第 29 卷第 3 期
2009 年 3 月

文章编号:1001 - 9081(2009)03 - 0817 - 06

基于自封闭代码块的软件保护技术

周立国,熊小兵,孙　洁
(信息工程大学 信息工程学院,郑州 450002)
(bingxiaoxiong@ yahoo. com. cn)

摘　要:针对传统的基于垃圾指令插入的花指令技术在软件保护应用中的不足,提出了一种基于自封闭代码块的软件反静态分析和动态调试的软件保护技术。重点介绍了自封闭代码块的相关概念,阐述了自封闭代码块的自动生成技术,包括基于指令编码表的随机指令序列生成技术和基于指令逆向思想的逆指令序列生成技术,并给出了相关算法和实例分析。

关键词:软件保护;反调试;自封闭代码块;逆指令序列
中图分类号: TP309.7;TP311　　**文献标志码**:A

Software protection technology based on self-sealing code block

ZHOU Li-guo, XIONG Xiao-bing, SUN Jie
(*College of Information Engineering, Information Engineering University, Zhengzhou Henan* 450002, *China*)

Abstract: With regard to the shortage of trash instruction inserting using software protection of conventional technology, a new software protection technology was proposed based on the concept of self-sealing code block. With the help of the new technology, the software could be protected from static analysis and dynamic debugging. The generation method of random instruction sequence and relevant reverse instruction sequence for constructing self-sealing code blocks were emphasized, including random original instruction sequence generation based on instruction encode table and reverse instruction sequence generation based on reverse engineering. The corresponding algorithms and sample analysis were provided.

Key words: software protection; anti-debug; self-sealing code block; reverse instruction sequence

0　引言

随着各种软件逆向分析技术的发展,商业软件均面临着被破解的危险,给软件厂商造成重大的经济损失,也不利于软件行业的良性发展。为此,计算机软件保护技术越来越受到各软件厂商的重视[1]。

反跟踪调试技术在软件保护中有广泛的应用,主要包括反动态调试和反静态分析两个方面。当前的软件保护技术有信息隐藏、分段加密及花指令技术等[2-3],本文主要对花指令技术进行探讨。花指令技术是软件反跟踪调试的重要技术手段,即在程序中插入一些指令序列,这些指令序列不会改变软件的运行效果,但能干扰软件破解者对软件的逆向分析,加大软件破解难度。

传统的花指令技术多采用插入垃圾指令的方式[4],为了不破坏原解密代码的执行现场,需采取一定的保护措施,为此,所插入的垃圾指令具有一定的规律:

1)在插入的垃圾指令代码块头尾处分别插入 pushad 和 popad 指令,或是将各寄存器顺序入栈和出栈的指令,从而保证垃圾指令执行前后寄存器状态的一致性。

2)每次插入垃圾指令代码块前都插入一条 jmp 跳转指令,跳过所插入的垃圾指令,使其不被执行。

采用上述插入垃圾指令的软件保护技术,能较好地对抗基于静态分析的软件破解技术,但对于基于动态调试的破解技术,保护效果并不明显,借助动态调试工具(比如 Ollydbg),能有效检测软件中的垃圾指令,从而突破软件保护。

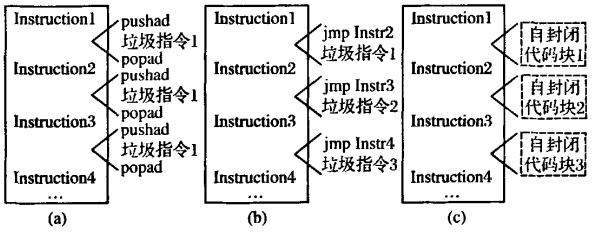

图 1　基于自封闭代码块的与基于垃圾指令插入的软件保护技术对比

为了加强软件的自我保护功能,本文提出了一种基于自封闭代码块的软件保护技术。基本思想是:在程序中随机插入不改变软件执行状态的自封闭代码块指令组,这些自封闭代码块在软件调试者看来,属于软件的一部分,与其他功能模块没有区别,但是其执行前后,软件的系统状态保持不变。该技术不仅能有效实现软件的静态反汇编分析,又能加大软件动态调试的难度,更好

收稿日期:2008 - 09 - 05;**修回日期**:2008 - 11 - 14。

作者简介:周立国(1983 -),男,吉林延吉人,硕士研究生,主要研究方向:网络信息安全;　熊小兵(1985 -),男,江西丰城人,硕士研究生,主要研究方向:网络信息安全;　孙洁(1984 -),女,硕士研究生,主要研究方向:网络信息安全。

地实现软件自我保护功能。基于自封闭代码块的软件保护技术和传统的基于垃圾指令插入的软件保护技术的应用效果对比如图1所示,可以看出传统的垃圾指令(图1(a,b)所示)规律性太强,易于分析破解,且不适用于反动态跟踪分析。

基于自封闭代码块的软件保护技术的实现关键是随机构造不改变程序执行状态的自封闭代码块。本文引入了指令逆向分析的思想[5-6],将自封闭代码块分成原指令序列和逆指令序列两个部分,其构造过程也分成随机指令序列的自动生成和逆指令序列的自动生成两个方面。

1 基本概念介绍

在具体讨论自封闭代码块生成技术之前,先对本文将要涉及的一些基本概念作简要介绍。

定义1 自封闭代码块。是指这样一组指令序列,其执行结果不会改变任何寄存器的值,也不改变内存单元的值[7]。图2中列举了一些典型的简单自封闭代码块,其中R表示寄存器,X表示立即数。本文构造的自封闭代码块并不是图2中几种典型自封闭代码块的简单组合。

