软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(4):730−748 [doi: 10.3724/SP.J.1001.2013.04234]

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

软件迷惑变换的鲁棒性量化评价*

付剑晶[1,2], 王 珂[1]

[1](浙江大学 遥感与信息技术应用研究所,浙江 杭州 310058)

[2](浙江传媒学院 新媒体学院,浙江 杭州 310018)

通讯作者: 付剑晶, E-mail: fjjmsn@163.com, http://www.zju.edu.cn

摘 要: 为了方便程序员比较多种迷惑变换方案的优劣,提出了一种量化评价迷惑变换鲁棒性的方法.该方法从软件复杂度变化与代码功能模糊性两个相对独立的层面来刻画迷惑变换导致的鲁棒性.首先,从系统的复杂性与信息的多样性角度建立软件系统复杂度模型,模型包含软件结构、信息流、分支、循环以及元素的嵌套层次,力求从复杂性层面更准确地反映变换对软件的保护;之后,为量化描述迷惑变换的功能模糊度,根据专家指标评分法建立单种迷惑变换模糊度模型,在此基础上建立多种迷惑变换复合模糊度模型;然后,阐述了如何联合所提出的模型实现对单种迷惑变换技术有效性判定与多种迷惑方案的选优,也给出了模型的实现算法及一些示例;最后,通过实例仿真详细展示了模型的工作过程.

关键词: 代码迷惑;迷惑变换;反向工程;软件保护;复杂性度量

中图法分类号: TP311 **文献标识码**: A

中文引用格式: 付剑晶,王珂.软件迷惑变换的鲁棒性量化评价.软件学报,2013,24(4):730−748. http://www.jos.org.cn/1000-9825/4234.htm

英文引用格式: Fu JJ, Wang K. Quantitative evaluation on the robustness for software obfuscating transformation. Ruanjian Xuebao/Journal of Software, 2013,24(4):730−748 (in Chinese). http://www.jos.org.cn/1000-9825/4234.htm

Quantitative Evaluation on the Robustness for Software Obfuscating Transformation

FU Jian-Jing[1,2], WANG Ke[1]

[1](Institute of Remote Sensing and Information System, Zhejiang University, Hangzhou 310058, China)

[2](College of New Media, Zhejiang University of Media and Communications, Hangzhou 310018, China)

Corresponding author: FU Jian-Jing, E-mail: fjjmsn@163.com, http://www.zju.edu.cn

Abstract: A method to quantitatively evaluate the robustness of the obfuscated software is proposed in order for programmers to make a choice in different obfuscating schemes. This method aims to measure software robustness during the obfuscating transform from software complexity change and the increase of code functional obscurity, which are relatively independent, each other. First, a system complexity model is constructed from the perspective of system complexity and the diversity of software information. The model contain such elements as software structure, information flow, branch, and loop at nested level, and tries to reflect the robustness from the obfuscating transform on the complexity level. Second, to quantitatively measure the functional obscurity for the obfuscating transform, the experts index score constructs a model for a single transform is constructed by the experts index score. On this basis, transform obscurity composition models for multiple transforms are proposed. Next, the paper describes how applying these two kinds of models can be used to evaluate whether an obfuscating technique is effective, and can sort the given obfuscating scheme set to choose the best one. Also, some examples and corresponding model algorithms are given. Finally, an instance simulation demonstrates in detail the work process for the proposed models.

Key words: code obfuscation; obfuscating transformation; reverse engineering; software protection; complexity metrics

* 基金项目: 国家自然科学基金(30800703); 国家高技术研究发展计划(863)(2006AA10Z204)
收稿时间: 2011-04-20; 定稿时间: 2012-03-30

软件迷惑变换是在保持软件功能的同时,改变代码的外观和结构的变换,其目的是降低代码的可理解性,使得静态分析和反向工程很困难.不需要硬件的支持以及平台的独立性,极大地提高了该技术应用的灵活性;而且实践证明,好的软件迷惑技术确实能使得攻击者在限定的环境下理解软件非常困难,甚至不可能.目前已提出很多种迷惑变换技术,它们通常被分成 4 类:词法变换、布局变换、数据变换、控制流变换[1].不同的变换技术在不同的位置或时刻,如源代码级、中间代码级、可执行代码级和编译时,起着不同程度的保护效果.

直至目前,研究人员普遍关注的是如何构建新的迷惑技术来保护代码,关于如何量化评价迷惑变换鲁棒性的研究却非常少.面对众多的变换方法,程序员该如何选择,显得非常重要.对程序代码,我们既可以对不同的代码段单独应用不同的变换方法,也可以对多个代码段复合应用多种变换方法.如果对变换的鲁棒性评价完全出于定性方面,不同的人评价结果可能差别很大;而且面对多种变换技术以及变换技术复合应用,程序员更是难以比较不同变换方案的优劣.所以,建立一种统一、量化、快速的评价模型很有现实意义.

由于文献[2]最早提出用力度(potency)来刻画理解变换后代码的难度,用弹力(resilience)来刻画变换阻止反编译工具的能力,因此,研究人员普遍使用力度和(或者)弹力作为迷惑变换有效性的评价标准.虽然阻止反编译工具的自动进行能够降低黑客的工作效率,但却难以阻止黑客应用动态分析技术来解密代码.相比之下,力度在这个评价标准中显得尤其重要.

一般来说,软件的复杂度越高,代码的理解难度越大.文献[2]就是直接借用传统的软件复杂制,如文献[3−7],来量化度量力度的,而且这种方式一直被后来的一些文献所引用,但这会带来以下问题:

(1) 传统的度量模型主要是从软件质量与成本的角度来考虑的,旨在降低软件复杂性,而迷惑变换的目的是增加复杂性防止被攻击,两种模型建立的出发点差异较大.因此,用基于传统的复杂制建立的力度来反映变换鲁棒性往往存在偏差;

(2) 传统的度量方法主要是用于全局性评价,而变换保护主要是从局部角度来度量.因而用全局度量代替局部度量的结果来表示变换的鲁棒性往往是不准确的,这好比链条的强度是由最薄弱的环节所决定一样;

(3) 基于不同的复杂制计算出来的力度值不同,在应用保护技术时,不便于对不同变换方案的比较和选择;

(4) 由于不同的复杂制是出于不同的侧面来构建的,单独应用其中的一种复杂制来表示有效性可能会显得构建基础信息有所缺乏.

为度量变换的鲁棒性,上述存在的问题启发我们在传统复杂制的基础上,可以有针对性地构建一种新的复制性度量模型.这种模型在本文定义为系统复杂度模型(system complexity model,简称 SCM),构建 SCM 的目的是为了从复杂性角度更准确地度量变换的鲁棒性.不妨把 SCM 看作一个函数,即 $V=SCM(S)$,它接受一个软件 S 作为输入,输出 S 的相应复杂性度量值 V.

值得注意的是,仅有 SCM 是不够的.让我们来看看这样一种情形:对一段给定的软件代码 C,分别应用两种不同的迷惑变换技术 t_1,t_2,变换后的代码分别表示为 $t_1(C)$与 $t_2(C)$.从理论上讲,存在以下两种情形:

(1) 当 $SCM(t_1(C))$与 $SCM(t_2(C))$相等或很接近时,变换后代码 $t_1(C)$与 $t_2(C)$的理解难度可能相差很大.如,当 t_1 为词法变换、t_2 为数据变换时可能出现此类情形;

(2) 当 $SCM(t_1(C))$值与 $SCM(t_2(C))$值相差较大时,理解 $t_1(C)$与 $t_2(C)$的难度可能相当.如,t_1,t_2 分别为不同的数据变换时可能出现这种情况.

这两个问题主要是由变换 t_1,t_2 引起的,因为迷惑变换不但改变软件系统的基本信息,即改变系统复杂度,重要的是它使系统所完成的功能(语义信息)更模糊;而且一般来说,不同的变换对二者的改变程度各不一样.此外,代码功能的模糊性不是由 SCM 所能完全反映的.在此,针对文献[2]中所提出的力度,我们把模型 SCM 所不能刻画的那部分信息,即代码功能的模糊性,其对应的模型在本文称作迷惑变换模糊度模型(transform obscurity model,简称 TOM).与 SCM 类似,在本文可把 TOM 看作一个函数,即 $O=TOM(Ts(S))$,它接受施加在软件 S 上的一个变换集 Ts 作为输入,输出功能模糊度值 O.

本文联合 SCM 与 TOM 一起刻画文献[2]中的力度,并以此来度量迷惑软件的鲁棒性.主要贡献如下:

(1) 提出了一种由 SCM 与 TOM 构成的度量方法,基于此方法,既能评价一种变换技术的有效性,也能对多种变换方案的鲁棒性进行排序;

(2) 所提出的 SCM 包含丰富的软件元素信息,能够较好地从复杂性层面来反映迷惑软件的鲁棒性.建立的 TOM 能够适应多种迷惑变换复杂的应用情形,从人的理解力角度为变换提升软件的鲁棒性提供评价信息;

(3) 用平均局部基本代码块模型度量值代替以往的软件总量复杂度,用以评价迷惑软件鲁棒性的度量值;

(4) 为计算 SCM 与 TOM,分别给出了相应的算法.

本文第 1 节概述相关研究.第 2 节和第 3 节分别阐述所提出的 SCM 与 TOM 模型,并举例帮助理解(参见附录).第 4 节介绍如何联合 SCM 与 TOM 来评价单种变换的有效性,以及对多种迷惑变换方案进行选优.实例仿真放在文本的第 5 节,以一个具体的实例详细展示如何应用本文的模型.最后是全文的总结.

1 相关工作

评价迷惑变换软件的鲁棒性非常有意义,但因涉及到软件的复杂性和人的理解力,评价工作十分困难.直至目前,国内还没有关于这方面的研究资料,而国外关于这方面的文献也很少.

Collberg 等人[2]最先在这方面进行了研究,文中提出了用传统的软件复杂制来帮助开发者选择不同的迷惑变换技术.Goto 等人[8]基于编程语言编译器的语法分析树,提出了一种关于迷惑变换增加阅读代码难度的量化度量方法,该方法依据代码在分析树中的深度作为权重来量化计算.通过实验表明,在插入哑块、函数替换以及循环修改中,循环修改更能增加分析者的阅读难度,但是 Goto 提出的方法限于局部语句块的单语法分析树.2005 年,Udupa 等人[9]联合静态分析与动态分析,根据自动反迷惑器的运行时间来评价“控制流扁平迷惑”的有效性.Udupa 的工作更多地是从弹力[2]角度来度量迷惑技术的有效性,很难全面而真实地反映变换带来的鲁棒性.

