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

doi:10.11772/j.issn.1001-9081. 2021071166

基于底层虚拟机的标识符混淆方法

田大江，李成扬，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: 针对现有代码混淆仅限于某一特定编程语言或某一平台，并不具有广泛性和通用性，以及控制流混淆和数据混淆会引入额外开销的问题，提出一种基于底层虚拟机（LLVM）的标识符混淆方法。该方法实现了 4 种标识符混淆算法，包括随机标识符算法、重载归纳算法、异常标识符算法以及高频词替换算法，同时结合这些算法，设计新的混合混淆算法。混淆方法首先在前端编译得到的中间文件中候选出符合混淆条件的函数名，然后使用具体的混淆算法进行处理，最后使用具体的编译后端将混淆后的文件转变为二进制文件。实验结果表明基于 LLVM 的标识符混淆方法适用于 LLVM 支持的语言，不影响程序正常功能，时空开销在 1.5 倍内，且标识符混淆程度稳定，平均混淆比率在 77.5%。相较于已有的随机标识符混淆方法和重载混淆方法，具有性能开销小，隐蔽性强，通用性广的特点。

关键词: 软件保护；代码混淆；标识符混淆；底层虚拟机（LLVM）；混淆方法

中图分类号:TP312　　**文献标志码: A**

Identifier obfuscation method based on low level virtual machine

TIAN Dajiang, LI Chengyang, HUANG Tianbo, WEN Weiping*

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

Abstract: Most of the existing code obfuscation solutions are limited to a specific programming language or a platform, which are not widespread and general. Moreover, control flow obfuscation and data obfuscation introduce additional overhead. Thus identifier obfuscation method was proposed based on the Low Level Virtual Machine (LLVM). Four identifier obfuscation algorithms were implemented in the method, including random identifier algorithm, overload algorithm, exception identifier algorithm, and high-frequency word replacement algorithm. And a new hybrid obfuscation algorithm was designed by combining these algorithms. Firstly, in the intermediate files compiled by the front-end, the function names, which met the obfuscation criteria, were selected by the obfuscation method. Secondly, the function names were processed by using specific obfuscation algorithms. Finally, the obfuscated files were transformed into a binary file using specific compilation back-ends. The experimental results show that the LLVM-based identifier obfuscation method applies to LLVM-supported languages, does not affect the normal function of the program. It has a time and space overhead within 1.5 times and a stable degree of identifier obfuscation with an average obfuscation ratio of 77.5%. Compared to existing random identifier obfuscation methods or overloaded obfuscation methods, it has low performance overhead, high concealment, and wide versatility.

Keywords: software protection; code obfuscation; identifier obfuscation; Low Level Virtual Machine (LLVM); obfuscation method

0　引言

软件保护涉及政府、企业和个人的利益，尽管从上世纪 80 年代[1]至今已经出现了众多保护技术，包括软件防篡改[2]、代码混淆[3]、软件多样性[4]等，但软件保护仍是当今时代极具挑战的问题之一。商业软件联盟（the Software Alliance，BSA）2018 年发布的全球软件研究报告表明，个人计算机领域中使

收稿日期: 2021-07-07;　**修回日期**: 2021-09-14;　**录用日期**: 2021-09-18。

基金项目：华为-北京大学校企合作项目（2020001763）。

作者简介: 田大江(1997—)，男，湖北黄冈人，本科生，CCF 会员，主要研究方向：代码混淆；李成扬(1996—)，男，山东临沂人，硕士研究生，主要研究方向：代码混淆；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。

用的盗版软件高达 37%，造成全球公司近 3590 亿美元损失的同时，致使用户数据处于易被窃取的危险状态。

源于软件保护的需求，代码混淆在 20 世纪 90 年代被提出用于保护 Java 代码，增大软件逆向的难度，进而保护代码的核心知识产权，避免泄露用户的隐私数据。因其在安全性方面低开销，高回报[5]的特性，在商业软件中得到广泛的使用[6]。虽然 Barak[7]指出并不存在混淆器可以完全隐藏程序的内部信息，因为除程序的输入和输出外，逆向人员还是可以获取更多有助于程序理解的信息。但文献[1]从实际需求出发认为相较于抽象的完美工具，是否存在实用的混淆工具用于具体的软件保护更加被看重。

Collberg[8]将混淆技术分为布局混淆，数据混淆，控制流混淆和预防性混淆，本文关注于布局混淆中的标识符混淆方法，在几乎无时空开销的前提下提高逆向分析的难度，以增强软件抵抗静态分析[9]的能力。为了突显性能和跨平台的通用性，本文主要研究基于底层虚拟机（Low Level Virtual Machine，LLVM）[10]的标识符混淆方法。

本文贡献点在于给出四种独立的标识符混淆方案和一个融合四种技术的创新混合方案。在保证正常语义的前提下，强化标识符混淆方法的保护能力。基于 LLVM 实现的标识符混淆方法不仅有着性能开销小，隐蔽性强的优点，而且适用于 LLVM 平台所支持的所有语言[11]，更具通用性和广泛性。

