

基于符号执行和 N -scope 复杂度的代码混淆度量方法

肖玉强, 郭云飞, 王亚文
(信息工程大学, 河南 郑州 450001)

摘要: 代码混淆可有效对抗逆向工程等各类 MATE 攻击威胁, 作为攻击缓和性质的内生安全技术发展较为成熟, 对代码混淆效果的合理度量具有重要价值。代码混淆度量研究相对较少, 针对代码混淆弹性的度量方法与泛化性、实用性度量方法相对缺乏。符号执行技术广泛应用于反混淆攻击, 其生成遍历程序完整路径输入测试集的难度可为混淆弹性度量提供参考, 然而基于程序嵌套结构的对抗技术可显著降低符号执行效率, 增加其混淆弹性参考误差。针对上述问题, 提出结合符号执行技术和 N -scope 复杂度的代码混淆度量方法, 该方法首先基于程序符号执行时间定义程序混淆弹性; 其次提出适配符号执行的 N -scope 复杂度, 定义程序混淆强度同时增强符号执行对多层嵌套结构程序的混淆弹性度量鲁棒性; 进而提出结合动态分析与静态分析的混淆效果关联性分析, 通过对程序进行符号执行与控制流图提取量化混淆效果。面向 C 程序构建了该度量方法的一种实现框架并验证, 实验对 3 个公开程序集及其混淆后程序集约 4 000 个程序进行混淆效果度量, 度量结果表明, 提出的度量方法在较好地刻画混淆效果的同时拥有一定的泛化能力与实用价值; 模拟真实混淆应用场景给出了该度量方法的使用样例, 为混淆技术使用人员提供有效的混淆技术度量与技术配置参考。

关键词: 代码混淆; 混淆度量; 符号执行; N -scope

中图分类号: TP393

文献标志码: A

DOI: 10.11959/j.issn.2096-109x.2022085

Metrics for code obfuscation based on symbolic execution and N -scope complexity

XIAO Yuqiang, GUO Yunfei, WANG Yawen
Information Engineering University, Zhengzhou 450001, China

Abstract: Code obfuscation has been well developed as mitigated endogenous security technology, to effectively resist MATE attacks (e.g. reverse engineering). And it also has important value for the reasonable metrics of code obfuscation effect. Since symbolic execution is widely used in anti-obfuscation attacks, metrics for code obfuscation

收稿日期: 2021-07-07; 修回日期: 2022-05-11

通信作者: 肖玉强, 516162560@sjtu.edu.cn

基金项目: 国家重点研发计划 (2021YFB1006200, 2021YFB1006201), 国家自然科学基金(62072467)

Foundation Items: The National Key R&D Program of China (2021YFB1006200, 2021YFB1006201), The National Natural Science Foundation of China(62072467)

引用格式: 肖玉强, 郭云飞, 王亚文. 基于符号执行和 N -scope 复杂度的代码混淆度量方法[J]. 网络与信息安全学报, 2022, 8(6): 123-134.

Citation Format: XIAO Y Q, GUO Y F, WANG Y W. Metrics for code obfuscation based on symbolic execution and N -scope complexity[J]. Chinese Journal of Network and Information Security, 2022, 8(6): 123-134.

resilience can refer to the efforts of generating input test set for executing all program paths. However, some adversarial techniques could reduce the symbol execution efficiency significantly based on the nested structure of the program and increase the error of the resilience reference. To solve the above problems, a metrics for code obfuscation was proposed based on symbolic execution and N-scope complexity. The obfuscation resilience was defined with symbolic execution time and obfuscation potency was defined based on the proposed N-scope complexity for better robustness in measuring the resilience of multi-nested structure programs. Furthermore, the correlation analysis of obfuscation effect was proposed and the effect was quantified by symbolic execution and control flow diagram extraction of programs. Over 4000 obfuscated programs from 3 open-sourced assemblies were evaluated with proposed metrics in the experiment, which indicated the generalization performance and practicality of the metrics. And an example of this metrics application was presented in a simulated obfuscation scenario which provided references of obfuscation technology metrics and obfuscation configuration for obfuscation users.

Keywords: code obfuscation, obfuscation metrics, symbolic execution, *N*-scope

0 引言

代码混淆是以对抗软件面临的逆向工程和篡改攻击等威胁为目的,对程序进行等价语义转换的安全技术。近年来,软件技术的发展与软件市场需求的扩大带来了巨大的经济效益,攻击者和安全人员随之进行相应的安全博弈和技术更新,代码混淆技术作为软件安全技术受到了关注。具体来说,以软件篡改和逆向攻击为代表的 MATE (man-at-the-end)^[1]攻击建立在软件和主机的使用权限之上,其多样性的攻击形式难以被预测,使传统的防御形式难以招架,而代码混淆作为内生安全技术,在不改变软件功能的情况下更改软件的内容与结构,以增加 MATE 攻击的难度和成本,受到安全攻守两方的重视并广泛应用。

代码混淆技术发展了近 30 年,研究人员开发了包括控制流混淆^[2]、关联混淆^[4]、数据混淆^[5]、布局混淆^[7]等方法在内的众多混淆技术,以 Tigress、LLVM-obfuscator、VMProtect 等为代表的各类混淆集成工具,为具有软件安全需求的用户提供了多样化选择。值得注意的是,代码混淆本质上并非绝对安全,而是混淆者与反混淆者基于资源的技术博弈,研究表明拥有充足资源的攻击者在足够长时间内可以攻破绝大多数混淆^[8]。尽管代码混淆的各类技术趋于成熟,除对混淆技术进行逻辑说明^[9]外,目前对混淆效果的相关度量方法与研究仍相对较少,缺乏实用性混淆度量方法,使混淆使用者无法较准确地评价混淆效果^[10],造成为安全“贪得无厌式”叠加大量混淆方法或为保

效率“投鼠忌器式”少加或不加混淆的窘境^[11]。从目的而言,不论逆向工程、篡改攻击或复制攻击,都需要对程序完整或部分逻辑进行分析,即对程序“理解”。最早由 Collberg 等^[12]定义的 3 个经典度量指标中:Potency 强度指基于人类理解能力的程序混淆复杂度,Resilience 弹性指基于机器反混淆技术的混淆抵抗力,Cost 开销指混淆带来的额外资源开销。现有的混淆效果度量研究主要针对混淆强度,而针对混淆弹性的度量相对较少,各种度量研究大致可分为以下 4 类。

1) 其他度量指标的迁移:研究人员常基于经典软件度量指标与混淆强度的概念相似或出于经验使用信息论度量指标定义混淆强度指标。如 Anckaert 等^[13]提出了一种迁移一系列软件复杂度指标,建立评估二进制程序指令、控制流、数据流和数据在内的复合度量方法,通过每个对象的对应指标综合评价混淆强度;Kirk 等^[14]提出基于信息论的混淆度量方式,将 Kolmogorov 复杂度迁移应用到二进制文件上进行混淆效果计算;Mohsen 等^[15]则进一步用约束的 Kolmogorov 复杂度确定代码混淆的规律性并给出代码混淆的相应规范化定义,进而度量无自动化工具影响下的混淆强度。

