计算机科学
COMPUTER SCIENCE

抗语义分析的脚本融合技术

田博文, 杨巨, 熊小兵, 段爽, 魏然

引用本文

田博文, 杨巨, 熊小兵, 段爽, 魏然. 抗语义分析的脚本融合技术[J]. 计算机科学, 2025, 52(1): 393-400.

TIAN Bowen, YANG Ju, XIONG Xiaobing, DUAN Shuang, WEI Ran. Anti-semantic Analysis Script Fusion Technology [J]. Computer Science, 2025, 52(1): 393-400.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.231100181

抗语义分析的脚本融合技术

田博文[1,2]　杨　巨[2]　熊小兵[2]　段　爽[2]　魏　然[2]
1 郑州大学网络空间安全学院　郑州 450001
2 信息工程大学网络空间安全学院　郑州 450001
(tbw1999@gs.zzu.edu.cn)

摘　要　近年来,脚本程序被广泛应用于计算机领域。脚本程序因其功能强大,执行效率高,相比二进制程序编写更为简单,体积更小,所以在当前网络环境中的使用愈加频繁。目前脚本的混淆技术主要包括编码混淆、结构混淆和加密混淆3种主要类型。然而,现有的脚本混淆方式特征较为明显,存在被反混淆风险,一旦脚本被反混淆,其功能很容易被分析和理解。因此,提出了一种抗语义分析的脚本融合技术,通过将具有普通功能的掩体代码与需要保护的目标代码分块后进行深度融合,融合后的代码同时包含两个脚本的代码,不同脚本之间的语义和逻辑相互交错、相互依赖,使语义分析变得更加困难。对融合后代码的理解和分析需要更加强大的语义推理和上下文理解能力。针对PowerShell脚本的实验表明,融合后脚本程序的控制流循环复杂度平均提升了81.51%,极大提高了代码的混淆强度。该技术能够有效地模糊脚本语义,改变控制流特征,在面对ChatGPT的语义分析中表现出良好的效果,目标代码的核心功能难以被分析理解,从而提高了脚本程序的存活性和持久性。

关键词:码保护;混淆;代码分块;融合;脚本程序
中图分类号　TP311

Anti-semantic Analysis Script Fusion Technology

TIAN Bowen[1,2],YANG Ju[2],XIONG Xiaobing[2],DUAN Shuang[2] and WEI Ran[2]
1 School of Cyber Science and Engineering,Zhengzhou University,Zhengzhou 450001,China
2 School of Cyber Science and Engineering,Information Engineering University,Zhengzhou 450001,China

Abstract　In recent years,script programs have been widely used in the field of computer science. Script programs are increasingly being used in the current network environment due to their powerful functionality and high execution efficiency,simpler writing and smaller file size than binary programs. Currently,the main types of script obfuscation techniques include encoding obfuscation,structural obfuscation,and encryption obfuscation. However,existing script obfuscation methods have obvious features and are at risk of being deobfuscated. Once a script is deobfuscated,its functionality can be easily analyzed and understood. To address this issue,an anti-semantic analysis script fusion technique is proposed. By deeply merging camouflage code with the target code that needs to be protected after dividing them into blocks,the fused code contains the code from both scripts,and the semantics and logic of different scripts are intertwined and interdependent,making semantic analysis more difficult. Understanding and analyzing the fused code requires stronger semantic reasoning and contextual understanding capabilities. Experimental results on PowerShell scripts show that the control flow complexity of the fused script programs is increased by 81.51% on average,and the obfuscation strength of the code is greatly enhanced. This technique effectively blurs the script's semantics,alters control flow characteristics,and performs well in the face of semantic analysis by ChatGPT.

Keywords　Code protection,Obfuscation,Code division,Fuse,Script program

1　引言

脚本程序具有非常强大的功能[1],脚本语言通常具有大量的内置函数和库,可以轻松地处理常见的任务和操作。同时,脚本程序还具有很高的灵活性,可以在不同的平台和环境中使用,例如在Windows,Unix,Linux等操作系统中使用。这种跨平台的特性使得脚本程序具备广泛的适用性。此外,脚本语言还具备与其他编程语言集成的能力,能够实现更复杂的功能。

在PowerShell等脚本语言越来越多地被使用的今天,由于脚本语言无需编译,直接以源码形态进行解释执行,因此,如何在网络行为中增强脚本程序的隐蔽性,提高脚本程序的抗分析能力是当前研究的重点。目前脚本的混淆技术主要包括编码混淆、结构混淆、加密混淆3种主要类型。编码混淆是对脚本代码部分通过十六进制、八进制、二进制、异或等方式进行数据化处理,使其失去原来直观的代码形式,以数据形式

到稿日期:2023-11-27　返修日期:2024-05-03
通信作者:熊小兵(bingxiaoxiong@163.com)

存在于源码中。结构混淆采用各种技术手段来隐藏或修改程序真正的执行流、控制流，达到阻止攻击者分析的目的，常见的方法主要是语法树变形。加密混淆主要使用 Base64、AES 或自身语言特有的加密方式对源码进行加密处理，在运行时再解密执行。

然而，现有的脚本混淆技术存在一定的局限性。传统混淆技术有着混淆方式单一、混淆特征明显等问题，容易被针对特定技术的反混淆技术攻击[2]。随着代码混淆检测技术的发展，静动态检测[3]的方式极大地提升了对脚本代码的反混淆能力。当下，人工智能、数据科学等技术[4-5]不断被运用在混淆检测方面，这些传统的混淆技术已经无法对脚本的语义提供良好的保护作用。本文拟提出一种抗语义分析的脚本融合方法，通过分块融合的方法，该技术实现了脚本程序的高度语义混淆和保护。其在掩体代码中嵌入需要保护的脚本代码，能够有效地提高抗静动态反混淆的能力，有助于保护核心代码语义，阻止恶意分析人员和攻击者理解和修改代码。