最近,Anckaert 等人[10]认为,程序有 4 种相对独立的属性:指令、控制流、数据流以及数据.基于 4 种属性对应用迷惑变换的程序建立了一种抽象的比较与评价框架,并通过实验表明,所提出的框架能够在一定程度上反映由迷惑变换导致的软件复杂性.但文中未阐明 4 种属性之间的关系以及不同的迷惑变换法对复杂性的影响,而且 4 种属性复杂性评价也是基于那些传统的软件复杂制.2009 年,Hsin-Yi 等人[11]建立了一种形式化抽象框架,认为现有的大多数迷惑变换算法能够由所建立的框架内的原子操作复合而成,这有利于对现有迷惑算法的理解和将来对新迷惑算法的构建.为量化描述基于控制流迷惑变换程序的复杂性,文中提出了建立在控制流图基础上的二元矢量 *DP* Vector(distance,potence),用最大公共图 MGE 表示 distance,用 *N*-Scope 制[11]表示 potency,并基于 *DP* 来评价控制流迷惑程序的性能及稳健性.*DP* 矢量的建立是研究变换鲁棒性很有价值的探索,因为它包含了除软件复杂制以外的距离信息.此外,Ceccato 等人[12]通过两个实验发现,标识符重命名技术能够有效地降低攻击效率,迷惑变换能够缩小初级黑客与高级黑客解密代码的水平差别.

归纳起来,以往对迷惑变换的量化度量主要是直接借用传统的软件复杂制,而且度量目的主要是集中在评价一种变换技术的有效性上.其中,部分研究工作限于定性方面,少数定量研究也只是针对具体类别的变换技术或特定的一种变换技术,因此目前还未形成统一的、实用的、适应面广的量化评价框架.这不利于开发者对众多迷惑变换技术应用效果的比较与选择.

2 系统复杂度模型(SCM)

在本文第 1 节已指出,如果直接借用传统的软件复杂制来度量迷惑变换的力度[2]会存在一些问题,为了更准确地从软件复杂性的角度评价迷惑变换的鲁棒性,本节详细阐述第 1 节中提到的模型——系统复杂度模型(SCM).

在软件工程中,为了突出程序的控制流,人们通常使用控制流图(control flow graph,简称 CFG)来研究程序,如传统的方法 McCabe 制[4]与 *N*-Scope 制[11]都是基于 CFG 的.本节在 CFG 的基础上,从系统科学的角度构建一种多因素的复杂度模型,旨在从系统固有的复杂性层面来反映文献[2]中所提出的力度.接下来定义模型中引入的相关概念.

2.1 相关概念

控制流图 CFG(control flow graph):一种描述程序控制逻辑的有向图,定义为 $G=(V,E)$,其中,V 和 E 分别是节点集与有向边集.图中的每个节点对应于一个顺序代码块或分支块,有向边对应于程序中的转移,并且每个节点都可以从入口节点到达,从每个节点也可以到达出口节点,例如下面的 C 语言代码片段(来自第 5.1 节):

```
int i,j;
int n;
int x[30];
printf (“Please input n(<=20):”);
scanf (“%d”, &n);
i=0;
if (n>20) n=20;
while (i<n)
   {printf (“Please input: x[%d]=”, i+1);
   scanf (“%d”, x+i); i++;}
```

其 CFG 如图 1 所示,图中只存在顺序代码节点,表示为 S_i,分支代码节点,表示为 B_i;T 代表节点的逻辑“真”出口,F 代表节点的逻辑“假”出口.

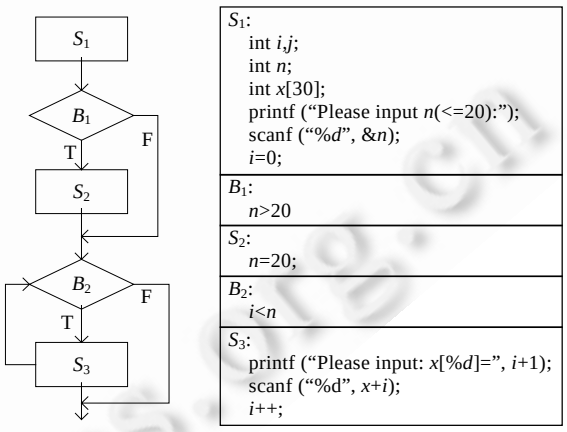


Fig.1 An example for CFG

图 1 控制流图实例

节点关系度:对 CFG 中的每个节点 v_i,节点入度 $D_{in}(v_i)$是指直接射入节点 v_i 的边数;节点出度 $D_{out}(v_i)$是指从节点 v_i 射出的边数;节点关系度定义为 $RD(v_i)=D_{in}(v_i)\times D_{out}(v_i)$,它代表进入节点与流出节点的关系组合数.例如在图 1 中,$RD(B_1)=1\times 2=2$;$RD(B_2)=2\times 2=4$.为了避免 $RD(v_i)=0$,规定 $D_{in}(v_i)=\max(1,D_{in}(v_i))$,$D_{out}(v_i)=\max(1,D_{out}(v_i))$.

节点数据流:对 CFG 中的每个节点 v_i,节点流入数据(node in-data)$NID(v_i)$定义为节点语句块中读取不同的变量与常量总数.同理,节点流出数据(node out-data)$NOD(v_i)$定义为节点语句块中写入不同的变量总数,其中的变量包括全局变量和局部变量.节点 v_i 的数据流(data flow)可定义为 $DF(v_i)=NID(v_i)+NOD(v_i)$,它代表节点与周围环境的信息交互的数目.例如在图 1 中,节点 S_3 读取了变量 i,x 以及常量 1;写入的变量有 i,$x[i+1]$,因此,$DF(S_3)=3+2=5$.

分支数 NB(number of branch nodes):$NB(G)=\sum B_i$,是指 CFG 中分支节点的总和.图 1 中有 B_1,B_2 这两个分支节点,因此 $NB(G)=2$.

循环指数:循环路径长度 LPL(loop path length)是循环中所包含的节点数量;CFG 中所有的 LPL 总和构成图的循环指数 LI(loop index):$LI(G)=\sum LPL_i$. 如图 1 所示,$LI(G)=LPL_1(B_2,S_3,B_2)=3$.

节点嵌套级别:程序代码是由顺序、分支、循环这 3 种基本结构复合而成,一般来说,很多编译器规定在进入过程/函数时的代码复合(嵌套)级别为 0,3 种基本结构每复合 1 次,代码的嵌套级别加 1.在本模型中,为了避免

乘以嵌套级别为 0 的权重,规定在过程/函数入口处代码的嵌套级别为 1.CFG 中的每个节点 v_i 的嵌套级别(nest level)$NL(v_i)$是节点 v_i 的语句序列在源代码中的复合级别.图 1 中,节点 B_1 与 S_3 的节点嵌套级别分别为 $NL(B_1)=1$, $NL(S_3)=1+1=2$(因为 while 语句的嵌套级别为 1).

2.2 度量模型

节点的关系度 $RD(v_i)$反映控制流图 CFG 中节点的连接特性,节点关系度数越大,表明与之关联的节点越多,这是软件局部结构复杂性的一种表征.所有节点的关系度数总和 $RD(G)=\sum_{i=1}^{n}D(v_i)$,可在一定程度上体现 CFG 中节点和边所构成网络结构的复杂性.软件的 CFG 可以看成是一个开放系统,它与外界的信息交互表现为软件的功能.这种功能由系统内部结构与内部节点间的信息交互支撑.

节点数据流 $DF(v_i)$是节点之间交互的一种体现,CFG 的总体数据流 $DF(G)=\sum_{i=1}^{n}DF(v_i)$ 反映软件内部交互的流复杂性.

分支 $NB(G)$与循环 $LI(G)$也是影响 CFG 控制结构复杂性的重要因子.分支节点数量以及分支节点水平扩展宽度,循环结构数量以及循环路径长度,都影响分析软件的逻辑路程,体现了结构复杂性.

在软件的 CFG 中,节点上的代码都是简单代码,它们与有向边的关系共同实现软件的功能(语义).虽然节点都是由简单代码组成,但不同的节点对实现语义的理解复杂度不一定相同.本文用节点的嵌套层次 $NL(v_i)$来代表节点 v_i 的理解复杂度.

上面提出的关系度数总和 RD、总体数据流 DF、分支数 NB、循环指数 LI 与节点嵌套层次 NL,是从不同的侧面反映软件的复杂性.为了综合度量软件的全局复杂度,我们要同时联合这 5 个元素来加以量化.由于前 4 个分量具有一定的独立性,而且 NL 可以作为一个权重施加在前 4 个分量上,为此引入由 4 维空间组成的全局复杂度矢量 $\boldsymbol{C}(\boldsymbol{G})$,如下定义:

$$\boldsymbol{C}(\boldsymbol{G})=(RD,DF,NB,LI),$$

其中,

$$RD=\sum_{i=1}^{n}D(v_i) \tag{1}$$

$$DF=\sum_{i=1}^{n}DF(v_i)\times NL(v_i) \tag{2}$$

$$NB=\sum B_i \tag{3}$$

$$LI=\sum LPL_i$$
$$LPL_i=\sum_{j=1}^{l}NL(v_j) \tag{4}$$

其中,$NL(v_i)$代表节点的嵌套层次数,l 是第 i 个循环($Loop_i$)中所包含的节点数量,j 代表循环 i 中的第 j 个节点.

