

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<sup>[1]</sup>。

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<sup>[2-3]</sup>,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<sup>[4]</sup>,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。

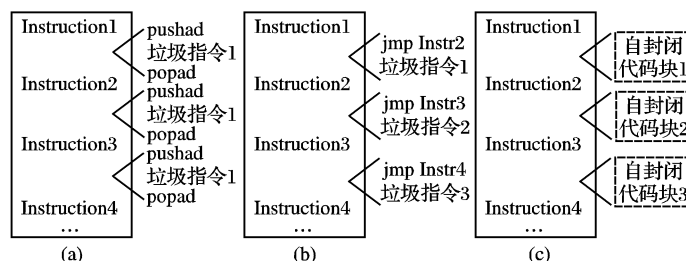


图1 基于自封闭代码块的与基于垃圾指令插入的软件保护技术对比

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<sup>[5-6]</sup>,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<sup>[7]</sup>。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。

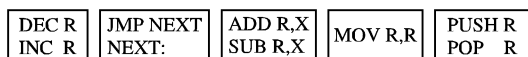


图 2 几种典型的简单自封闭代码块

**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  $S = (S_1, S_2, \dots, S_n)$ ,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  $I = (\alpha_1, \alpha_2, \dots, \alpha_n)$  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  $\alpha_j$ ,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  $\alpha_j$  的逆指令组,即  $RIG_i$  为使处理器从执行指令  $\alpha_j$  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  $S = (S_1, S_2, \dots, S_n)$ ,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  $S' = (S_n, S_{n-1}, \dots, S_1)$ ,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,  $\times \times \times \times$ ,其中  $\times \times \times \times$  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。

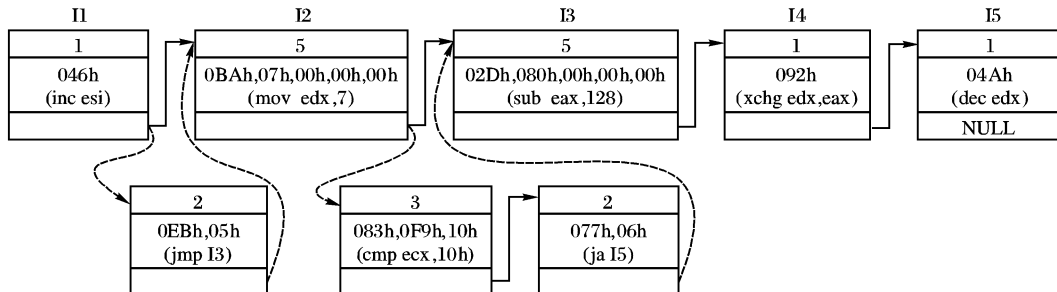


图 3 基于指令链的转移指令插入示意图

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 I5 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 I3
BA07000000 mov eax,7
83F910      cmp ecx,10h
7706        ja I5

I3:
2D80000000 sub eax,128
92          xchg edx,eax

I5:
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。

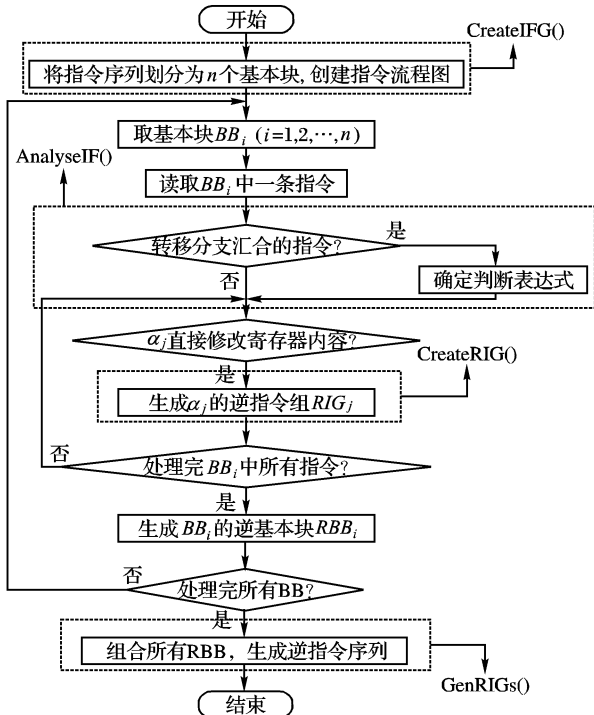


图4 生成逆指令序列算法流程

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  $\alpha = \text{Read\_Next\_Instruction}()$
- 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