1 相关知识

本节主要介绍标识符混淆的相关知识，概要介绍代码混淆，标识符混淆方法的贡献与不足，以及针对 LLVM 的必要说明。

1.1 代码混淆

代码混淆技术[8, 12]是一种软件保护技术，在保证不改变程序原有功能的同时，通过对源代码或者汇编码进行修改，包括但不限于控制流和数据流方面的修改，降低源码的可理解性，阻止逆向工具的处理，以达到软件保护的目的。代码混淆技术从上世纪 90 年代开始出现，最开始关注于指令替换和花指令的添加[13]；之后 Collberg[8]对代码混淆技术进行分类，并给出混淆效果的评估指标，包括混淆程度[12]（potency）、混淆强度（resilience）、性能开销（cost）和隐蔽性（stealth）；为了加强安全性，随后出现控制流混淆[14]和数据混淆[15]的方案；尽管 Barak[7]从理论上证明并不存在足够安全的混淆器，但随着工业界对代码混淆需求的上涨，后续出现了众多实用的功能增强方案，包括混淆中结合加密方案[16]；添加垃圾指令，构建虚假控制流和虚假跳转表阻止静态反编译[17]；结合指针分析和数值分析的方法来研究 x86 可执行程序的静态分析算法[18]，防止二进制文件的恶意混淆技术的攻击；针对切片的混淆[19]；在 LLVM 层实现的 OLLVM (Obfuscator Low Level Virtual Machine) [20]给出了跨平台，跨语言混淆的可行性。同时出现了针对模型的混淆方案[21]和新的混淆参数的评定指标[22]。

广义的代码混淆目前研究方向，一方面是实用性混淆方法的提出，包括现有方法的强化和创新，以适用于新的场景；另一方面是理论性研究，结合密码学的研究思路，追求可证明的混淆方式。狭义的代码混淆可以等效理解为实用性混淆方法，本文研究的标识符混淆方法即归属于这一类。

1.2 标识符混淆

标识符混淆的定义[23]：设 p 是一个给定的程序，U_p 是 p 中出现的所有名称的集合，$N_p(\subset U_p)$ 是针对混淆的名称集合。p 的标识符混淆方法是将 p 中的每个标识符 $n \in N_p$ 替换为另一个标识符 $n'(=t(n))$，获得混淆程序 p'，其中 t 是一对一的映射 $(t: N_p \to N_{p'}(N_{p'} \subset U_{p'}))$，$t$ 可以视为不同的标识符混淆方法。

标识符混淆方法为了隐藏程序逻辑，降低代码的可读性，多数采用替换的思路。其中霍建雷等[24]提出 Java 标识符重命名混淆算法及其实现，用四种算法构造出 Java 混淆器(Java Identifier Renaming Obfuscator，JIRO)，但实现上存在通用性不强的缺点，仅限于 Java，且未考虑算法间的协同处理以达到最优效果。Ceccato 等[25]通过具体的逆向实验证明，标识符混淆增大了逆向的难度，使得有经验的逆向人员和经验欠缺的人员都面临同等棘手的问题。Al-Hakimi 等[26]提出混合混淆的策略，在字符串混淆阶段，先将标识符替换为垃圾代码，然后用 Unicode 字符替换系统关键字，最后结合字符串加密技术实现字符串混淆，混淆方案本身取得较好的效果，但在标识符混淆实现部分，更多的是替换方案，思路过于单一。Cimato 等[27]提出针对标识符混淆的对抗方法，核心思路为针对字节码优先进行处理，提出两种实现思路：第一种是利用现有的工具，将字节码文件中的标识符转变为单一的合法标识符（如字母或数字）后，使用反混淆工具将字节码转变为源码；第二种思路是实现 ADAM(Another Decompilation Assistant Methodology)，在处理字节码文件时，使用实体-标识符-信息（Entity_Number_Info）的形式进行标识符重命名，再将其转变为源码。但思路仅限于 Java 语言，较为单一且反混淆的质量取决于 info，即类相关信息的抽取，而混淆是可以对这部分信息进行处理的。

综上所述，现有的标识符混淆技术大多针对于某一具体平台或语言，不具通用性；围绕隐藏标识符信息，标识符算法主要分为替换和重载两种思路，但现有的文章中并未对两者结合后的理论或实际效果进行分析。针对现有问题，本文所提出的基于 LLVM 的标识符混淆方法，适用于 LLVM 平台所支持的所有编程语言和后端架构，目前前端有 Ada、C、C++、D、Delphi、Fortran、Haskell、Julia、Objective-C、Rust

and Swift，后端有 x86、x86-64、ARM、ARM64 等指令集。后文实验部分对结合替换和重载思路的算法效果进行分析。

1.3 LLVM 框架

LLVM[10]的命名最早源自于底层虚拟机（Low Level Virtual Machine）的首字母缩写，随着发展，原生的内涵也发生了转变，现在 LLVM 代表着模块化和可重用的编译器和工具链技术的集合。

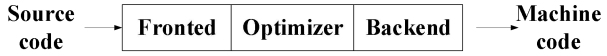

图 1 LLVM 三段设计

Fig.1 LLVM three-stage design

LLVM 是基于传统的三段式架构（前端、优化、后端）设计[28]的编译器。如图 1 所示，模块间更易于解耦，一方面利于拓展新的前端和后端，支持更多的语言和设备；另一方面，可以高度复用优化阶段的工作，在优化过程中对 LLVM IR(Intermediate Representation)[29]的工作可以无缝链接到不同的前端和后端。本文提出的方法针对 IR 文件进行修改和优化，因而理论上适用于 LLVM 支持的所有前端和后端。