对 PowerShell 代码的实验表明，融合后脚本程序的控制流循环复杂度平均提升了 81.51%，极大提高了代码的混淆强度。该技术将两个脚本的语义进行深度融合，在面对 ChatGPT 的语义分析中表现出良好的效果。同时，由于脚本语言之间具有相似的结构特点，该方法同样可以适用于其他脚本语言。

本文的主要贡献总结如下：

1）提出了一种基于控制流分析的融合点选取方法，通过控制流图找到掩体脚本中的必经路径和必经节点，为目标脚本提供可用的融合点位置。

2）提出了一种基于抽象语法树的代码分块方法，利用抽象语法树中提取到的关键信息对目标脚本进行细粒度的分块处理。

3）提出了一种脚本融合方法，用于对脚本程序的核心代码进行语义混淆和验证，提高脚本程序的抗语义分析能力。

2 相关工作

本章讨论脚本混淆技术和程序融合技术的相关工作。

2.1 脚本混淆

本程序的混淆技术研究主要集中在对 JavaScript 的混淆[6]，2020 年 Herrera[7]在对 JavaScript 进行的混淆检测中总结了 JavaScript 字符串切分、关键字替换、字符编码、变量模糊等常见的混淆策略，且这些混淆方式对其他脚本语言同样适用。

1）字符编码混淆

字符编码混淆通过 ASCII 码编码、八进制、十六进制、异或等方式对脚本内容进行处理。然而，Liu 等[8]提出的字符编码混淆方式并没有达到理想的混淆效果。为了改进这种情况，Sharif 等[9]将编码升级为加密，他们对代码进行加密，并在需要时解密后运行。然而，编码混淆会将代码转为数字形式，这不仅会增加代码长度，还会提高数字字符频率，具有明显的混淆特征。

2）语法树结构混淆

抽象语法树（Abstract Syntax Tree，AST）是源代码的抽象语法结构的树状表示，每个节点表示源代码中的一种结构。混淆技术中的热点是对代码结构进行混淆。2019 年，Fass 等[10]通过对 JavaScript 的基本 AST 结构进行同等语义的重写，使其在结构上具有较大的变化但不影响功能。

Bohannon 等[11]实现了一种名为 Invoke-Obfucation 的 PowerShell 混淆框架，它使用基于 AST 语法树的混淆技术来保护 PowerShell 代码。该方法通过对敏感操作数进行特殊符号、字符串格式化等混淆操作来对 PSToken 进行处理，以模糊检测程序对 PSToken 关键字的检测。

3）代码加密混淆

Ismanto 等[12]提出了一种使用 AES 加密对脚本代码进行混淆的方法，利用 AES 算法的强安全性来保护脚本代码。AES 算法是一种分组密码算法，根据密钥长度分为 3 种常见类型：AES-128，AES-192 和 AES-256。传统的混淆方法在保持代码安全性的同时还要保持源代码功能和行为的一致性，而加密方法则是将代码转换为加密密文，执行时先解密密文再执行。然而，该方法不适用于可见的脚本代码。

2.2 程序融合

程序融合技术将两个程序在源码或二进制层面融合成一个新程序，保留原有功能并通过加密或分段的方式保护恶意代码。目前对于程序融合技术的研究较为有限。与之相比，软件水印技术[13-14]是一种由嵌入器、提取器和识别器 3 个组件组成的软件保护技术。通过嵌入器引入秘钥，生成带有水印保护的程序。Chen 等[15]提出的新型动态水印框架应用了程序融合思想，通过控制流混淆连接混淆分支，隐藏动态水印并用神经网络预测执行路径。水印通过特殊输入值绑定存储，只有符合预测范围的输入值才能执行隐藏分支，实现水印与程序的深度融合。

文献[16]提出了一种基于融合编译的软件多样化保护方法，解决了传统保护措施易反编译和易分析的问题。该方法结合掩体代码和内存化目标程序，通过 LLVM 框架和中间语言插桩加密目标程序，并在插桩点插入解密、加载和执行功能的代码，使程序具备掩体和目标程序功能。Yu 等[17]提出了一种基于代码融合的保护技术，通过将 C 语言代码分片后融合，改变控制流特征，提升了对抗语义分析和控制流相似性检测的能力。

然而，现有的脚本混淆技术存在一些明显的问题，如混淆特征容易被识别和混淆方式单一等。这些问题导致传统脚本混淆技术在对抗语义分析时面临挑战。近年来，随着大语言模型[18-20]（如 ChatGPT）等的发展，脚本程序的语义分析迎来了重大突破。大语言模型具备强大的上下文感知能力，可以更好地理解脚本程序的语义，面对传统混淆技术混淆后的脚本，它们能够准确地分析出混淆前脚本的语义。

3 方法

3.1 概述

本文提出了一种基于分块融合的脚本融合方法，该方法主要包含源代码处理、融合点选取、代码分块和脚本融合 4 个部分。源代码处理的主要目的是解析掩体代码和目标代码并对可能影响融合过程的代码进行处理；掩体代码融合点选取则为目标代码分块提供可插入位置的选择；代码分块负责将

