软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(6):1778−1792 [doi: 10.13328/j.cnki.jos.005429]

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

基于指令交换的代码混淆方法*

潘 雁, 祝跃飞, 林 伟

(数学工程与先进计算国家重点实验室,河南 郑州 450001)

通讯作者: 潘雁, E-mail: z11pany@qq.com

摘 要: 软件程序是按一定顺序排列的指令序列,指令的排列组合构成了千变万化的程序语义.指令顺序重排通常会相应地导致程序语义的变化,通过分析相邻指令序列的相对独立性,可以在不影响程序语义的前提下交换相邻指令序列,增大指令距离,改变程序特征,在一定程度上增加逆向分析代价.通过改进程序的形式化定义论证相邻指令交换的充分条件,采用模拟退火算法实现随机化的指令乱序混淆方法,并将指令乱序方法与虚拟机代码保护技术融合,实现基于指令乱序的虚拟机代码保护系统 IS-VMP,使用加密算法实例进行系统测试,验证了指令乱序混淆算法的可行性与有效性.

关键词: 指令交换;代码混淆;语义等价;虚拟机保护;模拟退火

中图法分类号: TP311

中文引用格式: 潘雁,祝跃飞,林伟.基于指令交换的代码混淆方法.软件学报,2019,30(6):1778−1792. http://www.jos.org.cn/1000-9825/5429.htm

英文引用格式: Pan Y, Zhu YF, Lin W. Code obfuscation based on instructions swapping. Ruan Jian Xue Bao/Journal of Software, 2019,30(6):1788−1792 (in Chinese). http://www.jos.org.cn/1000-9825/5429.htm

Code Obfuscation Based on Instructions Swapping

PAN Yan, ZHU Yue-Fei, LIN Wei

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

Abstract: The program is a sequence of instructions in a certain order, and the permutation and combinations of instructions constitute the ever-changing program semantics. Although reordering instructions usually changes the program semantics, it is possible to swap adjacent instruction sequences without changing the program semantics via analyzing the relative independence of adjacent instruction sequences. Instructions swapping increases the distance of instructions and change characteristics of the program, which raises the cost of reverse analysis to a certain extent. Sufficient conditions of instructions swapping are proven by the improvement of the formal definition of the program, upon which the randomize method of instructions reordering based on simulated annealing is proposed in the study. Furthermore, a prototype of IS-VMP (virtual machine protection system based on instructions reordering) is implemented. In addition, the experiments are carried out with a set of encryption algorithms. Experiment results show that instruction reordering is effective and applicable for anti-reversing.

Key words: instructions swapping; code obfuscation; semantic equivalence; virtual machine protection; simulate anneal

据 2016 年软件联盟(the software alliance,简称 BSA)发布的全球软件调查显示,2015 年,全球范围内有高达 39%的已安装软件未经合法授权,这些非法授权软件导致的损失高达几千亿美元.与此同时,代码逆向分析技术的发展,使得软件保护技术的不断发展与创新尤为重要.

Collberg 于 1997 年提出通过将源代码中逻辑相近的变量、数组等进行物理分离[1],物理分离的思想迅速被

* 基金项目: 国家重点研发计划(2016YFB08011601)
Foundation item: National Key Research and Development Program of China (2016YFB08011601)
收稿时间: 2017-06-26; 修改时间: 2017-08-28, 2017-09-22; 采用时间: 2017-10-19

应用于二进制代码混淆算法中.Wroblewski 提出了指令与指令序列乱序的思想[2],但其给出乱序的充分条件不够准确;Birrer 等人提出了基于程序切分的代码混淆方法,将代码块切分成多个指令片段,并将其随机打乱分布在物理地址中,利用跳转地址表保存切片的保存顺序[3].而在工程实现中,指令乱序是通过无条件跳转的 jmp、变形的短跳转 call 等连接指令片段,能较大程度影响逆向分析人员的逻辑.但随着自动化逆向分析工具的不断发展,Ollydbg 等工具的 Trace 功能能够较好地还原执行流程,辅助逆向人员进行分析.其中,指令序列交换的思想被多次提出[2,4],但是未进行系统化地分析与实现,直至 2012 年,z0mbie 提出了基于 X86 指令架构的对象集的概念,开发了 XDE 反汇编引擎分析 X86 指令的对象集,并简单探讨了相邻指令交换的条件,但是不够完善和准确.

软件程序是由指令构成的序列,序列不同使得程序语义不同,代码混淆即通过将指令替换、乱序、膨胀等实现与原始程序相同的语义.本文的指令乱序是通过改变一些物理相邻但相互逻辑独立的指令执行序列来混淆原始程序,为了区分传统的指令乱序方法,本文将传统方法称为代码切分乱序.X86 指令序列中,存在部分独立的指令或运算逻辑可交换的指令,即相邻的指令互换后不影响序列语义,其既与程序编写思路又与编译器编译优化相关.通过迭代交换指令,尽可能地将相邻指令距离增大,也即通过物理分离逻辑.

本文基于 Wroblewski 构建的计算机架构的形式化定义并予以改进,论证相邻指令序列交换的充分条件;在分析 X86 指令的基础上改进 XDE 反汇编引擎,实现了相邻指令交换充分条件的判断;同时,以基本块内的指令序列为对象实现了指令乱序算法,使得每次生成的二进制代码具有随机性,且能较大程度地保持与原始程序的差异性.更进一步,将指令乱序混淆算法应用于虚拟机代码保护技术,对虚拟机解释函数进行随机乱序,增强其随机性,以此为基础设计,并实现了改进型虚拟机软件保护系统 IS-VMP(VM-based protection with instructions swapping).

本文的主要贡献在于:

(1) 改进 Wroblewski 提出的形式化定义,论证相邻指令序列交换的充分条件;

(2) 改进 XDE 反汇编引擎,采用随机化算法实现基本块内的指令乱序,通过实验验证其可行性及其效果,为代码克隆提供了一种新的自动化实现方法;

(3) 对虚拟机解释函数进行指令乱序,实现两种混淆算法的融合,增强了虚拟机保护技术的随机性,并通过实验验证了其可行性及其抗逆向分析的效果.

1 相关工作

现阶段软件保护的方法主要为源码混淆和二进制代码混淆.对于软件保护者,源代码是难以获得的,为了更好地实现功能与混淆的剥离,研究者更多地将精力集中于二进制代码混淆.但由于逆向分析技术及工具的发展,使得传统的代码混淆算法有效性大大降低.虚拟机代码保护技术的出现,使得代码保护进入了一个新的阶段,也被认为是未来发展的主要方向.

虚拟机代码保护系统首先将 PE 文件反汇编为 X86 指令流,并从中提取目标指令(KeyCode)序列,而后将目标汇编指令流转换成字节码(ByteCode).转换之后,PE 文件中被保护代码的正确运行需要虚拟机解释器对字节码解释执行,因此,整个虚拟机其实是内嵌在 PE 文件的,它包括跳转表、虚拟指令调度器(dispatcher)、字节码和虚拟指令解释函数(handler),各模块名称及意义分别如下.

- 目标指令:目标指令为待保护程序中被保护的 X86 指令序列;
- 字节码:虚拟机系统定义的一套指令构成的指令序列;
- 虚拟指令解释函数:用于解释字节码的 X86 指令序列;
- 虚拟指令调度器:调度虚拟指令解释函数的执行程序;
- 跳转表:虚拟指令解释函数与字节码的对应关系;
- 虚拟机上下文(VMcontext):用于存储真实寄存器、虚拟寄存器、跳转表的结构.

虚拟机保护机制如图 1 所示,具体保护步骤如下.

Step 1. 提取待保护程序 *P* 中使用 SDK 标识的 KeyCode;

Step 2. 将 KeyCode 转化成 ByteCode;

Step 3. 构造 Handler 集合和跳转表;

Step 4. 重建可执行程序文件,将 VMContext,ByteCode,Handlers,Dispatcher 重新构成新节或加至最后一节,并在 KeyCode 指令处填充垃圾代码.

