软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(4):1650−1665 [doi: 10.13328/j.cnki.jos.006719]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于分片融合的代码隐式混淆技术*

于 璞, 舒 辉, 熊小兵, 康 绯

(数学工程与先进计算国家重点实验室, 河南 郑州 450001)

通信作者: 舒辉, E-mail: shuhui123@126.com

摘 要: 目前, 在代码保护技术研究领域, 传统的混淆方法具有明显的混淆特征, 分析人员可根据特征对其进行定制化的去混淆处理. 为此, 提出了一种基于分片融合的代码保护技术, 通过在源代码层面将目标代码按照语法规则进行代码分片, 依据执行顺序与语法规则, 将分片插入另一程序的不同位置, 在修复函数调用过程与数据关系后, 形成可正常运行两个代码功能的融合后代码. 在实验部分, 对混淆后的代码, 从运行效率、代码复杂度影响、代码相似性这 3 个维度, 与其他混淆技术进行对比. 从测试结果可以看出: 基于分片融合的代码隐式混淆技术能够有效地模糊代码语义, 改变控制流特征, 且没有明显的混淆特征. 因此, 融合技术在对抗多种相似性对比算法的能力上有明显优势.

关键词: 代码保护; 混淆; 代码分片; 融合; 混淆特征

中图法分类号: TP311

中文引用格式: 于璞, 舒辉, 熊小兵, 康绯. 基于分片融合的代码隐式混淆技术. 软件学报, 2023, 34(4): 1650−1665. http://www.jos.org.cn/1000-9825/6719.htm

英文引用格式: Yu P, Shu H, Xiong XB, Kang F. Implicit Code Obfuscation Technique Based on Code Slice Fusion. Ruan Jian Xue Bao/Journal of Software, 2023, 34(4): 1650−1665 (in Chinese). http://www.jos.org.cn/1000-9825/6719.htm

Implicit Code Obfuscation Technique Based on Code Slice Fusion

YU Pu, SHU Hui, XIONG Xiao-Bing, KANG Fei

(State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China)

Abstract: At present, in the field of code protection technology research, traditional obfuscation methods have obvious obfuscation characteristics, and analysts can perform customized de-obfuscation processing based on these characteristics. For this reason, this study proposes a code protection technology based on code slice fusion. This technology slices the target code into code fragments according to grammatical rules at the source code level, and inserts the fragments into different positions of another program according to the execution order and grammatical rules. After repairing the function call process and the data relationship, the fusion code that can run the two code functions normally is formed. A comparative experiment for the fusion code is carried out from three perspectives , namely, resource overhead, code complexity impact, and code similarity. The test results demonstrate that the implicit code obfuscation technique based on code slice fusion can effectively obfuscate code semantics, change control flow characteristics, and has no obvious obfuscation characteristics. Therefore, fusion technology has obvious advantages in the ability to fight against multiple similarity comparison algorithms.

Key words: code protection; obfuscation; code slice; fuse; obfuscation characteristics

在代码保护技术领域的研究中, 代码混淆技术是研究的热点. 1997 年, Collberg[1]提出了程序混淆的概念, 并将混淆技术分为预防性混淆、数据混淆、布局混淆与控制流混淆. 近年来, 混淆技术虽然得到较大发展,但仍然存在混淆算法固定、有明显混淆特征的问题. 例如: 控制流扁平化技术[2]会将循环结构转化为大量的条件

* 基金项目: 国家重点研发计划(2016YFB08011601)
收稿时间: 2021-09-02; 修改时间: 2021-12-23, 2022-03-22; 采用时间: 2022-05-17

分支, 添加冗余代码会使程序出现大量不可达基本块等. 而若程序存在明显的混淆特征, 会显著提高程序被破解的风险. 因此, 本文提出了一种基于分片融合的代码隐式混淆技术.

该技术能够将两个不同的代码进行深度融合, 生成同时具有两个代码功能的新代码, 以此提高代码的抗分析能力. 其中, 被保护的代码称为目标代码, 另一个为母体代码. 分片融合技术就是在源代码层面, 将目标代码函数分片插入到母体代码的函数中. 设置区分变量屏蔽函数中的库函数调用、指针参数等可能改变环境状态的语句, 与目标函数与母体函数的参数共同组成新函数的参数, 完成函数与函数的融合. 具体融合效果如图 1 所示, 母体函数 *func*1 与目标函数 *func*2 融合成为 *func*3. 最终, 在修复函数调用关系后, 完成母体代码与目标代码的融合.

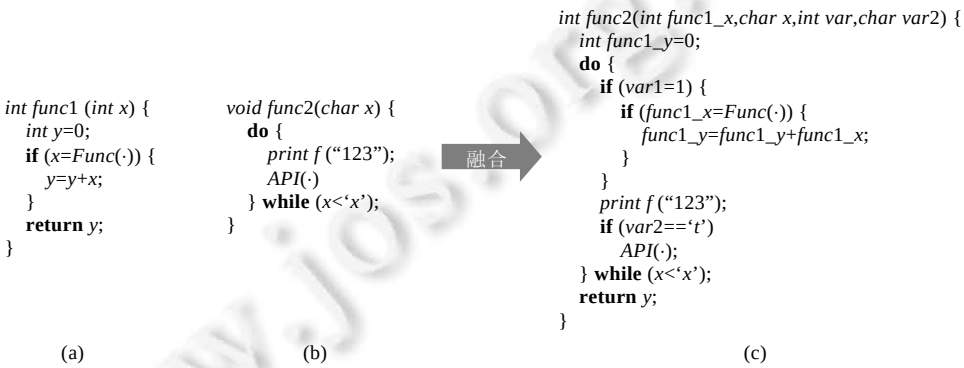

图 1 融合示例

通过融合技术生成的新代码, 母体代码与目标代码交替执行, 难以分离出完整的原目标代码, 而且混淆后的代码不具有明显的混淆特征, 更能满足目前对隐式混淆的需要. 由于融合后代码运行中会执行母体代码, 因此为保证母体代码不会对目标代码功能产生影响, 适用本文研究内容的母体代码需满足以下条件.

(1) 可与目标代码在相同环境下正常编译、运行;

(2) 代码内容以复杂算法为主, 不调用系统资源写入类的函数, 不影响目标代码功能;

(3) 不具有恶意性或敏感操作, 以免被防病毒软件误报;

(4) 提供足够多语义丰富的函数, 确保目标代码中的主要函数(或者绝大多数函数)均被融合.

本文第 1 节介绍有关代码混淆的相关方法与研究现状. 第 2 节介绍基于分片融合的混淆技术所涉及的基本概念与相关原理. 第 3 节介绍分片融合的整体框架与具体实现思路. 第 4 节通过实验验证分片融合方法的有效性. 最后总结全文.

1 代码混淆相关工作

依据混淆对象的不同, 可将代码混淆技术分为两种: (1) 源代码层面混淆; (2) 中间语言或二进制层面混淆.源代码层面的混淆技术主要包括加密、编码、语法树混淆这 3 种类型. 加密可分为数据加密、代码加密、标识符加密, 代码加密是使用强加密算法将代码文本加密, 在运行时, 通过对应的解密算法还原出源代码再进行解释执行[3], 例如文献[4]曾提出使用 AES 对 PHP 代码进行加密, 但这种方式只有通过 eval, Invoke 等特定函数或方法, 才可将解密后的代码文本作为代码执行. 标识符加密主要以随机生成的无意义字符串替换原有的标识符, 提高代码分析者对代码语义的理解难度[5]. 编码是将代码中的敏感字符替换为其他等价字符, 以此规避部分基于敏感字符查找的检测机制. 以上这些方法主要是为了降低源代码的可读性以及可理解性, 且只适用于解释型代码, 对于 Java、C#等托管类型代码的保护能力较弱. 语法树混淆主要是利用部分语法结构在控制流层面的等价关系, 通过等价替换的方式将部分结构进行变形[6]. 这种方式虽然能够有效规避部分基于语法树的特征检测技术, 但不会对编译型语言产生有效的保护.