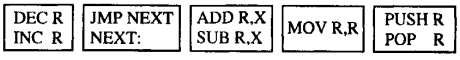

图2 几种典型的简单自封闭代码块

定义2 基本块。是程序中按序执行的一组指令,除基本块中最后一条指令外,基本块内不包含有转移指令。可以按以下规则划分为基本块:指令 i_k 和 i_{k+1} 在同一基本块中,当且仅当若指令 i_k 在第 n 步执行,则指令 i_{k+1} 必在第 $n+1$ 步执行;并且若指令 i_{k+1} 在第 $n+1$ 步执行,则指令 i_k 必在第 n 步执行。

定义3 程序状态序列。是指一组状态的集合 $S=(S_1, S_2, \cdots, S_n)$,其中的每一个状态 $S_i=(PC_i, M_i, R_i)$ 都表示程序运行时对处理器、内存和寄存器的改变。该状态序列与一个指令序列 $I=(\alpha_1, \alpha_2, \cdots, \alpha_n)$ 相关联,其中 $\alpha_j \in I$ 是使 P 从状态 S_{i-1} 转移到 S_i 的指令。

定义4 逆指令组(RIG)。假定对I中的任一指令 α_j,存在一个包含一条或多条指令的指令组(RIG_j),当 P 的状态为 S_i 时,执行 RIG_j 能使 P 从 S_i 回转到 S_{i-1},则称 RIG_j 为指令 α_j 的逆指令组,即 RIG_i 为使处理器从执行指令 α_j 时的状态 S_i 回转到 S_{i-1} 的指令序列。

定义5 逆指令序列(RIS)。令指令序列 IS_1 在处理器 P 上执行时,P 可达到的状态序列为 $S=(S_1, S_2, \cdots, S_n)$,在 S_i 与其前一状态 S_{i-1} 之间,有且仅有一条指令直接改变存储器或寄存器的值。假设存在另一个指令序列 IS_2,其执行使 P 的状态序列为 $S'=(S_n, S_{n-1}, \cdots, S_1)$,且对于 P 的任一状态 $S_i=(PC_i, M_i, R_i)$,IS_2 中都存在一个指令组,它的执行可使 P 的状态恢复到 S_{i-1},则称 IS_2 是 IS_1 的逆指令序列RIS。

2 自封闭代码块生成技术

自封闭代码块是由随机指令序列及其逆指令序列构成的,其构造过程包括随机指令序列的生成和逆指令序列的生成两部分内容,以下分别介绍生成这两种序列所采用的相关技术。

2.1 随机指令序列生成方法

生成随机指令时,本文采用了一种基于指令编码表的随机指令生成方法。该方法事先按照指令长度构造出各类指令和寄存器的编码表,需要生成指令时,随机选取编码表中的相应指令编码,对于需要填入立即数、内存地址或寄存器的指令,生成随机数将其填充成完整指令。表1为1字节和5字节的部分指令编码表。

表1 部分指令编码表

1 Byte		5 Byte	
机器码	汇编指令	机器码	汇编指令
046h	inc esi	0B8h	mov eax
040h	inc edx	0BEh	mov esi
04Ah	dec edx	02Dh	sub eax
092h	xchg edx, eax	025h	and eax

此时,如果根据产生的随机数在1字节编码表中选取第2个元素,则生成指令 inc edx;若是选取5字节编码表中的元素0B8h,它代表指令 mov eax, × × × ×,其中 × × × × 为任意立即数或某一内存单元。为简化实现步骤,本文中生成的随机指令均为非访存类指令。

采用上述方法能随机生成不同的指令,且无需先列出候选指令集,但指令序列比较简单,不存在指令间跳转等情况。为了提高随机指令序列的复杂性,本文还讨论了一种基于指令链的转移指令插入方法,该方法需添加一份转移指令的编码表。下面举例说明在生成的随机指令序列中插入转移指令的方法。表2给出部分转移指令的编码表。

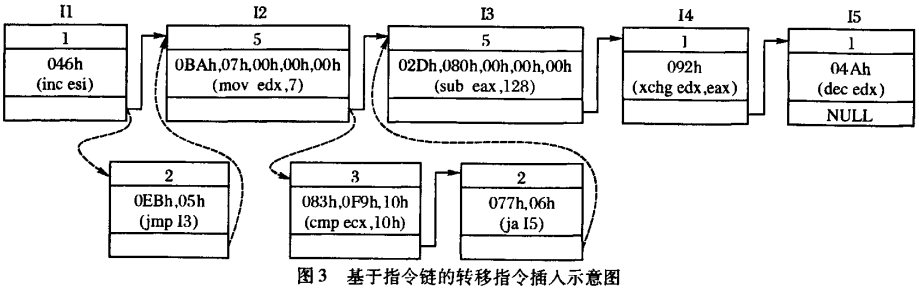

图3 基于指令链的转移指令插入示意图

表2 部分转移指令编码表

机器码	汇编指令	机器码	汇编指令
EBh,00h	JMP SHORT	7Dh,00h	JGE/JNL
74h,00h	JZ/JE	70h,00h	JO
75h,00h	JNE/JNZ	7Ah,00h	JP/JPE
73h,00h	JAE/JNB/JNC	7Bh,00h	JNP/JPO
79h,00h	JNS	78h,00h	JS

按照前面介绍的随机指令序列生成步骤,从指令编码表中随机选取若干条指令,假设已选取了5条指令(I1 ~ I5),它们构成的指令结构链如图3所示。在这5条指令之间选择任意位置插入转移类指令,图中示意了在I1与I2、I2与I3之间分别插入转移指令 jmp I3 和 ja I5 的方法。插入 jmp I3 指令时,根据跳转长度(即I2的指令长度)填充该跳转指令为0EBh,05h,并修改I1和该跳转指令结构中的指针;对于ja类的条件转移指令,在其结构前还要产生一条 cmp 比较指令