根据 $\boldsymbol{C}(\boldsymbol{G})$矢量制,可以对两个不同的软件系统复杂性进行比较.因不同的软件系统 $\boldsymbol{C}(\boldsymbol{G})$的每个元素可能有较大的差别,矢量差只说明系统间的因素差异性.比如 $\boldsymbol{C}(\boldsymbol{G}_A)-\boldsymbol{C}(\boldsymbol{G}_B)=(-5,10,-20,17)$,这说明系统 A 的 RD 及 NB 维度比系统 B 的要小,而在 DF 与 LI 维度上系统 A 比系统 B 要大,但这说明不了系统 A 与 B 哪一个的复杂度要高.为了比较不同软件系统复杂度大小,首先要进行归一化,本文定义如下表达式:

$$\|C_d(G)\|=\sqrt{RD^2+DF^2+NB^2+LI^2},\ SCM(C)=\|C_{avg}(G)\|=\frac{\|C_d(G)\|}{n} \tag{5}$$

其中,C 表示控制流图 G 对应的代码;$\|C_d(G)\|$代表在 4 维空间中,矢量 $\boldsymbol{C}(\boldsymbol{G})$到原点的距离,如果以它作为复杂度评价模型,则运算结果会很大,尤其是在较大的软件系统中;而且,它随着 G 中节点数 n 的增加而增大(规模的扩大),不能完全体现系统的真实复杂度.

$\|C_{avg}(G)\|$则从全局上反映软件系统中节点的平均复杂度,我们称其为软件复杂度密度,也就是本文提出的系统复杂度模型 SCM.以 SCM 作为软件复杂度评价模型,从软件防止受攻击的层面来说,该值越大越好.当模型 $SCM=\|C_{avg}(G)\|$中的 G 取 CFG 当中的一个子图时,则 $SCM(C_{sub})=\|C_{avg}(G_{sub})\|$,度量的是软件局部复杂度密度,有利于反映软件中关键部分代码的安全程度.

2.3 模型分析

Collberg 等人[2]把迷惑变换分成 4 类:词法变换、布局变换、数据变换、控制流变换.其中,词法变换和布局变换主要是用于妨碍反编译工具的编译效果,这两类变换对软件鲁棒性的增强主要体现在变换模糊度 TOM 上(见本文第 3 节);而数据变换、控制流变换除了增加变换模糊度,还会使软件在矢量 $\boldsymbol{C}(\boldsymbol{G})=(RD,DF,NB,LI)$的 4 个分量上的部分或全部提升,表现为系统复杂性的增强.

传统的软件复杂制如代码行制(LOC)、Halstead[3]、McCabe[4],以及之后提出来的一些复杂制模型,主要在软件工程领域发挥着各自的作用.但因为这些模型各自包含的大部分信息是我们所提出模型的子集,如果单独用它们来度量迷惑变换软件的鲁棒性,就不如所提出的 SCM 全面.这里,我们并无否定那些模型的价值之意,因为模型建立的出发点影响了它的应用领域.

所提出的模型 SCM(如公式(5)所示)是从系统学的角度来描叙软件的,它涉及软件系统的结构、信息流、分支、循环以及元素的嵌套层次;模型的信息丰富多样,计算结果能够更准确地反映软件复杂性表示的理解困难性.

2.4 SCM计算实现方案

本文在基于软件源代码建立控制流图(CFG)的基础上,提出了一种软件复杂性度量方法.由源代码到 CFG 的转换涉及程序设计语言编译知识,从逻辑上来说,编译器要对源代码依次进行词法分析、语法分析、语义分析、中间代码生成、代码优化和目标代码生成这 6 个阶段的处理.而对普通的基于源代码的 CFG 转换器,至少涉及编译器前 3 个阶段的工作,即从词法分析到语义分析阶段,工作量非常大.因此,本文不研究具体的 CFG 生成算法,而是要计算 $SCM(\|C_{avg}(G)\|)$,在现有的 CFG 生成算法的基础上给出详细的修改方案.

假定 CFG 生成器算法接受高级语言源代码字符流的输入,并输出对应的控制流图 CFG 以及图中节点对应的代码块.代码块中的代码可以是源代码、中间代码或汇编代码,为了便于理解,本文采用源代码形式.要实现 SCM 的自动计算,CFG 生成器要输出以下参数值:

(1) 控制流图中节点数量 n;
(2) 每个节点 v_i 的入度 $D_{in}(v_i)$和出度 $D_{out}(v_i)$,以便计算 RD;
(3) CFG 中分支节点的数量 NB;
(4) 确定每个节点 v_i 的嵌套层次级别 $NL(v_i)$;
(5) 输出所有的简单循环及其包含的节点序列,实现 LI 的计算;
(6) 分析节点 v_i 中代码块所包含的不同的变量(常量)数目 $DF(v_i)$.

为了计算这些参数,要对普通的 CFG 生成器算法进行如下改进:

第 1 步:计算节点嵌套层次级别 $NL(v_i)$.在现有的 CFG 生成器算法中设置一个全局变量 Lev 来记录当前语法分析器扫描的代码块所处的嵌套层次.语法分析器按以下规定来处理 Lev:

(1) 函数(过程)说明语句中,最外层的嵌套层次级别的 Lev 在本文规定为 1.如果存在嵌套定义,从外往里地,函数(过程)的嵌套层次级别依次加 1(参考第 2.1 节);
(2) 函数(过程)入口代码的嵌套层次级别等于函数(过程)说明语句的层次,因此,主程序及其入口代码的嵌套层次级别 Lev=1;
(3) 进入函数(过程)后,在从上到下扫描源代码时,每当遇到选择语句(如 if-else-endif)和循环语句(如 while,for)两类语句时,$Lev=Lev+1$;退出语句时,$Lev=Lev-1$;
(4) 对于选择语句和循环语句,它们自身的复合或之间的复合,每复合 1 次与退出复合语句时进行类似处理;
(5) 当语法分析器生成一个 CFG 节点 v_i 时,令 $NL(v_i)=Lev$.

第 2 步:标记 CFG 中节点类型 $Type(v_i)$.本文规定 CFG 中的节点分为两大类,即顺序节点(用 S 表示)与分支节点(用 B 表示);而 B 类型节点又可细分为选择分支(用 CB 表示)与循环分支(用 LB 表示).当现有的 CFG 生成

器算法经过语法分析生成 CFG 节点时,可以根据代码块的上下文环境唯一地确定节点的类型.

第 3 步:计算 CFG 中节点数据流 $DF(v_i)$.在生成 CFG 节点时,对当前代码块中的被读取与被写入不同的变量/常量($NID(v_i)$,$NOD(v_i)$)分别统计总数,并计算 $DF(v_i)=NID(v_i)+NOD(v_i)$.

第 4 步:基于 CFG 存储结构计算其他参数.CFG 生成器算法能够输出软件系统的 CFG,它是一个有向连通图;CFG 的存储结构有多种形式,如邻接矩阵、邻接表等.在此基础上,可以计算如下参数:

(1) 统计 CFG 中节点总数 n;

(2) 统计分支节点总数 NB;

(3) 计算每个节点 v_i 的入度 $D_{in}(v_i)$和出度 $D_{out}(v_i)$;

(4) 根据节点类型 $Type(v_i)$,查找所有的循环分支(LB)节点,对每个这样类型的节点 v_j 搜索以它为起点和终点的简单最大环(如 $v_{j1},v_{j2},\ldots,v_{j(n-1)},v_{jn},v_{jn}=v_{j1}$),用以计算 LPL_j.

3 变换模糊度模型(TOM)

本文第 1 节已说明,复杂性并不能全面描述变换带来的功能模糊性,即迷惑变换增加代码语义的识别难度.如何进一步定量地描述变换导致的功能模糊性,将是本节的研究内容.由于迷惑变换既可以单独应用、复合应用,也可将多种迷惑变换在软件的不同代码段形成更为复杂的综合应用,因此,本节要为迷惑变换的这些应用分别建立功能模糊度量化模型.

3.1 单种变换模糊度

由于量化评价单种迷惑变换模糊度涉及到人的理解力,为了简化处理这类复杂问题,本文采用管理学中普遍应用的综合评价法.关于综合评价法有很多种,有兴趣的读者可以参阅相关管理学、系统工程学内容.在本节,我们采用的是指标评分法中的多比例打分法[13].

在实践中,我们可以请多位领域专家对不同的迷惑变换技术的变换模糊度进行多比例打分[13].设有迷惑变换集 $T=\{t_1,t_2,\ldots,t_n\}$,对迷惑技术进行两两比较,对 t_i 与 $t_j(1\leqslant i,j\leqslant n,i\neq j)$的比较得分比例设置如下:1:0,0.9:0.1,0.8:0.2,0.7:0.3,0.6:0.4,0.5:0.5.假定有 m 位专家参与打分,则第 $i(1\leqslant i\leqslant m)$位专家对迷惑变换集 T 变换模糊度比较评价结果见表 1.

其中,$x_{jk}^{(i)}$ 表示第 i 位专家对变换 t_j 相对变换 t_k,变换模糊度的比例分配.其中,

$$x_{jk}^{(i)}(1\leqslant j,k\leqslant n)\in\{0,1,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9\},\text{且 } x_{jk}^{(i)}+x_{kj}^{(i)}=1.$$

然后,对 m 位专家评分结果进行综合,结果见表 2,其中,

$$x_{jk}=\frac{\sum_{i=1}^{m}x_{jk}^{(i)}-\max_i(x_{jk}^{(i)})-\min_i(x_{jk}^{(i)})}{m-2},\ j,k=1,2,\ldots,n,0\leqslant x_{jk}\leqslant 1 \tag{6}$$

x_{jk} 表示在 m 位专家的评价结果中除去最大值与最小值后的平均值.其中,C_{Ti} 定义如下:

$$C_{Ti}=\sum_{j=1}^{n}x_{ij},i=1,2,\ldots,n \tag{7}$$

表示迷惑变换 t_i,在表 2 中的横向求和得分.经过以上调整后,仍然满足 $x_{jk}+x_{kj}\equiv 1$.为避免以后计算中除 0,如果 $x_{jk}=0$,则令 $x_{jk}=0.01$,$x_{kj}=0.99$,于是有 $0<x_{jk}<1$.由此,对迷惑变换集 $T=\{t_1,t_2,\ldots,t_n\}$中任意单种变换技术 t_i,其变换模糊度 $O(t_i)$可定义为

$$O(t_i)=\frac{C_{Ti}}{\sum_{j=1}^{n}C_{Tj}}\times 100,\ i=1,2,\ldots,n,0<O(t_i)<100 \tag{8}$$

Table 1 Evaluation results of the *i*th expert

表 1 专家 *i* 对 *T* 集变换模糊度的评价

t_i:t_j	t_1	t_2	…	t_n
t_1	0.5	$x_{12}^{(i)}$	…	$x_{1n}^{(i)}$
t_2	$x_{21}^{(i)}$	0.5	…	$x_{2n}^{(i)}$
…	…	…	…	…
t_n	$x_{n1}^{(i)}$	$x_{n2}^{(i)}$	…	0.5

Table 2 Evaluation results combining *m* experts

表 2 *m* 位专家对 *T* 集变换模糊度的综合评价

t_i:t_j	t_1	t_2	…	t_n	C_T
t_1	0.5	x_{12}	…	x_{1n}	C_{T1}
t_2	x_{21}	0.5	…	x_{2n}	C_{T2}
…	…	…	…	…	…
t_n	x_{n1}	x_{n2}	…	0.5	C_{Tn}

3.2 多种变换模糊度

在软件的整个代码中,我们既可以针对同一个局部代码块应用多种迷惑变换技术,也可以对不同的代码块应用不同的迷惑变换技术.迷惑变换技术根据其变换原理,不同的变换技术之间可以线性组合使用,有些还可以复合使用.