在中间语言与二进制层面的混淆研究中, 控制流混淆是研究的热点. 控制流混淆通过改变或隐藏程序执行过程, 使分析人员难以分析程序的运行逻辑. 目前, 针对控制流的混淆技术主要分为两种: (1) 函数内的控制流混淆; (2) 函数间的调用关系混淆. 常见的函数内控制流混淆技术有循环变换、添加冗余代码[7]、使用不透明谓词[8]等, 其中, 循环变换主要是通过进行等价结构变换, 在保持原有功能不变的前提下修改循环的执行过程. 例如, 文献[2]针对循环结构提出的控制流扁平化, 通过将循环和条件转移语句展开, 通过条件分支进行连接, 并在循环中通过迭代器选择每次循环中运行的分支, 以此保证代码运行的相对顺序不变. 在此基础上, 研究人员[9]针对 Android 程序提出了利用分支合并过程中所产生的常量信息, 对基本块跳转关系进行加密的方法, 通过将不透明谓词与控制流扁平化相结合, 提高软件保护效果. 其中: 不透明谓词是指运算结果始终为“true”或“false”的条件谓词, 通常使用数论定理作为谓词, 在程序中引入虚假条件分支, 增加程序分析的成本. 文献[10]在此基础上提出了“动态不透明谓词”, 通过动态变化的值干扰基于动态测试和形式语义的检测技术. O-LLVM[11]是一款基于 LLVM 的混淆框架, 在中间语言层面对代码进行混淆变形. 框架中集成了指令替换、虚假控制流插入、控制流扁平化、基本块拆分等多种混淆策略. 该框架被广泛应用于多种场景, 例如: Lim 等人[12]使用 Android NDK[9], 利用 C 和 C++编写 Android 程序的主要部分, 在中间语言层面, 使用 O-LLVM 对由 C++编写的程序核心部分进行混淆, 以此提高 Android 程序的抗分析能力.

以上提到的控制流混淆方法均是对函数内部的运行逻辑进行混淆, 由于混淆算法固定, 混淆后, 程序中每个函数均具有明显的混淆特征, 且诸如冗余代码与不透明谓词等技术在控制流中加入的虚假分支, 难以有效对抗符号执行、LOOP[13]等动态分析技术与工具.

而对于函数间的调用关系混淆技术, 主要包括函数拆分、函数内联[14]、外联、函数合并等. 函数内联是将函数调用过程替换为函数内容, 函数外联是将指令序列进行部分修改形成函数体[15], 文献[16]在此基础上提出在内联时创建出多个不同的函数版本的方法, 进一步隐藏函数调用信息. 函数合并也称 Function Merging[17], 与本文提出的代码融合有相似之处, 在实现效果上, 均是将不同的函数合并为一个函数. 但不同点在于: Function Merging 是将多个函数的函数体封装到 switch-case 分支中, 通过参数的值, 判断执行哪个函数的功能. Balachandran[14]提出将来自不同函数的代码存储到一个函数中实现多函数合并, 通过跳转指令执行不同的代码片段以实现对应功能. Function Merging 虽然在一定程度上能够提高对函数调用过程的分析难度, 但是在分支下完整保留了合并前的函数体, 易被提取出完整的代码. 而代码融合是一种细粒度的合并方式, 融合后的函数中, 母体函数代码与目标函数代码之间没有显式的分支, 且在运行过程中交替运行, 难以分离出完整原函数代码, 因此具有更好的保护效果.

2 基础知识

本节将对代码融合过程进行形式化描述, 对相关名词进行解释说明, 对融合过程中亟待解决的问题进行系统阐述. 其中, 对代码融合中的相关定义将从控制流图的角度展开, 定义程序 P 的控制流图为 $GP=\{N,E,\varepsilon\}$, 其中, N 为控制流图中基本块的集合; E 为边的集合; ε表示基本块与基本块之间跳转条件的集合, 该跳转条件可分为条件跳转和无条件跳转.

2.1 代码分片和代码融合

定义 2.1. 若程序 P 有控制流图 $G_P=\{N,E,\varepsilon\}$, 若去除其中连接基本块 N_i 与 N_j 的边 E_{ij}, 使图 $G_P=\{N,E,\varepsilon\}$从起始节点无法到达终止节点, 则称 E_{ij} 是图 G_P 的必经路径, 而节点 N_i 与 N_j 为图 G_P 的必经节点.

定义 2.2. 若图 G_1 是图 G_2 的子图, 且图 G_1 的起始节点与终止节点均为图 G_2 的必经节点, 且节点间的跳转条件相同, 则称 G_1 是 G_2 的分片.

代码融合在控制流图层面表现为: 将两个控制流图合并为一个控制流图, 融合后的代码包含母体代码与目标代码中的任意代码与数据, 并添加新的跳转条件γ, 用于母体程序代码与目标程序代码之间的相互调用以及数据传递. 融合后的代码, 目标代码与母体代码均保持各自原有的代码执行顺序, 且数据的访问没有受到影响, 因此能够保证融合前后代码功能的一致性.

加壳后的程序可以看作是壳程序与目标程序融合后的程序, 加壳后的目标程序通过壳程序启动执行, 此时的目标程序作为一个整体存在, 即分片数量为 1, 如图 2 所示. 我们将这种在某一时间或位置将目标程序整体释放并执行的融合方式称为整体融合. 这种融合方式下, 母体程序与目标程序可以在很大程度上保留了自身的控制流结构与执行过程, 并且对彼此的执行过程影响较小, 但容易被剥离出完整的原目标程序.

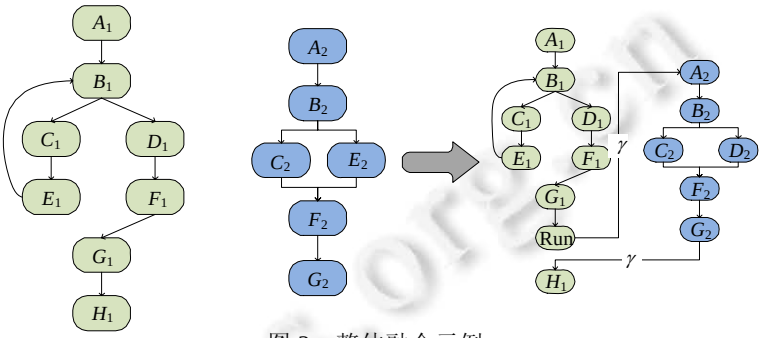

图 2 整体融合示例

本文在融合的基础上进一步提出分片融合, 分片融合流程如图 3 所示.