目标代码切分为融合所需的细粒度代码块；脚本融合算法实现了目标代码分块和掩体代码融合点的结合并进行功能性验证。该方法的框架如图 1 所示。

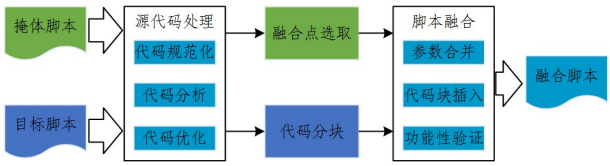

图 1　方法框架图

Fig. 1　Framework of the proposed method

该方法接收一个掩体代码和一个需要保护的目标代码作为输入。首先，对目标代码和掩体代码进行代码规范化、代码分析、控制流分析和代码优化，完成融合前的准备工作。同时，提取出两个脚本中的 AST，并根据 AST 获取脚本的控制流信息，使用融合点选取算法选出掩体脚本中的可融合位置，根据脚本程序中的关键字将目标代码切分为若干个代码块。最终，通过脚本融合算法，将这两个不同的脚本在控制流上深度融合为一个脚本，并对脚本的功能进行验证，生成融合后的脚本。该脚本结合了目标代码和掩体代码的功能，融合后脚本的语义与目标代码有较大差异，从而提高了混淆程度，增强了脚本的保护性。本文将以 PowerShell 代码为例，介绍脚本融合技术的方法和实验结果。

3.2　源代码处理

为了准确地进行融合点选取和代码分块并实现控制流上的深度融合，首先需要对代码进行分析以及格式化处理，以消除影响脚本信息判断的内容。如图 2 所示，首先对输入的脚本文件进行规范化处理，然后进行代码分析提取出 AST，通过控制流分析提取出 CFG，最后对可能影响融合的代码进行优化处理，为后续融合奠定基础。

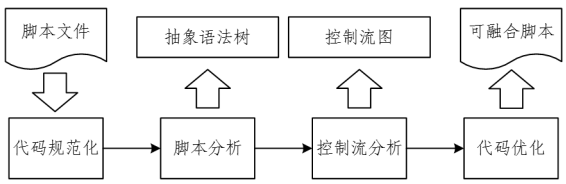

图 2　源代码处理

Fig. 2　Source code processing

1）代码规范化

代码规范化是源代码处理的第一步，其目的是将脚本中的注释、标识符命名、缩进等信息转换为统一的表达方式，从而使脚本的代码块结构更清晰，减少不必要的干扰。规范化脚本的编写风格和结构，有助于提高后续分析和融合的准确性和效率。

代码规范化的处理主要包括以下 3 个方面。首先是注释的清理，脚本中的注释内容可能包含函数功能、输入输出等重要信息。这些信息不仅可能会被攻击者利用，还会影响融合过程中对代码信息的识别。脚本程序的注释通常分为行注释和块注释。在 PowerShell 脚本中，行注释以“#”开头，其后的内容将被忽略。使用“〈#”和“#〉”包围的多行注释被视为块注释，块注释中的所有内容都会被忽略。通过正则表达式匹配注释内容，并将其替换为空白字符串从而清除注释。其次，标识符命名的统一化是另一个重要方面。统一的标识符命名规范有助于降低掩体代码和目标代码的区分度，使得目标代码更好地在掩体代码中进行伪装。最后，对脚本的缩进和换行进行整理，规范的缩进和换行可以为选取融合点位置提供更准确的信息。使用 PowerShell 脚本分析器解析脚本，获取脚本中的不同代码块和语句，并根据一组预定义的规则来自动化格式脚本，从而达到缩进和换行的一致性。

2）代码分析

代码分析的首要目的是获取掩体代码与目标代码的控制流图（Control Flow Graph，CFG）和代码块信息，为后续融合点选取以及脚本分块提供依据。而 AST 为此提供了重要支撑，AST 是源代码的一种表示形式，用于在计算机程序的编译、解析和分析过程中描述其语法结构。它是一种树状结构，其中每个节点表示源代码中的一个语法结构或表达式。CFG 是一个过程或程序的抽象表现，是用在编译器中的一个抽象数据结构，代表了一个程序执行过程中会遍历到的所有路径。它用图的形式表示一个过程内所有基本块执行的可能流向，也能反映一个过程的实时执行过程。AST 展示了代码的层次结构和语法关系，而 CFG 揭示了代码的执行跳转和控制流程。

通过分析脚本的 AST，可以构建脚本的 CFG。由于 AST 与 CFG 中节点和边的含义不同，因此需要进行转换。使用 Parser 类的 ParseFile 方法，可以解析脚本文件，并通过递归处理子节点来构建 AST。对于每个子节点，若是 AST 类型，则作为单一子节点添加；若是 AST 数组，则作为多个子节点添加；若是元组类型，则将其两个元素作为子节点添加，从而构建出完整的 AST。此外，还需要识别 AST 中的控制结构，如条件语句、循环语句和函数调用等。CFG 的基本单位是基本块，表示一组顺序执行的语句。通过分析 AST 中的控制结构，结合条件判断、循环条件或函数调用等信息，可以确定控制流的转移关系，并构建控制转移边。最终，基于基本块和控制转移边的信息，构建出完整的 CFG。

3）代码优化

为确保融合后的目标代码能够正常执行，代码优化的目标是清除或替换掩体代码中会影响目标代码正常执行的代码块。例如，在 PowerShell 中有一些代码会使脚本提前结束运行或者陷入死循环。其中，exit 语句是一种强制终止脚本的方式。同时，使用 while($true)或 for(;;)构造的循环结构会导致脚本进入死循环。这意味着循环内的代码会一直重复执行，导致后续的目标代码永远得不到执行。如果掩体代码中包含这些代码块，它们会阻塞后续目标代码的执行。若掩体代码中包含上述代码，则在融合后都会使后续的目标代码无法正常执行。

通过匹配脚本中是否包含相应代码段，针对脚本程序终止的代码进行删除操作。同时，根据 AST 中 WhileStatementAst和 ForStatementAst 节点的条件值，判断脚本中是否包含死循环。如果出现死循环情况，使用一个有限的 for 循环来进行替换。通过替换掉这些死循环，并检测修改后的脚本能否在指定时间内执行完毕，从而筛选出可用于融合的掩体