LLVM 支持三种等效的 IR 格式：可读的汇编格式（.ll）；内存中编译 IR 格式；适用于即时编译（Just-In-Time compilation, JIT）快速加载的 bitcode 格式（.bc）。为了实现程序持久性的优化，本文主要针对于.ll 和.bc 文件进行操作，通过编写趟（Pass）实现具体的混淆功能。

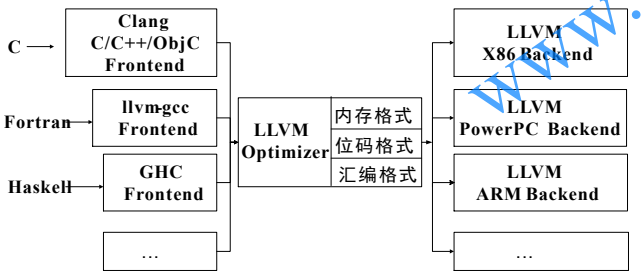

图 2 LLVM 编译器架构

Fig.2 LLVM compiler architecture

2 标识符混淆方法

本文针对文献[24]提出的标识符混淆算法进行重新设计。不同于以往的算法实现，在 LLVM 层进行开发，使其不再局限于 Java 等某一特定编程语言；对算法细节进行调整，包括随机标识符、非法标识符的构造，增强算法本身的效果；使用现有生活用语构造混淆后的字符，一方面隐藏真实标识符信息，另一方面增强混淆字符的隐蔽性。并对提出的四种算法进行混合，在保证程序正常执行的情况下，令四种算法效果达到最佳，在不会带来额外开销[30]的同时，让攻击者几乎不可能将标识符恢复到原始状态。在 2.3.5 节中给出了具体的分析。

2.1 标识符混淆域及混淆对象

在源代码中，当一个标识符被混淆时，与之相关的函数、调用等都要随之变化，否则会影响代码的正常执行。并且涉及到标准库和第三方库的方法，因为版权等方面原因，不能随意混淆。因此，有必要提出混淆域[31]的概念：在不影响程序正常执行的前提下，可以混淆的代码范围。

标识符混淆的对象理论上应是混淆域内的所有标识符，但源程序中的局部变量名、注释、自定义类型等信息在编译后已经被去除[32]。在 LLVM 层，源代码转化为 LLVM IR 后，局部变量以“%”字符开头，全局变量以“@”字符开头，并且为了编译器快速给出临时变量，避免符号表冲突，会配合无符号整数作为标识符的名称（如%12、@2），所以在正式混淆前，标识符已经产生变化。但程序的符号表会持有函数名称的引用，加之函数名在程序理解中的重要作用[33]，本文提出的标识符混淆方法主要对程序中的函数名进行处理。

本文标识符混淆方法针对的混淆域为当前文件或项目生成的 IR 文件，混淆对象为混淆域内自定义并初始化的函数名、全局变量名和自定义的结构体，不包括库函数和第三方 API。

2.2 标识符混淆方法设计

混淆方法设计分为三部分，分步骤完成混淆目标：

1）中间语言处理部分：通过 Clang 等前端将程序源码编译成对应的 LLVM IR 文件，再通过 LLVM-Link 或 WLLVM（Whole-program-LLVM）将所有的 IR 文件链接成一个包含源程序所有信息的 IR 文件。

2）标识符混淆部分：作为整个混淆算法的核心部分，主要用于对链接后生成的 IR 文件进行优化处理，可以分为以下 3 个子步骤完成标识符混淆处理：

① 全局数据处理。获取到代码的全局变量、函数名称等标识符信息。

② 标识符筛选。对①中获取到的标识符进行条件筛选，排除标准库和第三方库方法的标识符，保留满足混淆域要求的标识符。

③ 标识符混淆。对②中筛选后的标识符进行混淆处理，利用实现的 LLVM Pass 对标识符进行混淆。不同的 Pass 文件对应着不同的混淆算法。

3）生成部分：通过用户指定或者目标平台识别，将优化处理后的 IR 文件，由后端编译成适用于目标机器的可执行文件。

2.3 标识符混淆算法介绍

本文混淆方法给出 4 种标识符混淆算法，包括随机标识符算法、高频词替换算法、异常标识符算法、重载归纳算法，最后给出了将 4 种算法混合使用的混合标识符算法。其中随

机标识符算法和重载归纳算法基于已有的设计[24]，在 LLVM 平台进行实现；高频词算法的混淆字典、异常标识符混淆的异常字符选取属于原创性设计，同时为了充分利用现有混淆算法，在方法的结合上进行了新的尝试，即混合混淆算法。每种算法均为 LLVM Pass 文件，且对 IR 文件进行操作，生成优化后的 IR 文件以达到混淆目的。

算法 1：Identifier Rename
输入：LLVM IR file
输出：obfuscated IR file
necessary initial work for algorithm T
for global_variable in module; do
    global_variable ← T(global_variable)
    upgrade whole module
end
for struct_type in module; do
    struct_type ← T(struct_type)
    upgrade whole module
end
for funcname in module; do
    funcname ← T(funcname)
    upgrade whole module
end

算法 1 中的 T 即标识符重命名方法，包括随机标识符算法、高频词替换算法、异常标识符算法。重载归纳算法如算法 2 所示。

算法 2：Overload
输入：LLVM IR file
输出：obfuscated IR file
necessary initial work for algorithm T
functions ← all function name in module except main
vect<Function*, string> ← mapping function pointer to its real name
sort(vect)
for i = 1 to vect.size; do
    funcname = vect[i-1].second
    if vect[i].second != funcname; then
        Replacename(vect, i)
    end
    upgrade whole module
end