不妨把软件代码看成一个有序序列集,它由 n 个代码块顺序组成.令 C 代表整个软件代码,而 $cb_i(i=1,2,\ldots)$ 代表其中顺序第 i 个代码块,则有,$C=\langle cb_1,cb_2,\ldots,cb_n\rangle$.另设 G 为 C 对应的控制流图,那么 $G(cb_i)$表示代码块 cb_i 在 G 中对应的子图,$N(G(cb_i))$表示图 $G(cb_i)$中的节点数量.下面我们先讨论迷惑变换的两种简单应用情形:线性组合与简单复合,它们是多种迷惑变换技术综合应用的基础.

3.2.1 变换线性组合模糊度

设 C'是由 C 中 $k(k\leqslant n)$个互不重合的代码块组成的代码块集合,即有

$$C'\subseteq C, C'=\{cb_{i_1},cb_{i_2},\ldots,cb_{i_k}\},1\leqslant i_1,i_2,\ldots,i_k\leqslant n.$$

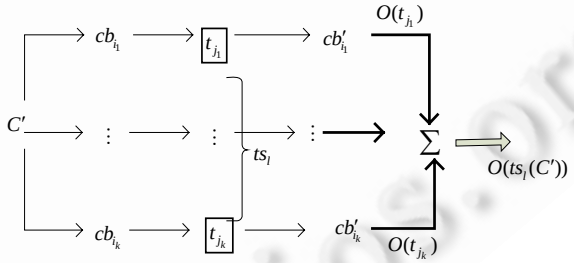
如果对C'中的每个代码块单独使用一种迷惑变换技术,这种多个迷惑变换技术的k次使用在本文中称为变换的线性组合应用,这种迷惑变换方案可形式化表示为 ts_l.

$$ts_l(C')=\{t_{j_1}(cb_{i_1}),t_{j_2}(cb_{i_2}),\ldots,t_{j_k}(cb_{i_k})\},1\leqslant i_1,i_2,\ldots,i_k\leqslant n,t_{j_m}\in T,1\leqslant m\leqslant k \tag{9}$$

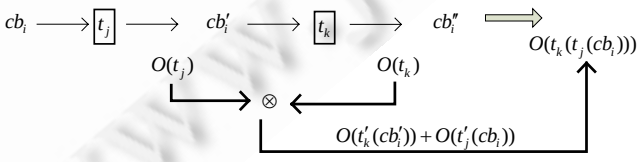
在公式(8)的基础上,对 C'本文定义变换线性组合模糊度为

$$O(ts_l(C'))=\sum_{l=1}^{k}O(t_{j_l}) \tag{10}$$

如图 2(a)所示,变换线性组合模糊度等于单种变换模糊度的总和.



(a) 线性组合



(b) 简单复合

Fig.2 Multiple transforms application

图 2 多种迷惑变换的应用

3.2.2 变换简单复合模糊度

我们先来考虑$\forall cb_i \in C'$,$1 \leqslant i \leqslant n$,先对 cb_i 应用迷惑变换技术 t_j,然后再应用变换技术 t_k(如图 2(b)所示),求复合变换模糊度 $O(t_k(t_j(cb_i)))$.

由公式(8)可知,$O(t_j(cb_i))=O(t_j)$,$O(t_k(cb_i))=O(t_k)$.为了量化评价 $O(t_k(t_j(cb_i)))$,我们很容易想到根据第 3.1 节中的内容建立 $O(t_k(t_j(cb_i)))$与 $O(t_j)$和 $O(t_k)$之间的函数关系,但也可能存在不依据 $O(t_j)$和 $O(t_k)$即可重新建立其他量化表达式的情况,本文只限于对前者的研究.为从 $O(t_j)$和 $O(t_k)$建立 $O(t_k(t_j(cb_i)))$表达式,我们先来看看它们之间的关系表达式.

从反向工程者角度而言,要理解复合变换 $t_k(t_j(cb_i))$语义,依据经验,我们认为其复杂度高于对同一代码块分别进行 t_j,t_k 单项变换语义复杂度的线性之和,但应低于这两次单项变换复杂度的乘积.我们将这种关系作如下表示:

$$\begin{aligned} O(t_j)+O(t_k) &\leqslant O(t_k(t_j(cb_i))) \\ O(t_k(t_j(cb_i))) &\leqslant O(t_j)\times O(t_k) \end{aligned} \tag{11}$$

注意,公式(11)只是一个经验公式,是否存在根据严格的数学推理建立的关系将是很有价值的研究内容.

此外我们都知道,在数学的复合函数中,要理解其中被复合的简单函数的难度,不低于这样的简单函数的单独应用的难度.与此类似,我们认为:对变换 t_j,从复合体 $t_k(t_j(cb_i))$中理解 $t_j(cb_i)$的复杂度,要高于单独应用环境中的复杂度,对变换 t_k,情况也是如此.这种关系表示如下:

$$\begin{aligned} O(t_j(cb_i)) &\leqslant O(t_j(cb_i))in[t_k(t_j(cb_i))] \\ O(t_k(cb_i)) &\leqslant O(t_k(cb_i'))in[t_k(t_j(cb_i))] \end{aligned} \tag{12}$$

根据公式(8),可得单种变换单独应用的模糊度 $O(t_j)$,$O(t_k)$,为求解 $O(t_k(t_j(cb_i)))$,本文定义

$$O(t_k(t_j(cb_i))) = O(t_k'(cb_i')) + O(t_j'(cb_i)) \tag{13}$$

公式(13)的理解过程如图 2(b)所示.

注意,为满足关系式(11)、关系式(12),从 $O(t_j)$到 $O(t_j')$, 以及从 $O(t_k)$到 $O(t_k')$ 可分别建立多种关系式.在第 3.1 节的基础上,我们按如下方式来建立它们的关系.在此要说明的是,我们所建立的公式不是唯一的表示方式.

从后文表 3 可知,由于$\langle t_j:t_k\rangle=x_{jk}$,$\langle t_k:t_j\rangle=x_{kj}$,为使公式(13)满足公式(11)、公式(12),定义如下关系:

$$\begin{aligned} O(t_k') &= O(t_k)\times\left(1+\frac{x_{kj}}{x_{kj}+x_{jk}}\right) \\ O(t_j') &= O(t_j)\times\left(1+\frac{x_{jk}}{x_{kj}+x_{jk}}\right) \end{aligned} \tag{14}$$

由于 $x_{jk}+x_{kj}\equiv 1$(见第 3.1 节),公式(14)可简化为

$$\left.\begin{aligned} O(t_k') &= O(t_k)\times(1+x_{kj}) \\ O(t_j') &= O(t_j)\times(1+x_{jk}) \end{aligned}\right\} \tag{15}$$

至此,我们在第 3.1 节的基础上,根据 $O(t_j)$与 $O(t_k)$,提出了一种求解 $O(t_k(t_j(cb_i)))$的方法(公式(13)、公式(15)).是否还存在更合理的其他形式的表达式,值得进一步加以研究.

3.2.3 变换复杂复合模糊度

本节将第 3.2.2 节的内容加以推广.下面我们考虑这样一种更为复杂的复合情况:设迷惑变换集 $T=\{t_1,t_2,\ldots,t_n\}$,且 C'是由 C 中 $k(k\leqslant n)$个互不重合的代码块组成的代码块集合,即有

$$C'\subseteq C, C'=\{cb_{i_1},cb_{i_2},...,cb_{i_k}\},1\leqslant i_1,i_2,\ldots,i_k\leqslant n.$$

对 C'的复合迷惑变换方案 ts_c 定义为

$$ts_c(C')=t_{j_m}(...t_{j_2}(t_{j_1}(\sum_{k=1}^{k=p}t_{i_k}(cb_{i_k})))...),k,m,p=1,2,...;1\leqslant j_m,i_k\leqslant n \tag{16}$$

按照公式(13)、公式(15)建立的方式,方案 ts_c 的变换模糊度可表示如下:

$$O(ts_c(C'))=\sum_{k=1}^{k=p}O(t_{i_k})\times(1+x_{i_k j_1})+\sum_{l=1}^{l=m-1}O(t_{j_l})\times(1+x_{j_l j_{l+1}})+O(t_{j_m})\times(1+x_{j_m j_{m-1}}) \quad (17)$$

3.3 变换综合应用的模糊度

在第 3.2 节中,公式(9)、公式(10)与公式(16)、公式(17)分别给出了迷惑变换模糊度的两类基本模型:线性组合模型与复合模型.在实际应用中,一种迷惑变换方案对软件代码的不同局部既可以采取线性组合应用,也可以施加复合应用,或者将二者混合应用.如果将公式(9)、公式(16)进行适当组合应用,则能产生更为复杂的变换模型,以适应变换综合应用需求,如图 3 左侧所示.

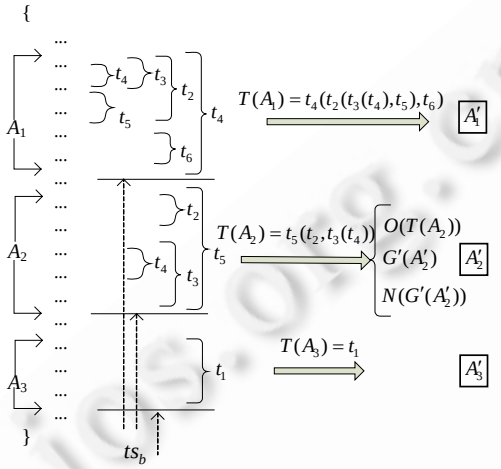


Fig.3 Calculating procedure for TOM

图 3 TOM 的计算过程

如果用形式化方式表示变换综合应用模型会显得非常复杂,所以本节采用算法的方式来计算此类应用.注意,本算法建立的基础来自第 3.1 节与第 3.2 节所提出的模型,而且在算法建立后我们会发现,它们的计算过程会是本算法的一种特殊情形.因此,以后关于变换模糊度的计算都采用接下来要介绍的算法.