2) 基于控制流图的度量:控制流混淆在一些度量指标中取得较高的混淆强度、弹性得分^[16],而基于控制流图的度量方法可以较准确地描绘程序逻辑,表现出与程序语义的强相关性。Tsai 等^[18]将控制流混淆分解为一系列原子操作符并对众多

控制流混淆算法进行形式化, 提出了基于控制流图的距离度量方案 MGE 并结合基于嵌套级别的混淆强度度量 *N-Scope* 作为混淆效果的度量方案; Gao 等^[19]提出一种结合符号执行与定理证明器的图同构技术对程序控制流进行分析, 从而对比二进制文件的语义差异; Luo 等^[20]通过构建符号执行与定理证明器检测二进制基本块的等价语义, 以等价基本块为基本元素计算混淆程序与原程序线性独立路径的最长公共子序列, 并将其比例作为路径得分, 从而计算函数与程序的相似性。

3) 基于抽象语义的度量: Pucella 等^[21]通过对混淆防御能力及其面临攻击进行精确定义, 给出以多样性为防御前提的攻击性质描述, 通过类比展示了类型系统与混淆器的良好近似能力; Dalla 等^[22]推导了一个基于抽象解释的格点隐藏性质进行混淆效能度量的理论模型, 通过抽象指标比较攻击者, 从而进一步通过对比抽象解释的隐藏属性对代码混淆的强度进行测量, 同时对代码混淆度量模型进行了一般性推广, 即任意程序转换都可能作为潜在的代码混淆器使用。

4) 基于具体攻击的度量: Viticchi'e 等^[23]对代码混淆应用的真实攻击进行对比实验, 通过比较攻击混淆和原程序时间、攻击成功效率对代码混淆弹性进行度量; Ceccato 等^[24]模拟真实逆向场景, 实验中被测试人员理解经过混淆后被反编译与未经过混淆反编译的两种 Java 代码, 反映出代码混淆对抗逆向的有效程度以及缓解基于系统特性攻击的价值; Banescu 等^[25]通过对比混淆前后符号执行遍历程序所有路径执行时间, 来模拟自动化攻击与动态分析场景, 描述混淆程序对抗符号执行攻击的代码混淆弹性; Majumdar 等^[26]对防御切片技术的切片混淆方法提出了 5 个度量指标并证明切片混淆的有效性。

各种度量方式有其优点和缺点, 如其他度量指标的迁移展现出与混淆强度有较强的相关性, 然而对混淆弹性的刻画则可能存在较大偏差; 基于控制流图的度量通过结合基本块符号执行可以较好地刻画程序间静态相似性, 然而该方法实际操作过程中需要遍历程序不同路径对基本块一一比较, 方法消耗的资源巨大, 因此实际场景中不

适合直接用作混淆度量, 基于抽象语义的度量可将复杂混淆转换通过语义近似及抽象属性进行描述, 但该方法在使用时需根据实际情况进行语义精确匹配, 尚不能实际应用该方法; 基于具体攻击的度量对真实攻击场景进行了良好的复现, 然而人力的攻击度量无法大规模应用且度量结果较依赖于攻击人员的水平。

本文提出一种基于符号执行 *N-scope* 复杂度的代码混淆度量方法, 结合对程序的动态执行与静态基本块级控制流分析, 通过计算程序 *N-scope* 复杂度量混淆强度与符号执行评估混淆强度弹性, 将二者相互补充, 能够较好地描述混淆效果, 实现真实场景下考虑自动化分析的混淆效果度量方法。本文度量方法对总数超过 4 000 个混淆程序的程序集进行了充分评估, 表明该度量方法能较好地刻画混淆效果的同时拥有一定的泛化能力与实用价值。

1 度量方法的提出

1.1 基于符号执行的混淆弹性度量

符号执行技术是通过符号值代替真实值作为程序输入执行程序的技术, 当执行到分支语句, 则添加约束分别执行分支路径, 尽可能遍历所有路径之后利用约束求解器及可达性推理求解程序各路径输入与路径信息, 达到进一步分析程序与获取程序执行结果的目的。

在分析大规模混淆程序时, 对分析工具的时间成本与泛化分析能力有一定要求, 通过符号执行技术可以较高效地寻找系统漏洞^[27], 其动态分析的特性提供了优秀的泛化分析能力并使一部分代码混淆技术无效化, 因而符号执行被广泛应用于程序测试分析、二进制分析及漏洞挖掘等安全相关领域, 尤其对 MATE 攻击者而言, 代码混淆作为其面对的主要防御手段之一, 有重要的实用价值。例如, 一些先进的自动化攻击经常针对如控制流混淆和代码虚拟化等代码混淆防御手段实现对抗措施, 这些自动化攻击依赖于以程序有效输入为先决条件的动态分析, 因此符号执行通过生成测试集遍历完整代码路径的能力是这些分析技术的前提, 其执行程序并生成完整测试集的难度可以作为混淆弹性指标。

符号执行技术的目的在于完整的测试集生成, 而路径爆炸是其面临的主要限制之一。具体来说, 由于现实场景中软件路径数量庞大, 执行单个路径的开销, 导致有限时间内符号执行引擎仅能执行有限的路径, 从而降低符号执行引擎的测试集生成效率; 另外, 研究人员为对抗符号执行面向路径爆炸特点的混淆算法, 如 Wang 等^[28]基于数学难题提出一种将符号值与循环边界关联的程序混淆方法, 致使分析程序时符号执行引擎路径爆炸, Banescu 等^[25]提出基于软件多样化方法的输入相关等价分支构建混淆技术, 有效降低符号执行效率。基于以上研究, 对抗符号执行的混淆技术与程序中的循环结构存在较强的依赖或相关性, 混淆技术研究者常借助该结构增加符号执行预测分支的数量和执行次数而达到降低符号执行效率的目的。

1.2 *N*-scope 复杂度

N-scope 最早由 Harrison 和 Megal^[29]提出以程序的嵌套程度来计算连续程序的控制流图复杂度, 而嵌套程度与控制流图的大小可以较恰当描述程序控制流混淆方法造成的控制流图变化。具体来说, 嵌套复杂度可以计算程序的不同层次中的循环数量和大小, 在实际应用中研究人员给出了不同的定义方式^[30]: 采用以循环层数加权的循环计算方法作为嵌套值, 将程序混淆前后嵌套差值与原程序嵌套值的比例作为嵌套复杂度的得分。

$$C_{\text{nesting}} = \sum_{n=1}^N \text{level}_n \text{count}_n \quad (1)$$

这种计算方式未考虑控制流平坦化等方法降低程序嵌套度的控制流混淆方法, 在得分为负值时以零计算; 文献[18]将一个基本块的 *N*-scope 复杂度则定义为:

$$ns(\psi) = \frac{\sum_{B_i \in B} |R(\psi, B_i)|}{\sum_{B_i \in B} |R(\psi, B_i)| + |NC_\psi|} \quad (2)$$