代码，按照预期顺利执行。

掩体代码和目标代码中可能存在相同命名的标识符，如果冲突的标识符被用于不同的上下文，可能会导致逻辑错误。因此，代码预处理的最后一步就是对两个脚本中的标识符进行检查，避免发生冲突。在 PowerShell 的 AST 中，通过 FunctionDefinitionAst 节点的 Name 值来获取函数的名称，而 VariableExpressionAst 节点的 VariablePath 值对应于参数和变量的名称。获取这些标识符名称并进行对比，如果发现掩体代码和目标代码中存在相同命名的标识符，那么使用一组随机字符串来替换目标代码中的标识符名称，以避免冲突的出现。PowerShell 中有许多内置的标识符名称，在对比时需要去除这些标识符。完成上述准备工作后，接下来可以对脚本进行融合点选取和分块操作。

3.3 融合点选取

融合点为目标代码提供了可嵌入代码块的位置，为了保证融合后脚本语法和功能的正确性，融合点选取的位置至关重要。融合点的选取必须为掩体代码的必经路径上的必经节点，且保证插入的目标代码块按固定顺序执行一次。程序的必经路径指程序 CFG 中从起始节点到达终止节点必须经过的边，而必经路径上的节点则为该程序的必经节点。

在代码分析过程中，使用 AST 构建了 CFG。CFG 由节点和边组成，节点代表代码块或语句，边表示程序执行的流向。首先，从入口点开始，通过遍历 CFG，找到能从入口点到达出口点的所有路径。在每条路径中都按相同先后顺序经过的路径即为必经路径。对于每条必经路径，标记其中经过的节点作为必经节点，这些必经节点被记为掩体代码中的融合点。同时，记录下这些必经节点的执行顺序以及在脚本中的位置。融合点选取的具体算法如算法 1 所示。

算法 1 融合点选取算法

输入：掩体代码 bscript

输出：新的掩体代码 bscript；融合点列表 points

```
1. function pointSelect(bscript)
2.   List pathList=dfs(bscript)//搜索掩体代码中的所有路径
3.   List points;//融合点列表
4.   for(point in pathList[0]) do//遍历路径列表中第一条路径的所有节点
5.     for(path in pathList)//遍历每一条路径
6.       if point not in path then//该节点不在其他路径中，不是必经节点
7.       continue//跳过该节点
8.     if(point.type=condition) then//条件分支节点
9.       while(case[i]) do//遍历每一个分支
10.      bscript=bscript.setPoint//在每个分支中设置融合点
11.      i=i+1
12.    else if(point.type=loop) then//循环结构节点
13.    bscript=bscript.setFlag//在循环结构设置标志位
14.  points.append(point)//加入融合点列表
15. return bscript,points.
```

为了增强两个脚本在控制流上的融合复杂性并提高目标代码的抗语义分析能力，进一步引入了新的融合点，并将其嵌入在条件分支和循环结构中。在必经路径上的条件分支，其所有分支路径中总有一条路径会被执行。将代码块分别插入到每个分支路径中，则该代码块也会被执行一次。而对于必经路径上的循环结构，则可以在循环体执行前引入一个标志位。同时，在循环体内和循环体后增加对该标志位的判断，在判断条件中设置一个融合点，当判断条件为 True 时执行一遍插入的代码块，并修改标志位的值，以确保代码块被正确执行一次。

如图 3 所示，图 3(a)为掩体代码的代码块，其中虚线矩形内即为一个条件分支。对该条件分支构建融合点后的效果如图 3(b)所示，其中 E，F，G，H 这 4 个代码块必有一个被执行一次。分别在这 4 个代码块后构建一个融合点，则插入到该位置的代码块也会被执行一次。图 3(c)为掩体代码的代码块，其中虚线矩形内即为一个循环结构。对该循环结构构建融合点后的效果如图 3(d)所示，在 α 处新增一个标志位 $flag=$true，β 处对标志位的值进行判断，若为 True 则执行插入的代码块，在代码块执行结束后，在 γ 处修改标志位的值为 False，然后继续执行原有的掩体代码 D。在记录新增的融合点的同时，还需要更新原有的融合点集合。

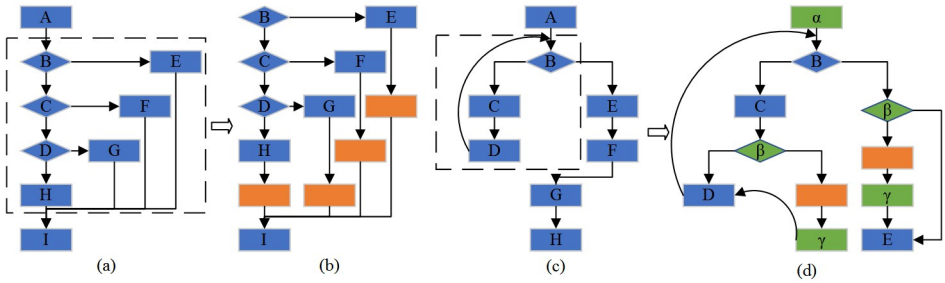

图 3 融合点构建

Fig. 3 Fusion point construction

3.4 脚本分块

在进行代码融合的过程中，需要将目标代码划分成多个细粒度的代码块，以便进行代码的组合和重用。但是，分块的粒度过小可能会破坏代码功能的完整性，因此在进行目标代码分块时要使切分的代码块粒度尽可能小，同时保持代码的完整性。脚本分块的具体算法如算法 2 所示。