(在比较类指令编码表中随机选取),同时修改 ja 指令的跳转长度和相关结构的指针,将生成的 cmp ecx,10h 和 ja l5 两条指令插入指令结构链中。

图 3 中采用基于指令链的转移指令插入法生成的含有转移分支的随机指令序列如下:

```
46                  inc esi
EB05                jmp l3
BA07000000          mov eax, 7
83F910              cmp ecx, 10h
7706                ja l5
              l3:
2D80000000          sub eax, 128
92                  xchg edx, eax
              l5:
4a                  dec edx
```

2.2　逆指令序列生成方法

利用 2.1 节所示方法生成随机指令序列后,采用逆指令生成技术构造该指令序列的逆序列,将这两个指令序列合并成完整的自封闭代码块。

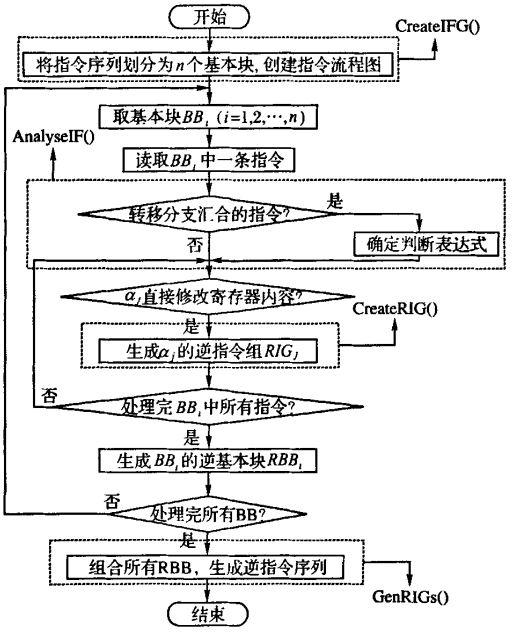

图 4　生成逆指令序列算法流程

逆指令生成技术的基本思想是:将指令序列划分为多个基本块,根据各基本块之间的相互依赖关系,建立相应的指令流程图(IFG)(即根据指令的执行过程构造的流程图);以基本块为基本单位,逐级恢复指令状态集中的状态 S_i,生成 BB_i 中每条指令 $\alpha_j \in I$(I 为改变程序状态的指令集)对应的逆指令组(RIG_j),然后按一定的规则整合所有的 RIG_j,得到原指令序列的逆指令序列(RIS),本文给出一种逆指令序列生成算法 CreateRIS,算法流程如图 4 所示。

算法中 CreateIFG( )用来划分指令序列中的基本块,并建立指令流程图(IFG);AnalyseIF( )是根据 IFG 中的指令流区间(IFIs)来分析指令流程信息,找出条件分支,并确定分支的判断条件;CreateRIG( )用来产生 BB_i 中每条指令 $\alpha_j(\alpha_j \in I, I$ 为改变寄存器内容的指令集)的逆指令组;GenRIGs( )用来组合所有的逆指令组,生成最终的逆指令序列。以下通过实例讨论 CreateIFG( )、AnalyseIF( )、CreateRIG( )和 GenRIGs( )函数的功能和采用的相关技术。

2.2.1　CreateIFG 函数

将 2.1 节中生成的随机指令序列以链表形式存储,链表首指针作为 CreateIFG( )的输入,将链表中的指令划分成若干个基本块单元,构建该指令序列的指令流程图 IFG。IFG = (N, E, start, exit),其中 N 是由基本块组成的节点集,E 是所有边的集合,start 和 exit 分别是该基本块的输入和输出节点。以下给出 CreateIFG( )的算法描述。

CreateIFG( ):创建随机指令序列的指令流程图 IFG
输入:随机指令序列链表的首指针
输出:该指令序列的 IFG

```
Begin
  1   遍历输入指令链,划分基本块 BBs
  2   IFG = NULL
  3   添加 start 块到 IFG 中
  4   MakeLabel(start)          /*给 start 块的输出边赋值*/
  5   repeat
  6     α = Read_Next_Instruction( )
  7     if 当前 BB 结束 then
  8       将该 BB 添加到 IFG 中
  9       MakeLabel(BB)         /*给 BB 节点的输出边赋值*/
  10    end if
  11  until 指令处理完毕
  12  添加 exit 块到 IFG 中
end
```

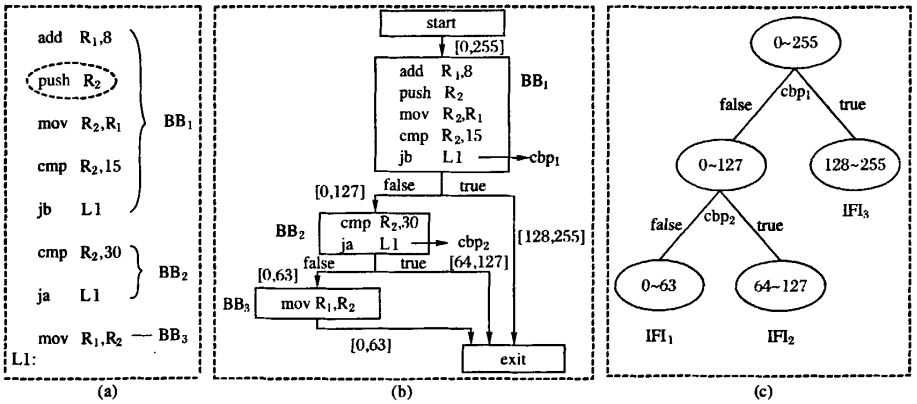

图 5　生成 IFG 实例图

如图 5 所示,(a)是产生的随机指令序列,其中 R_1 和 R_2 是任意的通用寄存器,按照基本块的定义,可将其划分为三个

基本块($BB_1 \sim BB_3$),其中 push R_2 是为保存 R_2 的原始状态而插入的指令。利用 CreateIFG( )依次将 start 块、三个基本块和 exit 块加入该指令序列的 IFG 中。为了产生 IFG 的指令流区间,以便 AnalyseIF( )对指令流序列进行分析,每加入一个基本块时,调用 MakeLabe( ),根据指令流给各边赋权值,赋值方法遵循二叉树原则。