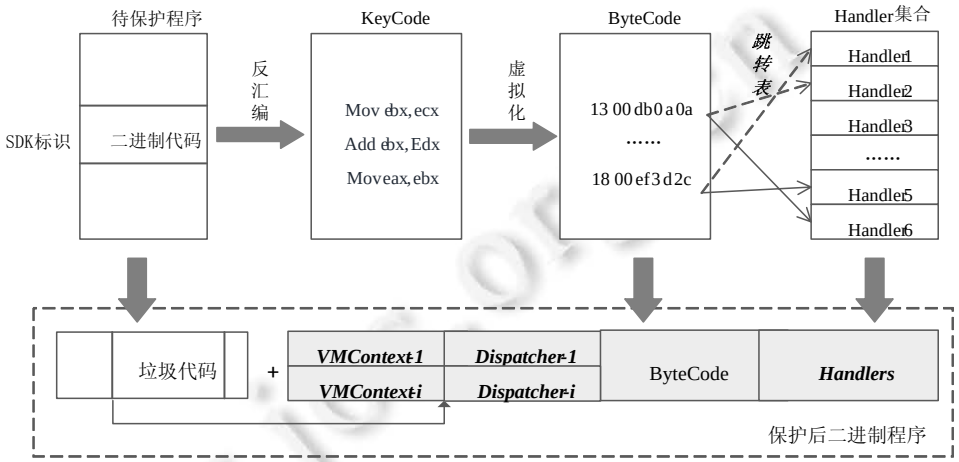


Fig.1 Mechanism of VM-based code protection

图 1 虚拟机代码保护机制

随着逆向分析技术的发展以及逆向分析者对虚拟机结构、解释函数的持续分析,虚拟机代码保护技术也逐渐被攻破[5−7].与此同时,正向保护研究人员不断将传统代码混淆技术与虚拟机代码保护技术进行融合:房鼎益等人提出通过设计数据流混淆引擎对 Handler 进行数据流混淆,增大数据流结构的复杂性[8];Sebastian 等人提出添加依赖于程序输入的分支指令,以增大符号执行的难度[9];谢鑫等人对所有 Handler 进行变长切分和随机乱序[10];吴伟明等人借鉴传统混淆技术,提出虚拟花指令序列与虚拟指令模糊变换技术,对虚拟机的虚拟指令系统做了改进[11].都在一定程度上加强了对关键代码的保护.

除此之外,研究人员基于虚拟机代码保护技术的结构,提出将随机化、动态多样化思想应用于虚拟机代码保护技术,如图 1 中字体为斜体的模块,其中包括:

(1) Handler 多样化:采用代码克隆、语义等价等方法生成多个形态不同但语义等价的 Handler 序列,来增强多 handler 序列间的差异性,并通过随机选择路径以抵御累积攻击[6,12,13];

(2) 虚拟寄存器多样化:使用多个虚拟寄存器代替单个,模糊虚拟寄存器与主机寄存器的关系,增大数据流分析难度[14];

(3) 指令集随机化:每次执行保护时打乱字节码与解释函数的对应关系,使得不同的保护程序中字节码语义不同,延长攻击者的分析时间[15,16];

(4) 调度器多样化:使用多个虚拟机调度器,将单一的循环调度结构扩充为循环结构与链式结构混合调度结构[14].

本文研究基本块内的指令乱序,并将此随机化因素添加到Handler构造中,尽可能地减少攻击者分析同一个虚拟机代码保护系统保护的不同软件时所拥有的先验知识.

2 指令序列交换的基础理论

X86 架构下的软件程序是由 X86 指令集中的指令构成的序列,千变万化的指令序列构成了不同的程序语义,代码混淆即通过指令序列替换、乱序、膨胀等实现与原始程序相同的语义.本文通过对基本块中的指令进行分析,在仅改变指令序列顺序的情况下保持原始程序语义,增加序列的差异性.在详细描述指令序列交换的算

法之前,先给出相关定义.

2.1 相关定义

Wroblewski 构建了基于系统指令的形式化定义[2],此形式化定义使用数学定义:集合、变量、函数、笛卡尔积、向量等描述代码语义.为了更好地描述基于 X86 系统指令的混淆算法,本文将其定义的外沿缩小为 X86 系统,并由此给出相关的形式化定义、定理与相关推论,表 1 是该定义中使用到的相关符号.

Table 1 Description of used notation

表 1 符号描述表

符号	实例	定义
大写字母	W	X86 指令影响的寄存器值、内存值的集合
大写粗体字母	$\boldsymbol{S}$	$\boldsymbol{S}=W_1\times W_2\times\ldots\times W_N$,寄存器值、内存值集合构成的状态空间
小写粗体字母	$\boldsymbol{v}$	向量 $\boldsymbol{v}\in\boldsymbol{S}$,是集合 $\boldsymbol{S}$ 中的元
小写字母+上下标	$\boldsymbol{v}_j^i$	某个 N 维向量 $\boldsymbol{v}_j$ 的第 i 个分量
×	$W_1\times W_2$	笛卡尔积运算
α	α	描述向量 $\boldsymbol{v}$ 中不重要的元素
大写字母+(…)	$I(\boldsymbol{v})$	N 维向量 $\boldsymbol{v}$ 经过指令 I 执行后的结果,仍为 N 维向量
大写字母+下标	$I_1\|I_2$	指令序列:I_1,I_2;也即组合 $I_2(I_1(\boldsymbol{v}))$
等号+上标	$\boldsymbol{v}_1\overset{!i}{=}\boldsymbol{v}_2$	除了第 i 个分量不等,N 维向量相等:$\boldsymbol{v}_1=\boldsymbol{v}_2$,等价于 $\bigwedge_{k\neq i}\boldsymbol{v}_1^k=\boldsymbol{v}_2^k\wedge\boldsymbol{v}_1^i\neq\boldsymbol{v}_2^i$

2.1.1 基础定义

X86 体系架构是冯·诺依曼体系架构中应用极为广泛的一种,其核心思想在于通过二进制指令执行处理系统存储的数据,其核心的两个要素为指令与数据.因此,可以进行如下定义.

定义 2.1(计算机体系架构 $\boldsymbol{A}(\boldsymbol{I},\boldsymbol{S})$). 在计算机体系架构 $A(\boldsymbol{I},\boldsymbol{S})$中,$\boldsymbol{I}=\{I_1,I_2,\ldots,I_M\}$为指令集合,指令实际为映射 $\boldsymbol{I}:\boldsymbol{S}\to\boldsymbol{S}$,其中,$\boldsymbol{S}=\{(\boldsymbol{v}^1,\boldsymbol{v}^2,\ldots,\boldsymbol{v}^N)|\boldsymbol{v}^1\in W_1,\ldots,\boldsymbol{v}^N\in W_N\}=W_1\times W_2\times\ldots\times W_N$ 是 $\boldsymbol{v}^1,\boldsymbol{v}^2,\ldots,\boldsymbol{v}^N$ 所有可能值构成的向量空间.

定义 2.2(程序 $\boldsymbol{P}$). 程序 P 是指令的有序集合,记做 $P=\{I_{i_1},I_{i_2},...,I_{i_t}\,|\,I_{i_t}\in I\}$,也常记做 $P=I_{i_1}\,|\,I_{i_2}\,|...|\,I_{i_t}$.即程序 P 为复合映射 $I_{i_t}\cdot I_{i_{t-1}}\cdot....\cdot I_{i_1}:\boldsymbol{S}\to\boldsymbol{S}$.

本文中,$\boldsymbol{S}$ 是指令操作所依据和影响的对象构成的向量空间,简称对象空间,包含所有通用寄存器、标志寄存器和内存引用标志等.以 X86 体系架构的 32 位指令为例,其对象空间即为 $\boldsymbol{S}=EAX\times EBX\times ECX\times EDX\times ESI\times EDI\times ESP\times EBP\times flags\times memory$,是一个 10 维向量空间.

定义 2.3(输入对象空间 $\boldsymbol{S_I}(\boldsymbol{P})$). 输入对象空间是一段程序 P 操作所依据或读取的所有对象的向量空间,记为 $\boldsymbol{S}_I(P)=V_1\times V_2\times\ldots\times V_N$,其中,

$$V_i=\begin{cases}W_i, & \text{if } V_{\boldsymbol{v}_1\overset{!i}{=}\boldsymbol{v}_2\in\boldsymbol{S},k\neq i}(I(\boldsymbol{v}_1)^k\neq I(\boldsymbol{v}_2)^k\vee I(\boldsymbol{v}_1)^i\neq\boldsymbol{v}_1^i)\\ \{\alpha\}, & \text{otherwise}\end{cases}.$$

当输入的第 i 个分量经过程序 P 执行后影响了某个分量的输出,或者输出的第 i 分量与输入的第 i 个分量不等,则认为其在输入对象空间重要,否则认为该分量是不重要的,即为$\{\alpha\}$.

定义 2.4(输出对象空间 $\boldsymbol{S_O}(\boldsymbol{P})$). 输出对象空间是一段程序 P 执行后影响或写入的所有对象,记做 $\boldsymbol{S}_O(P)=V_1\times V_2\times\ldots\times V_N$,其中,

$$V_i=\begin{cases}W_i, & \text{if } V_{\boldsymbol{v}_1\in\boldsymbol{S}}(I(\boldsymbol{v}_1)^i\neq\boldsymbol{v}_1^i)\\ \{\alpha\}, & \text{otherwise}\end{cases}.$$

如果第 i 个分量的输入/输出相同,则认为其在输出对象空间中是不重要的.