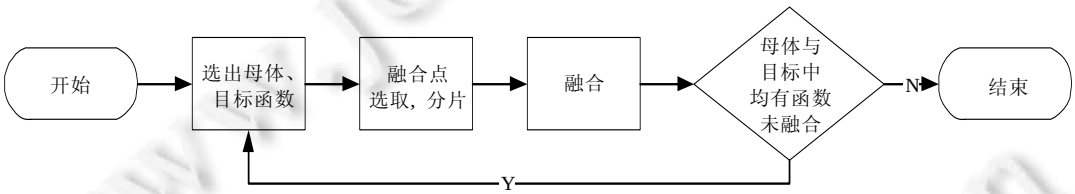

图 3 分片融合流程

首先, 在母体代码与目标代码中各选出一个还没有融合的函数, 将母体函数中的必经点作为可融合点, 目标函数则依据必经点切分为若干分片, 且分片数量小于母体函数的可融合点数量. 将目标函数分片按顺序插入到母体函数的可融合点, 形成一个新的函数, 新函数包含母体函数与目标函数的全部参数. 在修复函数调用方式后, 继续查找没有融合的函数, 直到有一方代码中的所有函数均被融合. 融合后, 母体代码与目标代码均参与到程序的执行过程中, 并且保留了各自完整的功能, 但难以区分出母体与目标代码.

同时, 分片融合对分支、Do-Loop 循环结构设置不同的融合方法. 分支结构的融合即是将两个分支中的所有条件分支进行两两组合, 以图 4 为例, 这样生成的分支语句中, 各个基本块的跳转关系没有改变, 分支内容所依赖的条件区间也没有改变, 因此既可以保证融合前后功能的正确性, 又强化了母体与目标代码之间的耦合性.

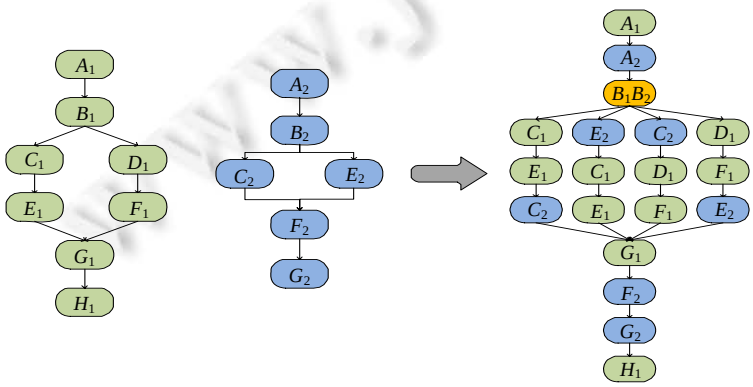

图 4 分片融合示例

循环结构主要包含两个主要类型: 一种是先判断后执行, 称为 Loop-Do 循环; 另一种是先执行后判断,称为 Do-Loop 循环. 对于 Do-Loop 的循环, 无论循环条件是否满足, 循环体都会被至少执行一次. 显然, 在这种循环的循环体内必然存在必经节点. 因此, 在母体函数中的 Do-Loop 循环体内可以存在可融合点. 但对于目标函数, 在 Do-Loop 循环体内进行代码切片会破坏代码结构, 造成融合后语法和功能不正确的问题. 因此,若母体函数中有 Do-Loop 循环, 则在进行可融合点选取时, 对在 Do-Loop 循环体内的可融合点进行特殊标识.若目标函数中有 Do-Loop 循环, 则将整个循环作为一个分片.

而对于 Loop-Do 循环, 循环体执行前会先进行条件判断, 整体的控制流形态与分支结构相似, 循环体内部不会存在必经节点. 因此, 对于 Loop-Do 循环采用直接插入分片的方法进行融合.

2.2 融合后程序正确性

本节将从控制流与数据流两个维度, 对融合前后函数功能的一致性进行讨论. 在控制流图角度, 通过对融合前后函数控制流图基本块的可达性影响进行讨论分析, 证明融合前后函数功能的一致性. 在数据流角度, 对融合后数据流的完整性进行讨论.

设母体函数 $F_S(x)$的控制流图 $G_S=\langle N_S,E_S\rangle$有必经点集合 $n=\{n_1,n_2,\ldots,n_i\}$, 其中, 任意必经点都有 $n_i\in N_S$. 设目标函数 $F_P(y)$的控制流图 $G_P=\langle N_P,E_P\rangle$有分片集合 $g=\{g_1,g_2,\ldots,g_j\}$, $j\leqslant i$, 其中, 任意分片都有 $g_j\subseteq G_P$, 即 g_j 是 G_P 的子图, 其中, 分片的起始基本块和结束基本块均为目标函数控制流图中的必经点.

设 $F_S(x)$与 $F_P(y)$进行分片融合后的函数为 $F(x,y)$, 若 $F_S(x)$控制流图的某一必经点 n_j 为融合点, 对应的 $F_P(y)$融合分片为 $g_j=\langle N_j,E_j\rangle$, 则此位置融合后产生新的代码块 $g'=\langle(N_j,n_j),E_j\rangle$, 其中, n_j 为新代码块的结束基本块. 因此, 分片 g_j 中任意基本块均可到达 n_j. 且因为 n_j 是必经点, 所以在 n_j 前的任意代码块对分片 g'为必达的. 同理, 在分片 g'前的分片对 g'也是必达的.

以此类推, 目标函数 $F_P(y)$的分片集合 g 在融合后的函数 $F(x,y)$中, 任意基本块之间的可达性没有发生变化, $F_S(x)$同理. 因此可以认为: 在融合后, 函数 $F_S(x)$与函数 $F_P(y)$的功能均不受影响.

分片融合作用在源代码层面, 以函数中的代码结构作为操作目标, 在保证母体代码与目标代码之间变量不存在冲突的情况下进行融合. 为防止函数中存在可能影响系统环境变化的因素干扰运行过程, 在对代码进行分析的过程中引入了区分变量, 对母体与目标代码的数据空间进行了隔离, 以保证融合后不会影响母体代码与目标代码各自的数据关系. 区分变量的引入策略和方法将在第 3.1 节中进行阐述.

综上所述, 融合后代码各自的执行过程与数据流均没有受到影响, 因此, 融合前后代码的功能是一致的.

在本节中, 对融合所涉及的相关名词进行了形式化定义, 并对相关技术进行了系统阐述, 本文将在第 4 节中对基于分片的代码融合过程与关键技术的实现方式进行说明.

3 基于分片的代码融合过程

本文提出的代码融合技术为源代码层面的融合, 在本节中, 以 C 语言代码为目标语言类型, 实现分片融合的原型系统. Clang 是一个基于 LLVM 框架的前端编译器, 用于 C 代码的词法分析、语法分析、编译等. 其中, libclang 提供了对 C 代码的抽象语法树解析与遍历接口. 因此, 选择 libclang 所提供的语法树解析接口, 对 C 代码进行分析处理. 主要过程如图 5 所示. 主要包括以下 4 个步骤.

(1) 代码分析. 该部分利用 libclang 提供的语法树分析接口, 解析母体、目标代码中所有函数的抽象语法树, 并形成控制流图, 为融合点选取与目标代码分片提供分析依据. 同时, 在遍历的过程中, 对代码中存在“指针参数”、“全局变量访问”、“库函数调用”的位置设置区分变量;

(2) 融合点选取. 使用必经点选取算法分析母体函数的控制流图, 选取出函数的可融合位置, 确保可融合位置的代码在执行过程中能够按照固定的顺序完整执行;

(3) 目标代码分片. 通过必经点选取算法, 选出目标函数中的必经点, 以必经点作为切分位置, 将目标函数切分成若干个分片, 且分片数量小于母体函数中可融合位置的数量, 并对每个分片的结构类型进行标识;

(4) 代码融合. 使用代码融合算法将目标函数分片与母体函数进行融合, 并修复函数调用方式.

融合整个过程均在源代码层面进行, 以代码的抽象语法树作为分析对象. 在第 3.1 节至第 3.4 节中, 将对每个步骤进行分别阐述.