设有迷惑变换集 $T=\{t_1,t_2,\ldots,t_n\}$,ts_b 表示一种复杂的迷惑变换方案,对一段给定的代码 C,执行迷惑方案 ts_b 后,代码的变换模糊度 $TOM(ts_b(C))$计算步骤如下:

第 1 步:求最大变换代码块集合 A.程序员在软件代码 C 的不同局部可以施加复杂的迷惑变换,对 C 中自上而下、从外到里地找出没有被其他迷惑变换复合的变换(最外层的变换)序列 T_{out}:

$$T_{out}=\{t_{i_1},t_{i_2},\ldots,t_{i_k}\},1\leqslant i_1,i_2,\ldots,i_k\leqslant n,k=1,2,\ldots$$

再根据 T_{out} 集中的每个元素 $t_m(1\leqslant m\leqslant k)$的应用范围,找出其对应的代码块 A_m,由此可以得出:

$$A=\{A_1,A_2,\ldots,A_k\},k=1,2,\ldots$$

A 为本步骤所求的最大变换代码块集合.

例如在图 3 中,由于 $T_{out}=\{t_4,t_5,t_1\}$,得出 $A=\{A_1,A_2,A_3\}$.

第 2 步:求代码块的复合变换表达式.对$\forall A_i\in A$,描述其复合变换表达式如下:

$$T(A_i)=t_{i_1}(\ldots t_{i_j}(t_{i_{(j+1)}},t_{i_{(j+2)}}(t_{i_{(j+3)}}),\ldots)\ldots).$$

描述规则如下:在代码块 A_i 中:

(1) 如果 $t_{i_{(j+1)}}$ 被 $t_{i_{(j)}}$ 直接复合,可描述为 $t_{i_j}(t_{i_{(j+1)}})$;

(2) 如果 $t_{i_{(j+1)}},t_{i_{(j+2)}},\ldots,t_{i_{(j+n)}}$ 是并列关系且同时被 $t_{i_{(j)}}$ 直接复合,可描述为 $t_{i_j}(t_{i_{(j+1)}},t_{i_{(j+2)}},\ldots,t_{i_{(j+n)}})$.

例如在图 3 中,最大变换代码块集 A 中每个元素对应的复合变换表达式如下:

$$T(A_1)=t_4(t_2(t_3(t_4),t_5),t_6),T(A_2)=t_5(t_2,t_3(t_4)),T(A_3)=t_1.$$

第 3 步:计算 A 的模糊度:$O(T(A))$.

根据公式(10),我们采用变换模糊度线性相加规则,即 $O(T(A))=\sum_{i=1}^{i=k}O(T(A_i))$. 为了计算 $O(T(A_i))$,要从左到右地扫描第 2 步求出的 $T(A_i)$表达式.初始设 $O(T(A_i))=0$,对扫描到的每个 $t_j(1\leqslant j\leqslant n)$作如下 3 种情形之一的处理:

(1) 如果 t_j 的左边与右边都未被其他变换复合(如图 3 所示,t_1 in $T(A_3)$),根据公式(8)计算 $O(t_j)$;

(2) 如果 t_j 的左边未被任何变换复合,而右边直接复合 t_l(如图 3 所示,$t_5(t_2)$ in $T(A_2)$,$j=5$,$l=2$),根据公式(18),

$$O(t_j)=O(t_j)\times(1+x_{jl}).$$

(3) 如果 t_j 的左边直接被 t_l 复合(如图 3 所示,$t_2(\ldots,t_5)$ in $T(A_1)$,$j=5$,$l=2$),根据公式(18),$O(t_j)=O(t_j)\times(1+x_{jl})$.

接着执行 $O(T(A_i))=O(T(A_i))+O(t_j)$,继续扫描,直到表达式结束.

第 4 步:计算 $TOM(ts_b(C))$.在第 1 步中已求得 $A=\{A_1,A_2,\ldots,A_k\}$,应用变换方案 ts_b 后,集合相应地变为

$$A'=\{A'_1,A'_2,...,A'_k\}.$$

A 对应的控制流图(CFG)G 也变为 G'.代码集 A'的 CFG 中节点数定义为

$$N(G'(A'))=\sum_{i=1}^{k}N(G'(A'_i)).$$

到此,我们能给出本文所提出的变换模糊度模型 TOM(迷惑变换模糊度密度):

$$TOM(ts_b(C))=O_{avg}(T(A))=\frac{O(T(A))}{N(G'(A'))}.$$

4 变换鲁棒性的度量

本节研究的是,对一段给定的代码 C,如何判断一种特定的迷惑变换方法的有效性,以及对应用在 C 上的多种迷惑变换方案产生的鲁棒性进行排序,以便从中选出最优方案.

设有迷惑变换方案集 $TS=\{ts_1,ts_2,\ldots,ts_n\}$,其中,$ts_i(1\leqslant i\leqslant n)$属于以下情形之一:

(1) 单种迷惑变换技术在 C 的不同代码段单独应用;

(2) 不同的迷惑变换技术在 C 的不同代码段单独应用;

(3) 不同的迷惑变换技术在 C 的不同代码段复合应用.

当 ts_i 应用在 C 后,经过模型 SCM 与 TOM 计算形成一个保护力矢量:$\boldsymbol{PS}_i=(SCM(ts_i(C)),TOM(ts_i(C)))$.$\boldsymbol{PS}_i$ 是一个二维矢量,第 1 维表示迷惑变换方案 ts_i 带来的系统复杂度,第 2 维表示变换模糊度.

4.1 单种变换有效性判断

当程序员应用一种新的迷惑变换技术时,最先关注的是新的变换技术对待保护的代码是否有效.因单种的迷惑变换技术可以看成是一种特殊的迷惑变换方案,所以问题归结为如何判定一种迷惑变换方案的有效性.

对一段给定的代码 C,设 ts_0 表示不作任何操作的空变换方案,则有 $ts_0(C)=C$,$SCM(ts_0(C))=SCM(C)$;令

$$TOM(ts_0(C))=TOM(C)=0,$$

其中,ts_0 不改变对 C 的理解难度.对任意给定的一种迷惑变换技术(方案)ts_1,其有效性问题可转化为对 C,比较变换方案 ts_0,ts_1 优劣的问题.此问题的求解过程将在第 4.2 节中加以讨论,如果 ts_1 评价结果大于 ts_0,说明变换方案 ts_1(单种迷惑变换)是有效的.

4.2 迷惑变换方案的排序

为了简化说明,我们令 $\boldsymbol{PS}_i=(x_i,y_i)$,其中,$i$ 代表第 i 种迷惑变换方案,$1\leqslant i\leqslant n$.方案选择过程如下:

(1) 把 TS 中的每种方案分别应用于 C 后,可计算出 n 个矢量:$(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)$.由于矢量 $\boldsymbol{PS}_i$ 的第 1 维与第 2 维的量纲不同,为了比较 n 个矢量的大小,要分别对这两维数据进行标准化处理,可将数据映射到区间[0,1]上.本文采用的标准化方法定义如下:

$$PS'_i=(x'_i,y'_i)=(x_i/sum(x),y_i/sum(y)).$$

(2) 计算 PS'_i 的得分,计算公式定义如下:

$$E(PS_i') = w_1 \times x_i' + w_2 \times y_i',$$

其中,w_1,w_2 分别为分配给矢量 $\boldsymbol{PS}_i$ 第 1 维与第 2 维的权重,且 $w_1+w_2=1$,它们的具体值可以根据实际情况来确定.由于 $\boldsymbol{PS}_i$ 的第 2 维表示变换模糊度,根据迷惑变换的目的,w_1 应小于 w_2.在本文中,不妨设 $w_1=0.4$,$w_2=0.6$.

(3) 根据各方案的得分,对方案进行降序排序,可得到如下关系:

$$E(PS'_{j_1}) \geqslant E(PS'_{j_2}) \geqslant ... \geqslant E(PS'_{j_n}).$$

从而有

$$ts_{j_1} \geqslant ts_{j_2} \geqslant ... \geqslant ts_{j_n},$$

其中,j_i=1,2,…,n.因此,在变换方案集 $TS=\{ts_1,ts_2,\ldots,ts_n\}$中,$ts_{j_1}$ 为所求的最优迷惑变换方案.

5 实例仿真

根据前面提出的软件系统复杂度模型(SCM,第 2 节)、迷惑变换模糊度模型(TOM,第 3 节)以及各自的计算算法,本节以选择排序软件为例进行仿真分析,详细展示模型的工作过程.

5.1 代码实例

程序 I 中,Part *A* 从键盘上任意输入不多于 20 个的整数,Part *B* 对输入序列数据进行选择排序,Part *C* 将排序结果打印输出.

```
/*Program I. Selection sort*/
void main(·)
{/******** Part A: input*******************/
   int i,j;
   int n;
   int x[30];
   printf ("Please input n(<=20):");
   scanf ("%d", &n);
   i=0;
   if (n>20) n=20;
   while (i<n)
   {printf ("Please input: x[%d]=", i+1);
      scanf ("%d", x+i); i++;}
   /******Part B: select_sort**********/
   int min, tmp;
   for (i=0; i<n–1; i++)
   {min=i;
      for (j=i+1; j<n; j++)
         if (x[j]<x[min]) min=j;
      if (min!=i)
         {tmp=x[i]; x[i]=x[min]; x[min]=tmp;}
   }
   /******** Part C: output*******************/
   i=0;
   while (i<n)
   {printf ("x[%d]=%d\n", i+1, x[i]); i++;}
```

}

5.2 迷惑变换集T

为说明问题的方便性,我们选择 8 种常用的迷惑变换技术,即 $T=\{t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8\}$,其中,

t_1="Identifier scrambling[2]";

t_2="Variable splitting[2]";

t_3="Extend Loop Conditions[2]";

t_4="Dummy code insertion[8]";

t_5="Modification of loop[8]";

t_6="Basic block fission[14]";

t_7="Intersecting loop[14]";

t_8="Flattening control-flow[15]".