其中, $|NC_\psi|$ 为程序 ψ 的节点数量, B 为 ψ 的分支集合, $|R(\psi, B_i)|$ 代表分支 B_i 的循环体节点数量。*N*-scope 复杂度能较好地描述程序嵌套程度而被研究人员用来描述程序控制流变化的混淆强

度, 然而作为静态评估方法缺少对程序动态执行的描述, 且孤立的控制流度量缺少对混淆程序的整体性评估。

1.3 代码混淆效果度量方法

本文提出基于两种关联性指标的代码混淆度量方法, 在综合符号执行混淆弹性度量和控制复杂度度量优势的基础上, 针对性地提出改进的基于程序多层嵌套结构的 *N*-scope 复杂度, 完善了现有符号执行分析在代码混淆弹性度量上的不足, 同时给出符号执行作为弹性度量指标基于相对执行时间的规范性定义。

通过对两种指标的关联性分析, 不仅能够清晰刻画程序混淆前后的混淆弹性变化与控制流复杂度变化趋势, 还可以解决单独使用符号执行弹性度量对多层嵌套程序的度量限制问题, 提高了对此类程序的度量准确性与度量效率。诸如 *N*-scope 复杂度的程序控制流复杂度度量方法仅能对程序的静态控制流进行评价分析, 而符号执行作为动态程序分析技术可以为整体混淆度量提供更全面的关联性分析参考, 平衡此类静态分析指标对控制流混淆技术的高敏感特征。

相对现有的单一代码混淆度量方法存在的缺点和限制, 组合具有关联性指标的度量方法的评估更全面, 如文献[18]将 *N*-scope 与混淆程序距离相结合, 混淆距离计算提高了度量方法对程序执行路径的变化敏感度, 改善了 *N*-scope 复杂度对某些混淆方法不敏感的特点。需要说明的是, 研究人员多遵循 Collberg 等^[12]定义代码混淆的指标方法, 即从混淆强度、混淆弹性、混淆开销 3 个方面进行混淆效果度量框架设计, 然而在实际场景中, 攻击者常将自动化代码分析工具和人工逆向分析结合, 现有的代码混淆度量方法研究则缺少对此场景的适配度量方法, 从实用的角度出发, 混淆度量方法应对特定混淆有较好的适配性和敏感性, 同时将度量开销限制在合理范围内。

基于以上考量, 本文提出一种结合符号执行和 *N*-scope 复杂度的实用性代码混淆效果度量方法: 给定原始程序 ψ , 经历混淆 Tr 后转换为混淆程序 ψ' , 二者控制流图分别为 G, G' , 则以 ξ 为参数的符号引擎 E 执行 ψ, ψ' 所有路径所需时间分别为 t_0, t_1 , 则基于以 ζ 为参数的符号执行引

擎 E 的程序混淆弹性定义为

$$R(\psi, \psi', E_\xi) = \frac{t_1 - t_0}{t_0} \quad (3)$$

复杂混淆程序在符号执行过程可能出现路径爆炸等问题，导致符号执行时间过长或超时，使得到的程序混淆弹性失去参考意义，因而设置最大符号执行时间 t_m 为时间约束，得到约束的程序混淆弹性为

$$R_{\text{norm}}(\psi, \psi', E_\xi) = \frac{\min(t_1, t_m) - t_0}{t_0} \quad (4)$$

本文考虑符号执行在执行程序嵌套逻辑的路径与循环体的多层嵌套结构，首先定义式(1)中基于控制流图 G 的程序嵌套值中嵌套层数为

$$\text{level}_i = 1, \text{ } G \text{ 中循环 } i \text{ 为最外层} \quad (5)$$

$$\text{level}_i = \max(\text{level}_j) + 1, \text{ } G \text{ 中循环 } i \text{ 嵌套于 } j \quad (6)$$

则基于程序 ψ 控制流图 G 的多层嵌套结构的 N -scope 复杂度定义如下。

$$\text{NS}(G) = \frac{\alpha C_{\text{nesting}}}{\alpha C_{\text{nesting}} + (1 - \alpha) | \text{NC}_G - \text{count}_{\text{nesting}} |} \quad (7)$$

NC_G 为控制流图 G 的节点数量，本度量方法中视基本块为一个节点， $\text{count}_{\text{nesting}}$ 为程序嵌套结构节点数量， α 为 0 到 1 之间实数，作为节点嵌套度的平衡系数，与 N -scope 复杂度为反比例关系。基于该 N -scope 复杂度的程序混淆强度定义为

$$P(G, G') = \text{NS}(G') - \text{NS}(G) \quad (8)$$

以 ψ 为原始程序的混淆程序 ψ' 的代码混淆效果，定义为包含程序混淆强度与混淆弹性的二元组：

$$E(\psi, \psi', G, G', E_\xi) = \langle P(G, G'), R(\psi, \psi', E_\xi) \rangle \quad (9)$$

本文中 $\text{NS}(\psi)$ 节点设置为程序基本块节点计算，计算 C_{nesting} 的伪代码如算法 1、算法 2 所示。

算法 1 DFS

输入 控制流图 G ，当前节点 v_i ，初始节点 v_0 ，当前循环 path ，所有循环路径 $\text{loop}_{\text{list}}$

- 1) FOR each v pointed by v_i and large than v_i
- 2) IF $v.\text{visited} = \text{FALSE}$
- 3) IF v points to v_0 in G

- 4) append v to path
- 5) append path to $\text{loop}_{\text{list}}$
- 6) $\text{path.pop}()$
- 7) $v.\text{visited} = \text{TRUE}$
- 8) append v to path
- 9) $\text{DFS}(G, v, v_0, \text{path}, \text{loop}_{\text{list}})$
- 10) $\text{path.pop}()$
- 11) $v.\text{visited} = \text{FALSE}$

算法 $C_Nesting(G)$ 计算了程序控制流图 G 中以基本块为节点的程序嵌套值 C_{nesting} ，该问题可抽象为求解有向图中所有环和计算环的嵌套度两个子问题。本文首先对程序节点按照名称排序并设置所有节点为未访问状态（算法 2 第 3~5 行），按此排序遍历图中节点，将自环加入循环路径后（算法 2 第 8~9 行），以当前节点为根节点对图 G 进行深度优先搜索，在遍历时只将排序大于根节点的邻接点加入 path 中（算法 1 第 1 行），保证了 DFS 主循环只搜索最小节点为根节点的环，同时访问状态的约束与设置，保证了环内部不相交（算法 1 第 2、7、11 行），在求解得到所有环后对环列表遍历计算每个环的被嵌套数，按此嵌套数对环长度加权求和得到程序的嵌套度。