2.2.2 AnalyseIF 函数

AnalyseIF( )根据 IFG 中的 IFIs 分析指令序列的执行流程。由于各基本块内部的指令都是顺序执行的,其逆指令序列就是该基本块中 RIGs 的反序,但对于基本块之间存在条件分支的情况,则要找出逆指令序列中分叉点,确定其逆指令序列中的分支及分支转移的判断条件。

仍以图 5 为例,(b)中的汇合点是 exit 块,它有三条权值分别为[0,63]、[64,127]和[128,255]的输入边。从(c)中指令流区间(IFIs)二叉树可看出,[0,63]为 IFI_1,表示条件分支 cbp_2($R_2 > 30$)为假的情况;[64,127]是 cbp_2 为真且 cbp_1($R_2 < 15$)为假时的指令流区间 IFI_2;[128,255]是 cbp_1 为真的 IFI_3。可见,在逆指令序列中 exit 块成为分叉点,存在三条输出边分别到达三个基本块。由于 R_2 的值作为 cbp_1 和 cbp_2 的判断条件,在各基本块汇合到 exit 块之前都没有被改变,所以可将其作为逆序列中各分支的判断条件。由此可确定逆指令序列中 exit 作为三条分支的分叉点,存在两个判断表达式 $R_2 > 30$ 和 $R_2 < 15$,分别对应原 IFG 中的 cbp_2 和 cbp_1。

如图 6 所示,R_2 作为 cbp_1、cbp_2 的判断条件,$R_2 < 15$ 和 $R_2 > 30$,其值在 BB_2 和 BB_3 中分别被指令 dec R_2 和 sub R_2,3 改变,通过两条输入边[0,63]和[64,127]到达 exit 块时 R_2 已经发生变化,所以不能将原来的分支条件作为逆序中该分叉点的判断表达式。这时采用一种简单的处理方法:去除 exit 块之前各基本块中改变 R_2 内容的指令,从而使所求的判断表达式与原 IFG 中的分支判断条件相同。

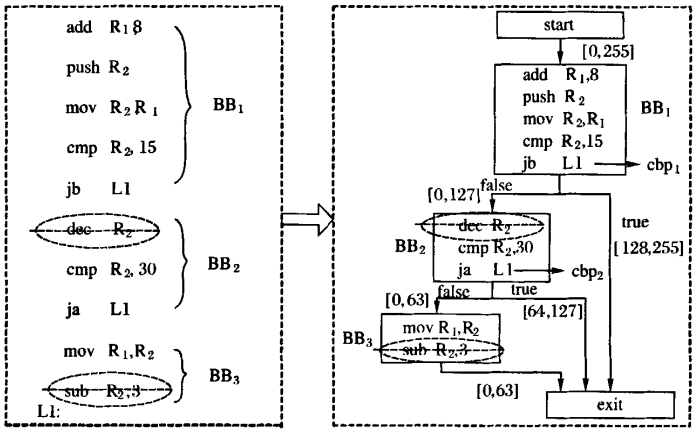

图 6 针对分支条件寄存器被改变的情况处理示意图

2.2.3 CreateRIG 函数

CreateRIG( )是 RIS 算法中最重要的函数,用来产生 BB_i 中每条指令 α_j($\alpha_j \in I$,I 为改变寄存器内容的指令集)的逆指令组。

为了区分各寄存器不同阶段的值,该函数首先采用重命名的方法,将指令序列中被改变的寄存器命名为 R_i^j($i, j = 0, 1, 2, \cdots$,i 是不同寄存器的编号,j 表示某个寄存器在不同阶段的状态编号,初始状态时 $j = 0$)。根据原指令序列的 IFG 和 IFIs,记录指令中寄存器被改变的情况,然后产生用来恢复指令 α_j 中被破坏的寄存器的一条或多条指令,得到 α_j 的逆指令组 RIG_j,最后组合基本块 BB_i($i = 2,3,\cdots$)中所有指令的逆指令组,生成该基本块的逆基本块 RBB_i。

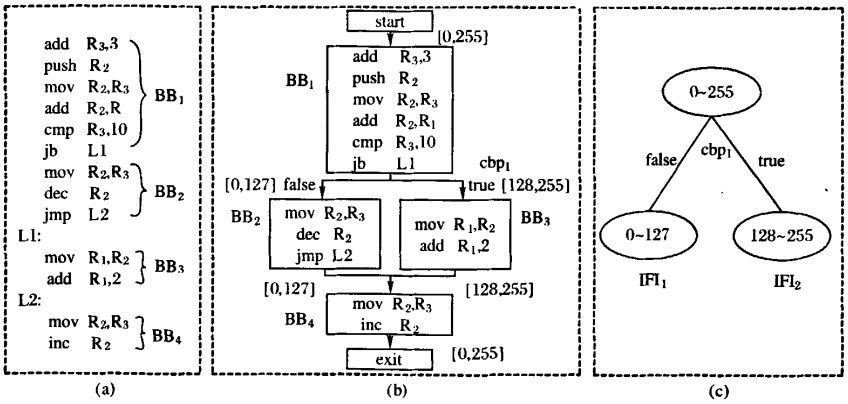

图 7 用于生成重命名表的 IFG 和 IFIs 示意图

如图 7(a)中给出的指令序列,$R_1 \sim R_3$ 是不同的寄存器,(b)和(c)分别是该指令序列的 IFG 和 IFIs。根据 IFIs 对寄存器重命名处理,R_i 的初值为 R_i^0($i = 1,2,3$),在不同基本块中其内容被改变,根据原指令序列的 IFG 和 IFIs,可得图 8 所示的寄存器变化过程图,依此记录寄存器内容变化情况,构建重命名表,如表 3 所示。