算法 2 脚本分块算法

输入：目标代码 tscript；

输出：代码块列表 blocks；

```
1. function scriptDivide(List tscript)
2.   List keywords;
3.   for(keyword in tscript.ast.keyword)//遍历 AST 中的关键字
```

```
4.      keywords.append(keyword);//将关键字放入列表中
5.      keywords.sorted(cfg);//根据 CFG 排序
6.   for(keyword in keywords)
7.      startPosition=keyword.startPosition;//关键字的起始位置
8.      endPosition=keyword.endPosition;//关键字的结束位置
9.      for(i in range(startPosition,endPosition+1))
10.        block=block+tscript[i];//一个关键字的起始和结束为一个代码块
11.        blocks.append(block);//将代码块加入代码块列表
12.  while(len(points)<len(blocks))//如果融合点数量少于分块数量
13.  i=random(0,len(blocks)-2)//获取随机下标
14.  blocks[i]=blocks[i]+blocks[i+1]//随机将两个相邻代码块组合
15.  blocks.pop(i+1)//更新代码块列表
16. return blocks.
```

在 PowerShell 脚本中,关键字是一组具有特殊含义的保留单词,用于控制脚本的语法和结构。通过检查 AST 中的关键字将脚本划分为多个代码块,每个拆分后的代码块都可独立运行。每个条件分支和循环结构都被视为一个完整的代码块。

如图 4 所示,图中的代码被拆分为 a,b,c,d,e 5 个代码块。融合后应确保各个代码块的执行顺序不变,因此需要记录每个代码块的执行顺序、大小以及在脚本中的位置,在融合过程中按顺序插入到融合点的位置。在代码块划分完毕后,还需根据融合点的数量确定最终代码分块的数量,如果代码块的数量多于融合点的数量,则随机将相邻的代码块组合,使最终分块的数量不多于融合点的数量,以确保融合过程的正确性。

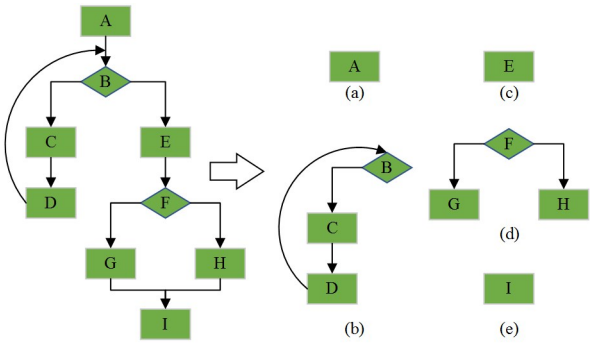

图 4 脚本分块

Fig. 4 Script division

分块是脚本融合的最小单位,在融合后的脚本中,目标代码的分块与掩体代码的代码交织在一起。从分析者的角度来看,掩体代码与目标代码的代码交错执行,难以区分出掩体代码与目标代码,因此难以从融合后的脚本中还原出原目标代码的语义和功能。

3.5 脚本融合

本节将详细介绍融合的具体过程。融合策略的首要任务是合并参数,在 PowerShell 中参数的定义有两种形式。如图 5(a)和图 5(b)所示,一种使用 Param 关键字来定义参数,另一种则在函数或脚本的参数列表中使用 $ 符号加上参数名来声明参数。如图 5(c)所示,将两种参数转化为统一的形式并合并,将掩体代码和目标代码的参数共同作为融合后新脚本的参数。

```
function Add {
  Param(
    [Parameter(Position = 0, Mandatory = $true)]
    [int]$Num1,

    [Parameter(Position = 1, Mandatory = $true)]
    [int]$Num2
  )

  $result1 = $Num1 + $Num2
  return $result1
}
```

(a)

```
function Subtract ([int]$Num3, [int]$Num4) {

  $result2 = $Num3 - $Num4
  return $result2
}
```

(b)

```
function Add-Subtract {
  Param(
    [Parameter(Position = 0, Mandatory = $true)]
    [int]$Num1,

    [Parameter(Position = 1, Mandatory = $true)]
    [int]$Num2,

    [Parameter(Position = 2, Mandatory = $true)]
    [int]$Num3,

    [Parameter(Position = 3, Mandatory = $true)]
    [int]$Num4
  )

  $result1 = $Num1 + $Num2
  $result2 = $Num3 - $Num4

  return $result1, $result2
}
```

(c)

图 5 脚本分块

Fig. 5 Script division

融合算法首先对输入的脚本结构进行判断,根据脚本中是否包含函数,可以将 PowerShell 的结构分为无函数类型和有函数类型两种。根据不同的脚本结构类型的组合,融合可以分为 3 种情况。对于每种情况需要采用不同的融合策略和逻辑来保证融合的正确性和有效性。具体的融合方法如算法 3 所示。

算法 3 脚本融合算法

输入:掩体代码 bscript,目标代码 tscript

输出:融合后的脚本 fscript

```
1. function Fuse(bscript,tscript)
2.    List fscript;
3.    if(bscript.funcNum ! = 0)//如果脚本有函数
4.       bscript=bscript.content;//提取函数中代码
5.    if(tscript.funcNum ! = 0)//如果脚本有函数
6.       tscript=tscript.content;//提取函数中的代码
7.    fscript.params=bscript.params+tscript.params;//参数合并
8.    List bscript,List points=call pointSelect(bscript);//调用融合点选取算法获取融合点列表
9.    blocks=call scriptDivide(tscript);//调用脚本分块算法获取代码块列表
10.   int j=0;//记录条件分支融合点的数量
11.   for(i=0 to len(points)) do//遍历列表中的每个融合点
12.      if(points[i].type=condition) then//如果该融合点为条件分支融合点
```

```
13.      j=j+1;//条件分支融合点的数量加 1
14.    fscript=fscript.insert(points[i],blocks[i-j]);//按顺序将目标代码块插入到融合点位置
15.    if len(blocks)=i-j then//如果所有分块均已融合完毕,则结束
16. fscript.rets=bscript.rets+tscript.rets//返回值修改为两个脚本的返回值
17. fscript.modifyFunctionCalls//修改函数调用
18.  return fscript.
```