算法 2 $C_Nesting$

输入 控制流图 G

输出 嵌套值 C_{nesting}

- 1) $\text{loop_list}, \text{path} = []$
- 2) $C_{\text{nesting}} = 0$
- 3) $\text{sort}(G.V)$
- 4) FOR v in $G.V$
- 5) $v.\text{visited} = \text{FALSE}$
- 6) FOR v in $G.V$:
- 7) append v to path
- 8) IF edge (v, v) in $G.E$
- 9) append path to $\text{loop}_{\text{list}}$
- 10) $\text{DFS}(G, v, v, \text{path}, \text{res})$
- 11) $\text{path.pop}()$
- 12) FOR loop in $\text{loop}_{\text{list}}$:
- 13) $\text{cnt} = 1$
- 14) FOR loop_t in $\text{loop}_{\text{list}}$:
- 15) IF $\text{loop} < \text{loop}_t$
- 16) $\text{cnt} = \text{cnt} + 1$

17) $C_{\text{nesting}} += \text{cnt-loop.length}()$

18) RETURN C_{nesting}

本文提出的度量方法从混淆弹性和混淆强度两个方面出发, 结合动态分析与静态分析对代码混淆效果综合度量, 一方面结合符号执行在真实逆向等攻击场景的常用性与代码混淆中程序控制流的关键性, 使本度量方法具有真实场景下的应用价值, 另一方面针对单一度量方法的局限性, 本度量方法使用符号执行技术反映程序路径执行难度, 提出适配符号执行的 $N\text{-scope}$ 复杂度算法, 在准确表达程序嵌套度的同时, 针对符号执行对抗技术依赖程序循环结构的特点, 提高符号执行度量的鲁棒性。

2 实验与讨论

2.1 度量方法实现

本文使用程序混淆工具 Tigress 学术版本对实验程序进行混淆, 采用 LLVM9 版本的 KLEE 符号执行引擎及与其适配的 SMT 约束求解器、klee-uclibc 等作为本文实验的符号执行环境, 使用 Jakstab 开源二进制反编译工具与 python 库 pydot 作为混淆文件的 $N\text{-Scope}$ 计算接口, 度量方法实现流程如图 1 所示。本文使用的实验环境为 64 位 Ubuntu18.04 操作系统, 4 核频率为 3.20 GHz 的 Intel Core i7-8700 CPU 及 4 GB 物理内存, Window7 操作系统, 12 核频率为 3.20 GHz 的 Intel Core i7-8700 CPU 及 16.0 GB 物理内存。

Tigress 为 C 语言的源码混淆工具, 具备防御静态与动态逆向、反虚拟化攻击等混淆功能, 本文实验主要使用程序控制流混淆相关功能, 使在混淆后的静态控制流或动态执行路径产生变化, 增加分析程序真实控制流的难度; KLEE 是基于 LLVM 编译框架开发的动态符号执行引擎, 对 LLVMbitcode 进行操作, 使用 uclibc 库支持对常规 C 文件的符号执行, 本实验的符号执行时间通过其内置工具 klee-stats 获得, 实验中 KLEE 的 `-sym-arg` 参数统一设置为 3; Jakstab 是一个基于抽象解释的反汇编和静态分析集成框架, 采用自定义指令译码, 可应用于多种操作系统, 基于 x86 处理器处理 32 位的 WindowsPE 或 linuxELF 可执行文件, 基于此特性, 本文使用 Windows 系统下

MingW64 支持的 GCC 将 C 语言源文件编译为 32 位 PE 文件并交由 Linux 下 Jakstab 处理 (如图 1 所示)。Jakstab 动态地将机器码转换为低级中间语言, 同时对增加的控制流进行分析得到数据流信息, 进而定位预测分支目标与代码位置, 本文实验使用该工具分析 32 位可执行文件得到其 .dot 格式的控制流图文件, 进而使用 python 的 pydot 模块对其进一步处理。

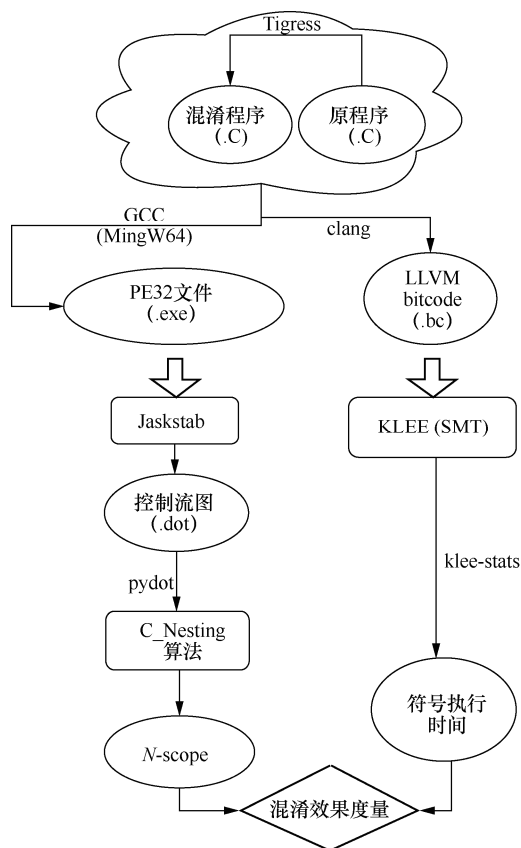


图 1 度量方法实现流程

Figure 1 Flow chart of measurement method implementation

本文使用该度量方法对两个公开程序集及其对应的混淆程序集共约 4 000 个程序进行了度量, 程序集 1 中程序为各种 if 分支和循环结构的简单组合, 程序集 2 为常见的基本算法, 程序集 3 为使用 Tigress 混淆工具自动生成的程序集。由于混淆程序代码量可能为原程序的几倍至几十倍, 受限于实验环境, 程序集 1、程序集 2 的原程序在 100 行以内, 程序集 3 的原程序介于 100 行与 200 行之间, 在防止程序集混淆版本程序过大导致本文实验环境中符号执行引擎运行超时的前提下,