算法 2 中初始化工作包括混淆对象的选取，主体逻辑如下：标识符替换的算法通过编写 Module Pass 进行实现，在获取模块（module）后首先针对全局变量和自定义的结构体名称进行换名，module 提供对应的 API 进行处理：全局变量可以通过遍历 module.begin()和 module.end()进行获取；结构体可以通过 StructTypes.run(module, true)获取相关的信息；函数名，则通过遍历模块内的所有函数进行筛选，这里只选取用户自定义的函数，通过 LLVM 提供的 API 结合函数类型和链接类型进行有效的判断。

对于重载归纳算法，还要多做一步，因为在 C/C++中存在名称改写（name mangling）机制，因此在重载时，会先对函数名进行处理，获取去除不相关字符后真正的函数名再进行处理。最后，每更改一次，都需要进行全局更新，防止后续运行时找不到链接的函数。

2.3.1随机标识符算法

随机标识符算法：遍历混淆域内的所有标识符，随后通过自定义的随机函数产生随机且无意义的名字替代之前的标识符命名。其中随机函数通过随机选取大小写字母和下划线等字符，随机组合成 X 位的毫无意义[34]的字符串（如统一设置 11 位字符串 jK2iOy3yewc）。

这里的代码示例中源码选择 avl-tree（https://www.geeksforgeeks.org/avl-tree-set-1-insertion/）代码。通过查看混淆前后的 IR 文件中的符号表信息，统计相关函数名信息如表 1，后文中的算法效果示例均如此。

2.3.2高频词替换算法

高频词替换算法：将生活中常见的英文单词用于重命名标识符，在掩盖标识符本身含义的情况下，诱导攻击者以为程序未被混淆。使用 2019 年经济学人的词频统计作为源数据，在构造混淆字典时，为了更具真实性以便抵抗混淆检测算法，借鉴当前识别标识符混淆的研究，如文献[35]将代码中是否存在标识符长度小于 3 作为是否被混淆的指标之一，因此选择字符个数不少于 3 且不属于保留字的字符。在整理的 109932 项条目中，最后仅保留 1071 项作为混淆字典的条目。对于标识符数目庞大的代码，实现上通过补充下划线的方式进行扩容，使得混淆库足够丰富，以备替换。

高频词替换算法产生的标识符名称是具有现实意义的，只不过和源程序的函数意义是没有任何关联的，进而引申出追求更高层级的标识符混淆，不是去除标识符，也不是简单的使用无意义的标识符进行替换，而是使用有意义但无实际联系的字符进行替换，更能迷惑逆向分析人员。这里针对如何使用有意义但无实际联系的标识符进行概念说明：

有意义但无实际联系的标识符，指的是源于生活、相关领域认可且具有共识认知的词汇或者表示，但和函数本身的功能不存在指向关系。应当满足特性：首先，“有意义”体现在标识符本身是被认可的，且被该领域或者相关领域的主体认同；其次，“无实际联系”体现在标识符本身和函数本身的功能没有任何关联，可以没有任何联系，也可以相反，也可以相同，因为本身混合在一个系统中，真假难辨反而更容易迷惑逆向人员。

2.3.3异常标识符算法

异常标识符算法：针对混淆域内的所有标识符，使用形似保留字的标识符或下划线进行替换，以混淆视听。该算法分为两种思路，本节将分别叙述。

异常标识符算法 1：该思路旨在生成形似保留字的标识符以迷惑攻击者，过程分为三步，第一步生成一个带有希腊字母“保留字”的集合：先找出所有和英文字母类似的希腊字母，用这些希腊字母对保留字中的英文字母进行替换；第二步获取程序中出现的标识符；第三步使用自定义集合里的“保留字”替代程序中的标识符，以混淆视听。

异常标识符算法 2：通过由下划线组成的标识符（如：__）替代原有标识符，通过下划线个数作区分，随着标识符的递增，标识符区分度也逐渐降低。而程序可以依照个数的不同调用目标函数，实现对程序的保护目的。

表 1 混淆算法示例

Tab. 1 Identifier algorithm demo

原函数	随机标识符算法处理结果	高频词替换算法处理结果	异常标识符算法 1 处理结果	异常标识符算法 2 处理结果	重载归纳算法处理结果
height(Node*)	ieqitsHnYrg	took	export	_____	newNode(Node*)
max(int, int)	e42sWLoECD9	london	short	_____	newNode(int, int)
newNode(int)	m34nK7081V0	tech	operator	_____	newNode(int)
leftRotate(Node*)	q7U24yHN4W2	later	long	_____	leftRotate(Node*)
insert(Node*, int)	msauw40i8tj	rich	auto	_____	leftRotate(Node*, int)

注：异常标识符算法 2 一列数据为不同长度的下划线，用于标识不同的函数

2.3.4 重载归纳算法

重载归纳算法：利用面向对象语言的重载特性对标识符进行易名，使程序含有尽可能多的相同名字、不同参数类型的函数。

重载归纳算法不同于上述的替换思路，算法通过获取程序中出现的函数名，对其进行排序，根据参数类型或数量不同，采用同名替换的方式使得程序尽可能多的含有同名函数。程序可以采用重载的方法调用目标函数，并不会影响到程序的正常运行，而攻击者却难区分函数，这样就可以对调用的函数实现保护作用。