根据对脚本结构类型的判断,若两个脚本均为无函数类型的脚本,则首先进行参数合并,在完成参数合并后直接对掩体代码进行融合点选取,并对目标代码进行分块。按顺序遍历掩体代码中的每个可融合点,将目标代码分块按顺序插入到融合点位置。若融合点在条件分支中,则将同一代码块插入到条件分支中的每个融合点;若融合点在循环结构内,则将代码块插入到循环结构中构造的融合点位置上,确保在融合后每个目标代码分块之间的相对顺序和执行次数不变;若两个脚本分别为无函数类型和有函数类型脚本,则保留脚本中原有的函数,将无函数脚本中的参数添加到函数的参数中,然后把无函数类型脚本中的代码融合到函数内部;若为两个有函数类型脚本,则将两个脚本中的函数一一对应,分别进行融合。将来自两个脚本中的两个函数合并为一个函数,并修改原有的函数名,然后将两个函数内的代码进行融合。修改融合后函数中的返回值类型,新函数中的返回值应同时包含掩体代码和目标代码中函数的返回值。最后修改脚本中对函数的调用方式,完成掩体代码与目标代码的融合。

4 实验

4.1 实验样本

为了验证基于分块融合的脚本融合方法的有效性,本文将 GitHub 上的 PowerShell 脚本作为实验对象,根据融合前后脚本的代码插桩结果、运行时间和控制流循环复杂度的变化,验证融合对目标代码功能完整性、运行效率和混淆强度的影响。根据 ChatGPT 对不同技术混淆后的脚本进行语义分析,验证融合对比其他 PowerShell 混淆技术在抗语义分析能力上的优势。

用于融合测试的代码均来自于 GitHub 公共存储库中的 PowerShell 代码。通过查找“PowerShell”等关键词和扩展名为“.ps1”的文件,并删除重复文件和过小的文件,从中选取了 100 个 PowerShell 语言开源代码组成测试用例集,其中 50 个目标代码测试样本中包含多种复杂系统功能操作。为了避免掩体代码的行为对目标代码产生影响,选择的 50 个掩体代码主要包含多种算法运算、数据处理、字符操作等,不包含系统或资源写入功能等。为了更好地评估融合方法的适用性,选取的目标代码规模不等。对于掩体代码,若代码量过小则会导致融合点数量少于目标代码分块数量,使融合后的效果变差。因此,对于每个目标代码样本都选择了相对较大的掩体代码。最终选择的目标代码平均代码行数为 240.1,平均代码块数量为 84.46。掩体代码平均代码行数为 276.12,平均融合点数量为 109.28。测试主机环境为:PowerShell 5.1,Invoke-Obfuscation 1.8.2,Xencrypt 1.0,32G 内存,Intel I7 处理器,Windows 11 x64 系统。

4.2 程序正确性验证

在将掩体代码和目标代码融合后生成的新脚本中,保证融合后的目标代码的功能完整性是至关重要的。为了确保融合后的新脚本能够正常执行完毕,需要对目标代码的程序正确性进行验证。

本文在实验样本选择以及源代码处理环节已经采取了一系列措施来处理可能影响目标代码正常运行的代码,确保掩体代码与目标代码之间不存在冲突,并通过融合点选取和代码分块算法确保目标代码在融合后依然能保持其程序正确性。本文利用代码插桩技术来检查融合后目标代码的正确性。首先,选取目标代码分块的位置作为插桩点,在融合前后目标代码的相同位置插入生成的验证代码来检查目标代码的正确性。验证代码包括变量监测、执行顺序记录。分别运行目标代码和融合代码后,检查每个目标代码块处插桩代码的执行结果。如果验证代码中检测到变量或执行顺序不一致,说明目标代码在执行过程中存在问题。在两个相同环境的 Windows 虚拟机上进行测试,所有 50 个测试样本均正常执行完毕后退出执行流程。代码插桩的结果表明目标代码的功能在融合后并没有发生改变,证明了该方法具有良好的鲁棒性。

4.3 运行效率影响分析

融合算法将掩体代码和目标代码融为一体,使两者内容交替执行,这必然会对运行效率产生一定影响。因此,在分析运行效率时,还需要考虑掩体代码的运行效率。本节将通过测试掩体代码、目标代码、融合后脚本的执行时间,来验证融合对脚本执行效率的影响。所测量的执行时间指脚本在相同环境下正常运行结束所消耗的时间。执行效率的实验结果如图 6 所示。

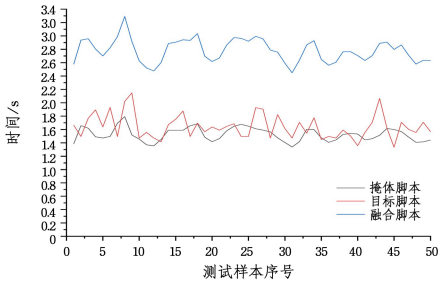

图 6 执行效率

Fig. 6 Implementation efficiency

由实验结果可以观察到,融合算法对脚本整体的运行效率产生了一定程度的影响。融合脚本的平均运行时间比目标脚本多了约 1.138 s。产生该问题的原因在于,掩体代码与目标代码融合后,新脚本同时包含了两者的全部代码,导致执行的代码量接近于两个脚本的总和,从而影响了融合后脚本的运行效率。但是从分析者的角度来看,融合后脚本的执行过程变得更加复杂,对其进行分析的难度也大大提升。这是因为分析者需要同时考虑掩体代码和目标代码的逻辑,并协调二者之间的交替执行,这增加了分析过程中的复杂度,需要更多的时间和精力来理解融合后脚本的行为。因此,综合考虑实验结果和给分析者带来的困难,可以认为这种运行效率的影响是可以被接受的。