保证了程序样本的多样性。

2.2 实验结果分析

实验 1 不同结构原程序符号执行

使用程序集 1 具有不同结构的未混淆程序进行符号执行，实验样本结构主要由 if 和 loop 两种结构进行叠加或嵌套组成（不同结构的程序样本如图 2 所示），同时为了保证符号执行的时间差异性，约 30% 的程序样本与输入无关，而按比例将约 70% 程序输入与循环判断条件及 if 分支判断条件结合，增加符号执行时间。

```
int main(int argc, char* argv[]) {
    int i;
    int sum = 0;

    for (i = 0; i < argv[1][0]; i++) {
        if (argv[1][0] > 127)
            sum += i;
        else
            sum *= i;
    }

    printf("Sum is: %d\n", sum);
    return 0;
}

int main(int argc, char* argv[]) {
    int i, j;
    int sum = 0;

    const char *str = argv[1];
    register const char *s, *t;

    for (i = 0; i < argv[1][0]; i++) {
        for (t = str; *t; ++t) {
            sum += *t;
        }
    }

    printf("Sum is: %d\n", sum);
    return 0;
}
```

图 2 不同结构的程序样本

Figure 2 Program samples with different structures

如表 1 所示，不同结构程序符号执行时间展现出明显差异，在无嵌套状态下 loop 结构程序的平均符号执行时间约为 if 结构程序的 2~5 倍，而在有嵌套的状态下 if 嵌套，loop 中嵌套 if 与 loop 嵌套 loop 展现出数量级的递增差异，表明 loop 结构及 loop 嵌套结构具有增加符号执行时间的显著优势。

当两种结构的分支判断语句与程序输入相关联时，if 结构程序的符号执行时间变化可忽略，而执行 loop 结构程序的时间则有较大增加，表明与输入相关联的 loop 结构相比单一 loop 结构具有更强的抗符号执行能力。需要注意的是，在 loop 嵌套层数增加时，不同的 loop 结构执行时间将展

现出更强的差异性，使标准差显著增大。

表 1 不同结构程序符号执行时间
Table 1 Symbolic execution time of different structure programs

单位：秒

程序符号	均值	标准差
if×1	0.03	0
if×2	0.04	0.007
if 嵌套 if	0.04	0.005
loop×1	0.06	0.05
loop×2	0.21	0.292
loop+if	0.068	0.021
loop 嵌套 if	0.52	0.50
loop 嵌套 loop	4.31	9.57
loop 输入无关	0.03	/
loop 输入有关	0.12	/

实验 2 程序集 1 混淆与效果度量

为验证本文混淆效果度量方法并探究不同混淆技术在本度量方法下的表现，本文使用 tigress 混淆工具对程序集 1 进行混淆，由于本文度量算法侧重于对程序的符号执行难度与控制流进行评估，同时为了保证混淆程序与 klee 引擎的兼容性，选取 5 种 Tigress 混淆技术（如表 2 所示）并对其进行排列组合共 24 种混淆方案，由于 tigress 混淆需指定函数名的特点，函数拆分仅单独使用或作为后置技术与其他技术组合使用。

表 2 Tigress 混淆技术介绍
Table 2 Introduction of Tigress obfuscation techniques

混淆技术	模块名称	描述
模糊谓词	AddO4/16	在控制流中添加模糊谓词，谓词数量为 4 或 16
函数拆分	Split	将函数拆分为小片段并进行外联
控制流平坦	Flatten	将顺序执行的程序结构转换为多分支循环结构
代码虚拟化	Virtualize	将函数转换为拥有唯一字节码的解释器进行执行
数学运算加密	Encode Arithmetic	将整数运算转换为相同语义下更复杂的表达式

实验中对程序集 1 进行混淆得到约 1 000 个混淆程序，将这些程序按照使用的混淆方法分类，ubuntu 实验环境下使用 klee 进行符号执行，执行

共计约 30 h, 设置最大执行时间为 10 min, 使用 klee 内置工具 klee-stats 统计符号执行时间并根据原程序执行时间对每种混淆方法下混淆程序集进行混淆弹性统计, 程序集 1 各混淆弹性统计如图 3 所示; 将混淆程序置于 Windows 环境下编译为 32 位 PE 可执行文件, 在 ubuntu 下使用 Jakstab 得到控制流(.dot)文件, 计算其 N -scope 复杂度, 各混淆方法程序集 $1N$ -scope 复杂度如图 4 所示。为进一步分析不同混淆方法对混淆弹性与混淆强度关联性的影响, 按序将同种混淆方法的弹性与 N -scope 复杂度进行逐一对比, 如图 5 所示。

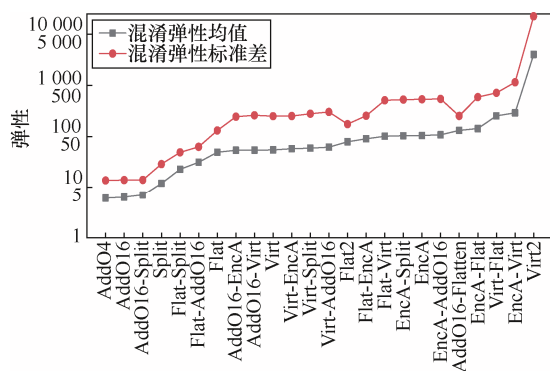


图 3 程序集 1 混淆弹性统计
Figure 3 Obfuscation resilience statistics of assembly 1

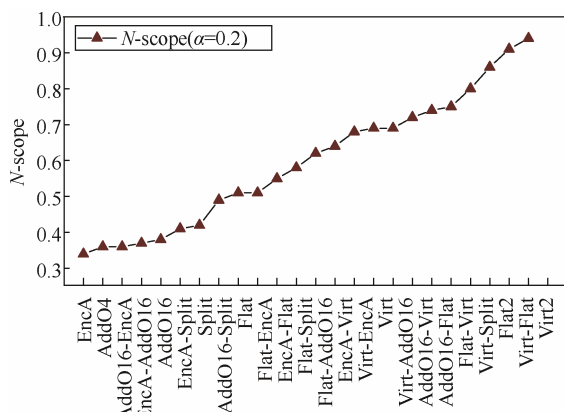


图 4 程序集 1 N -scope 复杂度统计
Figure 4 N -scope complexity statistics of assembly 1

由于程序集与混淆技术使程序样本中嵌套结构较多, 为得到更清晰的 N -scope 复杂度分布, 调整 α 参数将其设置为 0.2, 避免嵌套结构过多造成 N -scope 复杂度差异性过小。由于使用的混淆技术特性, 原程序的 N -scope 复杂度与混淆程序最小 N -scope 复杂度可视作相同。

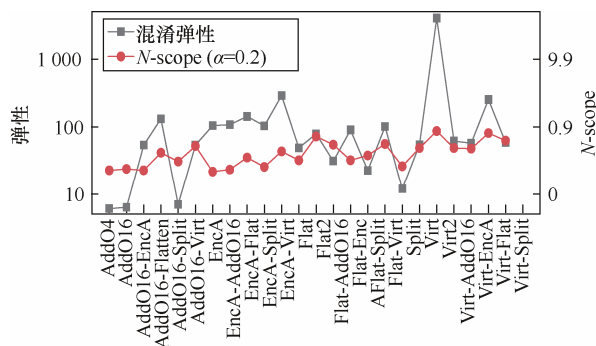


图 5 程序集 1 混淆弹性与混淆强度关联性
Figure 5 Assembly 1 obfuscation resilience and potency correlation