不同于前三种算法，重载算法并没有对混淆域内的所有函数名进行保护，会保留源程序的部分标识符信息。

前四种算法，对混淆域内的标识符提供了一定的保护能力，为了进一步增加混淆的隐蔽性和复杂性，可以将算法进行有效的组合，即混合标识符混淆算法。

2.3.5混合标识符混淆算法

前三种算法虽然达到隐藏标识符的目的，但是标识符依旧具有区分度；而重载归纳法虽然降低了区分度，但是依旧保留了标识符的部分信息。为了充分利用现有的标识符混淆算法，取得最佳的混淆效果，将上述四种算法混合使用，命名为混合标识符混淆算法。

如果仅是线性的将四种算法进行组合，如前三种算法进行混淆后，再进行重载归纳算法混淆可以得到可执行文件，但使用 IDA（Interactive DisAssembler professional）等逆向工具进行逆向处理时，会因为函数缺乏必要的参数信息，而对相同的函数添加后缀以示区分，所以为了对抗类似于 IDA 这样的机制，在将两类算法进行混合的时候，对前三种算法进行修改，考虑名称改写机制，仅对函数名称进行处理，而保留原本的参数列表信息，基于此，再使用重载归纳算法进行处理，以此，一方面因为添加重载函数名，强化第一类替换算法的效果；另一方面，替换算法进一步抹除原本的函数名信息，强化了重载算法的安全性。

算法 3：Mixed Identifier Rename

输入：LLVM IR file

输出：obfuscated IR file

initial work for algorithm T1, T2, T3, T4

Ti ← Randomly select from T1, T2, T3

for global_variable in module; do

    global_variable ← Ti(global_variable)

    upgrade whole module

end

for struct_type in module; do

    struct_type ← Ti(struct_type)

    upgrade whole module

end

for funcname in module; do

    funcname ← Ti(funcname)

    upgrade whole module

end

carry out T4

算法 3 中，首先初始化工作，筛选程序中的标识符，获取到 4 种算法可处理的混淆对象；然后，根据动态生成的随机数在替换类算法（算法 1，2，3）中进行随机选择，对全局变量、结构体和函数名称分别进行混淆处理，同时注意这里的算法实现上进行了部分调整，在针对函数名称替换时，会保留参数列表信息，以供下一步的重载混淆使用；最后，针对替换类算法处理后的程序，调用重载算法，基于替换后的函数名称做进一步的复用工作，使得攻击者无法有效恢复标识符信息。因为在混淆时仅涉及标识符的替换，没有增加额外的逻辑处理，所以理论上时空开销几乎无影响。

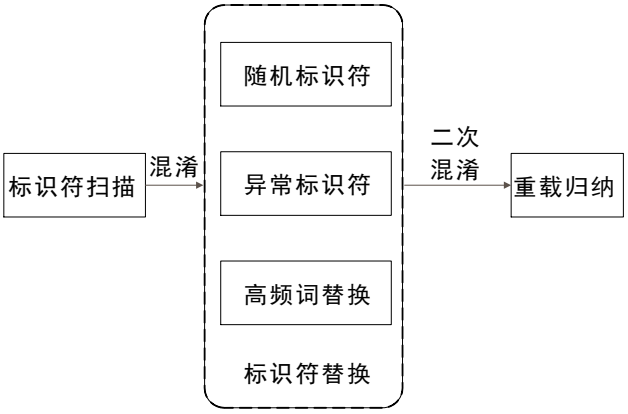

图 3 混合标识符混淆算法示意图

Fig.3 Mixed identifier algorithm process

3 标识符混淆方法实验

3.1 实验环境和数据

实验在 Inter(R) Core(TM) i7-9750H、16G 内存的 Ubuntu18.04 64 位机上进行实现，基于 LLVM10 release 版进行开发，使用 IDA Free 7.6 进行效果的对比。实验分为性能分析和混淆效果分析：性能分析关注混淆后程序在时空上增加的开销[36]；混淆效果分析关注混淆的标识符数量占符号表标识符总量的比率。为了保证实验数据的准确性，性能分析中使用脚本进行数据的获取；效果分析使用 IDA Free 7.6 对混淆前后的可执行文件进行逆向处理，统计更改名称的函数所占的比率。

其中，性能分析以 C++实现的算法库（https://github.com/xtaci/algorithms）和 Rust 算法库（https://github.com/TheAlgorithms/Rust）为测试数据，一方面说明混淆方法对算法的时空开销，另一方面说明本文提出方法的通用性——LLVM 支持众多的前端，本次实验选取目前为止始终拥有良好维护的 C++和 Rust 以展示效果；混淆效果分析中，选取项目文件，包括 std 的 map 实现，数独游戏 sudoku（https://github.com/mayerui/sudoku），字符转换工具 flowchart（https://github.com/Gusabary/FlowChar）和 Google 开源的 leveldb（https://github.com/google/leveldb），以展示在多文件中的使用效果。

3.2 性能分析

选取 C 语言和 Rust 语言上的算法实现，以时空增长作为指标，评估性能分析。本节先理论分析，后面辅以实验说明。

依据 Collberg[8]提出的混淆指标对本文方法进行理论评估，在性能代价（cost）上表现为无代价（free）。性能开销分为 dear、costly、cheap 和 free 四种等级，由于本文技术仅针对于标识符混淆层面，所以在性能开销上是 free 级别。