例 2.1:假设计算机的状态空间为 $\boldsymbol{S}=W_1\times W_2\times W_3$,假设一段程序为如下映射:

$$P:(\boldsymbol{v}^1,\boldsymbol{v}^2,\boldsymbol{v}^3)\to(\boldsymbol{v}^1+\boldsymbol{v}^2,\boldsymbol{v}^2,\boldsymbol{v}^3).$$

其中,$\boldsymbol{v}^1\in W_1,\boldsymbol{v}^2\in W_2,\boldsymbol{v}^3\in W_3$.简单的,可以得到该程序的输入对象空间和输出对象空间:

$$\boldsymbol{S}_I(P)=W_1\times W_2\times\{\alpha\},\boldsymbol{S}_O(P)=W_1\times\{\alpha\}\times\{\alpha\}.$$

根据上述定义,当程序为单条指令时,可以得到指令的输入/输出对象空间.本文将 z0mbie 提出的 X86 指令的对象集进行修正,为了简洁,将对象空间简化后结果见表 2.X86 指令按照功能可以分为算术运算、逻辑运算、数据传送、串操作、控制转移、处理器控制、保护方式指令,本文将后 3 类指令视为特殊指令,其输入/输出对象空间均为 *EAX*×*EBX*×*ECX*×*EDX*×*ESI*×*EDI*×*ESP*×*EBP*×*flags*×*memory*.

Table 2 Input context and output context of commonly used X86 instructions

表 2 常用 X86 指令的输入对象空间和输出对象空间

指令	输出对象空间	输入对象空间
nop	∅	∅
mov *R*1, *R*2	*R*1	*R*2
cmp *R*1, *R*2	*flags*	*R*1×*R*2
add *R*1, *R*2	*R*1×*flags*	*R*1×*R*2
xchg *R*1, [*R*2]	*R*1×*memory*	*R*1×*R*2×*memory*
push *R*1	*ESP*×*memory*	*R*1
pop *R*1	*ESP*×*R*1	*memory*
rep lods	*ESI*×*EAX*×*ECX*×*flags*	*flags*×*memory*
rep stos	*EDI*×*ECX*×*memory*×*flags*	*EAX*×*flags*
rep cmps	*ESI*×*EDI*×*ECX*×*memory*×*flags*	*memory*

2.1.2 对象空间操作

为了分析程序与指令的输入对象空间和输出对象空间的关系,首先定义对象空间的并、交、差操作.

定义 2.5(对象空间的并). 设 $\boldsymbol{S}_1$ 和 $\boldsymbol{S}_2$ 是两个对象空间,则对象空间的并可记做 $\boldsymbol{S}_1\cup\boldsymbol{S}_2=V_1\times V_2\times\ldots\times V_N$,其中,

$$V_i=\begin{cases}W_i, & \text{if } \Lambda_{\boldsymbol{v}_1\in\boldsymbol{S}_1,\boldsymbol{v}_2\in\boldsymbol{S}_2}(\boldsymbol{v}_1^i\in W_i\vee\boldsymbol{v}_2^i\in W_i)\\ \{\alpha\}, & \text{otherwise}\end{cases}..$$

定义 2.6(对象空间的交). 设 $\boldsymbol{S}_1$ 和 $\boldsymbol{S}_2$ 是两个对象空间,则对象空间的交可记做 $\boldsymbol{S}_1\cap\boldsymbol{S}_2=V_1\times V_2\times\ldots\times V_N$,其中,

$$V_i=\begin{cases}W_i, & \text{if } \Lambda_{\boldsymbol{v}_1\in\boldsymbol{S}_1,\boldsymbol{v}_2\in\boldsymbol{S}_2}(\boldsymbol{v}_1^i\in W_i\wedge\boldsymbol{v}_2^i\in W_i)\\ \{\alpha\}, & \text{otherwise}\end{cases}.$$

同时,利用对象空间的交定义对象空间的包含关系.

定义 2.7(对象空间的包含). 对象空间的包含即为集合的包含,记做 $\boldsymbol{S}_1\subset\boldsymbol{S}_2\Leftrightarrow\boldsymbol{S}_1\cap\boldsymbol{S}_2=\boldsymbol{S}_1$.

定义 2.8(对象空间的差). 对象空间的差即为集合的差,记做 $\boldsymbol{S}_1-\boldsymbol{S}_2=V_1\times V_2\times\ldots\times V_N$,其中,

$$V_i=\begin{cases}W_i, & \text{if } \Lambda_{\boldsymbol{v}_1\in\boldsymbol{S}_1,\boldsymbol{v}_2\in\boldsymbol{S}_2}(\boldsymbol{v}_1^i\neq\alpha\wedge\boldsymbol{v}_2^i=\alpha)\\ \{\alpha\}, & \text{otherwise}\end{cases}.$$

为了便于理解,集合$\{\alpha\}$具有空集的运算性质,也即$\{\alpha\}\cup W_i=W_i$,$\{\alpha\}\cap W_i=\{\alpha\}$,$W_i-\{\alpha\}=W_i$.

性质 2.1. 若假设 $\boldsymbol{S}_1=V_{1,1}\times V_{1,2}\times\ldots\times V_{1,N}$,$\boldsymbol{S}_2=V_{2,1}\times V_{2,2}\times\ldots\times V_{2,N}$,则有:

$$\boldsymbol{S}_1\cup\boldsymbol{S}_2=(V_{1,1}\cup V_{2,1})\times(V_{1,2}\cup V_{2,2})\times\ldots\times(V_{1,N}\cup V_{2,N}),$$

$$\boldsymbol{S}_1\cap\boldsymbol{S}_2=(V_{1,1}\cap V_{2,1})\times(V_{1,2}\cap V_{2,2})\times\ldots\times(V_{1,N}\cap V_{2,N}).$$

定理 2.1. 给定程序 $P=P_1|P_2$,每段程序的输入对象空间分别为 $\boldsymbol{S}_I(P_1)$,$\boldsymbol{S}_I(P_2)$,输出对象空间为 $\boldsymbol{S}_O(P_1)$,$\boldsymbol{S}_O(P_2)$,则程序的输入对象空间为

$$\boldsymbol{S}_I(P)=(\boldsymbol{S}_I(P_2)-\boldsymbol{S}_O(P_1))\cup\boldsymbol{S}_I(P_1).$$

定理 2.2. 给定程序 $P=P_1|P_2$,每段程序的输出对象空间为 $\boldsymbol{S}_O(P_1)$,$\boldsymbol{S}_O(P_2)$,则程序的输出对象空间为

$$\boldsymbol{S}_O(P)=\boldsymbol{S}_O(P_1)\cup\boldsymbol{S}_O(P_2).$$

特别地,当 P_1,P_2 同时为指令时,也即 $P=I_1|I_2$,程序的输入/输出对象空间分别为

$$\boldsymbol{S}_I(P)=(\boldsymbol{S}_I(I_2)-\boldsymbol{S}_O(I_1))\cup\boldsymbol{S}_I(I_1),$$

$$\boldsymbol{S}_O(P)=\boldsymbol{S}_O(I_1)\cup\boldsymbol{S}_O(I_2).$$

恰为文献[2]中描述的两条指令的输入/输出对象空间.

例 2.2:假设一个计算机的状态空间为 $\boldsymbol{S}=W_1\times W_2\times W_3\times W_4$,两条指令的映射分别如下:

$$I_1:(\boldsymbol{v}^1,\boldsymbol{v}^2,\boldsymbol{v}^3,\boldsymbol{v}^4)\to(\boldsymbol{v}^1,\boldsymbol{v}^2,\boldsymbol{v}^1+\boldsymbol{v}^2,\boldsymbol{v}^4),$$
$$I_2:(\boldsymbol{v}^1,\boldsymbol{v}^2,\boldsymbol{v}^3,\boldsymbol{v}^4)\to(\boldsymbol{v}^1,\boldsymbol{v}^3+\boldsymbol{v}^4,\boldsymbol{v}^3,\boldsymbol{v}^4),$$

其中,$\boldsymbol{v}^1\in W_1,\boldsymbol{v}^2\in W_2,\boldsymbol{v}^3\in W_3,\boldsymbol{v}^4\in W_4$,则两条指令的输入对象空间分别为

$$\boldsymbol{S}_I(I_1)=W_1\times W_2\times\{\alpha\}\times\{\alpha\},$$
$$\boldsymbol{S}_I(I_2)=\{\alpha\}\times\{\alpha\}\times W_1\times W_2.$$

输出对象空间为

$$\boldsymbol{S}_O(I_1)=\{\alpha\}\times\{\alpha\}\times W_3\times\{\alpha\},$$
$$\boldsymbol{S}_O(I_2)=\{\alpha\}\times W_2\times\{\alpha\}\times\{\alpha\}.$$

给定程序 $P=I_1|I_2$,由定理 2.1 和定理 2.2,可得程序 P 的输入、输出对象空间分别为

$$\boldsymbol{S}_I(P)=W_1\times W_2\times\{\alpha\}\times W_4,$$
$$\boldsymbol{S}_O(P)=\{\alpha\}\times W_2\times W_3\times\{\alpha\}.$$

即程序 P 使用输入 $\boldsymbol{v}^1,\boldsymbol{v}^2,\boldsymbol{v}^4$ 改变了 $\boldsymbol{v}^2,\boldsymbol{v}^3$.