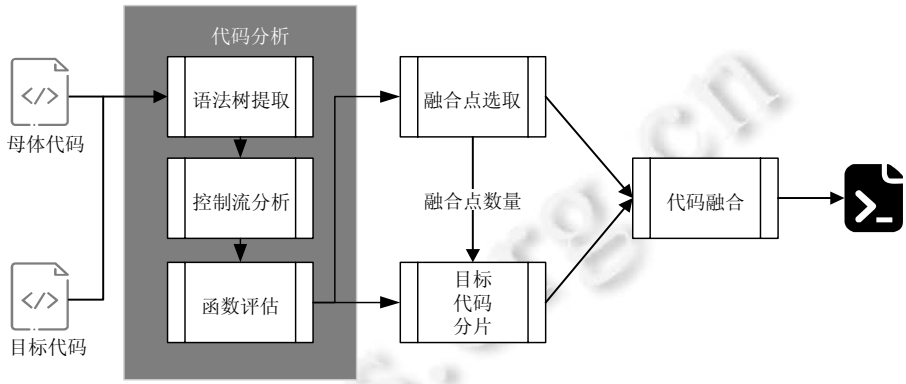

图 5 融合主要过程

3.1 代码分析

代码分析的首要目的是获取母体与目标函数的控制流图. 在源代码层面获取函数的控制流图需要分析源代码的语法结构, 而抽象语法树是代码结构的重要表示方法. 因此, 本节将通过分析函数代码抽象语法树的方式获取函数控制流图.

控制流图与抽象语法树有相似的组成结构, 以图 1 所示的(a)代码为例, 其语法树和控制流图的联系如图 6 所示, 图中每个圆形节点表示代码中的一个表达式, 而方形节点表示语法树中的 Token. α表示在语法树中节点间为兄弟节点, β表示父子节点. ε表示在控制流图中节点间是无条件跳转, 而 True 和 False 表示条件跳转.

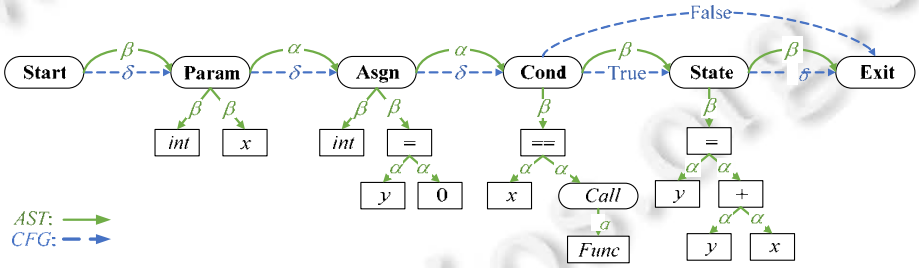

图 6 AST CFG 关系图

由此可见, 控制流图和抽象语法树中节点和边所表示的意义不完全相同, 因此需对语法树的节点和边向控制流图中进行转换. 对于节点的转换, 本文选取语法树中的 DeclStmt, BinaryOperator, ReturnStmt 等表达式类型的节点作为控制流图的基本块. 而对于边的转换, 则依据语法树节点间关系的不同设置不同的转换方法: 若在语法树中两个节点为兄弟关系, 则将其转换为控制流图中的无条件跳转; 若为父子关系, 则转换为有条件跳转. 至此完成抽象语法树向控制流图的转换, 并以此方式获取到每个函数的控制流图.

在进行代码融合后, 融合后的函数会既在母体函数调用链中, 也在目标函数调用链中. 因此, 融合后的函数可能会被多次调用. 若函数中存在修改系统或环境状态的操作, 例如修改系统配置、改写全局变量、内存操作等, 则可能会影响程序的正常执行. 因此, 在代码分析阶段需要对函数代码进行评估, 评估内容包括函数的参数是否含有指针、执行过程中是否调用了非自定义的函数、是否访问了全局变量. 若函数满足以上的条件之一, 则在这些位置设置“区分变量”. 区分变量作为函数的一个参数, 当函数中某一位置存在上述情况, 根据该参数的值或属性判断是否执行该语句. 如图 7 所示: 函数通过参数中包含的区分变量 *va* 的值来判断是否

执行 API 函数, 当函数被目标程序调用时, 使区分变量的判断为真, 执行 API 函数. 由于添加的区分变量与普通变量没有明显区别, 且添加区分变量是在源代码层面, 从开发者角度来说, 在代码编写过程中, 为代码添加 if 结构的情况非常常见, 并且可使用多组变量构建条件谓词, 提高区分变量的多样性, 还可以在代码其他位置也添加这样的区分变量, 使分析人员难以分析出添加此类分支的规律. 因此, 从逆向分析者的角度, 添加区分变量而增加的条件分支, 与程序中原有的其他分支相比并不存在显式区别, 分析者难以分辨出普通分支与区分变量的分支.

```
void test(·) {
   //Other Statements;
   HANDLE file=
   CreateFile (
   L“C:\\a.txt”,
   GENERIC_READ,
   FILE_SHARE_READ,
   NULL,
   OPEN_EXISTING,
   NULL,
   NULL
   );
}
```

```
void test(int va) {
   //Other Statements;
   HANDLE file=NULL;
   If (va==1) {
      File=CreateFile (
         L“C:\\a.txt”,
         GENERIC_READ,
         FILE_SHARE_READ,
         NULL,
         OPEN_EXISTING,
         NULL,
         NULL
      );
   }
}
```

图 7　对 API 函数位置设置区分变量

在 C 代码语法中, 大多数情况下直接通过标识符进行变量访问与函数调用, 且全局变量定义在函数外部. 因此, 只需对函数的语法树中对所有 Token 的标识符(spelling)进行检查, 若标识符不是该函数体中的参数名、关键字、其他自定义函数的函数名或 DeclStmt 结构声明的变量名, 则认为该位置可能修改环境状态, 对该位置设置区分变量并记录, 并将区分变量加入到函数的参数列表.

在对所有母体函数与目标函数完成代码分析后, 即可对函数进行融合点选取或分片操作.

3.2 融合点选取

选取融合点是指依据母体函数的控制流图, 从代码中选取出可以进行融合的位置. 本节将介绍融合点的选取原则与方法, 并提出一种融合点选取算法.

函数与函数之间的融合既要保证融合后的代码符合语法规范, 也要保证融合后各自功能的正确性. 若融合点在母体函数中的分支、循环等结构内部, 则会使插入的代码发生不执行或多次执行的情况, 影响融合后代码功能的正确性. 另外, 目标函数分片具有固定的顺序, 若在融合后, 分片的运行顺序发生变化, 同样会造成程序运行失败. 基于以上两点, 函数内融合点的选择应满足以下条件: 第一, 该融合点在母体函数控制流图中属于必经点; 第二, 选出的融合点具有固定顺序. 为满足以上两个条件, 本文提出一种融合点判断算法. 具体算法如下.

算法 1. 函数内融合点选择算法.