对 60 个 C 文件和 24 个 Rust 文件进行混淆处理。为了减小机器运行时造成的时间误差，每个测试用例执行 100 次求平均值作为统计的运行时间。同时，为了更好的展示不同方法间的差异，对算法库文件，单独使用混淆算法后，对每个文件统计其时空增长倍率，然后在下图中以各自均值进行表示，如针对于 C++项目的 60 个文件，使用 abuse 算法后，时间增长倍率的均值在 1.07。

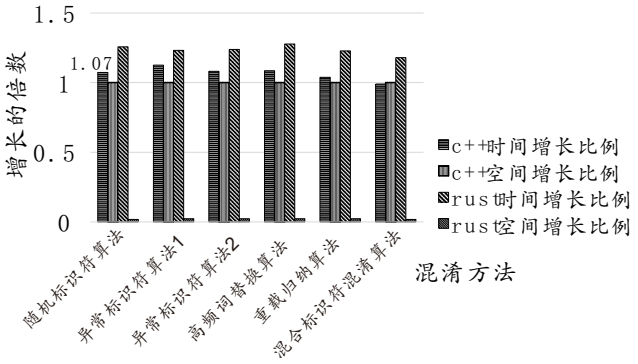

图 4 性能分析

Fig.4 Performance Analysis

横坐标是 5 类混淆算法（异常标识符算法存在两种实现，共 6 种算法），纵坐标表示混淆后相较于混淆前的增长倍数。对应可以得到如下的结论：

1）混淆算法对 C++处理后，在时空上几乎无影响，特别地，考虑到机器运行的时间误差，混淆增加的时间倍率可以进一步减小。

2）混淆算法对 Rust 处理后，增加 20%时间开销的同时，极大的减小了空间开销，空间上平均变为原本的 2%。

3）虽然混淆算法针对 C++和 Rust 在时空开销上影响不同，但是可以证明本文提出算法的通用性。

3.3 混淆效果分析

为了更客观的评估混淆效果，选取项目文件进行混淆。选择不同标识符量级的项目，涵盖几十到几万量级，同时兼顾项目的不同类型，最终以 std 的 map 实现，数独游戏，字符转换工具 flowchart 和 Google 开源的 leveldb 作为实验数据。通过 IDA 对混淆后的可执行文件逆向处理，得到混淆后的标识符所占的比例。下文先对标识符混淆的混淆强度进行理论分析，再辅以实验说明。

依据 Collberg[8]提出的混淆指标对本文方法进行理论评估，在混淆强度（potency）上表现为强（strong）。混淆强度指标主要是为了度量混淆后的程序对自动化反混淆器的抵抗能力，包括两个方面的表现，一方面是有针对性的编写一个自动化反混淆器付出的代价，另一方面是使用自动化反混淆器进行反混淆所付出的代价[12]。

第一个方面的代价分为多项式时间复杂度和指数时间复杂度，本文提供的标识符技术复杂度是 $O(n^2)$，属于多项式时间复杂度，而对应的第二个方面所指代的反混淆付出的代价分为 full、strong、weak 和 trivial 四个级别，分别对应 interprocess、interprocedural、global 和 local 四个层面[8]。本

文会将所有的中间文件链接在一起统一对其混淆，可以确定该技术是在 interprocedural 层面进行的，所以在混淆强度指标上表现为强（strong）。

从混淆算法角度出发，本文提出的 6 种具体的混淆算法，又可以归于三大类算法：替换算法，包括随机标识符算法、高频词替换算法、异常标识符算法（所能影响的标识符数量是相同的，只是标识符替换的策略不同）；重载算法；替换和重载结合的混合算法。

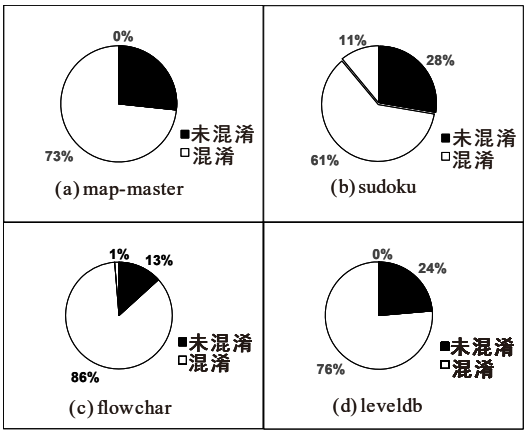

图 5 混淆算法混淆效果图

Fig.5 identifier rename algorithm effect

图 5 中白色空白区表示已混淆标识符的数量，黑色区域表示未混淆的数量。部分标识符未混淆，因为算法设计时仅针对用户自定义的函数，对于库函数和第三方 API 并不进行处理。

针对 4 个项目分别使用三类算法（第一类替换算法）运行时通过随机数随机指定）得到如下结论：

1） 第三类算法影响的标识符数量和第一类算法是相同的，第二类算法影响的数量是最少的。如图 5 四个子图中均存在较小的白色区域，即重载所占的比重；

2） 第三类算法保留了前两类算法的优点，对其各自的缺点进行了弥补：第一类算法可以混淆更多的标识符，但是标识符仍然具有唯一性，第二类算法使用重载，标识符不再具有唯一性，但是混淆的数量较少，且会保留部分源程序的语义信息。而第三类在取得极大混淆数量的同时，彻底去除原本的语义信息，且借助于重载使得标识符不再具有唯一性。