分析实验结果如下。

1) 两次代码虚拟化的混淆弹性最高且远高于其他混淆组合, 其 N -scope 得分亦最高, 代码虚拟化后函数内代码以字节码的形式由函数中解释器执行, 相当于程序内部代码与解释器构成了多层嵌套循环结构, 而两次虚拟化则使得循环执行路径指数化增加, 导致符号执行引擎时间的爆炸式增加。值得注意的是, 虽然代码虚拟化相关混淆 N -scope 复杂度明显高于其他混淆, 然而其对混淆弹性的增益可能低于其他混淆 (如 AddO16-EncA), 表明增加符号执行的路径求解难度与增加程序嵌套度都可能对混淆弹性产生较大增益。

2) 应用单次混淆时, 数学运算加密对混淆弹性的增益较为明显, 然而其对程序控制流影响较小, 因而对 N -scope 复杂度增益可忽略, 而模糊谓词技术虽然绝对意义上增加了程序分支数量, 相比于其他控制流混淆技术对 N -scope 的增益却较为有限。

3) 应用混淆组合时, 混淆组合的不同顺序可能对混淆弹性和混淆强度产生不同影响。如 Flat-AddO16 与 AddO16-Flat 相比混淆弹性与 N -Scope 复杂度均较低, 而 Virt-AddO16 和 AddO16-Virt 则与单混淆技术具有相似的混淆效果。造成上述差异的原因可能是本程序集在程序分支数量提升时控制流平坦化, 使增加代码嵌套的能力更强, 而代码虚拟化解释执行则对该变化不敏感。

4) 使用函数拆分作为后置混淆技术时, 程序 N -Scope 复杂度增加明显, 而程序混淆弹性则变化不一致。如与代码虚拟化组合的混淆程序弹性

有所提升, 而控制流平坦化的混淆弹性则下降, 说明函数拆分可能降低了控制流平坦混淆程序的路径求解难度。

5) 随着程序混淆弹性增加, 混淆弹性的标准差呈现上升趋势, 表明具有多样性结构的程序样本在混淆弹性增加时其弹性差异随之变大, 即混淆方法提升的弹性在一定程度上依赖程序样本的原有结构。

6) 按序将同种混淆方法的混淆弹性与混淆强度进行关联性分析可知, 混淆强度与混淆弹性的相对变化存在多种情形, 这些变化与混淆程序的符号执行求解路径数量、路径求解难度强相关, 通过将符号执行时间与 N -scope 复杂度进行关联性分析归纳如下: ① 应用代码虚拟化与控制流平坦的混淆程序中, 混淆弹性与混淆强度都有明显提升; ② 应用数学加密的混淆程序中, 混淆弹性增加而混淆强度未见明显变化; ③ 应用模糊谓词为后置混淆技术, 混淆弹性无明显增加而混淆强度增加明显; ④ 应用函数拆分为后置混淆技术时, 混淆弹性表现与前置技术相关, 而混淆强度有所提升; ⑤ 应用数学加密为后置混淆技术时混淆强度普遍变化较小, 而部分混淆程序弹性增加明显。

实验3 程序集2混淆与效果度量

为检验本文混淆度量方法的泛化能力, 实验3中使用经典算法程序集对其进行混淆与泛化度量, 该数据集包含排序、字符串比较、计算公因数等40个经典算法, 相比程序集1程序结构更加多样化并有更高的实用价值。本实验其余实验配置与实验2相同, 经过混淆后得到其混淆弹性统计如图6所示。 N -scope复杂度统计如图7所示。从实验结果看, 对两个程序集的混淆效果整体上差异较小, 而程序集2的混淆弹性总体低于程序集1, 其中几种混淆技术的混淆效果有所不同。

1) 与程序集1相比, 单次使用数学运算加密的混淆弹性较低, 而单次使用控制流平坦、代码虚拟化等提升 N -Scope 复杂度较高的混淆时弹性提升较大, 造成该现象的原因可能是本程序集中原有数学运算复杂性较强而程序嵌套度较低, 使与后者相关的混淆技术在混淆弹性上表现较好。

2) 与程序集1相比, 模糊谓词对程序的混淆弹性提升更明显, 可能程序集2程序结构较复杂, 导致符号执行路径求解难度较大, 因而在增加绝对分支数量时提升更高的混淆强度; 然而由于程序本身嵌套结构较少及 N -Scope 参数限制, 模糊谓词对 N -scope 的增益不明显。

3) 与程序集1中 Flat 与 AddO16 不同顺序的技术组合产生显著安全增益差异不同, 程序集2中两种组合的混淆弹性差异不十分明显, 前者 N -Scope 甚至略高于后者, 一方面是程序集差异导致的混淆技术使用效果差异, 另一方面是在符号执行与控制流解析过程中可能存在一定误差, 导致混淆效果的度量偏差。

4) 按序将程序集2程序使用同种混淆方法的混淆弹性与混淆强度进行关联性分析可知(如图8所示), 相同混淆技术对混淆程序的弹性与强度增益与程序集1的混淆增益方向基本一致, 而在增益程度上表现出一定差异。

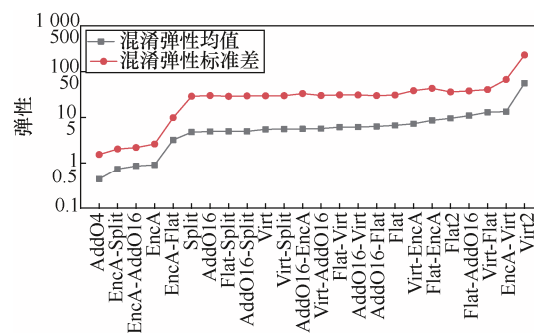


图6 程序集2混淆弹性统计
Figure 6 Obfuscation resilience statistics of assembly 2

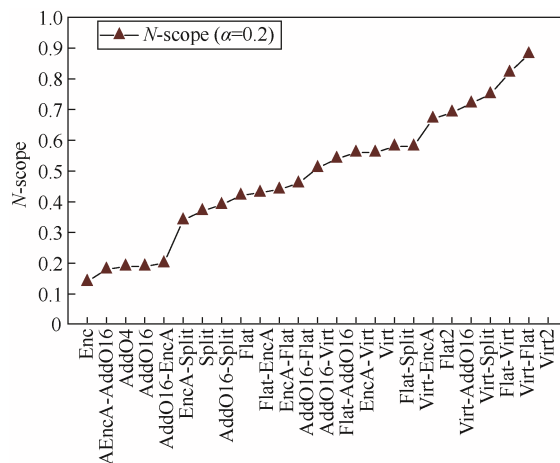


图7 程序集2 N -scope 复杂度统计
Figure 7 N -scope complexity statistics of assembly 2

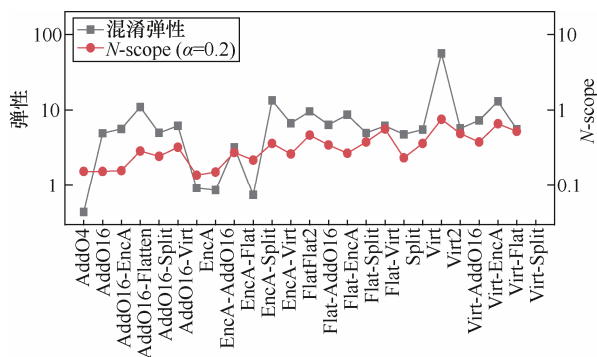


图8 程序集2混淆弹性与混淆强度关联性
Figure 8 Assembly 2 obfuscation resilience and potency correlation