为了在仿真中能够进行数值计算,我们邀请了多位熟悉迷惑变换技术的学者,站在反向工程者结合静态分析与动态跟踪技术攻击的角度,对这 8 种迷惑变换的模糊度进行了打分评价(见第 3.1 节),汇总评价结果如附表 2(见附录 B)所示.注意,附表 2 的数据只是在本文用于说明如何联合 SCM,TOM 工作的一个例子.若本文的成果要在实践中普遍应用,有待于以下工作的完成:

- 增加更多的常用迷惑变换技术,丰富迷惑变换集合 T;
- 按照第 3.1 节的方法,大量收集各国专家对变换集合 T 的评价结果.

5.3 迷惑变换方案

在我们的仿真中,对程序 I 中的 Part A,Part C 不作任何变换,而对 Part B 采用 4 种迷惑变换方案.Part B 经过不同的变换方案处理后,有 $B\Rightarrow B'$,$G\Rightarrow G'$,然后分别计算每种变换后:

(1) 整个软件的系统复杂度密度 $SCM(G')$;

(2) 局部系统复杂度密度 $SCM(B')$;

(3) 局部变换模糊度密度 $TOM(ts_i(B))$.

迷惑变换方案如下:

(I) $ts_1(B)=t_7(t_5(B))$;

(II) $ts_2(B)=t_5(t_5(B))$;

(III) $ts_3(B)=t_5(B)$;

(IV) $ts_4(B)=t_0(B)$,即对 Part B 不作任何变换,只计算 $SCM(G)$.

方案(I):执行 $ts_1(B)=t_7(t_5(B))$后,变换结果代码如图 4 右侧所示.

内层循环(for ($j=i+1\ldots$))经 t_5 变换后产生的代码被阴影围住;整个代码的真实控制流隐藏在模糊谓词 $\boldsymbol{P}^F$(if ($rand(\cdot)*i<0$)),$\boldsymbol{P}^T$(if ($min*rand(\cdot)>=0$))以及交叉流(goto label 1,goto label 2)中.在本仿真中应用的谓词比较简单,实际应用中应使用更为复杂的谓词.

对程序 I 执行方案 ts_1 后,Part A,Part B,Part C 对应的源代码再经过改进的 CFG 生成器算法(第 2.4 节)处理后,分别生成如图 5(a)、图 6、图 5(b)所示的 CFG,以及 SCM 所需的基础数据如表 3 所示.

表 3 中详细列出了图 5、图 6 中各节点用于 SCM 的参数信息,其中,节点 1~节点 5(A_1~A_5)来自图 5(a),节点 6~节点 25(B_1~B_{20})来自图 6,节点 26~节点 28(C_1~C_3)来自图 5(b).根据第 2.2 节提出的模型以及第 2.4 节中的算法,对表 3 的数据再次计算(详细方法参考附录 A),结果见表 4.其中,$G'(A')$,$G'(B')$,$G'(C')$分别表示程序 I 的 Part A,Part B,Part C 变换后对应的 CFG,n 表示 CFG 中节点的数量,它们的 SCM 值分别为 4.71,7.29,4.75.

此外,由附表 2(见附录 B)可知,$O(t_7)=17$,$O(t_5)=11.4$;$x_{75}=0.7$,$x_{57}=0.3$.程序 I 的 Part B 经方案 ts_1 变换后对应的 B'的 CFG 如图 6 所示,其节点信息见表 3 中的 B_1~B_{20},即 $N(G'(B'))=n=20$.按照第 3 节提出的 TOM 以及第 3.3 节中的算法,计算变换方案 $ts_1(B)$的变换模糊度如公式(18)所示(详细计算方法见附录 B).

$$TOM(ts_1(B))=\frac{O(t_7)(1+x_{75})+O(t_5)(1+x_{57})}{N(G'(B'))}=\frac{17\times(1+0.7)+11.4\times(1+0.3)}{20}=2.19 \quad (18)$$

由此,针对变换方案(I):$ts_1(B)=t_7(t_5(B))$,我们已得到一个矢量:$\boldsymbol{PS}_1=(SCM(ts_1(B)),TOM(ts_1(B)))=(9.29,2.19)$(参考第 4.2 节).

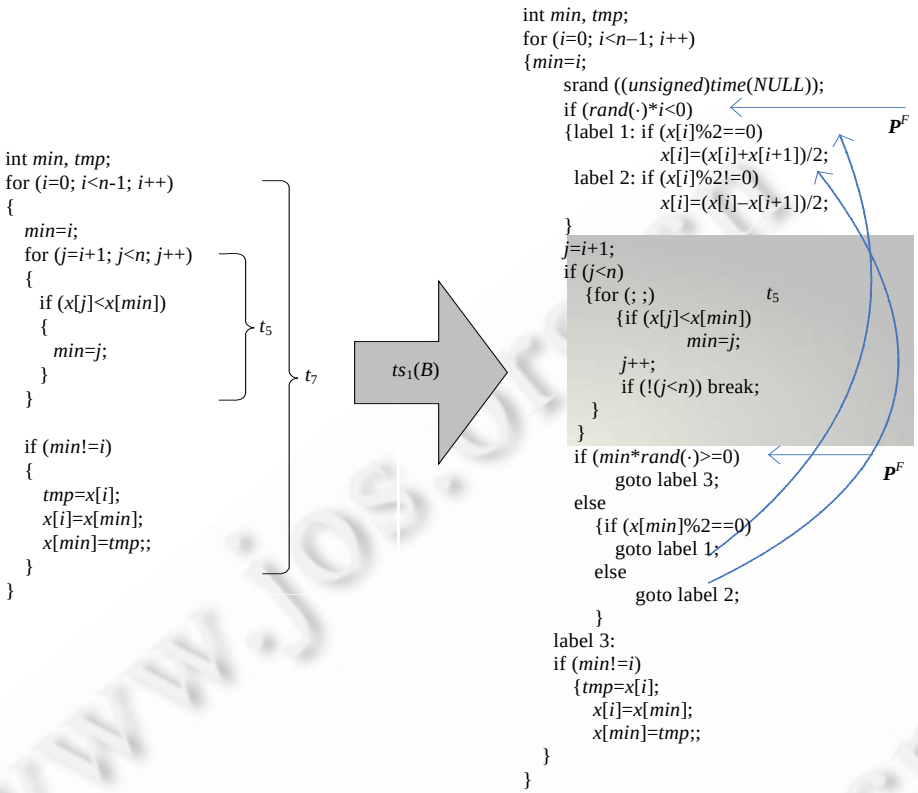
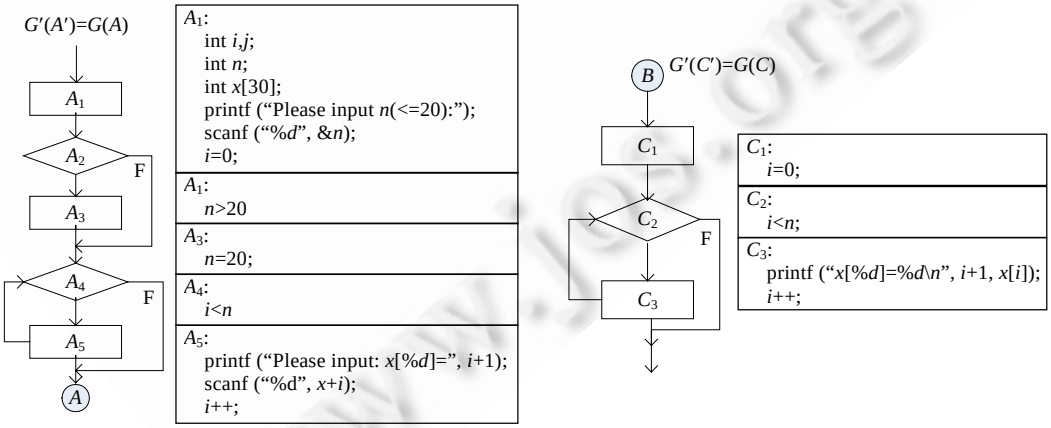


Fig.4 Obfuscated Part B in program I by $t_7(t_5(B))$

图 4 程序 I 中 Part B 的 $t_7(t_5(B))$变换过程



(a) 输入功能 $G'(A')$ (b) 输出功能 $G'(C')$

Fig.5 CFG for input function $G'(A')$ and output function $G'(C')$ in the obfuscated Program I

图 5 程序 I 迷惑变换后输入 $G'(A')$与输出 $G'(C')$部分对应的 CFG

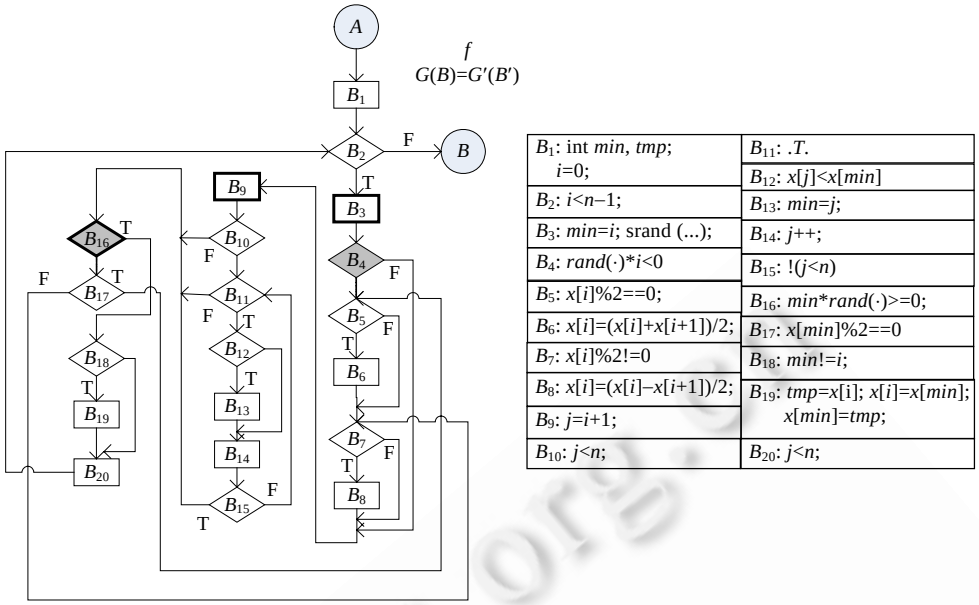


Fig.6 CFG for the sorting function $G'(B')$ in the obfuscated Program I

图 6 程序 I 的 CFG:排序功能 $G'(B')$

Table 3 Elementary data for $SCM(G')$ in scheme (I)