输入: 点横坐标 *row*, 点纵坐标 *col*, 结束节点标识 *end_value*, 当前遍历的节点 *s_key*;

输出: [*row*,*col*]是否必经路径, 是: true; 否: false.

begin

```
1.  Dictionary⟨int List⟨int⟩⟩ route;   //控制流图中的路径
2.  List⟨[int,int]⟩ used;   //遍历过的节点
3.  List⟨int⟩ circle;   //环
4.  procedure FindUseless(int row,int col,int end_value,int s_key)
5.      If (route.HasKey(s_key)):   //此节点是否有子节点
6.          bool r=false;
```

```
7.        foreach (int v in route[s_key]):   //遍历此节点的所有子节点
8.          if (used.has([s_key,v])):   //如果到下个节点的路径已走过则跳过
9.            circle.add(s_key);   //若下一条路径已走过, 则记录当前节点为环的开始节点
10.           continue;
11.         end if
12.         if (s_key==row && v==col):   //如果此路径是待验证路径则跳过
13.           continue;
14.         end if
15.         used.add([s_key,v]);   //将此路径加入到已验证路径列表中
16.         bool res=FindUseless(row,col,end_value,v);   //递归验证下个节点
17.         r=r||res;   //判断该节点是否能到达最终节点
18.       end foreach
19.     else if (s_key==end_value):
20.       return true;   //如果该节点是最终节点, 则返回 true
21.     else:
22.       return false||r;   //若不是, 则返回 false, 并与其他路径结果取并集
23.     end if
24. end procedure
end
```

该算法将控制流图抽象为矩阵, 若控制流图中 i 节点与 j 节点相连, 且从 i 可达 j, 则矩阵中[i,j]的值为 1, $i\rightarrow j$ 为从 i 到 j 的路. 而从矩阵中选取必经点的方法是:

首先选取矩阵中一个值为 1 的点[*row*,*col*], 验证从起始节点在不经过 $row\rightarrow col$ 的情况下, 能否到达最终节点 *end_value*: 若不能, 则表示 $row\rightarrow col$ 是必经路径. 当前控制流图中所有必经路径上点的集合即为该图的必经点, 这些点记为母体函数中的融合点. 在记录融合点的过程中, 同时记录融合点的位置、顺序以及所在位置的语法结构, 用于确定目标函数分片数量以及选择对应融合算法.

3.3 代码分片

依据第 2.1 节中对代码分片的定义, 分片的初始节点与结束节点均为控制流图中的必经节点. 因此, 算法 1 同样可以用于目标函数的切分位置选取. 不同的是, 目标函数在选取切分位置后, 直接按照切分位置将函数代码切分为若干个分片. 若融合后分片之间的相对位置发生变化, 会影响程序的正确性, 因此在分片的过程中, 需对每个分片的顺序进行记录. 如图 8 所示, *A*, *B*, *D*, *F*, *G*, *H* 节点均为控制流图中的必经点, 若以基本块 *B*, *D* 为切分点, 则切分为(a), (b), (c)这 3 个代码分片. 在融合时, 按照(a), (b), (c)的顺序进行融合, 以保证融合后相对位置不变.

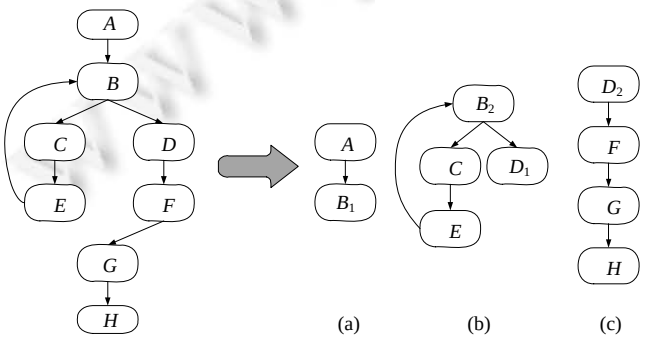

图 8 代码分片

分片是代码融合的最小单元, 在融合后的函数中, 目标函数的分片与母体函数的代码相互交叉, 从逆向分析者的角度, 母体代码与目标代码交错执行, 难以区分出母体代码与目标代码, 因此难以从融合后的函数中还原出原目标函数. 至此, 基本完成融合前的准备工作, 在下一节中, 即对具体的融合策略进行说明.

3.4 代码融合

在完成母体函数融合点选取, 目标函数分片之后, 即可开始进行融合. 根据输入的函数融合点信息, 将目标函数分片插入母体函数代码中组成新的函数体, 并修复函数调用关系. 本文针对不同分片结构设计了两种融合方式: 一种是直接将分片插入母体函数的融合点, 另一种是分支融合. 其中, 直接进行分片插入的融合方法如算法 2 所示.

算法 2. 分片融合算法.

输入: 目标函数的分片列表 *slice*, *func*1 中融合点列表 *points*, 母体函数 *sfunc*, 目标函数 *pfunc*;

输出: 融合后的函数 *func*.

begin

```
1.  Struct Func {
2.      char*argv[·];   //函数的参数
3.      Type[·] rets;   //函数的返回
4.  }
5.  Struct If_STMT {
6.      STMT condition;   //分支结构的条件
7.      STMT[·] body;   //当前条件下要执行的代码
8.  }
9.  Func sfunc; Func pfunc;
10.  procedure Fuse(List⟨Slice⟩ slices, List⟨point⟩ points)
11.     for (int j=0; j<len(points); j++)   //遍历母体中每个融合点
12.        if (len(points)–j>len(slices)–i)   //如果剩余的融合点数量大于剩余的分片量
13.           if (random(0,1)>0.8)   //随机决定是否在此进行融合
14.              continue
15.           end if
16.        end if
17.        if (slices[i] is 'if' && points[j].type is 'if')   //分片与融合点位置结构均为分支
18.           points[j]=fuseif(slices[i],points[j])   //进行分支融合
19.        else if (points[j].type is 'do-stmt')   //融合点位置为 Do-Loop 式的循环
20.           If_STMT if_loop
21.           if_loop.condition=Arg.varN   //使用区分变量作为判断依据
22.           if_loop.body=slices[i]+"Arg.varN=false;"   //确保该段只执行一次
23.           points[j].place.addr.insert(if_loop)   //将包含分片的新分支插入 Do-loop 循环
24.        else
25.           points[j].place.addr.insert(slices[i])   //将分片插入该融合点
26.        end if
27.        if (i==len(slices.slice))   //如果所有分片均已融合完毕, 则结束
28.           break
29.        end if
30.        i++
```

```
31.    end for
32.    sfunc.argv[·]=sfunc.argv[·]+pfunc.argv[·]+Arg   //整合母体与目标函数的参数, 并加入区分变量
33.    sfunc.rets=sfunc.rets+pfunc.rets   //通过结构体返回两个函数的返回值
34.    return sfunc
35. end procedure
end
```

算法 2 的核心思想是, 按顺序遍历母体函数中的每个可融合点, 将目标函数分片按顺序插入到融合点位置, 确保在融合后每个目标函数分片之间的相对顺序不变. 若融合点位置与目标函数分片均为分支结构, 则通过算法 3 进行分支融合. 若融合点位置是 Do-loop 式循环, 则创建一个区分变量, 以保证该分片在循环中不会被多次执行. 函数内容融合完成后, 将母体函数、目标函数的参数与引入的区分变量, 共同作为新函数的参数. 随后, 设置新结构体类型作为融合后函数的返回值类型, 新结构体的属性中包含母体与目标函数的返回值类型. 最后修改其他函数中对该函数的访问方式, 完成该母体函数与目标函数的融合.

算法 3. 分支融合算法.

输入: 母体函数中的分支条件组 *ifA*, 目标函数中的分支条件组 *ifB*;

输出: 融合后的分支条件组 *ifC*.

```
begin
1.  Struct If_STMT {
2.     STMT condition;   //分支结构的条件
3.     STMT[·] body;   //当前条件下要执行的代码
4.  }
5.  If_STMT[·] ifA; If_STMT[·] ifB;
6.  procedure fuseif(If_STMT[·] ifA, If_STMT[·] ifB)
7.     If_STMT[·] ifA
8.     for (int j=0; j<len(ifA); j++)
9.        for (int i=0; i<len(ifB); i++)
10.          If_STMT new_if
11.          new_if.condition=ifA[j].condition+ "&&" +ifB[i].condition
             //将两个条件谓词取并集, 作为新分支的条件谓词
12.          new_if.body=ifA[j].body+ifB[i].body   //将分支的内容进行合并
13.          ifA.add(new_if)
14.       end for
15.    end for
16.    return ifA
17. end procedure
end
```

算法 3 的核心思想是, 提取出两个分支结构中的所有条件谓词, 其中, else 对应的条件谓词, 设为其他条件并集的非集; 随后, 将母体函数中的分支与目标函数分片中的分支两两组合, 形成新的分支结构.