实验4 程序集3混淆与效果度量

为模拟本文代码混淆方法在实际场景中的应用,实验4中使用Tigress混淆工具Random Function模块随机生成的100个具有不同结构的差异化程序集。这些差异化包括但不限于:控制流嵌套度、控制流语法树节点数、基本块数量、程序、函数、基本块大小、程序输入输出类型、全局变量、局部变量、动态变量等各类型变量,分支条件与程序输出是否依赖等。本实验从Tigress超过40种混淆技术中随机选取19组混淆组合对程序进行混淆,每组5种基础混淆技术,Tigress混淆技术补充介绍如表3所示。

表3 Tigress混淆技术补充介绍
Table 3 Supplement to introduce Tigress obfuscation techniques

混淆技术	模块名称	描述
函数复制	Copy	复制一个函数
冗余清扫	CleanUp	删除未使用变量,非必要函数名等
谓词更新	UpdateOpagues	添加更新模糊谓词的代码
虚假控制流	ImplicitFlow	在基本块间构造冗余控制流
反别名分析	AntiAliasAnalysis	通过用间接函数调用替换直接函数调用来转换代码
反污点分析	AntiTaintAnalysis	结合虚假控制流对抗动态污点分析工具
熵增益	Generate Entropy	增加程序运行时的随机性
伪信息泄露	Leak	插入伪装成泄露秘密函数的代码

使用本文代码混淆效果度量方法对混淆程序进行混淆,由于本实验中使用混淆技术数量增加,故将N-scope复杂度参数设置为0.01,设置最大

符号执行时间 t_m 为4 min,其余设置与实验2相同。程序集3混淆效果分析如表4所示。不同结构、使用不同混淆组合的混淆程序在本文度量方法上表现出较大差异,表明本文度量方法的良好泛化能力;符号执行时间标准差显示程序集在同一混淆技术组合后的程序弹性增益较小,出现此现象一方面是多种混淆技术的使用使各程序的混淆弹性普遍较高,另一方面是程序集的实际代码量差异不大,结构差异性在多次混淆后的弹性对比不明显。

在实际应用场景中,通过分析不同混淆组合的混淆弹性与混淆强度指标,可以帮助混淆使用者更有效地对程序进行混淆。本实验中推荐混淆强度与混淆弹性都较强的混淆技术组合,通常选择较高的N-scope复杂度与较长符号执行时间中位数,推荐混淆技术组合为序号1,3,6,7,8的混淆组合。专业安全人员根据各基础混淆技术特性与安全需求可进一步进行选择。

3 结束语

现有混淆度量方法缺少对混淆弹性度量,复杂控制流图度量算法实现困难,抽象语义度量实用性不高,使得混淆技术使用者难以对混淆效果进行客观评价并及时调整混淆策略,造成混淆技术过度累加导致的资源浪费。为解决上述问题,本文利用符号执行技术在真实代码逆向、漏洞挖掘等攻击场景中的常用性,结合N-scope复杂度度量方法,对混淆程序进行动态的符号执行路径遍历与静态的程序基本块拓扑嵌套度分析,从混淆弹性和混淆强度两个方向进行综合度量,对超过4 000个混淆程序进行混淆度量的实验结果表明,使用不同混淆方法的混淆程序在本代码混淆度量方法中展现出差异化的混淆弹性与强度,同时在不同程序集上本方法表现出良好的泛化能力,可以为安全人员对混淆技术的评价以及混淆技术的组合提供合理参考。

本文设计的基于符号执行和N-scope复杂度的混淆度量方法目前仅在有限的程序集与特定混淆技术上展开验证,且本度量方法在使用上要求使用者具备一定程度的相关知识,如设置参数 α 并分析混淆弹性和N-scope复杂度,未来希望在

表 4 程序集 3 混淆效果分析
Table 4 Assembly3 obfuscation effect analysis

混淆技术	N -scope 复杂度	符号执行时间/min		
		均值	中位数	标准差
1. AddO16-Copy-Flatten-ImpF-Leak	0.57	3.95	2.76	3.40
2. EncA-Virt-ImpF-Leak-AntiTA-CleanUP	0.46	3.60	2.37	3.18
3. EncA-UpdateO-Virt-GenrEntr-EncL	0.88	3.33	1.71	3.05
4. EncA-UpdateO-GenrEntr-AntiAA-EncL	0.21	3.33	2.88	2.62
5. EncA-ImpF-AntiTA-EncL-CleanUP	0.21	3.27	2.99	2.52
6. EncA-UpdateO-Virt-AntiAA-Split	0.86	3.12	1.59	2.84
7. EncA-Virt-GenrEntr-Leak-AntiAA	0.85	2.99	1.51	2.85
8. Flat-GenrEntr-ImpF-EncL-CleanUP	0.51	2.96	1.58	2.55
9. Flat-UpdateO-ImpF-AntiTA-Split-EncL	0.46	2.80	1.55	2.51
10. ImpF-Flat-AntiTA-Split-CleanUp	0.43	2.59	1.26	2.45
11. Virt-GenrEntr-ImpF-EncL-CleanUp	0.83	2.56	1.22	2.28
12. Copy-UpdateO-Virt-Split-CleanUP	0.82	2.54	1.17	2.39
13. AddO16-GenrEntr-ImpF-EncL-CleanUP	0.22	2.48	1.26	2.21
14. EncA-ImpF-AntiAA-Split-EncL	0.20	2.47	1.34	2.10
15. AddO16-GenrEntr-Leak-EncL-CleanUP	0.22	2.47	1.40	2.07
16. AddO16-UpdateO-Virt-GenrEntr-ImpF	0.85	2.43	1.13	2.19
17. Flat-UpdateO-Virt-ImpF-Leak-AntiTA	0.84	2.37	1.08	2.21
18. UpdateO-GenrEntr-ImpF-AntiTA-Split	0.21	2.05	0.92	1.91
19. Copy-Flat-UpdateO-EncL-CleanUp	0.52	0.52	0.16	0.50