表 3 迷惑变换方案(I)中用于 $SCM(G')$模型计算的基础数据

v_i		$D(v_i)$		$DF(v_i)$		NL	Type	v_i		$D(v_i)$		$DF(v_i)$		NL	Type
No.	Name	D_{in}	D_{out}	NID	NOD			No.	Name	D_{in}	D_{out}	NID	NOD		
1	A_1	1	1	1	2	1	*S*	15	B_{10}	1	2	1	1	2	*CB*
2	A_2	1	2	2	0	1	*CB*	16	B_{11}	2	2	1	0	3	*LB*
3	A_3	1	1	1	1	2	*S*	17	B_{12}	1	2	2	0	4	*CB*
4	A_4	2	2	2	0	1	*LB*	18	B_{13}	1	1	1	1	5	*S*
5	A_5	1	2	3	2	2	*S*	19	B_{14}	2	1	2	1	4	*S*
6	B_1	1	1	1	1	1	*S*	20	B_{15}	1	2	1	1	4	*CB*
7	B_2	1	2	3	0	1	*LB*	21	B_{16}	3	2	3	0	2	*CB*
8	B_3	1	1	2	2	2	*S*	22	B_{17}	1	2	3	0	3	*CB*
9	B_4	1	2	3	0	2	*CB*	23	B_{18}	1	2	2	0	2	*CB*
10	B_5	2	2	3	0	3	*CB*	24	B_{19}	1	1	3	3	3	*S*
11	B_6	1	1	3	1	4	*S*	25	B_{20}	2	1	2	1	2	*S*
12	B_7	3	2	3	0	3	*CB*	26	C_1	1	1	1	1	1	*S*
13	B_8	1	1	3	1	4	*S*	27	C_2	2	2	2	0	1	*LB*
14	B_9	3	1	2	1	2	*S*	28	C_3	1	2	3	1	2	*S*

Table 4 Local SCM for obfuscating scheme (I)

表 4 执行方案(I)后代码块的 SCM

CFG	RD	DF	NB	LI	*n*	SCM
$G'(A')$	10	21	2	3	5	4.71
$G'(B')$	47	163	11	75	20	9.29
$G'(C')$	7	12	1	3	3	4.75

根据上面的方式,我们再对方案(II)~方案(IV)分别执行相关计算,可以得到相应的图形、表格、计算结果.限于篇幅,这里只列出它们的汇总数据,见表 5.

Table 5 Summary data for scheme (I)~(IV)

表 5 方案(I)~方案(IV)仿真数据汇总

Code block	Scheme	CFG	RD	DF	NB	LI	*n*	SCM	TOM	$E(\boldsymbol{PS}_i')$	SortID
Local code	I~IV	$G(A)$	10	21	2	3	5	4.71	—	—	—
		$G(C)$	7	12	1	3	3	4.75	—	—	—
Local code	I	$G'(B')$	47	163	11	75	20	**9.29**	**2.19**	0.318	2
	II	$G'(B')$	28	122	8	71	14	**10.29**	**2.44**	0.353	1
	III	$G'(B')$	24	106	6	50	12	**9.98**	**1.43**	0.250	3
	IV	$G(B)$	19	66	4	24	10	**7.29**	**0**	0.079	4
Globe code	I	G'(Prg.I′)	64	196	14	81	28	**7.93**	**2.19**	0.323	2
	II	G' (Prg.I′)	45	155	11	77	22	**8.14**	**2.44**	0.350	1
	III	G' (Prg.I′)	41	139	9	56	20	**7.78**	**1.43**	0.245	3
	IV	G (Prg.I)	3	99	7	30	18	**6.1**	**0**	0.081	4

5.4 仿真结果

表 5 汇总了第 5.3 节中 4 种迷惑变换方案主要的仿真数据.

本文所提出的 SCM 既可以应用于软件局部,也可以应用于整个软件.表 5 中浅灰色背景的数据(第 3 行~第 5 行)来自于模型应用于程序 I 的 Part *B* 部分后产生的数据,当所提出的模型应用于整个软件时,生成的汇总数据如表 5 全局代码块(globe code)指示的区域所示(第 6 行~第 8 行).表 5 中的"—"表示模型在对应处不产生数据.因 4 种仿真方案都是针对程序 I 的 Part *B*,而 Part *A* 与 Part *C* 对应的子控制流图在这 4 次仿真中保持不变,其对应的汇总数据见表 5 中头两行.

针对程序 I 的 Part *B*,应用第 5.3 节设计的 4 种变换方案后,经模型 SCM 与 TOM 计算,分别产生 4 个力矢量(参考附录 C):$\boldsymbol{PS}_1$=(9.29,2.19),$\boldsymbol{PS}_2$=(10.29,2.44),$\boldsymbol{PS}_3$=(9.98,1.43),$\boldsymbol{PS}_4$=(7.29,0),其中,$\boldsymbol{PS}_1$ 的生成过程在第 5.3 节已给出说明,而后 3 个矢量的构建过程与 $\boldsymbol{PS}_1$ 相同.然后按照第 4.2 节的方法,对这 4 个矢量进行综合评价,结果见表 5 中 $E(\boldsymbol{PS}_i')$ 列.SortID 列说明迷惑变换方案鲁棒性的优先顺序为 $ts_2>ts_1>ts_3>ts_4$(关于鲁棒性的度量请参考本文第 4 节).

类似地,当提出的 SCM 模型应用于整个程序 I 时,也能产生相关数据,如表 5 中 Globe code 指示的区域所示.与应用在软件局部相比,当 SCM 作用在整个软件时,表现的是相关参数值的增大与节点数量的增加,由于 SCM 模型计算的是平均节点度量值,因此不同方案的 SCM 值差距缩小了,但 SCM 值排序关系与方案应用在局部时是一致的.如表 5 中 SCM 列所示,从第 7 行~第 10 行的数值排序与从第 3 行~第 6 行的数值排序一致.此外,TOM 刻画的是迷惑导致的局部代码理解难度,当 TOM 应用于全局代码时,模型的输出值仍等于相应的局部值,这是因为程序 I 的 Part *A* 与 Part *C* 未施加迷惑变换操作.因而,如果用全局代码数据参与 SCM 与 TOM 计算,不同方案评价值的排序与局部应用时一致,仍有 $ts_2>ts_1>ts_3>ts_4$,如表 5 中 SortID 列从第 7 行~第 10 行的数值所示.

由于软件中的知识产权或秘密算法主要集中在代码的局部,而且当模型应用在软件局部时,不同方案的 SCM 值的差距高于全局应用时的差距,更能体现不同迷惑方案鲁棒性的强弱.因此,我们认为模型的局部应用更符合实际,这样也能降低模型的计算开销.

6 总 结

对软件知识产权保护的重视,使得软件迷惑变换技术发展迅速.然而,不同的迷惑变换对软件的保护能力各不相同,有些差异非常大.为了抵制软件被低价反向工程,如何选择具体的迷惑变换技术显得非常重要.以往对迷惑变换软件鲁棒性度量的研究大都是在定性方面,对定量方面的研究非常少,目前,仍缺乏一种统一、实用的量化评价框架,不利于比较同类迷惑变换方案的优劣,影响了迷惑变换技术的有效应用.

本文提出了一种选择迷惑变换方案的方法,该方法从软件系统复杂性与变换模糊性两个层面来共同刻画迷惑变换带给软件的鲁棒性.

我们所提出的系统复杂度模型实现了以下目标:

(1) 从复杂性代表理解的困难性角度建立模型;

(2) 以信息的多样性表示系统的复杂性,模型包含的软件信息有:软件结构、信息流、分支、循环以及元素的嵌套层次;

(3) 模型既能反映全局复杂性,也能用于局部代码复杂性度量,以增强应用的灵活性;

(4) 用基本代码块的平均度量值代替以往的软件总量复杂性,来刻画软件的稳健度.所建立的模型能够更准确地从复杂性刻画理解的困难性.

提出的迷惑变换模糊度模型,能够满足绝大多数变换技术复杂的应用场合.我们首先根据管理学中的综合评价法建立了基于多位专家指标评分法的单种变换模糊度形式化模型,这是变换模糊度模型的基础;然后建立多种变换线性组合应用和复合应用的模糊度形式化模型;在此基础上,针对复杂的多种变换应用情形,给出了变换综合应用模糊度算法.

此外,本文还给出了一些例子便于对模型的理解,实例仿真全面展示了模型的工作过程;提出的相关算法有利于自动评价工具的开发.

评价软件迷惑变换的鲁棒性因涉及到代码理解问题,度量工作非常困难,目前还不存在适用面广、完全客观的评价方法.本文所提出的模型是一种适用面广的量化评价模型,也是一种尽力的客观评价.因变换模糊度模型中的单种变换模糊度评价采用了专家指标评分法,存在部分主观性,其结果的准确性与收集专家数据的多少正相关.如何完全客观地度量单种变换模糊度,将是下一步的研究工作.

References:

[1] Wang YB, Chen YY. Progress of research on code obfuscation technology. Journal of Jilin University (Information Science Edition), 2008,26(4):386−390 (in Chinese with English abstract).

[2] Collberg C, Thomborson C, Low D. A taxonomy of obfuscating transformations. Technical Report, 148, Auckland: University of Auckland, 1997.