定义 2.9(程序语义相同). 在输入对象空间 $\boldsymbol{S}_I$ 和输出对象空间 $\boldsymbol{S}_O$ 相同情况下的程序 P_1 和 P_2 语义相同,定义集合 $J_I=\{x\mid V_{\boldsymbol{v}^x\in\boldsymbol{S}_I}\boldsymbol{v}^x\neq\alpha\},J_O=\{x\mid V_{\boldsymbol{v}^x\in\boldsymbol{S}_O}\boldsymbol{v}^x\neq\alpha\}$,如果有 $\Lambda_{\boldsymbol{v}_1,\boldsymbol{v}_2\in\boldsymbol{S}}(\Lambda_{j\in J_I}\boldsymbol{v}_1^j=\boldsymbol{v}_2^j\Rightarrow\Lambda_{k\in J_O}P_1(\boldsymbol{v}_1)^k=P_1(\boldsymbol{v}_2)^k)$,则认为两个程序 P_1 和 P_2 语义相同,记为 $P_1\equiv P_2$.其中,J_I 与 J_O 是 $\boldsymbol{S}_I$ 与 $\boldsymbol{S}_O$ 的另一种表示,以便后续的推导.

程序语义相同的公式描述的是当状态变量在当前程序所引用的分量上相等,则输出的状态变量在所影响的分量上一定相等,即相同语义的程序在其所关注的输入相同时,输出一定相同.

定理 2.3(程序交换充分条件). 假设程序 P_1,P_2 的输入/输出对象空间分别为 $\boldsymbol{S}_I(P_1),\boldsymbol{S}_I(P_2),\boldsymbol{S}_O(P_1),\boldsymbol{S}_O(P_2)$,如果:

$$\boldsymbol{S}_O(P_1)\cap\boldsymbol{S}_O(P_2)=\{\alpha\}\times\{\alpha\}\times\ldots\times\{\alpha\},$$
$$\boldsymbol{S}_O(P_1)\cap\boldsymbol{S}_I(P_2)=\{\alpha\}\times\{\alpha\}\times\ldots\times\{\alpha\},$$
$$\boldsymbol{S}_I(P_1)\cap\boldsymbol{S}_O(P_2)=\{\alpha\}\times\{\alpha\}\times\ldots\times\{\alpha\},$$

则有 $P_{12}=P_1|P_2\equiv P_{21}=P_2|P_1$.也即:相邻两段程序不会引用另一段程序所影响的分量,同时两者所影响的分量也正交,那么两者交换不影响程序语义.

证明:定义如下集合:

$$J_I(P_1)=\{x\mid V_{\boldsymbol{v}^x\in\boldsymbol{S}_I(P_1)}\boldsymbol{v}^x\neq\alpha\},J_O(P_1)=\{x\mid V_{\boldsymbol{v}^x\in\boldsymbol{S}_O(P_1)}\boldsymbol{v}^x\neq\alpha\};$$
$$J_I(P_2)=\{x\mid V_{\boldsymbol{v}^x\in\boldsymbol{S}_I(P_2)}\boldsymbol{v}^x\neq\alpha\},J_O(P_2)=\{x\mid V_{\boldsymbol{v}^x\in\boldsymbol{S}_O(P_2)}\boldsymbol{v}^x\neq\alpha\}.$$

由于 $\boldsymbol{S}_O(P_1)\cap\boldsymbol{S}_O(P_2)=\boldsymbol{S}_O(P_1)\cap\boldsymbol{S}_I(P_2)=\boldsymbol{S}_I(P_1)\cap\boldsymbol{S}_O(P_2)=\{\alpha\}\times\{\alpha\}\times\ldots\times\{\alpha\}$,

对于 P_{12} 和 P_{21},其输入对象空间和输出对象空间相同,同为:

$$\boldsymbol{S}_I(P_{12})=(\boldsymbol{S}_I(P_2)-\boldsymbol{S}_O(P_1))\cup\boldsymbol{S}_O(P_1)=\boldsymbol{S}_I(P_2)\cup\boldsymbol{S}_I(P_1)=(\boldsymbol{S}_I(P_1)-\boldsymbol{S}_O(P_2))\cup\boldsymbol{S}_O(P_2)=\boldsymbol{S}_I(P_{21}),$$
$$\boldsymbol{S}_O(P_{12})=\boldsymbol{S}_O(P_{21})=\boldsymbol{S}_O(P_1)\cup\boldsymbol{S}_O(P_2);$$

同时有 $J_O(P_1)\cap J_O(P_2)=\varnothing,J_O(P_1)\cap J_I(P_2)=\varnothing,J_I(P_1)\cap J_O(P_2)=\varnothing$.

可得: $\Lambda_{j\in J_I(P_1)\cup J_I(P_2)}\boldsymbol{v}_1^j=\boldsymbol{v}_2^j\Rightarrow\Lambda_{k\in J_O(P_1)}P_2(\boldsymbol{v}_1)^k=\boldsymbol{v}_2^k$.

由定义 2.9 可得: $\Lambda_{j\in J_I(P_1)\cup J_I(P_2)}\boldsymbol{v}_1^j=\boldsymbol{v}_2^j\Rightarrow\Lambda_{k\in J_O(P_1)}P_1(P_2(\boldsymbol{v}_1))^k=P_1(\boldsymbol{v}_2)^k=P_2(P_1(\boldsymbol{v}_2))^k$.

同理可得: $\Lambda_{j\in J_I(P_1)\cup J_I(P_2)}\boldsymbol{v}_1^j=\boldsymbol{v}_2^j\Rightarrow\Lambda_{k\in J_O(P_2)}P_1(P_2(\boldsymbol{v}_1))^k=P_1(\boldsymbol{v}_2)^k=P_2(P_1(\boldsymbol{v}_2))^k$.

因此有: $\Lambda_{j\in J_I(P_1)\cup J_I(P_2)}\boldsymbol{v}_1^j=\boldsymbol{v}_2^j\Rightarrow\Lambda_{k\in J_O(P_1)\cup J_O(P_2)}P_1(P_2(\boldsymbol{v}_1))^k=P_1(\boldsymbol{v}_2)^k=P_2(P_1(\boldsymbol{v}_2))^k$.

也即 $P_{12}=P_1|P_2\equiv P_{21}=P_2|P_1$.

证明完毕.特别地,当程序为一条指令时,可以得到指令交换的充分条件.

定理 2.4(指令交换充分条件). 假设指令 l_1,l_2 的输入、输出对象空间分别为 $\boldsymbol{S}_I(l_1),\boldsymbol{S}_I(l_2)$和 $\boldsymbol{S}_O(l_1),\boldsymbol{S}_O(l_2)$,如果:

$$\boldsymbol{S}_O(I_1)\cap\boldsymbol{S}_O(I_2)=\{\alpha\}\times\{\alpha\}\times...\times\{\alpha\},$$
$$\boldsymbol{S}_O(I_1)\cap\boldsymbol{S}_I(I_2)=\{\alpha\}\times\{\alpha\}\times...\times\{\alpha\},$$
$$\boldsymbol{S}_I(I_1)\cap\boldsymbol{S}_O(I_2)=\{\alpha\}\times\{\alpha\}\times...\times\{\alpha\},$$

则有 $P_1=I_1|I_2\equiv P_2=I_2|I_1$.

例 2.3:如图 2 中的粗斜体指令,指令 I_1=*mov esi,eax* 与 I_2=*add edx,ebp* 的输入对象空间与输出对象空间分别为(为了运算简洁,省略特殊值{α}):

$$\boldsymbol{S}_I(I_1)=eax,\boldsymbol{S}_O(I_1)=esi;$$
$$\boldsymbol{S}_I(I_2)=edx,\boldsymbol{S}_O(I_2)=edx\times flags.$$

满足条件:

$$\boldsymbol{S}_O(I_1)\cap\boldsymbol{S}_O(I_2)=\{\alpha\}\times\{\alpha\}\times...\times\{\alpha\},$$
$$\boldsymbol{S}_O(I_1)\cap\boldsymbol{S}_I(I_2)=\{\alpha\}\times\{\alpha\}\times...\times\{\alpha\},$$
$$\boldsymbol{S}_I(I_1)\cap\boldsymbol{S}_O(I_2)=\{\alpha\}\times\{\alpha\}\times...\times\{\alpha\}.$$

也即 $P_1=I_1|I_2\equiv P_2=I_2|I_1$.

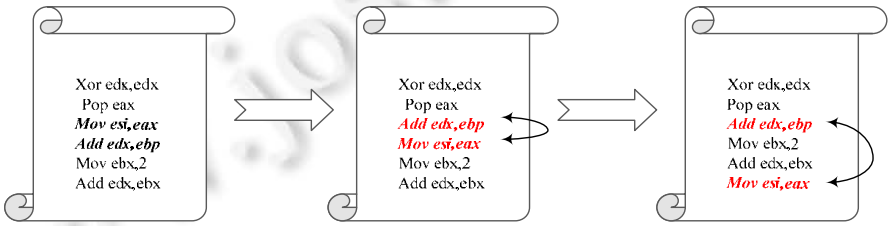


Fig.2 Diagram of adjacent instructions swapping

图 2 相邻指令交换示意图

2.1.3 讨 论

定理 2.3 与定理 2.4 仅从输入/输出对象空间考虑程序语义相同的充分条件,没有考虑指令的映射语义.而 X86 指令中的算术、逻辑运算等具有其特定的语义,可能满足指令交换后语义不变的条件.

