测试集的平均值汇总到表 7。其中，针对测试文件 word_search.cpp.out 和 bubbleSortDemo.cpp.out 的分析时间超过 5 分钟，认定为超时；针对其他文件，对待分析的路径可以有效进行分析，但相对应的时间开销上有所增长。

表 7　angr 符号执行的平均时间开销

Table 7　Average time overhead of angr symbol execution

混淆方法	时间开销
OLLVM 扁平化	1.13
嵌套 switch 混淆	1.18
嵌套 switch+入度混淆	1.27

4　结束语

本文提出基于 LLVM 中间代码的控制流混淆增强方案，通过构造嵌套的 switch 结构，隐藏真实的跳转变量，增强扁平化混淆的安全性；通过增大虚假块的入度，抵抗对基本块的入度分析，在尽可能的减少空间开销的前提下，隐藏相邻基本块的数据流和跳转关系。未来的工作包括：虽然本文提出的方法可以极大的增加程序控制流图的复杂度，但是对于符号执行对特定路径的分析上效果有限，可以添加针对符号执行的控制流混淆设计；在 LLVM 平台上实现低开销的虚拟机软件保护，以对抗针对软件的动态调试，增强动态保护能力。

参考文献:

[1] GABRIELSSON J, BUGEJA J, VOGEL B. Hacking a Commercial Drone with Open-Source Software: Exploring Data Privacy Violations[C]//2021 10th Mediterranean Conference on Embedded Computing, MECO 2021,2021.

[2] C. S. MACHADO R, R. BOCCARDO D, SÁ V G P, et al.Software control and intellectual property protection in cyber-physical systems[J].Eurasip Journal on Information Security,2016, 2016 (1).

[3] XU H, ZHOU Y, KANG Y, et al.On Secure and Usable Program Obfuscation: A Survey[J]. ArXiv, 2017, abs/1710.01139.

[4] COLLBERG C, THOMBORSON C, LOW D.A Taxonomy of Obfuscating Transformations[J].technical report,1997.

[5] LATTNER C, ADVE V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//2nd International Symposium on Code Generation and Optimization,2003: 75-86.

[6] SHY O, THISSE J F.A strategic approach to software protection[J].Journal of Economics & Management Strategy,1999, 8 (2): 163-190.

[7] COLLBERG C S, THOMBORSON C.Watermarking, tamper-proofing, and obfuscation - Tools for software protection[J].Ieee Transactions on Software Engineering,2002, 28 (8): 735-746.

[8] BERLATO S, CECCATO M.A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps[J].Journal of Information Security and Applications,2020, 52: 28.

[9] MERLO A, RUGGIA A, SCIOLLA L, et al.ARMAND: Anti-Repackaging through Multi-pattern Anti-tampering based on Native Detection[J].Pervasive Mob. Comput.,2021, 76: 101443.

[10] TEMIZKAN O, PARK S, SAYDAM C.Software Diversity for Improved Network Security: Optimal Distribution of Software-Based Shared Vulnerabilities[J]. Information Systems Research,2017, 28 (4): 828-849.

[11] LARSEN P, HOMESCU A, BRUNTHALER S, et al. SoK: Automated Software Diversity[C].35th IEEE Symposium on Security and Privacy (SP),2014: 276-291.

[12] BANIK S, BOGDANOV A, ISOBE T, et al.Analysis of Software Countermeasures for Whitebox Encryption[J]. Iacr Transactions on Symmetric Cryptology,2017, 2017 (1): 307-328.

[13] CHOW S, EISEN P, JOHNSON H, et al.White-box cryptography and an AES implementation, Nyberg K, Heys H,editor,Selected Areas in Cryptography[R], Berlin: Springer-Verlag Berlin,2003: 250-270.

[14] HAN S, SHIN W, PARK J H, et al. A bad dream: Subverting trusted platform module while you are sleeping[C].Proceedings of the 27th USENIX Security Symposium,2018: 1229-1246.

[15] YU Z L, ZHANG W P, DAI H J.A Trusted Architecture for Virtual Machines on Cloud Servers with Trusted Platform Module and Certificate Authority[J].Journal of Signal Processing Systems for Signal Image and Video

Technology,2017, 86 (2-3): 327-336.