更多样化的程序集中和混淆技术上探究本度量方法的普适性与准确性并加以改进，同时基于机器学习等方法对程序集的混淆度量结果进行更全面的分析并制定推荐混淆策略。

参考文献：

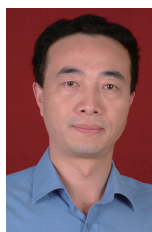
- [1] AKHUNZADA A, SOOKHAK M, ANUAR N, et al. Man-at-the-end attacks: analysis, taxonomy, human aspects, motivation and future directions[J]. Journal of Network and Computer Applications, 2015, 48(6):44-57.
- [2] VAN DEN BROECK J, COPPENS B, DE SUTTER B. Flexible software protection[J]. Computers & Security, 2022, 10(2): 6-36.
- [3] 耿普, 祝跃飞. 一种基于分支条件混淆的代码加密技术[J]. 计算机研究与发展, 2019, 56(10): 2183-2192.
GENG P, ZHU Y F. A code encrypt technique based on branch condition obfuscation[J]. Journal of Computer Research and Development, 2019, 56(10): 2183-2192.
- [4] 陈喆, 王志, 王晓初, 等. 基于代码移动的二进制程序控制流混淆方法[J]. 计算机研究与发展, 2015, 52(8): 1902-1909.
CHEN Z, WANG Z, WANG XC, et al. Using code mobility to obfuscate control flow in binary codes[J]. Journal of Computer Research and Development, 2015, 52(8): 1902-1909.
- [5] AHIRE P, ABRAHAM J. Mechanisms for source code obfuscation in C: novel techniques and implementation[C]//2020 International Conference on Emerging Smart Computing and Informatics. 2020.
- [6] PIZZOLOTTO D, FELLIN R, CECCATO M. OBLIVE: seamless code obfuscation for Java programs and Android Apps[C]//2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. 2019.
- [7] VAN DEN BROECK J, COPPENS B, DE SUTTER B. Obfuscated integration of software protections[J]. International Journal of Information Security, 2021, 20(1): 73-101.
- [8] CECCATO M, ANDREA C, PAOLO F, et al. A large study on the effect of code obfuscation on the quality of Java code[J]. Empirical Software Engineering, 2015, 20(6): 1486-1524.
- [9] LARSEN P, HOMESCU A, BRUNTHALER S, et al. SoK: automated software diversity[C]//Symposium on Security and Privacy. 2014.
- [10] XU H, ZHOU Y, MING J, et al. Layered obfuscation: a taxonomy of software obfuscation techniques for layered security[J]. Cybersecurity, 2020, 3(1): 1-18.
- [11] PENG Y, CHEN Y, SHEN B. An adaptive approach to recommending obfuscation rules for java bytecode obfuscators[C]//2019 IEEE 43rd Annual Computer Software and Applications Conference. 2019.

- [12] COLLBERG C, THOMBORSON C, LOW D. A taxonomy of obfuscating transformations[R]. 1997.
- [13] ANCKAERT B, MADOU M, DE SUTTER B, et al. Program obfuscation: quantitative approach[C]//ACM Workshop on Quality of Protection, 2007.
- [14] KIRK SR, JENKINS S. Information theory-based software metrics and obfuscation[J]. Journal of Systems and Software, 2004, 72(2):179-186.
- [15] MOHSEN R, PINTO AM. Evaluating obfuscation security: A quantitative approach[C]//International Symposium on Foundations and Practice of Security. 2016.
- [16] MOHSEN R. Quantitative measures for code obfuscation security[D]. London: Imperial College London, 2016.
- [17] RODRÍGUEZ-VELIZ M, NUÑEZ-MUSA Y, LIMA R S. Call graph obfuscation and diversification: an approach[J]. IET Inf Secur, 2020, 14(2): 241-252.
- [18] TSAI HY, HUANG YL, WAGNER D. A graph approach to quantitative analysis of control-flow obfuscating transformations[J]. IEEE Transactions on Information Forensics and Security, 2009, 4(2): 257-267.
- [19] GAO D, REITER MK, SONG D. BinHunt: automatically finding semantic differences in binary programs[C]//International Conference on Information and Communications Security. 2008.
- [20] LUO L, MING J, WU D, et al. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection[C]//ACM Sigsoft International Symposium on Foundations of Software Engineering. 2014.
- [21] PUCELLA R, SCHNEIDER FB. Independence from obfuscation: a semantic framework for diversity[J]. Computers & Security, 2010, 18(5):701-749.
- [22] DALLA M, GIACOBBAZZI R. Semantic-based code obfuscation by abstract interpretation[C]//International Colloquium on Automata, Languages, and Programming. 2005.
- [23] VITICCHIE A, REGANO L, TORCHIANO M, et al. Assessment of source code obfuscation techniques[C]//International Working Conference on Source Code Analysis and Manipulation. 2016.
- [24] CECCATO M, DI PENTA M, NAGRA J, et al. The effectiveness of source code obfuscation: an experimental assessment[C]//International Conference on Program Comprehension. 2009.
- [25] BANESCU S, COLLBERG C, GANESH V, et al. Code obfuscation against symbolic execution attacks[C]//Annual Conference on Computer Security Applications. 2016.
- [26] MAJUMDAR A, DRAPE S, THOMBORSON C. Metrics-based evaluation of slicing obfuscations[C]// International Symposium on Information Assurance and Security. 2007.
- [27] MOLNAR D, LI XC, WAGNER DA. Dynamic test generation to find integer bugs in x86 binary linux programs[C]//USENIX Security Symposium. 2009.
- [28] WANG Z, MING J, JIA C, GAO D. Linear obfuscation to combat symbolic execution[C]//European Symposium on Research in Computer Security. 2011.
- [29] ZUSE H. Software complexity: measures and methods[R]. Berlin, Walter de Gruyter GmbH & Co KG, 1991.
- [30] KARNICK M, MACBRIDE J, MCGINNIS S, et al. A qualitative analysis of Java obfuscation[C]//IASTED International Conference on Software Engineering and Applications. 2006.

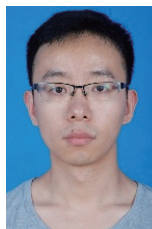
[作者简介]



肖玉强 (1997-), 男, 吉林辽源人, 信息工程大学硕士生, 主要研究方向为网络空间安全、代码混淆、机器学习。



郭云飞 (1963-), 男, 河南郑州人, 信息工程大学教授、博士生导师, 主要研究方向为云安全、电信网络安全、网络安全。



王亚文 (1990-), 男, 河南郑州人, 信息工程大学讲师, 主要研究方向为云计算、入侵容忍、网络安全。