[3] Mauri AR, Williams AH. Extending Halstead's software science for a more precise measure of APL. In: Proc. of the Int'l Conf. on APL (APL'82). New York: ACM Press, 1982. 207−213. [doi: 10.1145/390006.802245]

[4] McCabe TJ. A complexity measure. IEEE Trans. on Software Engineering, 1976,2(4):308−320. [doi: 10.1109/TSE.1976.233837]

[5] Woodward M, Hennel M, Hedley D. A measure of control flow complexity in program text. IEEE Trans. on Software Engineering, 1979,SE-5(1):45−50. [doi: 10.1109/TSE.1979.226497]

[6] Henry SM, Kafura D. Software structure metrics based on information flow. IEEE Trans. on Software Engineering, 1981,SE-7(5): 510−518. [doi: 10.1109/TSE.1981.231113]

[7] Harrison W, Magel K. A complexity measure based on nesting level. SIGPLAN Notices, 1981,16(3):63−74. [doi: 10.1145/947825.947829]

[8] Goto H, Mambo M, Matsumura K, Shizuya H. An approach to the objective and quantitative evaluation of tamper-resistant software. In: Proc. of the 3rd Int'l Workshop on Information Security (ISW 2000). Berlin, Heidelberg: Springer-Verlag, 2000. 82−96. [doi: 10.1007/3-540-44456-4_7]

[9] Udupa S, Debray S, Madou M. Deobfuscation: Reverse engineering obfuscated code. In: Proc. of the 12th Working Conf. on Reverse Engineering. Los Alamitos: IEEE Computer Society, 2005. 45−56. [doi: 10.1109/WCRE.2005.13]

[10] Anckaert B, Madou M, Sutter DC, Bus DB, Bosschere DK, Preneel B. Program obfuscation: A quantitative approach. In: Proc. of the 14th ACM Computer and Communications Security Conf. Alexandria: ACM Press, 2009. 15−20. [doi: 10.1145/1314257.1314263]

[11] Tsai HY, Huang YL, Wagner D. A graph approach to quantitative analysis of control-flow obfuscating transformations. IEEE Trans. on Information Forensics and Security, 2009,4(2):257−267. [doi: 10.1109/TIFS.2008.2011077]

[12] Ceccato M, Di Penta M, Nagra J. The effectiveness of source code obfuscation: An experimental assessment. In: Proc. of IEEE the 17th Int'l Conf. on Program Comprehension (ICPC 2009). Vancouver: IEEE, 2009. 178−187. [doi: 10.1109/ICPC.2009.5090041]

[13] Sun DC, Lin FY. Systems Engineering Introduction. Beijing: Tsinghua University Press, 2004. 192−194 (in Chinese).

[14] Hou T, Chen H, Tsai M. Three control flow obfuscation methods for Java software. IEE Proc. Software, 2006,153(2):80–86. [doi: 10.1049/ip-sen:20050010]

[15] Wang CX, Hill J, Knight J, Davidson J. Software Tamper Resistance: Obstructing Static Analysis of Programs. Charlottesville: University of Virginia, 2000.

附中文参考文献:

[1] 王一宾,陈意云.代码迷惑技术研究进展.吉林大学学报(信息科学版),2008,26(4):386–390.

[13] 孙东川,林福永.系统工程引论.北京:清华大学出版社,2004.192–194.

附录 A. SCM 计算实例

普通的 CFG 生成器算法经过第 2.4 节的改进,并在接受源代码输入后,可以自动输出计算 SCM 的所有基础参数数据.例如,对第 2.1 节给出的那段 C 语言代码,经过改进的算法(见第 2.4 节)处理后,能够生成如图 1 所示的 CFG,以及 SCM 相关参数见表 6.

图 1 中代表节点数的变量 n=5,其中有 3 个顺序节点(S_1,S_2,S_3,*type*=*S*),1 个条件分支节点(B_1,*type*=*CB*)和 1 个循环分支节点(B_2,*type*=*LB*),见表 6.对节点 B_2,分别有两条弧线:射入和射出,故在表 6 中,D_{in},D_{out} 列的相应行处分别为 2.因 B_2 的代码为"*i*<*n*",此语句读取两个不同的变量(NID),改写 0 个变量(NOD).根据第 2.4 节的计算方案,B_2 的嵌套层次 *NL*=1.此外,图 1 中只有一个循环(B_2,S_3,B_2).根据表 6 中 5 个节点的基础数据,按照公式(1)~公式(4),分别计算出 *RD*=10,*DF*=21,*NB*=2,*LI*=3.生成的全局复杂度矢量 $\boldsymbol{C}(\boldsymbol{G})$=(10,21,2,3),根据公式(5),CFG 中节点平均复杂度为$\|C_{avg}(G)\|$=4.71,即 *SCM*(*C*)=4.71.

Table 6 Parameters from the improved CFG algorithm for the C code in Section 2.1

表 6 计算 SCM 的基础数据(见第 2.1 节 C 代码)

v_i		$D(v_i)$		$DF(v_i)$		NL	Type
No.	Name	D_{in}	D_{out}	NID	NOD		
1	S_1	1	1	1	2	1	*S*
2	B_1	1	2	2	0	1	*CB*
3	S_2	1	1	1	1	2	*S*
4	B_2	2	2	2	0	1	*LB*
5	S_3	1	2	3	2	2	*S*
Result		*n*=5		LPL_i:{(B_2,S_3,B_2)}			
		RD=10		*DF*=21	*NB*=2		*LI*=3
		SCM(*C*)=$\|C_{avg}(G)\|$=4.71					

附录 B. TOM 计算实例

本附录内容以图 3 为例,详细地展示 TOM 的工作过程,以助于理解第 3 节的内容.

设有迷惑变换集 T={t_1,t_2,…,t_8},对一段给定的代码 *C* 施加图 3 所示的迷惑变换方案 ts_b.此外,假定已收集多位专家对变换集 *T* 的打分结果并对其汇总,数据见表 7.关于表 7 的形成过程,参考本文第 3.1 节.按照第 3.3 节提出的算法,对 *TOM*(ts_b(*C*))进行如下计算:

根据算法的第 1 步,可得出代码 *C*(如图 3 所示)的最外层变换序列 T_{out}={t_4,t_5,t_1},根据 T_{out} 便可求出最大变换代码块集合 A={A_1,A_2,A_3}.

执行第 2 步的计算,可形成如下复合变换表达式:

$$T(A_1)=t_4(t_2(t_3(t_4),t_5),t_6),T(A_2)=t_5(t_2,t_3(t_4)),T(A_3)=t_1.$$

扫描 $T(A_1)$,$T(A_2)$,$T(A_3)$后得到如下计算式(算法第 3 步):

$$O(T(A_1))=0+O(t_4)\times(1+x_{42})+O(t_2)\times(1+x_{24})+O(t_3)\times(1+x_{32})+O(t_4)\times(1+x_{43})+O(t_5)\times(1+x_{52})+O(t_6)\times(1+x_{64}),$$

其中,$O(t_i)$与 x_{jk} 的结果已列在表 7 中,例如 $O(t_1)$=8.3,$O(t_2)$=13;x_{42}=0.3,x_{64}=0.6.这样,便可得出 $O(T(A_1))$=100.34.同样的方式可计算 $O(T(A_2))$,$O(T(A_3))$:

$O(T(A_2))=0+O(t_5)\times(1+x_{52})+O(t_2)\times(1+x_{25})+O(t_3)\times(1+x_{35})+O(t_4)\times(1+x_{43})=64.43$,

$O(T(A_3))=0+O(t_1)=8.3$.

因此,

$$O(T(A))=O(T(A_1))+O(T(A_2))+O(T(A_3))=173.07.$$

Table 7 Comprehensive evaluation data for set T

表 7 专家对集合 T 的综合评价结果

x_{ij}	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	$O(t_i)$
t_1	**0.5**	0.4	0.3	0.55	0.4	0.3	0.1	0.1	**8.3**
t_2	0.6	**0.5**	0.6	0.7	0.5	0.55	0.4	0.3	**13**
t_3	0.7	0.4	**0.5**	0.5	0.4	0.3	0.2	0.1	**9.7**
t_4	0.45	0.3	0.5	**0.5**	0.5	0.4	0.2	0.2	**9.5**
t_5	0.6	0.5	0.6	0.5	**0.5**	0.35	0.3	0.3	**11.4**
t_6	0.7	0.45	0.7	0.6	0.65	**0.5**	0.3	0.3	**13.1**
t_7	0.9	0.6	0.8	0.8	0.7	0.7	**0.5**	0.45	**17**
t_8	0.9	0.7	0.9	0.8	0.7	0.7	0.55	**0.5**	**18**

参照第 3.3 节中第 4 步计算方法,在 $A=\{A_1,A_2,A_3\}$经过应用 ts_b 方案后,对应的代码块集合为 $A'=\{A_1',A_2',A_3'\}$. 假定 A'的 CFG 中节点总数是 50,即 $N(G'(A'))=50$,则有

$$\boldsymbol{TOM}(ts_b(C))=O(T(A))/N(G'(A'))=173.07/50=3.46.$$

附录 C. 变换鲁棒性度量实例

对一段给定的代码 C,设有 4 种候选迷惑方案(ts_1,ts_2,ts_3,ts_4)构成迷惑方案集 TS,如表 8 中的第 1 列所示.经过模型 SCM 与 TOM 计算后形成 4 个保护力矢量 $\boldsymbol{PS}_i(i=1,2,3,4)$,如表 8 中第 2 列所示.根据第 4.2 节中计算过程的第 1 步,产生 $\boldsymbol{PS}_i$ 对应的标准化数据矢量 $\boldsymbol{PS}_i'$,如表中第 3 列所示.然后,按照第 4.2 节中计算过程的第 2 步计算得分 $E(\boldsymbol{PS}_i')$,如表 8 中第 4 列所示.最后,对表 8 中第 4 列数据降序排列,生成排序编号,显示在 SortID 列.由此可得出结论,针对代码 C 迷惑方案鲁棒性的排序为 $ts_2>ts_1>ts_3>ts_4$,即方案 2 为首选方案,其鲁棒性最好.

Table 8 An example for the sort of obfuscating schemes

表 8 方案选择实例

TS	$\boldsymbol{PS}_i$	$\boldsymbol{PS}_i'$	$E(\boldsymbol{PS}_i')$	SortID
ts_1	(9.29,2.19)	(0.252,0.361)	0.318	2
ts_2	(10.29,2.44)	(0.279,0.403)	0.353	1
ts_3	(9.98,1.43)	(0.271,0.236)	0.250	3
ts_4	(7.29,0)	(0.198,0)	0.079	4

付剑晶(1976－),男,江西泰和人,博士,副教授,主要研究领域为信息安全,网络安全,数字水印.

E-mail: fjjmsn@163.com

王珂(1964－),男,博士,教授,博士生导师,主要研究领域为遥感信息技术,信息提取,资源环境遥感监测、评价与规划.

E-mail: kwang@zju.edu.cn