4.4 混淆强度影响分析

McCabe[21] 提出了控制流循环复杂度(Cyclomatic Com-

plexity)，并证明了其可以作为一个度量指标有效地衡量程序复杂度。Zhao 等[22]证明了该指标用来评估代码混淆强度的有效性。控制流循环复杂度记为 $V(G)$，计算式如下：

$$V(G)=e-n+2$$

其中，e 表示 CFG 中边的数量，n 表示 CFG 中节点的数量。控制流循环复杂度高的程序，攻击者理解其功能需要花费的时间和精力更多，破解时间更长。因此，本节将通过对比掩体代码、目标代码以及融合后脚本的平均控制流循环复杂度，进而评估融合算法的有效性。控制流循环复杂度的实验结果如图 7 所示。

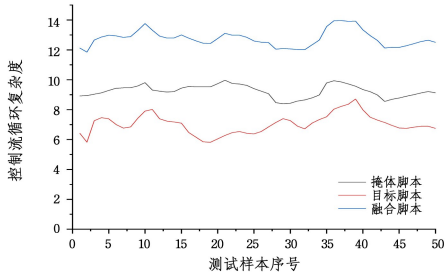

图 7 控制流循环复杂度

Fig. 7 Control flow cyclomatic complexity

理论上，对于掩体代码和目标代码融合后生成的新脚本，根据控制流循环复杂度计算公式推理，融合后脚本的平均控制流循环复杂度应高于融合前两个脚本的控制流循环复杂度，但应小于两者控制流复杂度之和。从图中展示的结果可以看出，融合后脚本的控制流循环复杂度与融合前复杂度较高的脚本更为接近，有些融合后的控制流循环复杂度略高于两个脚本的控制流复杂度之和。这是因为在融合前对掩体代码进行融合点选取时构建了新的分支结构，从而增大了控制流复杂度；此外，两个脚本原来的复杂度越大，则新构建的融合点更多，融合后脚本的控制流复杂度提升也会更加明显。总的来说，融合脚本的控制流循环复杂度相较于目标脚本平均提高了约 81.51%。同时，脚本的控制流循环复杂度提高会造成脚本运行过程的复杂性提高，对提高程序的混淆强度有正向作用。

4.5 语义影响分析

融合脚本中同时包含掩体代码和目标代码，且目标代码分块分散在掩体代码的各个必经节点上。掩体代码的逻辑、条件语句和函数调用等均为目标代码提供了保护。这种融合方式导致目标代码的上下文受到掩体代码的影响，给语义分析带来了更大的挑战。在编程语言分析中，语义分析可以通过考虑上下文中的变量作用域、类型推断等来实现更精确的推断代码功能。但由于目标代码分散插入到掩体代码中，因此目标代码的上下文受到掩体代码的影响。这使得语义分析需要考虑目标代码在不同上下文中的行为，确保其中的变量、函数等符号的正确性和一致性。此外，语义分析还需要能够正确地识别和处理控制流结构，以确保代码的逻辑正确性和执行顺序的准确性。4.4 节中证明了融合脚本具有更加复杂的控制流结构，使语义分析更难理解目标代码的控制流。

ChatGPT 是一种基于深度学习的自然语言处理模型，在脚本语言的语义分析方面展现出卓越的能力。ChatGPT 的自适应建模能力使其能够根据上下文准确地推断出代码的含义和意图。这种上下文理解能力使得 ChatGPT 与脚本语言的语义分析更加契合。因此，本文选择了 ChatGPT 来测试融合后脚本在对抗语义分析方面的能力。本节将对比融合脚本和 Invoke-Obfuscation、Xencrypt 方法进行混淆后的脚本对抗 ChatGPT 语义分析的效果。少样本提示(Few-Shot Prompting)[23]可以利用少量的样本数据来训练模型，使其能够生成有意义且准确的输出。这种方法能够帮助模型通过仅有的几个训练样本来学习代码的结构和特征，从而提升代码去混淆以及语义分析的效果。思维链提示(Chain-of-Thought Prompting)[24]通过提示模型生成一系列推理步骤来解决多步骤的问题，实现了复杂的推理能力。通过将思维链提示与少样本提示相结合，可以获得更好的语义分析结果。Few-shot-CoT 在提示样本中不仅要给出问题的答案，同时需还给出问题推导的过程，从而让模型学到思维链的推导过程，并将其应用到新的问题中。如果直接输入 prompt："Identify the core function of the following piece of code"，对于 3 种方式混淆后的代码，ChatGPT 都很难识别其功能。因此本文在实验中使用 Few-shot-CoT 输入 prompt："The following obfuscated code is deobfuscated, and the core functions of the deobfuscated code are obtained by semantic analysis"。然后针对每种混淆方法，输入 3 组样例进行学习，每组样例包括混淆后的代码、未混淆的代码以及代码的核心功能。最后输入测试样本，使 ChatGPT 按照上述推理过程去进行混淆和语义分析，测试 ChatGPT 进行语义分析后得到的代码功能与原目标代码功能是否相符，测试结果如表 1 所列。

表 1 语义分析识别率

Table 1 Semantic analysis recognition rate

混淆方法	识别脚本	未识别脚本	识别率/%
Invoke-Obfuscation	22	28	44
Xencrypt	30	20	60
Our Method	9	41	18

全部 50 个目标代码在经过 3 种不同的方法混淆后，Invoke-Obfuscation 和 Xencrypt 混淆后的脚本在使用 ChatGPT 进行语义分析后分别识别出了 22 个和 28 个脚本的核心功能。然而，融合后的脚本只识别出了其中 9 个脚本的核心功能。ChatGPT 在面对 Invoke-Obfuscation 和 Xencrypt 混淆后的代码时，因其混淆方式单一，在经过引导进行去混淆后也能分析出目标代码的核心功能。由于掩体代码的多样性，ChatGPT 并不能准确地学习到融合对源代码带来的影响。同时，融合后的代码受到掩体代码的影响，在语义上与目标代码产生了较大的区别。掩体代码引入了新的语义元素并修改了原有的代码结构，使得 ChatGPT 难以理解目标代码的核心功能。相比之下，在 3 种混淆方法中，融合后的代码表现出最好的抗语义分析效果。因此，该脚本融合方法能够有效地提高目标代码的抗语义分析能力。