[16] MA H Y, JIA C F, LI S J, et al.Xmark: Dynamic Software Watermarking Using Collatz Conjecture[J].Ieee Transactions on Information Forensics and Security,2019, 14 (11): 2859-2874.

[17] COHEN F B.Operating system protection through program evolution[J]. Computers and Security, 1993, 12 (6): 565-584.

[18] 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,1998: 184–196.

[19] ANCKAERT B, MADOU M, DE SUTTER B, et al. Program Obfuscation: A Quantitative Approach[C].3rd Workshop on Quality of Protection/14th ACM Computer and Communications Security Conference,2009: 15-20.

[20] BARAK B, GOLDREICH O, IMPAGLIAZZO R, et al. On the (Im)possibility of Obfuscating Programs[J]. Journal of the Acm,2012, 59 (2): 47.

[21] KUZURIN N, SHOKUROV A, VARNOVSKY N, et al. On the concept of software obfuscation in computer security[C].10th International Conference on Information Security,2007: 281-+.

[22] JUNOD P, RINALDINI J, WEHRLI J, et al. Obfus- cator-LLVM-Software Protection for the Masses[C]// IEEE/ACM 1st International Workshop on Software Protection (SPRO),2015: 3-9.

[23] 陈耀阳, 陈伟.采用隐式跳转的控制流混淆技术[J].计算机工程与应用,2021, 57 (20): 125-132.

Chen Y, Chen W.Control Flow Obfuscation Technology Based on Implicit Jump[J].Computer Engineering and Application,2021, 57 (20): 125-132.

[24] LIN Y. Novel Techniques in Recovering, Embedding, and Enforcing Policies for Control-Flow Integrity[M]. Springer,2021: 1-95.

[25] PHU T N, THO N D, HOANG L H, et al.An Efficient Algorithm to Extract Control Flow-Based Features for IoT Malware Detection[J].Computer Journal,2021, 64 (4): 599-609.

[26] MING J, XU D P, WANG L, et al. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code[C].22nd ACM SIGSAC Conference on Computer and Communications Security (CCS),2015: 757-768.

[27] XU D P, MING J, WU D H. Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method[C].19th Annual International Conference on Information Security (ISC),2016: 323-342.

[28] Majumdar A, Thomborson C. On the use of opaque predicates in mobile agent code obfuscation[C].Kantor P, Muresan G, Roberts F, Zeng D D, Wang F Y, Chen H, Merkle R C,editor,Intelligence and Security Informatics, Proceedings, Berlin: Springer-Verlag Berlin,2005: 648-649.

[29] Majumdar A, Thomborson C, Drape S. A survey of control-flow obfuscations[C]//2nd International Con- ference on Information Systems Security, 2006: 353-+.

[30] Ceccato M, Di Penta M, Nagra J, et al. Towards experimental evaluation of code obfuscation techniques[C]// Proceedings of the ACM Conference on Computer and Communications Security,2008: 39-45.

[31] Blazy S, Trieu A. Formal Verification of Control-Flow Graph Flattening[C].5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP),2016: 176-187.

[32] Kanzaki Y, Monden A, Collberg C, et al. Code Artificiality: A Metric for the Code Stealth Based on an N-gram Model[C].IEEE/ACM 1st International Workshop on Software Protection (SPRO),2015: 31-37.

[33] JOHANSSON B, LANTZ P, LILJENSTAM M, et al. Lightweight Dispatcher Constructions for Control Flow Flattening[C].7th Software Security, Protection, and Reverse Engineering Workshop (SSPREW),2017.

[34] 田大江,李成扬,黄天波,等.基于底层虚拟机的标识符混淆方法[J/OL].计算机应用 （2021-10-9）[2022-04-24]. http://kns.cnki.net/kcms/detail/51.1307.TP.20211008.1407.002.html

TIAN D J, LI C Y, HUANG T B, et al. Identifier obfuscation method based on low level virtual machine[J]. Journal of Computer Applications (2021-10-9) [2022- 04-24]. http://kns.cnki.net/kcms/detail/51.1307.TP.20211008.1407.002.html

[35] SHOSHITAISHVILI Y, WANG R Y, SALLS C, et al. (State of) The Art of War: Offensive Techniques in Binary Analysis[C].IEEE Symposium on Security and Privacy (SP),2016: 138-157.