考虑到 X86 指令中加减法具有交换律,因此,分析当两条指令的操作码为 ADD,输出对象空间相同时,是否具有可交换的性质,以扩展指令交换的充分条件.

在输入、输出对象空间的指导下,本文将输出对象空间相同的 ADD 指令分为 3 类,见表 3.

Table 3 Input context and output context of the instruction ADD

表 3 ADD 指令的输入/输出空间

X86 指令	映射	输入	输出
ADD reg1,reg1	I_1:($\boldsymbol{v}^1$,$\boldsymbol{v}^2$,*flags*)→($\boldsymbol{v}^1$+$\boldsymbol{v}^1$,$\boldsymbol{v}^2$,*f*(*flags*,$\boldsymbol{v}^1$+$\boldsymbol{v}^1$))	W_1	W_1×*flags*
ADD reg1,reg2	I_2:($\boldsymbol{v}^1$,$\boldsymbol{v}^2$,*flags*)→($\boldsymbol{v}^1$+$\boldsymbol{v}^2$,$\boldsymbol{v}^2$,*f*(*flags*,$\boldsymbol{v}^1$+$\boldsymbol{v}^2$))	W_1×W_2	W_1×*flags*
ADD reg1,imm	I_2:($\boldsymbol{v}^1$,$\boldsymbol{v}^2$,*flags*)→($\boldsymbol{v}^1$+*imm*,$\boldsymbol{v}^2$,*f*(*flags*,$\boldsymbol{v}^1$+*imm*))	W_1	W_1×*flags*

其中,函数 f 是有关标志寄存器的映射关系.由指令的映射可知:

$$I_1\,|\,I_2:(\boldsymbol{v}^1,\boldsymbol{v}^2,\ flags)\rightarrow(\boldsymbol{v}^1+\boldsymbol{v}^1+\boldsymbol{v}^2,\boldsymbol{v}^2,\ f(\ flags,\boldsymbol{v}^1+\boldsymbol{v}^1+\boldsymbol{v}^2)),$$
$$I_1\,|\,I_2:(\boldsymbol{v}^1,\boldsymbol{v}^2,\ flags)\rightarrow(\boldsymbol{v}^1+\boldsymbol{v}^2+\boldsymbol{v}^1+\boldsymbol{v}^2,\boldsymbol{v}^2,\ f(\ flags,\boldsymbol{v}^1+\boldsymbol{v}^2+\boldsymbol{v}^1+\boldsymbol{v}^2)).$$

I_1 与 I_2 是不能交换的.而:

$$I_2\,|\,I_3:(\boldsymbol{v}^1,\boldsymbol{v}^2,\ flags)\rightarrow(\boldsymbol{v}^1+\boldsymbol{v}^2+imm,\boldsymbol{v}^2,\ f(\ flags,\boldsymbol{v}^1+\boldsymbol{v}^2+imm)),$$
$$I_3\,|\,I_2:(\boldsymbol{v}^1,\boldsymbol{v}^2,\ flags)\rightarrow(\boldsymbol{v}^1+\boldsymbol{v}^2+imm,\boldsymbol{v}^2,\ f(\ flags,\boldsymbol{v}^1+\boldsymbol{v}^2+imm)).$$

I_2 与 I_3 是可交换的.同理,I_2 与 I_2、I_3 与 I_3 也是可交换的.同时,INC 指令是 *ADD reg*1,*imm* 的特殊情况,可以归为 ADD 指令.

同理,SUB 与 DEC 指令也满足同样的性质.

由于第 1 类指令与第 3 类指令的输入/输出对象空间条件相同,因此必须结合 X86 指令的操作码进行判断.不妨设表 3 中的第 1 类指令为 ADD1,第 2 类指令为 ADD2,第 3 类指令为 ADD3,对于 SUB 指令,同样定义 SUB1,SUB2 与 SUB3,则 ADD1 与 ADD2 的区分条件为 $\boldsymbol{S}_{I}\subset\boldsymbol{S}_{O}$;而 ADD1 与 ADD3 的区分条件为操作码,在 X86 指令中,ADD1 的操作码范围为 0x00~0x05,ADD3 的操作码范围为 0x80~0x83.

而在逻辑运算中,与、或运算所满足的结合律使得其也具有一定的可交换性,且由于任意二进制数与自身的与或结果都为自身,因此不需要如加减指令进行分类.输出对象空间相同的 AND 指令或 OR 指令可以互换.

2.1.4 指令序列距离

为了描述经过指令互换后指令序列的差异性,借鉴汉明距离(Hamming distance),给出指令序列距离的定义.

定义 2.10(指令序列距离). 设 P_1 为一个 X86 指令构成的基本块:$I_1|I_2|\ldots|I_n$,经过指令乱序的指令序列为 P_2: $I_{i_1}|I_{i_2}|\ldots|I_{i_n}$,其中,$i_1,i_2,\ldots,i_n$ 是 1 到 n 的一个有序排列,且满足 $P_1\equiv P_2(\boldsymbol{v}_I)=\boldsymbol{v}_O$,则指令序列距离为

$$D(P_1,P_2)=\frac{\sum_j^n(i_j-j)^2}{n}.$$

指令序列距离并未选取距离的绝对值作为指标描述其差异性,原因在于差异性需强调经过混淆前后指令序列的距离,例如原始序列的为 1-2-3,经过混淆后有两个序列 3-2-1 与 3-1-2,若用距离的绝对值作为衡量标准,则认为两个序列具有相同的差异度,而使用距离的平方值作为衡量标准,则认为序列 3-2-1 与原始序列的差异度更高,这更符合实际.

3 指令乱序混淆方法

本文研究的对象为指令基本块,是静态分析的结果.混淆分为两个步骤:一是对每条指令的输入/输出对象空间进行分析,以判断是否可以进行指令交换;二是通过模拟退火算法最大化指令序列距离,寻找最优交换策略.

3.1 指令交换

图 3 为相邻指令交换的具体流程.

Step 1. 通过反汇编引擎将二进制代码反汇编为汇编代码,也即指令序列;

Step 2. 对基本块中指令序列的每条指令,使用改进后的 XDE 引擎分析其输入、输出对象空间;

Step 3. 判断两条相邻的指令是否满足交换条件.

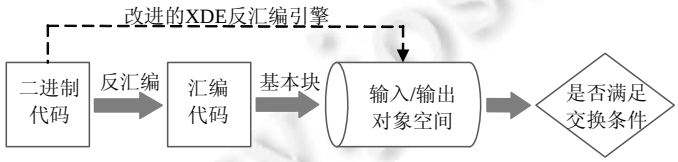


Fig.3 Flow chart of instructions swapping

图 3 指令交换流程图

其中,指令交换的充分条件由定理 2.4 和第 2.1.3 节构成,可进行如下定义:

$$\boldsymbol{S}_{O1}\cap\boldsymbol{S}_{O2}=\{\alpha\}\times\{\alpha\}\times\ldots\times\{\alpha\} \quad (1)$$

$$\boldsymbol{S}_{O1}\cap\boldsymbol{S}_{I2}=\{\alpha\}\times\{\alpha\}\times\ldots\times\{\alpha\} \quad (2)$$

$$\boldsymbol{S}_{I1}\cap\boldsymbol{S}_{O2}=\{\alpha\}\times\{\alpha\}\times\ldots\times\{\alpha\} \quad (3)$$

$$[1]\&\&[2]\&\&[3] \quad (4)$$

$$\boldsymbol{S}_{O1}==\boldsymbol{S}_{O2} \quad (5)$$

$$(\boldsymbol{S}_{O1}\cap\boldsymbol{S}_{O2})\subset\boldsymbol{S}_{O1} \quad (6)$$

$$(\boldsymbol{S}_{O2}\cap\boldsymbol{S}_{I1})\subset\boldsymbol{S}_{O2} \quad (7)$$

$$\left.\begin{array}{l}\#define\ Op(I)\ (I.op == ADD2 \| ADD3 \| SUB2 \| SUB3 \| INC \| DEC)\\ Op(I_1)\&\&Op(I_2)\&\&[5]\&\&[6]\&\&[7]\end{array}\right\} \tag{8}$$

则指令交换的充分条件为[4]&&[8].

3.2 指令乱序算法

为了最大化混淆前后指令序列的差异性,问题可抽象为最优化问题.给定由 X86 指令序列构成的基本块 $P=I_1|I_2|\ldots|I_n$,则问题转化为

$$\begin{cases}\max\ Z = D(P_1,P_2)\\ \text{s.t. } P_1 \equiv P_2(\boldsymbol{v}_I) = \boldsymbol{v}_O\end{cases}.$$

根据定义 2.10,该优化问题的变量为$(i_1,i_2,\ldots,i_n)$,其满足的条件为由此变量构成的序列 $P_1\equiv P_2$,即该问题为非线性优化问题,可以通过遍历可行解集合求解最优值.为了更好地判断可行解集合的大小,本文任意选取基本块指令,指令数分别为 40,45,50,55,60,65,遍历以求得可行解集合大小,实验环境为 Win7 操作系统,Intel(R) Core (TM) i7-6700 CPU@3.40GHz 处理器,32G 内存.计算结果如图 4,解空间与遍历时间随着指令数的增长呈指数型增长,如果指令数超过 80,遍历时间则难以满足要求.