结束语 本文提出了一种基于分块融合来混淆脚本的方法。该方法首先利用一组预定义的规则将脚本规范化并提取 AST 以及 CFG，然后通过融合点选取和脚本分块处理掩体代码和目标代码，最终融合算法将两个脚本融为一体。实验结果表明，该融合技术可以有效地对脚本程序进行混淆，从而

提高了抗分析的能力。同时，融合后的脚本在面对 ChatGPT 时表现出了较好的抗语义分析能力，并在多组评估实验中表现出良好的性能，证明了该方法的实用性和可靠性。本文的方法虽然实现了两个脚本在控制流和语义上的深度融合，但仍存在一定的局限性。该方法并未涵盖数据依赖关系的混淆，因此在面对基于数据依赖分析的攻击时可能存在一定挑战。在后续研究中，可以通过将目标代码的变量与掩体代码变量进行绑定，从而对代码中的数据流进行混淆。

参 考 文 献

[1] SUDHAKAR, KUMAR S. An emerging threat Fileless malware: a survey and research challenges[J]. Cybersecurity, 2020, 3(1): 1.

[2] CHAI H, YING L, DUAN H, et al. Invoke-deobfuscation: AST-based and semantics-preserving deobfuscation for PowerShell scripts[C]// 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2022: 295-306.

[3] MIMURA M, TAJIRI Y. Static detection of malicious PowerShell based on word embeddings[J]. Internet of Things, 2021, 15: 100404.

[4] RUSAK G, AL-DUJAILI A, O'REILLY U M. Ast-based deep learning for detecting malicious powershell[C]// Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018: 2276-2278.

[5] HENDLER D, KELS S, RUBIN A. Detecting malicious powershell commands using deep neural networks[C]// Proceedings of the 2018 on Asia Conference on Computer and Communications Security. 2018: 187-197.

[6] BLANC G, KADOBAYASHI Y. A step towards static script malware abstraction: Rewriting obfuscated script with maude [J]. IEICE Transactions on Information and Systems, 2011, 94(11): 2159-2166.

[7] HERRERA A. Optimizing away javascript obfuscation [C]// 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, 2020: 215-220.

[8] LIU W Y, FU A Y, DENG X. Exposing homograph obfuscation intentions by coloring unicode strings[C]// Asia-Pacific Web Conference. Berlin: Springer, 2008: 275-286.

[9] SHARIF M I, LANZI A, GIFFIN J T, et al. Impeding Malware Analysis Using Conditional Code Obfuscation [C]// NDSS. 2008.

[10] FASS A, BACKES M, STOCK B. Hidenoseek: Camouflaging malicious javascript in benign asts[C]// Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 1899-1913.

[11] BOHANNON D, HOLMES L. Revoke-obfuscation: powershell obfuscation detection using science [J/OL]. https://www.blackhat.com/docs/us-17/thursday/us-17-Bohannon-Revoke-Obfuscation-PowerShell-Obfuscation-Detection-And%20Evasion-Using-Science-wp.pdf.

[12] ISMANTO R N, SALMAN M. Improving security level through obfuscation technique for source code protection using AES algorithm[C]// Proceedings of the 2017 the 7th International Conference on Communication and Network Security. 2017: 18-22.

[13] 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.

[14] LYNN B, PRABHAKARAN M, SAHAI A. Positive results and techniques for obfuscation[C]// International Conference on the Theory and Applications of Cryptographic Techniques. Berlin: Springer, 2004: 20-39.

[15] CHEN Z, JIA C, XU D. Hidden path: dynamic software watermarking based on control flow obfuscation[C]// 2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC). IEEE, 2017, 2: 443-450.

[16] XIONG X B, SHU H, KANG F. Method of diversity software protection based on fusion compilation[J]. Chinese Journal of Network and Information Security, 2020, 6(6): 13-24.

[17] YU P, SHU H, XIONG X B, et al. Implicit Code Obfuscation Technique Based on Code Slice Fusion[J]. Journal of Software, 2023, 34(4): 1650-1665.

[18] WU T, HE S, LIU J, et al. A brief overview of ChatGPT: The history, status quo and potential future development[J]. IEEE/CAA Journal of Automatica Sinica, 2023, 10(5): 1122-1136.

[19] LIU Y, HAN T, MA S, et al. Summary of chatgpt-related research and perspective towards the future of large language models[J]. Meta-Radiology, 20231(2): 100017.

[20] ZHOU C, LI Q, LI C, et al. A comprehensive survey on pretrained foundation models: A history from bert to chatgpt[J]. arXiv: 2302.09419, 2023.

[21] MCCABE T J. A complexity measure[J]. IEEE Transactions on software Engineering, 1976(4): 308-320.

[22] ZHAO Y J, TANG Z Y, WANG N, et al. Evaluation of code obfuscating transformation[J]. Journal of Software, 2012, 23(3): 700-711.

[23] BROWN T, MANN B, RYDER N, et al. Language models are few-shot learners[J]. Advances in Neural Information Processing Systems, 2020, 33: 1877-1901.

[24] WEI J, WANG X, SCHUURMANS D, et al. Chain-of-thought prompting elicits reasoning in large language models[J]. Advances in Neural Information Processing Systems, 2022, 35: 24824-24837.

TIAN Bowen, born in 1999, master. His main research interests include reverse engineering and software protection.

XIONG Xiaobing, born in 1985, Ph.D, associate professor. His main research interests include reverse engineering and software protection.

（责任编辑：何杨）