表 3　寄存器重命名表

指令序列	R_1	R_2	R_3
IFI_1	R_1^0	$R_2^0, R_2^1, R_2^2, R_2^3$	R_3^0, R_3^1
IFI_2	R_1^0, R_1^1	R_2^0, R_2^1, R_2^3	R_3^0, R_3^1

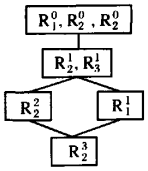

图 8　寄存器变化示意图

根据重命名表，通过逐级恢复表中寄存器各阶段的值，可生成破坏其值的指令 α_j 的逆指令组 RIG_j，本文提出了重定义和赋值引用两种方法来实现寄存器内容的恢复。

1）重定义法。通过查找 α_j 所改变的寄存器 R 的上一条赋值指令，根据与其改变相关的其他寄存器的状态改变情况来恢复该寄存器的值，得到 α_j 的逆指令组。

2）赋值引用法。通过查找 α_j 所改变的寄存器 R 的上一条赋值引用指令，根据被其赋值的寄存器的状态改变情况，来恢复该寄存器的值，得到 α_j 的逆指令组。

仍以图 5 中指令序列为例，为方便描述，将指令序列中各基本块转换为伪指令。图 9(a) 是图 5 中指令序列转化后的 IFG，BB_4 中的指令 α : $R_2 = R_3 + 1$ 是分支汇合指令，该指令改变了寄存器 R_2，结合重定义法和赋值引用法来恢复 R_2 的值。在 false 分支中找到改变 R_2 的上一条赋值指令 $R_2 = R_3 - 1$，分支 true 中找到 R_2 的赋值引用指令 $R_1 = R_2 + 2$，相关的寄存器分别是 R_3 和 R_1，在这两条指令与 α 之间 R_3 和 R_1 均未发生变化，利用重定义法可直接通过指令 $R_2 = R_3 - 1$ 来恢复 false 分支中 R_2 的值，而 true 分支中 R_2 的值可用赋值引用法通过指令 $R_2 = R_1 - 2$ 来恢复。在逆指令序列中，指令 α 成为分叉点，其逆指令 $R_2 = R_3 - 1$ 和 $R_2 = R_1 - 2$ 分别对应两个条件分支。由此可得 BB_4 的逆基本块 RBB_4，如图 9(b) 所示。

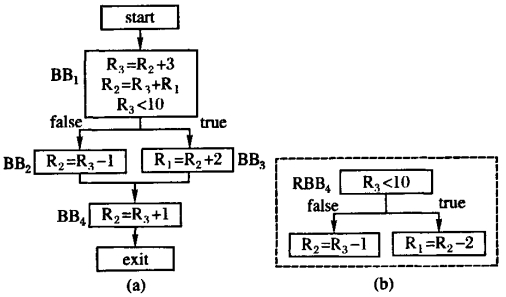

图 9　重定义法生成逆指令示意图

2.2.4　GenRIGs 函数

GenRIGs( ) 是将 CreateRIG( ) 得到的所有逆基本块 RBBs 组合，生成原指令序列的逆指令序列。

该函数在处理各基本块时遵循自底而上的原则，例如图 7(a) 中的指令序列，基本块顺序为 $BB_1 \rightarrow BB_2 \rightarrow BB_3 \rightarrow BB_4$，$BB_i$ 对应的逆基本块为 $RBB_i (i = 1,2,3,4)$，那么生成的逆指令序列，其基本块的顺序为 $RBB_4 \rightarrow RBB_3 \rightarrow RBB_2 \rightarrow RBB_1$。另外，原指令序列中的条件分支汇合点对应逆指令序列中的一个分叉点，并插入一个分支条件判断指令，其转移的判断条件与原指令序列中的分支判断条件相同，而原分叉点对应逆指令序列中的分支汇合点。为了实现这一过程，需要在逆指令序列中插入一些非条件跳转指令。

如图 9 中的例子，设由 CreateRIG( ) 生成的逆基本块为 $RBB_i (i = 1,2,3,4)$，通过插入不同的跳转指令，逆指令序列的基本块顺序为 $RBB_4 \rightarrow RBB_3 \rightarrow RBB_2 \rightarrow RBB_1$。图 10 是该指令序列的 IFG 与生成的逆指令序列的 IFG(即 RIFG) 对照图，图中 BB_4 中的交汇点 P_0 变成了 RIFG 中的分叉点 Fr，BB_1 中的分叉点成为 RIFG 中的交汇点 Pr，在组合各个逆基本块时，在分叉点 Fr 的输出边处插入一条分支判断指令，判断表达式与原指令序列中的分支判断条件相同($R_3 < 10$)，而在汇合点 Pr 的一条输入边上需要插入一条非条件跳转，使其到达汇合点。