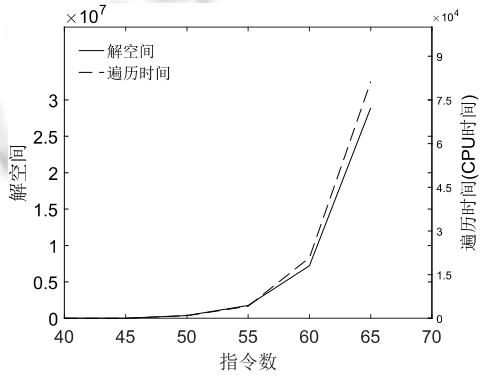


Fig.4 Relationship diagram between the solution space and the number of instructions

图 4 解空间与指令数关系图

具体分析遍历算法的复杂度,以 n 条指令为分析对象,其中,判断两条相邻指令的交换条件为基本操作,若满足条件则解空间增加 2 倍,也即基本操作增加 2 倍,因此,遍历算法的时间复杂度为 $O(2^{k\times n})=O(2^n)$,其中,$k\times n$ 为 n 条指令中可相互交换的指令数;关于空间复杂度,遍历过程中只需额外维护一个当前最优值的结构体,也即空间复杂度为 $O(n)$.复杂度分析结果与实验结果相符合.

为了解决上述最优化问题,同时增加指令乱序的随机性,本文采用模拟退火算法的思想对指令乱序进行最优化求解,也即:在指令交换过程中,以 Metropolis 准则接收新解,使用的符号见表 4,具体算法步骤如下.

Step 1. 初始化温度参数 T=1,r=0.95;

Step 2. 以原始序列 P 为初始解,计算目标函数 $D(P,P)$=0;

Step 3. 分析指令序列 $P(i)$中所有可交换的相邻指令,以第一条指令为起点,判断是否可与后一条指令交换,若可交换,则产生新解 $P(i+1)$,计算目标函数差值$\Delta D=D(P,P(i+1))-D(P,P(i))$;

Step 4. 若差值大于 0,则接收新解,仍以当前指令为对象进行 Step 3;否则,以 Metropolis 准则接受新解:$P(i)=P(i+1)$,同时进行降温:$T=r\times T$,以当前指令的下一条指令为对象进行 Step 3;

Step 5. 直到所有指令分析结束,跳出循环.

对于每次分析过程,对象仅为两条相邻语句,基本操作为判断两条相邻语句的交换条件,内循环次数不超过 2,大循环的次数是指令序列中指令数 n,即算法复杂度为 $O(n)$;算法分析过程中所需额外内存都是以指令数为单位,因此其空间复杂度为 $O(n)$.该混淆算法能在可接受时间内获得局部最优解,并为算法增加一定的随机性.

Table 4 Description of used notation in the disorder algorithm

表 4 指令乱序算法符号描述表

符号	描述
T	温度参数
r	降温速率
Ins_len	指令序列中指令数
P(*i*)	表示当前的指令序列
P(*i*+1)	表示新的指令序列
P	表示初始状态
D(*P*,*P*(*i*))	目标函数值

3.3 解释函数指令乱序

在虚拟机代码保护的各个阶段,都可融合指令乱序混淆算法.本文以解释函数构造为例,对解释函数进行指令乱序,这样每次保护便能得到语义相同但序列不同的解释函数,增加了虚拟机代码保护的随机性和动态性,增加了攻击者的分析时间.

4 实验及分析

4.1 理论分析

虚拟机代码保护技术的出现,使代码保护算法的评价更加复杂.目前,学术界并没有统一标准的方法或指标来评价某种方法或某个系统的保护效果,大多采用 Collberg 提出的强度、开销、抗逆向、隐蔽性这 4 个定性描述[1,17]来分析保护算法是否有效.本文基于这 4 个定性描述,对解释函数指令乱序方法进行定性分析.

指令乱序的对象为代码基本块,混淆前后的指令数量维持不变,因此在强度属性上无变化,例如强度属性中的指令执行率、控制流基本块指标等,指令乱序混淆算法不会改变上述指标.同样,对于空间开销和时间开销,由于经由混淆算法生成的保护后程序没有增加或修改指令,只是修改指令的执行顺序,因此相较于原始程序,保护后程序几乎不会改变任何时间和空间上的开销.

虚拟机代码保护技术的逆向分析集中在虚拟机保护技术的结构分析、解释函数语义还原,而现有的商业软件,如 Themida,Code Virtualizer 等,每个解释函数的指令数为百千数量级,针对解释函数的语义还原显得困难重重.现有的主要方法是尽可能自动化地压缩 Handler,而后通过符号执行对约减后的指令进行语义分析[18],其中,自动化压缩算法主要为模式匹配.Guillot 提出:由于 Handler 中大部分为算术运算与堆栈操作指令,可以通过常量传播、常量合并、运算合并、堆栈优化等方法对 Handler 进行压缩[19].如图 5 所示,可以对 3 条 add 指令进行约减.而指令乱序可能将图中可合并的指令打乱,如图 6 所示,原始程序中的 add 指令约减变得复杂,也即:指令乱序对此类基于模式匹配的自动化约减有较好的抵抗效果,增加了自动逆向分析的难度.

同时,对 Handler 指令序列进行随机化指令乱序,使得每次保护后程序的 Handler 不同,结合指令集随机化技术,极大地消除攻击者对 Handler 的先验知识,进一步延长攻击者逆向分析的时间.

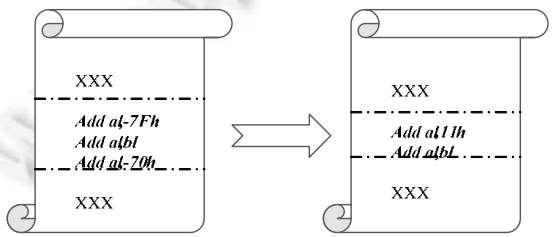


Fig.5 Reduction of the add computation

图 5 Add 指令的约减

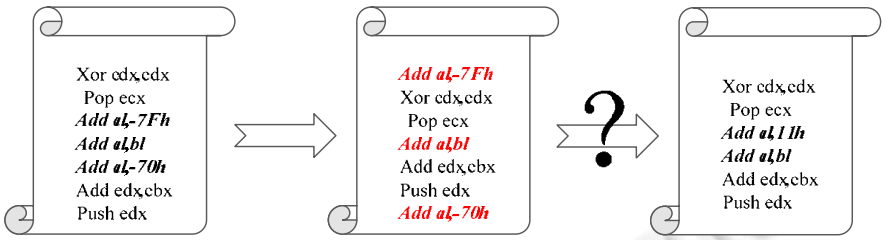


Fig.6 Effect of instructions swapping

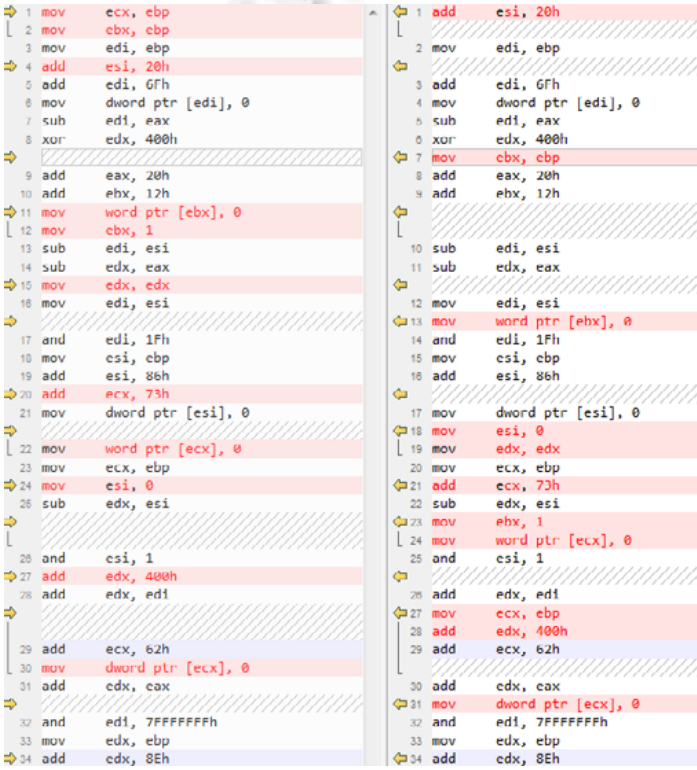
图 6 指令乱序的效果

隐蔽性强调混淆后程序与原始程序的相似度,其中,谢鑫等人[9]将指令序列之间的相似度作为隐蔽性属性的度量指标之一.而指令乱序能在对原始程序不进行语义分析的基础上增大指令序列的差异性,结合垃圾代码、等价指令替换等混淆方法,能极大地增大隐蔽性.