在整个融合的过程, 若目标函数的分片数量小于母体函数的可融合点数量, 即: 在遍历可融合点时, 随机选择是否在当前位置进行融合. 因此, 只有在融合位置与分片均为 if 分支、且同时随机数的值满足条件的情况下才会进行分支融合, 以此避免融合后代码出现大量的代码拷贝. 同时, 采用随机选择的方式, 能够使融合位置更加随机, 在一定程度上提高了融合的多样性.

4 实验分析

4.1 实验样本

为了验证基于代码融合的混淆方法的有效性，本文将以开源程序的 C 代码作为实验对象，根据融合前后的程序运行时间，测试融合对代码执行效率、代码复杂度的影响；并从不同角度对混淆前后代码的相似性进行测试，验证融合对比其他混淆技术在抗同源分析能力上的优势.

由于在本文所提的融合技术中要求母体代码不包含系统或资源写入功能，且不能对目标代码运行产生影响，因此使用本文作者的另一研究成果[18]，通过代码随机生成技术生成的 C 代码作为母体代码，生成的代码主要包含多种算法运算、数据处理、字符操作等. 在实验前随机生成了 500 个母体代码，这些代码的函数数量及融合点数量如图 9 所示. 由于融合点选取是在源代码层面进行，代码中的每个表达式均表示一个基本块，因此可选的融合位置较多. 在这些代码中选出的 5 个母体代码均不含有全局变量、指针类型参数以及 API 函数，因此实验中的母体代码未被添加区分变量.

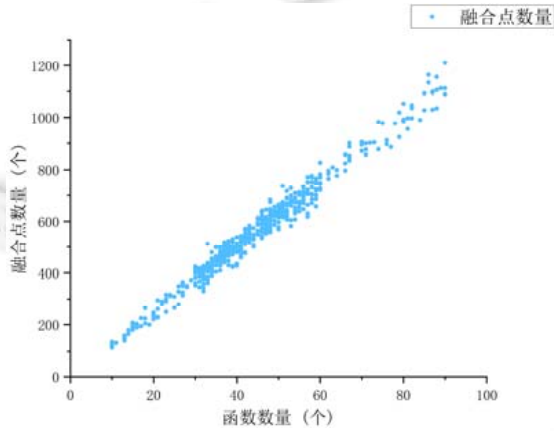

图 9 母体代码融合点数量及函数数量分布图

本文选取 5 个 C 语言开源工具组成目标代码测试用例集，测试样本中用到了多种复杂加密算法、指针调用、库函数调用等，由于融合受限于母体代码的代码量，因此选择了与母体代码量相近的工具作为本次测试的目标代码. 测试样本代码均来源于 GitHub 开源网站. 各测试用例详细说明见表 1.

表 1 实验测试样本集

序号	用例名称	函数数量	类型
1	File2Base64	11	目标代码
2	AES	19	目标代码
3	RSA	49	目标代码
4	gzip	52	目标代码
5	twoif	72	目标代码
6	SkinA	30	母体代码
7	SkinB	32	母体代码
8	SkinC	36	母体代码
9	SkinD	35	母体代码
10	SkinE	41	母体代码

将目标代码与母体代码分为 5 个融合组，每一融合组对应的融合信息见表 2，其中，函数数量差值为母体函数数量与目标函数数量的差值(为测试母体函数数量与目标函数数量的差值对融合效果的影响，特别进行 3 组母体代码数量小于目标代码函数数量情况下的实验). 因目标代码中使用了指针、全局变量和系统函数,在代码中设置了区分变量，并对数量进行了记录. 经过测试，1 组至 5 组的融合代码在目标代码功能上与融合前的

目标代码相同. 测试主机环境为: VS2019, BinDiff 6 (IDA 7.5), Kam1n0 V2 (IDA 7.5), 16 G 内存, Intel I5 处理器, Win 10 x64 系统.

表 2 测试样本分组融合详情

组序	目标代码	母体代码	目标函数	母体函数	函数数量差值	区分变量数量
1	Aes	SkinA	19	32	13	17
2	File2Base64	SkinB	11	35	24	3
3	gzip	SkinC	52	41	−11	37
4	RSA	SkinD	49	30	−19	24
5	twoif	SkinE	72	36	−36	32

4.2 运行效率影响分析

融合之后的程序会交错执行母体代码与目标代码, 所以在运行效率方面, 需要考虑到母体代码的执行效率. 本节将通过测试母体代码、目标代码、融合后代码的执行时间, 验证融合对代码执行效率的影响. 该时间为程序在相同环境下正常运行功能所消耗的时间. 实验结果如图 10 所示.

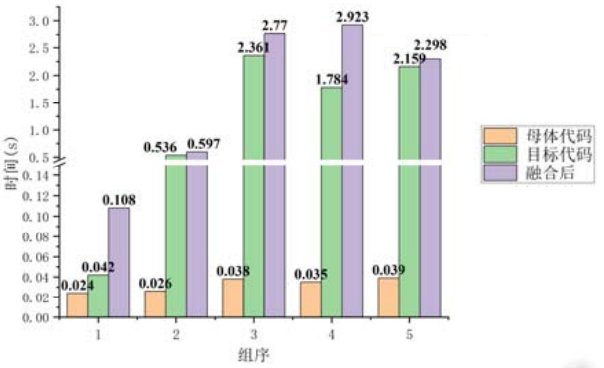

图 10 融合前后运行时间变化

从结果可以看出, 融合对代码整体的运行效率上产生了一定的影响. 产生该问题的原因在于: 母体函数与目标函数融合后, 新函数同时存在于母体代码与目标代码的函数调用过程中, 且该函数包含了母体代码与目标代码, 执行的函数数量与每个函数所需运行的代码量有所增加, 因此影响了融合后代码的运行效率. 从分析者的角度, 程序函数调用过程与函数的执行过程变得更加复杂, 对融合后代码的分析难度有较大提升. 因此从总体的实验结果来看, 该影响是可以被接受的.

4.3 复杂度影响分析

圈复杂度是一种表示代码复杂程度的度量值, 也被称为条件复杂度或循环复杂度, 计算公式为

$$Complexity=E-N+2,$$

其中, E 为控制流图中边的个数, N 为控制流图中节点的个数. 由此公式可以看出: 程序的复杂度与程序中包含的分支数量成正相关, 即程序中存在的分支结构越多, 代码的混乱程度越高,程序就越复杂.

本节将使用 SourceMonitor 对融合前后的母体代码与目标代码进行平均圈复杂度测试, 验证融合以及区分变量对代码复杂程度的影响. 实测结果如图 11 所示.