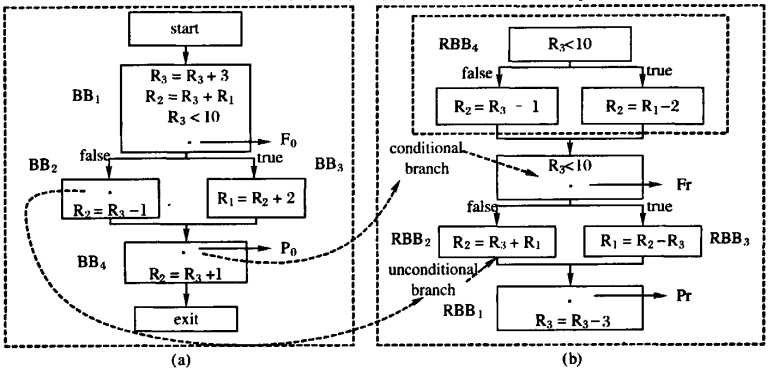

图 10　指令序列的 IFG 与 RIFG 对照图

综合以上逆指令序列的产生过程，即能生成一段随机指令序列的逆指令序列。图 11 举例说明了生成给定指令序列的逆指令序列的完整过程。

图 11 中，(a) 是产生的随机指令序列的部分代码，其中 $R_0 \sim R_2$ 是通用寄存器，X 是立即数；通过上述构造逆指令序列的方法，先由 CreateIFG( ) 划分该指令序列中的基本块，并根据程序流程给基本块各输出边赋值，生成指令流程图，如 (b) 所示，其中 F_0 和 P_0 是利用 AnalyseIF ( ) 找到的转移分支的分叉点和交汇点；(c) 中 $RBB_i (i = 1,2,3,4)$ 是 CreateRIG( ) 采用两种恢复寄存器值的方法产生的各基本块的逆指令块，在 GenRIGs( ) 综合 RBBs 时，原交汇点 P_0 变成分叉点 Fr，原分叉点成为交汇点 Pr，并在 Fr 处和 Pr 的任一输入边处分别插入条件分支指令和非条件跳转指令，构成原指令序列的逆指令序列的控制流程图，从而得到 (d) 中所示的逆指令序列。将图 11(a) 中的指令序列和 (d) 中的逆指令序列合并，即可构成一个完整的自封闭的代码块。

3　结语

本文分析了传统的基于垃圾指令插入的软件保护技术在程序反跟踪调试中存在的不足，提出了一种基于自封闭代码块的软件反静态分析和动态调试的软件保护技术。这种自封闭代码块与传统的垃圾指令不同，可以插入程序中的任何位

置,也可以和程序的其他指令一样被执行,但其执行不会改变程序的系统状态。

本文首先介绍了自封闭代码块的相关概念,然后重点介绍了自封闭代码块的自动生成技术,包括基于指令编码表的随机指令序列的生成技术和基于指令逆向思想的逆指令序列生成技术,并给出了相关实现算法和实例分析。不足之处在于生成的自封闭代码块只包含非访存类指令,不包含存储器操作指令,致使指令样式较少,这也是下一步工作的重点。

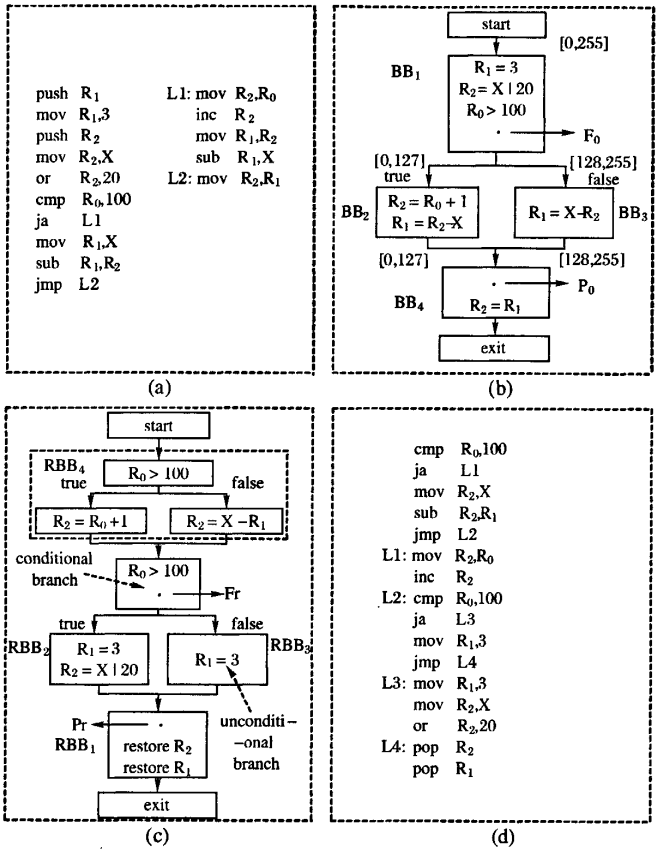

图 11 逆指令序列生成过程示例图

参考文献:

[1] 剑川. 软件保护技术[EB/OL]. [2008 - 09 - 01]. http://greentoad.bokee.com/viewdiary.13791529.html.

[2] MANA A, PIMENTEL E. An efficient software protection scheme[C]// Proceedings of IFIP SEC'01. Paris: Kluwer, 2001:385 - 401.

[3] GOLDREICH O, OSTROVSKY R. Software protection and simulation on oblivious rams[J]. Journal of ACM, 2003,43(3):431 - 473.

[4] 段钢. 加密与解密[M]. 2 版. 北京: 电子工业出版社, 2005.

[5] AKGUL T, MOONEY Ⅲ V J. Instruction-level Reverse Execution for Debugging [EB/OL]. [2008 - 09 - 01]. http://codesign.ece.gatech.edu/publications/tankut/paper/git-cc-02-49.pdf.

[6] CRESCENZI P. Reversible execution and visualization of programs with Leonardo[J]. Journal of Visual Languages and Computing, 2000, 11(2): 125 - 150.