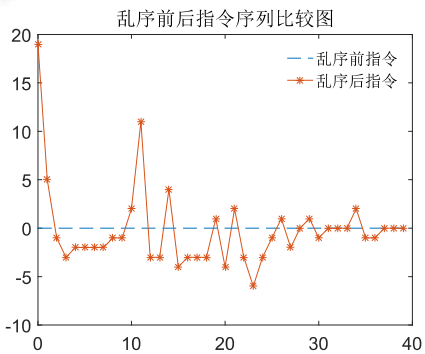
4.2 实验验证

本文实验环境为 Win7 操作系统,Intel(R) Core(TM) i7-6700 CPU@3.40GHz 处理器,32G 内存,选用的实例指令序列为 CV 的一个 Handler,以此分析指令乱序的效果.

由于 CV 的 Handler 序列长度过长,考虑到篇幅限制,本文截取了某个 Handler 的前 40 条指令,属于一个基本块,对 40 条指令执行指令乱序混淆算法,其中,图 7(a)所示为通过 BeyondCompare 工具对比混淆前后指令序列的差异,由图可知:两个指令序列差异较大,约一半的指令进行了指令交换操作.



(a) BC compare



(b) MATLAB R2016a

Fig.7 Comparison of instruction sequences

图 7 指令序列比较图

由图 7(b)可知,部分指令与原始指令序列的位置距离较远.例如第 1 条指令经过交换后移至第 20 条指令,混淆前后的指令序列距离为 17.8,以逆向分析者的角度来说,两段指令序列已经是语义不同的指令序列.

同时,本文采用的指令乱序算法是在保证局部最优值的前提下增加了一定的随机性,使得同一段指令序列每次经过指令乱序混淆得到的结果都不尽相同,如图 8 所示,若原始指令序列长度更长,效果更为显著.

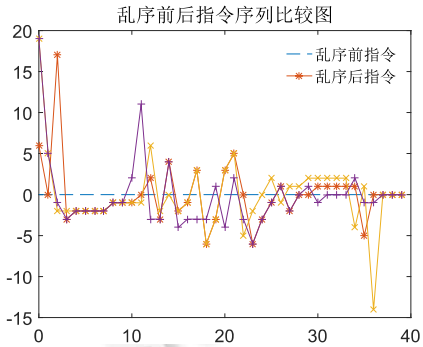


Fig.8 Comparison of effect of different instructions swapping

图 8 多次指令乱序效果对比图

上文主要阐述了指令乱序算法对指令乱序的效果,而代码混淆算法的优劣更应该从逆向分析人员的角度进行思考与分析[20−23].本文则借助 IDA 自动化分析逆向工具对指令乱序前后的程序进行比较,分析指令乱序抗逆向分析的效果.

以 MD5.exe(标准的 MD5 加密算法执行程序)为输入,经过指令乱序算法生成 MD5-IS.exe,借助 IDA 工具分析 MD5-IS.exe 相较于 MD5.exe 丢失的信息.经指令乱序模块分析,MD5.exe 共有 10 130 条汇编语句,共进行了 1 027 次指令交换过程,指令距离为 0.3,输出为 MD5-IS.exe.

图 9 是 MD5String 函数经过 IDA 还原后的伪代码,经过指令乱序后的函数参数以及名字都有较大的信息丢失,以加大逆向分析者的还原难度.而以 SHA1.exe(标准的 SHA1 加密算法执行程序)为输入时,IDA 的反编译功能对 SHA1-GO 函数的伪代码还原度极高,基本与 C++源代码相同,对于逆向分析者的后续工作有极大的帮助;而对于 SHA1-IS.exe,反编译功能则不能执行,对其汇编代码无法进行转换.

```
__int64_cdecl sub_402900(char*Str)
{
    int v1;                 //edx@1
    __int64 v2;             //ST04_8@7
    char v4;                //[sp+Ch][bp-16Ch]@1
    char*v5;                //[sp+10h][bp-168h]@1
    int i;                  //[sp+DCh] [bp-9Ch]@1
    size_t v7;              //[sp+E8h] [bp-90h]@1
    void*v8;                //[sp+F4h] [bp-84h]@1
    char v9[24];
    char v10;               //[sp+118h][bp-60h]@1
    unsigned int v11;
    int savedregs;
    memset(&v4,0xCCu,0x16Cu);
    v11=(unsigned int)&savedregs^__security_cookie;
    v5=unknown_libname_1(0x20u);
    v8=v5;
    v7=strlen(Str);
    sub_402860(&v10);
    sub_403C20(&v10,Str,v7);
    sub_402740(v9,&v10);
}
```

(a) MD5-IS.exe

```
__int64_cdecl MD5String(char*string)
{
    int v1;                 //edx@1
    __int64 v2;             //ST04_8@7
    char v4;                //[sp+Ch] [bp-16Ch]@1
    char*v5;                //[sp+10h] [bp-168h]@1
    int i;                  //[sp+DCh] [bp-9Ch]@1
    unsigned int len;
    char*output1;
    char digest[24];
    _MD5_CTX context;
    unsigned int v11;
    int savedregs;          //[sp+178h][bp+0h]@1
    memset(&v4,0xCCu,0x16Cu);
    v11=(unsigned int)&savedregs^__security_cookie;
    v5=(char*)operator new[](0x20u);
    output1=v5;
    len=strlen(string);
    MD5Init(&context);
    MD5Update(&context,string,len);
    MD5Final(digest,&context);
}
```

(b) MD5.exe

Fig.9 Pseudocode view of IDA

图 9 IDA 伪代码窗口

4.3 性能分析

将指令乱序混淆算法与虚拟机代码保护技术融合,本文采取的融合方法为对每条 Handler 进行指令乱序,在虚拟机代码保护系统 My-VMP 的基础上实现 IS-VMP 系统.实验环境同上,测试用例选用了标准的加密算法 MD4,MD5 与 SHA1 为测试用例,见表 5.在同样的实验环境下,对测试实例分别用商用软件 Code Virtualizer (CV),VMProtect 以及 IS-VMP 系统保护.其中:CV 版本号为 2.2.1.0,使用的虚拟机类型为 Tiger32 White;VMP 版本号为 2.13.8,采用最快速度策略进行虚拟机保护.表 6 为保护前后文件大小变化,表 7 为保护前后 KeyCode 执行时间变化.由于每次执行时间都有所不同,图表中执行时间为 10 次执行时间的平均值.其中,MyVMP 与 IS-VMP 的区别在于是否融合了指令乱序算法.

Table 5 Test case description

表 5 测试用例描述

测试程序	关键代码段描述	KeyCode 指令数	KeyCode 执行时间/μs
MD4.exe	MD4_update,加密字符串长度为 10	1 196	0.6
MD5.exe	MD5_Transform,加密字符串长度为 10	1 333	1.8
SHA1.exe	主函数,加密字符串长度为 10	34	6.0

Table 6 Comparison of file size

表 6 保护前后文件大小变化

被保护软件	保护前(KB)	保护后(KB)			
		CV2.2.1.0	VMP2.13.8	My-VMP	IS-VMP
MD4.exe	26	372	49	148	148
MD5.exe	41	406	65	173	173
SHA1.exe	22	288	34	37	36

Table 7 Comparison of exection time

表 7 保护前后 KeyCode 执行时间变化

被保护软件	保护前(μs)	保护后(us)			
		CV2.2.1.0	VMP2.13.8	My-VMP	IS-VMP
MD4.exe	0.6	1065.5	245.1	903.7	807.0
MD5.exe	1.8	1010	295.7	829.6	805.7
SHA1.exe	6.0	116.9	31.1	28.9	29.9

为了更加直观地比较不同方法之间的差别,如图 10 所示.

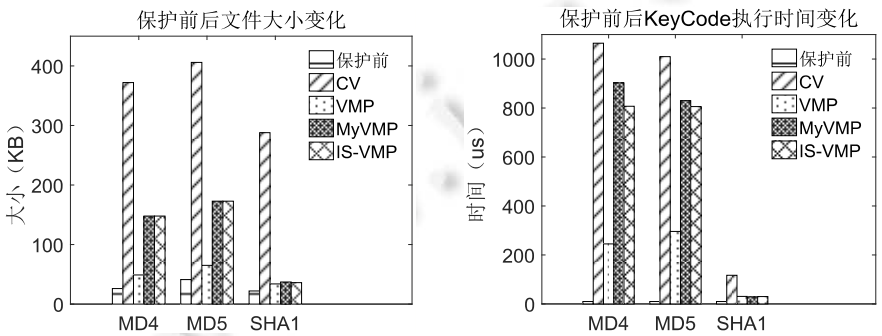


Fig.10 Performance of IS-VMP

图 10 IS-VMP 性能测试结果

由上述图表可知,

- IS-VMP 与商用保护软件的纵向对比:其膨胀比例与时间开销都处在 CV 与 VMProtect 两者之间;
- IS-VMP 与 MyVMP 的横向对比:融合指令乱序算法后,膨胀比例与时间开销几乎不变;而根据前文的分

析,IS-VMP 在抵抗逆向分析与消除攻击者的先验知识上都具有一定的效果.