在理想情况下, 若母体与目标代码中所有的函数均被融合, 且没有添加区分变量的情况下, 根据圈复杂度计算公式推理, 融合后, 函数的代码复杂度应在融合前两函数复杂度的中间值上下浮动, 且理论上更接近于函数数量多的代码. 而添加区分变量会增加函数中的分支结构, 使复杂度升高. 从图 11 展示的结果可以看出: 融合后的代码复杂度与融合前复杂度较高的代码更为接近, 且函数数量差值越大, 融合后代码复杂度与函数数量多的代码复杂度越接近. 同时, 区分变量最多的 3 号样本组复杂度增量最高, 区分变量数量最低的 2

号样本组增量最低. 因此可以判断, 融合后的代码复杂度更接近函数数量多的代码. 同时, 区分变量会提高代码的复杂度, 因此融合会在一定程度上提高代码的复杂度. 但同时, 代码的复杂度提高会造成代码运行过程的复杂性提高, 对提高程序的抗分析能力有正向作用.

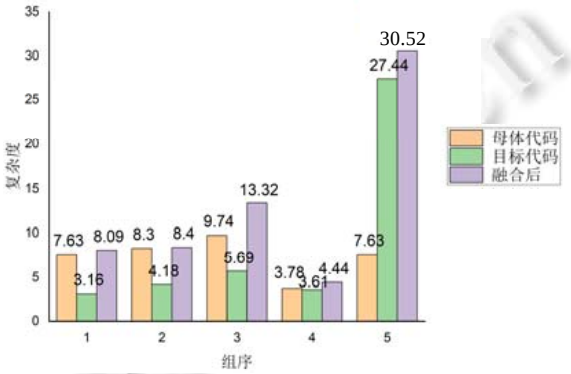

图 11 平均代码复杂度

4.4 基于语义的相似性对比分析

在混淆前后程序相似性的对比实验中, 为从多方面验证融合后程序的抗相似性检测能力, 本文将采用两种不同的相似性对比方式进行实验: 一种是基于语义的相似性对比, 另一种是基于控制流的相似性对比. 本节将使用基于语义的相似性分析算法 Asm2Vec[19]验证融合对代码语义的影响. Asm2vec 是 Ding SHH 在安全顶级会议 S&P 上公布的一个程序语义相似性对比算法, 该算法能够通过分析汇编代码将每个函数的语义形成一个特征向量, 通过不同程序函数间向量的余弦距离判断函数的相似性, 进而得出函数间的语义相似性, 能够在一定程度上突破混淆技术以及编译器优化, 在工程与研究领域均对其有较高的评价. 目前, 该算法集成在二进制对比系统 Kam1n0[20]中. 实验将融合后的程序与对应的目标程序、母体程序分别进行语义相似性检测. 该对比结果为判定相似的函数数量(matched functions)与自身总的函数数量(source functions)的比值, 比值越大, 则表示测试样本中检测到语义相似的函数越多(在 Kam1n0 系统中, 计算得出的相似度结果为相似的函数数量与库中程序的函数数量(index functions)的比值, 若库中函数数量较少, 则计算出的结果值可能超过 100%, 因此对其计算方式进行了修改). 具体计算方法为

$$Similarity=Matched\ Functions/Source\ Functions.$$

假设 A 程序有 100 个函数, 其中 35 个函数与库中 B 程序相似, 那么计算可得 A 与 B 程序的相似度为 35%. 在实验过程中, 阈值(threshold)均设置为 0.6, 即: 当两个函数之间的语义相似度达到 60%以上时, 则判定该两个函数是相似的.

该实验分为两组进行: 第 1 组实验进行融合后程序与融合前母体程序、原目标程序之间的语义相似性测试; 第 2 组实验使用 OLLVM 混淆框架对原目标程序分别进行控制流扁平化、冗余基本块添加、等价指令替换这 3 种技术处理, 并与代码融合做对比, 比较原程序与混淆后的程序之间的语义相似性.

第 1 组实验的测试结果见表 3.

表 3 融合后与母体、原程序语义相似性检测结果

组序	目标代码	与母体程序相似度(%)	与原目标程序相似度(%)
1	Aes	34.37	40.63
2	File2Base64	37.14	40.00
3	gzip	26.92	48.07
4	RSA	32.65	57.14
5	twoif	20.83	69.44

由结果可以看出: 经过融合后的程序与母体程序和目标程序均有较低的语义相似性, 其中, 组序为 3, 4, 5 的实验组相较于其他两组, 与原目标程序的相似度较高, 其主要原因是有部分函数没有被融合, 因此部分函数能够被完全匹配, 从而升高了相似度; 并且由该结果以及相似度计算公式可以得出, 融合后程序的语义相似性与参与融合的函数数量有较大关系, 如若未融合的函数中母体函数占得比重越高, 则会使融合后程序与母体代码的语义相似度越高. 因此, 选择函数数量多于目标代码的母体代码, 能够更好地干扰语义检测. 将此结果与其他混淆技术的检测进行对比, 对比结果见表 4.

表 4 不同混淆技术下语义相似性检测结果

组序	目标代码	控制流扁平化(%)	指令替换(%)	冗余基本块(%)	融合(%)
1	Aes	89.47	94.73	94.73	40.63
2	File2Base64	90.90	90.90	90.90	40.00
3	gzip	80.76	80.76	94.23	48.07
4	RSA	95.91	97.95	85.71	57.14
5	twoif	93.05	97.22	88.88	69.44

从表 4 中可以看出: 相比于其他混淆技术, 融合能够更有效地混淆程序语义. 其主要原因在于: 传统的混淆技术中, 只是对代码进行等价变化, 其语义并不会发生大的变化; 但融合技术通过引入其他程序的代码, 使目标代码与母体代码交替存在, 而 Asm2vec 在运行的过程中会不断根据上下文分析出的语义结果调整语义向量, 而融入的其他代码能够直接改变上下文结构, 进而影响语义分析的结果. 因此, 融合技术在对抗基于语义分析的相似性检测技术中有显著效果.

4.5 基于控制流的相似性对比分析

BinDiff 是谷歌的二进制文件分析对比工具, 能够从基本块、控制流图与函数调用关系等多个层面判断二进制文件之间的相似性. 该工具计算出两程序中函数的相似度(similarity), 并根据使用的算法强弱给出信任值(confidence). 但计算程序相似性的具体算法并未公布, 只能通过软件 UI 界面获取最终的程序相似性结果, 或者从产生的“.BinDiff 文件”(该文件为 BinDiff 用于存放测试结果的 sqlite 数据库)中获取精确的相似值.本实验中, 测试结果均为“.BinDiff 文件”中记录的相似值, 在保留两位小数后得出的最终结果. 在本实验中, 将使用 BinDiff 从基本块、控制流图与函数调用关系等角度, 检验融合前后程序的相似性. 本节将同样按照第 4.4 节中的分组方式进行实验: 第 1 组测试融合后程序与融合前母体程序、原目标程序之间的相似性, 第 2 组进行其他混淆技术与融合的对比实验. 具体实验结果见表 5、表 6.

表 5 融合后与母体、原程序控制流相似性检测结果

组序	目标代码	与母体程序相似度(%)	与原目标程序相似度(%)
1	Aes	67.30	32.98
2	File2Base64	87.87	43.34
3	gzip	50.69	28.98
4	RSA	48.21	22.32
5	twoif	52.25	27.55

表 6 不同混淆技术下控制流相似性检测结果

组序	目标代码	控制流扁平化(%)	指令替换(%)	冗余基本块(%)	融合(%)
1	Aes	49.85	96.22	50.81	32.98
2	File2Base64	60.37	97.54	54.18	43.34
3	gzip	37.80	95.79	48.77	28.98
4	RSA	53.02	97.89	52.53	22.32
5	twoif	36.49	95.73	41.96	27.55

从以上结果可以看出: 融合后的程序与母体程序的相似度偏高, 但与原目标程序的相似度较低. 经过分析, 其主要原因是: 在实验中选择的母体代码中, 比目标代码有更多的分支和循环结构(由第 4.3 节的复杂度实验的结果中可以看出), 而融合后的程序完整保留了这些结构, 因此 BinDiff 能够检测出较高的相似性. 但在第 2 组实验中, 与其他混淆技术相比, 使用融合进行混淆的代码与原目标代码的相似度更低. 其中, 使用控制

流扁平化与添加冗余基本块进行混淆的代码虽然在一定程度上会改变控制流的结构, 但多数基本块中代码没有发生大的变化. 而等价指令替换只能改变部分指令, 改动的部分有限, 因此相似性最高. 从实验结果可以看出: 基于融合的混淆技术相比于其他混淆技术, 在对抗基于基本块、控制流的相似性检测能力上有更明显的优势.