3） 针对不同标识符量级的项目，混淆算法的平均比率在 77.5%，如图 6 所示；

4） 在混淆的标识符中，重载归纳算法的比例较小，和程序本身的特性相关，极端情况下无效果。

5） 混淆效果和项目的标识符数量无关联，和程序本身自定义的函数和函数参数的设计相关。

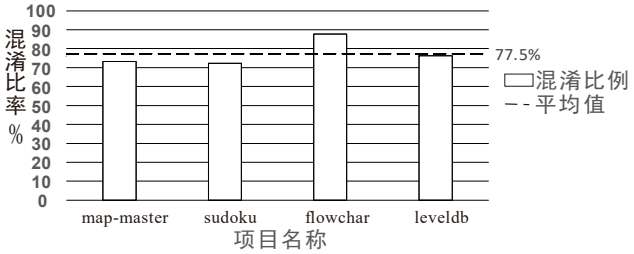

图 6 混合混淆算法混淆比例图

Fig.6 Mixed identifier algorithm ratio

同时，因为 strip 用于去除符号表，和本文所讨论的标识符替换存在关联。对两者进行分析，得到如下结论：

1） 在 Rust 程序上进行测试，除同 strip 一样可以极大减小符号表体积外，不同于 strip 直接去除符号表，本文方法可保留迷惑性的信息以干扰逆向人员的分析。

2） 本文提出的方法针对自动化逆向工具分析时，目前效果不大于 strip 的效果，甚至自动化工具会直接去除符号表信息，以规避标识符混淆的效果。

3） 目前 LLVM 支持的前端中包括静态编译语言和动态解释语言，本文方法在生成静态语言可执行文件时，可能受制于 strip，但如果动态解释语言得到进一步的优化和维护后，后人可以在本文方法上进行进一步的研究和推进。

4 结语

基于软件保护的应用场景和编程语言多样的需要，提出基于 LLVM 的标识符混淆方法。该混淆方法融合四种算法的优势，提升代码的混淆强度；采用 84 个测试用例证明算法性能的优越性，且运行在不同类型的项目中，均有较为稳定的混淆效果。算法有效地隐藏了标识符信息，提高代码的安全性；同时由于 LLVM 的跨平台性，算法适用于 LLVM 支持的所有语言和架构。

但是对于动态调试，本文所提供的技术无法提供足够的保护，这是接下来的工作之一。同时，本文标识符混淆方法多数停留于标识符替换层面，反编译人员可以经过分类等方法，大致确认函数的作用。接下来的工作，一方面是增强对动态调试的抵抗力，另一方面是提高抵抗机器学习攻击[37]的能力，如何消除 strip 的影响，提供功能更为丰富的混淆方法也是今后的重点工作。

参考文献

[1] VAN OORSCHOT P C. Revisiting software protection[J]. International Conference on Information Security, 2003, 2851: 1-13.

[2] 王朝坤, 付军宁, 王建民, et al.软件防篡改技术综述[J].计算机研究与发展,2011, 48 (06): 923-933.(WANG C K, FU J N, WANG J M, et al. An overview of software anti-tampering technologies [J]. Journal of Computer Research and Development, 2011, 48 (06): 923-933.)

[3] Rajba P , Mazurczyk W . Data hiding using code obfuscation[C]//Processing of ARES 2021: The 16th International Conference on Availability, Reliability and Security, New York: ACM 2021.