[7] 祝恩, 殷建平, 蔡志平, 等. 计算机病毒自动变形机理的分析[J]. 计算机工程与科学, 2002, 24(6): 14 - 17.

(上接第 738 页)

参考文献:

[1] 朱仲杰, 蒋刚毅, 郁梅, 等. 目标基视频编码中的运动目标提取与跟踪新算法[J]. 电子学报, 2003, 31(9): 1426 - 1428.

[2] MOTAMED C. Motion detection and tracking using belief indicators for an automatic visual-surveillance system[J]. Image and Vision Computing, 2006, 24(11): 1192 - 1201.

[3] DEMIRDJIAN D, KO T, DARRELL T. Untethered gesture acquisition and recognition for virtual world manipulation[J]. Virtual Reality, 2005, 8(4): 222 - 230.

[4] KIMA K, CHALIDABHONGSEB T H, HARWOOD D, *et al*. Real-time foreground - background segmentation using codebook model [J]. Real-Time Imaging, 2005, 11(3): 172 - 185.

[5] WANG JUNXIAN, BEBIS G, MILLER R. Robust video-based surveillance by integrating target detection with tracking[C]// IEEE Workshop on Object Tracking and Classification Beyond the Visible Spectrum. Washington, DC: IEEE Computer Society, 2006: 137 - 137.

[6] GUPTE S, MASOUD O. Detection and classification of vehicles [J]. IEEE Transactions on Intelligent Transportation Systems, 2002, 39(1): 37 - 47.

[7] LIPTON A, FUJIYOSH I H, PATIL R. Moving target classification and tracking from real-time video[C]// Proceedings of IEEE Workshop on Application of Computer Vision. Washington, DC: IEEE Computer Society, 1998: 8 - 14.

[8] WU YOUFU, SHEN JUN, DAI MO. Traffic object detections and its action analysis[J]. Pattern Recognition Letters, 2005, 26(13): 1963 - 1984.

[9] 李宏友, 汪同庆, 叶俊勇, 等. 基于 3D OGHM 的视频运动目标检测算法[J]. 电子学报, 2008, 36(3): 604 - 608.

基于自封闭代码块的软件

作者： 周立国， 熊小
作者单位： 信息工程大学,信息工程学院,郑州,450002
刊名： 计算机应用 ISTIC PKU
英文刊名： JOURNAL OF COMPUTER APPLICATIONS
年，卷(期)： 2009，29(3)
被引用次数： 0次

参考文献(7条)

1. 剑川 软件保护技术 2008

2. MANA A. PIMENTEL E An efficient software protection scheme 2001

3. GOLDREICH O. OSTROVSKY R Software protection and simulation on oblivious rams 2003(03)

4. 段钢 加密与解密 2005

5. AKGUL T. MOONEY Ⅲ V J Instruction-level Reverse Execution for Debugging 2008

6. CRESCENZI P Reversible execution and visualization of programs with Leonardo 2000(02)

7. 祝恩. 殷建平. 蔡志平 计算机病毒自动变形机理的分析[期刊论文]-计算机工程与科学 2002(06)

相似文献(5条)

1. 学位论文 秦杰 基于IAT加密的加壳程序研究 2009

软件保护是软件开发中一个不可忽视的环节，由于软件发布后要面对众多逆向分析人员的研究，给要发布的软件加一层壳现在几乎成了保护软件的一个必要的步骤，加壳软件的目的就是为软件穿上一层“盔甲”以阻碍对其逆向分析，随着各种逆向分析工具和技术的不断发展，壳对抗逆向分析的能力就直接反映出了它的保护强度。

现有的壳主要是用于保护Win32环境下的可执行程序、动态链接库、驱动程序等PE格式的文件，由于PE文件头中包含了很多重要的信息，为了尽量减少暴露在逆向分析人员面前的信息，加壳程序通常需要生成自己的PE文件头，并模拟操作系统对被保护的程序进行加载。逆向分析人员在对加壳保护后的程序进行分析时，会采用各种动态、静态的逆向分析工具，壳里就需要加入一些反逆向的手段来给他们增加一定的难度。壳除了对软件进行保护外，对自身也需要隐藏，由于常见的壳识别工具采用的是特征码检测的方式，所以在壳中加入多态变形技术来对抗特征码检测也是加壳程序编写过程中的一个重要步骤。

本文首先论述了软件保护技术的发展及对软件进行加壳的重要性，阐明了本研究课题的特点及实用价值；然后，对加壳程序开发所需的基础知识及关键技术需要的理论进行了全面的介绍；壳的核心技术是反调试，它直接影响到壳的保护强度，强度高的壳通常是因为采用了一些未公开的私有的反调试手段，本文接下来通过逆向跟踪几款知名的保护壳后，揭露了它们所用的反调试手段，对它们各自的优缺点进行了总结并给出了一些建议，在壳的IAT加密部分提出了完全模拟系统API的思想；最后，通过对BPE32多态变形引擎源码的详细研究，构建出一个能为壳所用的微型变形引擎框架并实现了相关部分代码，并对对抗多态变形常用的启发式代码仿真检测技术进行了介绍，总结了相应的突破方法和部分代码。

本文分析知名壳的反调试后整理出来的函数代码，可以直接用于加壳程序开发，在实际应用中可以根据具体情况灵活选择运用；实现的多态变形引擎框架经过扩充后也可以用在实际开发中。

2. 学位论文 伍祚春 软件保护相关技术研究及实现 2007

自计算机诞生之日起，其技术的发展可谓日新月异，各种新技术、新思路不断涌现。各种软件从最初的几个、几十个字节发展到现在的动辄几张光盘，成千上万的共享软件和商业软件越来越大，技术内涵也日趋复杂。一款优秀的软件，其技术秘密往往成为他人窃取的重点。随着软件以共享方式在网络上发布这种方式的流行，软件保护和数据加密技术的迫切性被越来越突出地表现出来。