5 总 结

针对现有的代码混淆技术具有明显混淆特征、难以对抗基于语义分析检测技术的问题, 本文提出了基于分片融合的代码隐式混淆技术, 利用其他无害程序代码作为母体, 根据必经点选择算法选出母体函数的可融合点, 并将分片后的目标函数插入到母体函数中, 目标代码以片段的形式隐藏于母体代码中, 使目标代码难以完整地从程序中剥离, 以此达到保护的效果. 本文以 LLVM 框架中的 libclang 代码解析库为基础, 阐述了包括代码分析、融合点选取、目标代码分片、融合这 4 个主要步骤的实现方法. 最后, 在实验部分选取了开源的 C 代码工具作为测试样本集, 在混淆前后的资源开销、代码复杂度、相似性等进行了测试. 从测试结果来看: 基于分片融合的代码隐式混淆技术相比于其他混淆技术, 在对抗基于语义分析、控制流的相似性检测方面均具有明显的优势. 该研究对以往只对代码做等价变换的混淆思路提供了新的混淆模式参考. 但在研究过程中发现: 融合对母体代码与目标代码的健壮性要求较高, 且由于对母体代码做了相关限制, 导致构建可融合的母体代码难度较高. 除此之外, 通过静态方式查找代码中可能修改全局环境的位置难度较大. 因此在接下来的研究中, 将进一步优化融合策略, 降低对母体代码的要求, 并且在数据环境上做到母体代码与目标代码的完全隔离, 以此减少代码之间的相互干扰.

References:

[1] Collberg C, Thomborson C, Low D. A taxonomy of obfuscating transformations. Technical Report, 148. Department of Computer Science the University of Auckland New Zealand, 1997. [doi: 10.1109/SEFM.2005.13]

[2] Blazy S, Trieu A. Formal verification of control-flow graph flattening. In: Proc. of the 5th ACM SIGPLAN Conf. on Certified Programs and Proofs. St. Petersburg: ACM, 2016. 176−187. [doi: 10.1145/2854065.2854082]

[3] Hosseinzadeh S, Rauti S, Laurén S, *et al*. Diversification and obfuscation techniques for software security: A systematic literature review. Information and Software Technology, 2018, 104: 72−93. [doi: 10.1016/ j.infsof.2018.07.007]

[4] Ismanto RN, Salman M. Improving security level through obfuscation technique for source code protection using aes algorithm. In: Proc. of the 7th the Int'l Conf. on Communication and Network Security (ICCNS 2017). Tokyo: ACM, 2017. 18−22. [doi: 10.1145/3163058.3163071]

[5] Qin J, Bai Z, Bai Y. Polymorphic algorithm of javascript code protection. In: Proc. of the 2008 Int'l Symp. on Computer Science and Computational Technology. Shanghai: IEEE, 2008. [doi: 10.1109/ISCSCT.2008.48]

[6] Fass A, Backes M, Stock B. HideNoSeek: Camouflaging malicious Javascript in benign asts. In: Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security. London: ACM, 2019. 1899−1913. [doi: 10.1145/3319535. 3345656]

[7] Popov IV, Debray SK, Andrews GR. Binary obfuscation using signals. In: Proc. of the USENIX Security Symp. 2007. 275−290.

[8] Darwish SM, Guirguis SK, Zalat MS. Stealthy code obfuscation technique for software security. In: Proc. of the 2010 Int'l Conf. on Computer Engineering & Systems. Cairo: IEEE, 2010. 93−99. [doi: 10.1109/ICCES.2010.5674830]

[9] Peng Y, Su G, Tian B, *et al*. Control flow obfuscation based protection method for android applications. China Communications, 2017, 14(11): 247−259. [doi: 10.1109/CC.2017.8233664]

[10] Xu D, Ming J, Wu D. Generalized dynamic opaque predicates: A new control flow obfuscation method. In: Bishop M, Nascimento ACA, eds. Proc. of the Information Security, Vol.9866. Cham: Springer Int'l Publishing, 2016. 323−342. [doi: 10.1007/978-3-319-45871-7_20]

[11] Junod P, Rinaldini J, Wehrli J, *et al*. Obfuscator-llvm—Software protection for the masses. In: Proc. of the 2015 IEEE/ACM 1st Int'l Workshop on Software Protection. Florence: IEEE, 2015. 3−9. [doi: 10.1109/SPRO.2015.10]

[12] Lim K, Jeong J, Cho S, *et al*. An anti-reverse engineering technique using native code and obfuscator- llvm for android applications. In: Proc. of the Int'l Conf. on Research in Adaptive and Convergent Systems. Krakow: ACM, 2017. 217−221. [doi: 10.1145/3129676.3129708]

[13] Ming J, Xu D, Wang L, *et al*. LOOP: Logic-oriented opaque predicate detection in obfuscated binary code. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security. Denver: ACM, 2015. 757−768. [doi: 10.1145/2810103.2813617]

[14] Balachandran V, Keong NW, Emmanuel S. Function level control flow obfuscation for software security. In: Proc. of the 8th Int'l Conf. on Complex, Intelligent and Software Intensive Systems. Birmingham: IEEE, 2014. 133−140. [doi: 10.1109/CISIS. 2014.20]

[15] Collberg C, Myles GR, Huntwork A. Sandmark-A tool for software protection research. IEEE Security & Privacy, 2003, 1(4): 40−49. [doi: 10.1109/MSECP.2003.1219058]

[16] Kulkarni A, Metta R. A code obfuscation framework using code clones. In: Proc. of the 22nd Int'l Conf. on Program Comprehension (ICPC 2014). Hyderabad: ACM, 2014. 295−299. [doi: 10.1145/2597008.2597807]

[17] Aravalli S. Some novice methods for software protection with obfuscation [MS. Thesis]. University of New Orleans, 2006.

[18] Yu P, Shu H, Xiong X, *et al*. A random code generation method based on syntax tree layering model. In: Proc. of the Int'l Conf. on Electronic Information Engineering and Computer Technology (EIECT 2021), Vol.12087. SPIE, 2021. 465−476.

[19] Ding SHH, Fung BCM, Charland P. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: Proc. of the 2019 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2019. 472−489. [doi: 10.1109/SP.2019.00003]

[20] Ding SHH, Fung BCM, Charland P. Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In: Proc. of the 22nd ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. San Francisco: ACM, 2016. 461−470. [doi: 10.1145/2939672.2939719]

于璞(1996－), 男, 硕士生, 主要研究领域为网络安全, 代码保护.

熊小兵(1985－), 男, 博士, 副教授, 主要研究领域为逆向工程, 软件保护.

舒辉(1974－), 男, 博士, 教授, 博士生导师, 主要研究领域为网络安全, 逆向工程.

康绯(1972－), 女, 教授, 主要研究领域为网络安全.