5 总结与展望

本文基于 Wroblewski 提出的形式化定义并加以改进,论证了相邻指令序列交换的充分条件,并在此基础上采用模拟退火算法实现了随机性的指令乱序算法,并以基本块内的指令序列为对象予以实现.通过实验验证了混淆算法的可行性与有效性,为代码克隆提供了一种新的自动化实现的方法.同时,分析现有的虚拟机代码保护技术的随机化与动态化思想,提出将指令乱序算法应用于解释函数的混淆,并实现了 IS-VMP 系统.测试实例结果表明:指令乱序算法能有效增强解释函数的随机性,且不增加性能开销.

指令乱序混淆算法属于传统代码混淆算法,其面向对象为代码基本块,因此,其可融合的方向不仅仅解释函数,研究其应用的场景将是本文未来研究的方向.针对相邻指令交换,本文所求取的是相邻指令交换的充分条件,其充要条件仍需更多的研究.同时,研究相邻指令序列的交换的实现也具有很强的实际意义,也为恶意代码多态变形技术的研究提供了借鉴意义.

References:

[1] Collberg C, Thomborson CD, Low D. A taxonomy of obfuscating transformations. Technical Report, 148, Department of Computer Science the University of Auckland New Zealand, 1997.

[2] Wroblewski G. General method of program code obfuscation [Ph.D. Thesis]. Wroclaw: Institute of Engineering Cybernetics, Wroclaw University of Technology, 2002.

[3] Birrer BD, Raines RA, Baldwin RO, *et al*. Program fragmentation as a metamorphic software protection. In: Proc. of the Int'l Symp. on Information Assurance and Security. IEEE, 2007. 369−374. [doi: 10.1109/IAS.2007.28]

[4] Li Y, Zuo ZH. An overview of object-code obfuscation technologies. Computer Technology and Development, 2007,17(4): 125−127 (in Chinese with English abstract). [doi: 10.3969/j.issn.1673-629X.2007.04.034]

[5] Ghosh S, Hiser J, Davidson JW. Replacement attacks against VM-protected applications. ACM SIGPLAN Notices, 2012,47(7): 203−214. [doi: 10.1145/2365864.2151051]

[6] Coogan KP. Deobfuscation of packed and virtualization-obfuscation protected binaries [Ph.D. Thesis]. University of Arizona, 2011.

[7] Coogan K, Lu G, Debray S. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In: Proc. of the ACM Conf. on Computer and Communications Security. ACM Press, 2011. 275−284. [doi: 10.1145/2046707.2046739]

[8] Fang DY, Zhang H, Tang ZY, Chen XJ. DAS-VMP: A virtual machine-based software protection method for defending against semantic attacks. Journal of Sichuan University (Engineering Science Edition), 2017,49(1):159−168 (in Chinese with English abstract). [doi: 10.15961/j.jsuese.2017.01.021]

[9] Banescu S, Collberg C, Ganesh V, *et al*. Code obfuscation against symbolic execution attacks. In: Proc. of the Conf. on Computer Security Applications. ACM Press, 2016. 189−200. [doi: 10.1145/2991079.2991114]

[10] Xie X, Liu FL, Lu B, *et al*. Virtual machine protection based on Handler obfuscation enhancement. Computer Engineering and Applications, 2016,52(15):146−152 (in Chinese with English abstract). [doi: 10.3778/j.issn.1002-8331.1410-0299]

[11] Wu WM, Xu WF, Lin ZY, *et al*. Software protection technique based on improved virtual machine. Computer Engineering & Science, 2014,36(4):655−661 (in Chinese with English abstract). [doi: 10.3969/j.issn.1007-130X.2014.04.015]

[12] Fang DY, Zhao Y, Wang HJ, Gu YX, Xu GL. Software protection based on virtual machine with time diversity. Ruan Jian Xue Bao/Journal of Software, 2015,26(6):1322−1339 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4592.htm [doi: 10.13328/j.cnki.jos.004592]

[13] Wang H, Fang D, Li G, *et al*. TDVMP: Improved virtual machine-based software protection with time diversity. In: Proc. of the ACM SIGPLAN on Program Protection and Reverse Engineering Workshop. 2014. 1−9. [doi: 10.1145/2556464.2556468]

[14] Wang H, Fang D, Li G, *et al*. NISLVMP: Improved virtual machine-based software protection. In: Proc. of the 2013 9th Int'l Conf. on Computational Intelligence and Security (CIS). IEEE, 2013. 479−483. [doi: 10.1145/2556464.2556468]

[15] Kuang K, Tang Z, Gong X, *et al*. Exploiting dynamic scheduling for VM-based code obfuscation. In: Proc. of the Trustcom/ Bigdatase/Ispa. IEEE, 2017. 489−496. [doi: 10.1109/TrustCom.2016.0101]

[16] Tang Z, Li G, Fang D, *et al*. Code virtualized protection system with instruction set randomization. Journal of Huazhong University of Science & Technology, 2016,44(3):28−33 (in Chinese with English abstract). [doi: 10.13245/j.hust.160306]

[17] Collberg C, Thomborson C, Low D. Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proc. of the ACM SIGPLAN—SIGACT Symp. on Principles of Programming Languages. 1997. 184−196. [doi: 10.1145/268946.268962]

[18] Pretschner A, Pretschner A, Pretschner A, *et al*. Code obfuscation against symbolic execution attacks. In: Proc. of the Conf. on Computer Security Applications. ACM Press, 2016. 189−200. [doi: 10.1145/2991079.2991114]

[19] Guillot Y, Gazet A. Automatic binary deobfuscation. Journal of Computer Virology and Hacking Techniques, 2010,6(3):261−276. [doi: 10.1007/s11416-009-0126-4]

[20] Lemay E, Ford MD, Keefe K, *et al*. Model-based security metrics using adversary view security evaluation (ADVISE). In: Proc. of the 8th Int'l Conf. on Quantitative Evaluation of Systems. IEEE Computer Society, 2011. 191−200. [doi: 10.1109/QEST.2011.34]

[21] Mavrogiannopoulos N, Kisserli N, Preneel B. A taxonomy of self-modifying code for obfuscation. Computers & Security, 2011, 30(8):679−691. [doi: 10.1016/j.cose.2011.08.007]

[22] Zhao YJ, Tang ZY, Wang N, *et al*. Evaluation of code obfuscating transformation. Ruan Jian Xue Bao/Journal of Software, 2012, 23(3):700−711 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3994.htm [doi: 10.3724/SP.J.1001.2012.03994]

[23] Wang H, Fang D, Li J, *et al*. The research and discussion on effectiveness evaluation of software protection. In: Proc. of the Int'l Conf. on Computational Intelligence and Security. IEEE Computer Society, 2016. 628−632. [doi: 10.1109/CIS.2016.0152]

附中文参考文献:

[4] 李勇,左志宏.目标代码混淆技术综述.计算机技术与发展,2007,17(4):125−127. [doi: 10.3969/j.issn.1673-629X.2007.04.034]

[8] 房鼎益,张恒,汤战勇,等.一种抗语义攻击的虚拟化软件保护方法.四川大学学报(工程科学版),2017,49(1):159−168. [doi: 10.15961/j.jsuese.2017.01.021]

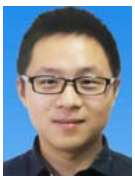
[10] 谢鑫,刘粉林,芦斌,等.Handler 混淆增强的虚拟机保护方法.计算机工程与应用,2016,52(15):146−152. [doi: 10.3778/j.issn.1002-8331.1410-0299]

[11] 吴伟民,许文锋,林志毅,等.基于增强型虚拟机的软件保护技术.计算机工程与科学,2014,36(4):655−661. [doi: 10.3969/j.issn.1007-130X.2014.04.015]

[12] 房鼎益,赵媛,王怀军,顾元祥,许广莲.一种具有时间多样性的虚拟机软件保护方法.软件学报,2015,26(6):1322−1339. http://www.jos.org.cn/1000-9825/4592.htm [doi: 10.13328/j.cnki.jos.004592]

[16] 汤战勇,李光辉,房鼎益,等.一种具有指令集随机化的代码虚拟化保护系统.华中科技大学学报(自然科学版),2016,44(3):28−33. [doi: 10.13245/j.hust.160306]

[22] 赵玉洁,汤战勇,王妮,等.代码混淆算法有效性评估.软件学报,2012,23(3):700−711. http://www.jos.org.cn/1000-9825/3994.htm [doi: 10.3724/SP.J.1001.2012.03994]



潘雁(1995－),男,安徽安庆人,硕士,主要研究领域为网络与信息安全,软件安全与保护.

祝跃飞(1962－),男,博士,教授,博士生导师,主要研究领域为网络安全,密码学.

林伟(1986－),男,博士,讲师,主要研究领域为软件保护与分析,网络安全.