目前的保护技术包括软加密和硬加密两种方式，用硬件保护是一种较安全的技术，但是成本很高，纯软件加密因经济方便而蓬勃发展，当前最常见的是外壳加密技术。本课题的目的正是针对这种情况，致力于剖析当前流行的脱壳技术的特点，并且确定加壳方案，从而编制相应软件。

本课题首先对目前Windows平台上的主流可执行文件格式——PE格式进行深入分析，从壳的概念和壳的加载过程入手，全面分析了反跟踪技术、手动脱壳技术和各种工具处理后的壳的特点，重点剖析了壳的构造以及执行机理。

在认识到传统加壳软件已被广泛深入地研究，保护能力不强的基础上，课题采用模块化的结构设计了一款简单的加壳工具。对文件的压缩处理采用”LIB压缩函数库，针对PE文件中资源区块的特殊性，对资源目录和资源数据分别进行了处理。同时，为了提高安全性，对原始的输入表也进行了必要的变形和转储，并设计了相应的结构。在确定大致方案后，使用VM技术保护重要代码，再使用加壳软件加壳，基本可以实现了对简单软件的有效保护，课题也从反调试技术的角度出发，深入地分析了如何更合理地保护程序不被破解的方法，进而在程序设计中采用相应的对策得以实现。

最后，对课题中的加壳工具存在的问题进行了分析，给出了当前提高应用软件安全性的主要方式。另外，本文给出了Windows环境下32位汇编语言程序设计的一些相关知识。

3. 学位论文 刘晓冬 软件加壳技术的研究与实现 2006

自计算机诞生之日起，其技术的发展可谓日新月异，各种新技术、新思路不断涌现。各种软件从最初的几个、几十个字节发展到现在的动辄几张光盘，成千上万的共享软件和商业软件越来越大，技术内涵也日趋复杂。一款优秀的软件，其技术秘密往往成为他人窃取的重点。随着软件以共享方式在网络上发布这种方式的流行，软件保护和数据加密技术的迫切性被越来越突出地表现出来。

目前的保护技术包括软加密和硬加密两种方式，用硬件保护是一种较安全的技术，但是成本很高，纯软件加密因经济方便而蓬勃发展，当前最常见的是外壳加密技术。本课题的目的正是针对这种情况，致力于剖析当前流行的脱壳技术的特点，并且确定加壳方案，从而编制相应软件。

本课题首先对目前Windows平台上的主流可执行文件格PE格式进行深入分析，从壳的概念和壳的加载过程入手，全面分析了反跟踪技术、手动脱壳技术和各种工具处理后的壳的特点，重点剖析了壳的构造以及执行机理。

在认识到传统加壳软件已被广泛深入地研究，保护能力不强的基础上，课题采用模块化的结构设计了一款简单的加壳工具。对文件的压缩处理采用APLIB压缩函数库，针对PE文件中资源区块的特殊性，对资源目录和资源数据分别进行了处理。同时，为了提高安全性，对原始的输入表也进行了必要的变形和转储，并设计了相应的结构。在确定大致方案后基本实现了对简单软件的有效保护，课题也从反调试技术的角度出发，深入地分析了如何更合理地保护程序不被破解的方法，进而在程序设计中采用相应的对策得以实现。

最后，对课题中的加壳工具存在的问题进行了分析，给出了当前提高应用软件安全性的主要方式。另外，本文给出了Windows环境下32位汇编语言程

序设计的一些相关知识。

4.期刊论文 杨宏群.陈春华 结构化异

SEH是软件保护中常用的技术.本文介绍了SEH的

5.学位论文 吴超 windows环境下隐蔽调试器的设计与实现 2009

在安全事件中，恶意代码造成的经济损失占有最大的比例。恶意代码的检测技术总是滞后于新恶意代码的出现。一方面是人们很难区别正常代码和恶意代码；另一方面，很多信息系统缺少必要的保护措施。因此，人们常常被恶意代码欺骗，而无意地执行恶意代码。可见恶意代码被引入系统并执行是不可避免的，监控、分析或检测二进制程序是否为恶意程序已经成为现阶段的研究热点。

恶意代码为了防止被分析，都有很强的自保护功能，现在主要的调试器很难对恶意代码进行分析。同时，软件保护技术的发展使大量的反调试工具出现，既有针对源代码的反调试工具，又有针对二进制程序的反调试工具。在反调试工具中，加壳工具功能比较突出，它综合使用多种技术对抗代码分析调试，而且加壳工具使用非常简单，越来越多的恶意代码使用加壳技术保护自己。

本文在分析了很多反调试技术后，包括检测断点技术、检测调试器技术和自动修改代码技术等，设计并实现了一种隐蔽性强的调试器。该隐蔽调试器利用windows分页管理机制，巧妙地对目标进程设置断点以获取控制，很好地实现了隐蔽调试的特性。

为获得对目标进程的控制权，本调试器利用了windows系统中进程的地址空间分为用户空间和系统空间，代码执行之前系统会进行权限检查的特点，通过修改了目标进程内存页面属性，使目标进程执行时出现异常，从而获得目标进程的控制权。为增强隐蔽性，本调试器还采用了多种设计技术，包括：通过向目标进程插入shellcode控制目标进程、不产生在windows系统中的注册信息等等。在功能使用性方面，本隐蔽调试器为系统增加了一些新的系统调用，以方便实现调试功能设置和用户交互。

实验和测试证明，本文提出的调试方法可以作为现行调试技术的一种有效的补充。