[4] LIU Z, ZHANG Z, LIU H, et al. Web service active defense mechanism based on automated software diversity[C]// Proceeding of International Conference on Computer Engineering and Application (ICCEA), Piscataway: IEEE, 2020: 241-249.
[5] BIN SHAMLAN M H, ALAIDAROOS A S, BIN MERDHAH M H, et al. Experimental evaluation of the obfuscation techniques against reverse engineering[M]//Advances on Smart and Soft Computing. Springer, Singapore, 2021: 383-390.
[6] 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.
[7] BARAK B, GOLDREICH O, IMPAGLIAZZO R, et al. On the (im) possibility of obfuscating programs[J]. Journal of the ACM (JACM), 2012, 59(2): 1-48.
[8] COLLBERG C, THOMBORSON C, LOW D. A taxonomy of obfuscating transformations[R]. Department of Computer Science, The University of Auckland, New Zealand, 1997.
[9] CANFORA G, DI PENTA M, CERULO L. Achievements and challenges in software reverse engineering[J]. Communications of the ACM, 2011, 54(4): 142-151.
[10] Chris Lattner, The LLVM compiler infrastructure[EB/OL]. 2021.07.01.https://llvm.org/.
[11] Chris Lattner, What source languages are supported?[EB/OL]. 2021.07.01 https://llvm.org/docs/FAQ.html#what-source-languages-are-supported.
[12] 杨宇波. 代码混淆模型研究[D]. 北京：北京邮电大学,2015.19-46.(YANG Y B. A study of code obfuscation models[D]. Beijing: Beijing University of Posts and Telecommunications, 2015.19-46)
[13] COHEN F B.Operating system protection through program evolution[J].Computers and Security,1993, 12 (6): 565-584.
[14] COLLBERG C, THOMBORSON C, LOW D. Manufacturing cheap, resilient, and stealthy opaque constructs[C]// Proceeding of Conference Record of the Annual ACM Symposium on Principles of Programming Languages, New York: ACM, 1998: 184-196.
[15] COLLBERG C, THOMBORSON C, LOW D. Breaking abstractions and unstructuring data structures[C]//Proceedings of the 1998 International Conference on Computer Languages (Cat. No. 98CB36225). Piscataway: IEEE, 1998: 28-38.
[16] CHOW S, EISEN P, JOHNSON H, et al. White-box cryptography and an AES implementation[C]//Processing of International Workshop on Selected Areas in Cryptography. Springer, Berlin: Heidelberg, 2002: 250-270.
[17] LINN C, DEBRAY S. Obfuscation of executable code to improve resistance to static disassembly[C]// Proceedings of the ACM Conference on Computer and Communications Security, Washington: CCS, 2003: 290-299.
[18] ROUNDY K A, MILLER B P. Binary-code obfuscations in prevalent packer tools[J]. ACM Computing Surveys (CSUR), 2013, 46(1): 1-32.
[19] MAJUMDAR A, DRAPE S, THOMBORSON C. Slicing obfuscations: design, correctness, and evaluation[C]//Proceedings of the 2007 ACM workshop on Digital Rights Management. New York: ACM, 2007: 70-81.
[20] JUNOD P, RINALDINI J, WEHRLI J, et al. Obfuscator-LLVM-software protection for the masses[C]//Processing of 2015 IEEE/ACM 1st International Workshop on Software Protection. Piscataway: IEEE, 2015: 3-9.
[21] GARG S, GENTRY C, HALEVI S, et al. Candidate indistinguishability obfuscation and functional encryption for all circuits (Extended Abstract)[J].2013 IEEE 54th Annual Symposium on Foundations of Computer Science (Focs),2013: 40-49.
[22] ANCKAERT B, MADOU M, DE SUTTER B, et al. Program obfuscation: a quantitative approach[C]//Proceedings of the 2007 ACM workshop on Quality of protection. New York: ACM, 2007: 15-20.
[23] ISOBE Y, TAMADA H. Design and evaluation of the de-obfuscation method against the identifier renaming methods[J]. International Journal of Networked and Distributed Computing,2018, 6 (4): 232-238.
[24] 霍建雷. 用于 Java 软件保护的代码混淆技术研究与实现[D]. 西安：西北大学,2009.24-48.(HUO J L. Research and implementation of code obfuscation techniques for Java software protection[D]. Xi'an : Northwest University, 2009.24-48)
[25] CECCATO M , MD PENTA, NAGRA J, et al. The effectiveness of source code obfuscation: an experimental assessment[C]// Proceeding of IEEE 17th International Conference on Program Comprehension. Piscataway: IEEE, 2009: 178-187
[26] AL-HAKIMI A , SULTAN A , GHANI A, et al. Hybrid obfuscation technique to protect source code from prohibited software reverse engineering[J]. IEEE Access, 2020, 8:187326-187342.
[27] CIMATO S, DE SANTIS A, FERRARO PETRILLO U. Overcoming the obfuscation of Java programs by identifier renaming[J].Journal of Systems and Software,2005, 78 (1): 60-72.
[28] Racordon D. From ASTs to machine code with LLVM[C]// Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming. New York: ACM, 2021: 68-76.
[29] Chris Lattner, LLVM language reference Manual[EB/OL].2021.07.01. https://llvm.org/docs/LangRef.html
[30] 潘雁. 基于虚拟机框架的代码混淆技术研究[D]. 郑州：战略支援部队信息工程大学,2018.11-24.(PAN Y. Research on code obfuscation techniques based on virtual machine framework[D]. Zhengzhou : Information Engineering University, 2018.11-24)
[31] ISOBE Y, TAMADA H. Design and evaluation of the de-obfuscation method against the identifier renaming methods[J].The International journal of networked and distributed computing (Online),2018, 6 (4): 232-238.
[32] JAFFE A, LACOMIS J, SCHWARTZ E J, et al. Meaningful variable names for decompiled code: A machine translation approach[C]//Proceedings of the 26th Conference on Program Comprehension. Piscataway: IEEE, 2018: 20-30.
[33] KASHIWABARA Y, ONIZUKA Y, ISHIO T, et al. Recommending verbs for rename method using association rule mining[C]//Processing of 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). Piscataway: IEEE, 2014: 323-327.
[34] LI G, LIU H, NYAMAWE A S.A survey on renamings of software entities[J].ACM Computing Surveys (CSUR),2020, 53 (2): 1-38.
[35] KUMAR R, VAISHAKH A R E. Detection of obfuscation in java malware[J]. Procedia Computer Science,2016, 78: 521-529.
[36] 张越. 基于代码混淆的软件保护方案研究与设计[D]. 成都：电子科技大学,2019.61-71.(ZHANG Y. Research and design of a software protection scheme based on code obfuscation[D]. Chengdu : University of Electronic Science and Technology of China, 2019.61-71)
[37] LACOMIS J, YIN P, SCHWARTZ E, et al. DIRE: a neural approach to decompiled identifier naming[C]// Proceeding of 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), Piscataway: IEEE, 2019: 628-639.

This work is partially supported by the Huawei-Peking University School-enterprise Cooperation Project (2020001763).

TIAN Dajiang, born in 1997, B. S. His research interests include code obfuscation.

LI Chengyang, born in 1996, M. S. candidate. His research interests include code obfuscation.

HUANG Tianbo, born in 1997, M. S. candidate. 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.