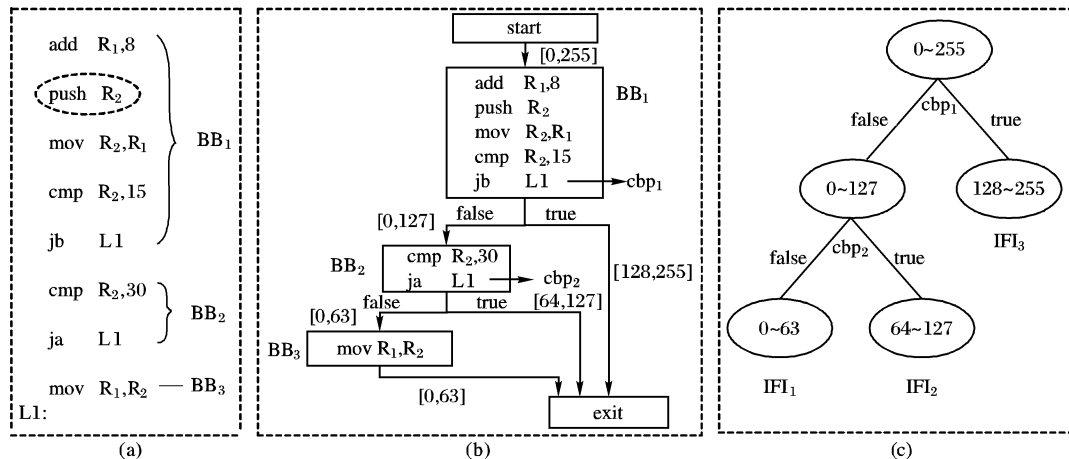


图5 生成 IFG 实例图

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  $\text{push } R_2$  是为保存  $R_2$  的原始状态而插入的指令。利用  $\text{CreateIFG}()$  依次将  $\text{start}$  块、三个基本块和  $\text{exit}$  块加入该指令序列的 IFG 中。为了产生 IFG 的指令流区间,以便  $\text{AnalyseIF}()$  对指令流序列进行分析,每加入一个基本块时,调用  $\text{MakeLabe}()$ ,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

$\text{AnalyseIF}()$  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  $\text{exit}$  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  $IF_1$ ,表示条件分支

$\text{cbp}_2(R_2 > 30)$  为假的情况; $[64,127]$  是  $\text{cbp}_2$  为真且  $\text{cbp}_1(R_2 < 15)$  为假时的指令流区间  $IF_2$ ;  $[128,255]$  是  $\text{cbp}_1$  为真的  $IF_3$ 。可见,在逆指令序列中  $\text{exit}$  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  $\text{cbp}_1$  和  $\text{cbp}_2$  的判断条件,在各基本块汇合到  $\text{exit}$  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  $\text{exit}$  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  $\text{cbp}_2$  和  $\text{cbp}_1$ 。

如图 6 所示, $R_2$  作为  $\text{cbp}_1$ 、 $\text{cbp}_2$  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  $\text{dec } R_2$  和  $\text{sub } R_2, 3$  改变,通过两条输入边  $[0,63]$  和  $[64,127]$  到达  $\text{exit}$  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  $\text{exit}$  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。

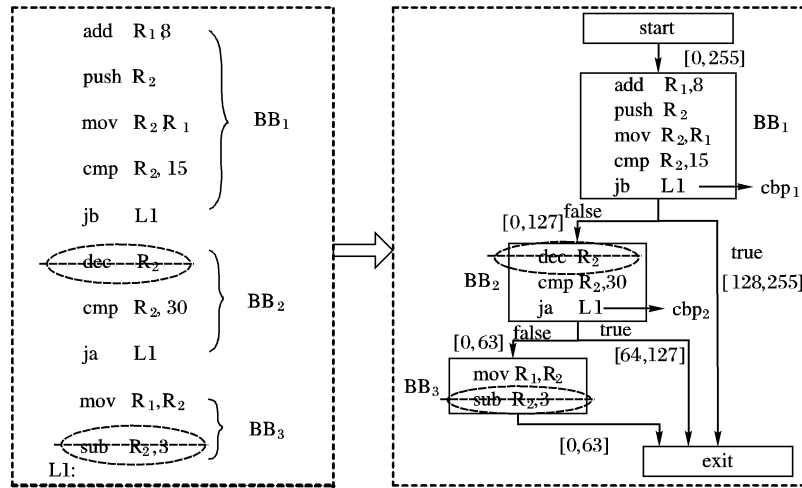


图 6 针对分支条件寄存器被改变的情况处理示意图

### 2.2.3 CreateRIG 函数

$\text{CreateRIG}()$  是 RIS 算法中最重要的函数,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。

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  $R_i^j$  ( $i, j = 0, 1, 2, \dots$ ,  $i$  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  $\alpha_j$  中被破坏的寄存器的一条或多条指令,得到  $\alpha_j$  的逆指令组  $RIG_j$ ,最后组合基本块  $BB_i$  ( $i = 2, 3, \dots$ ) 中所有指令的逆指令组,生成该基本块的逆基本块  $RBB_i$ 。

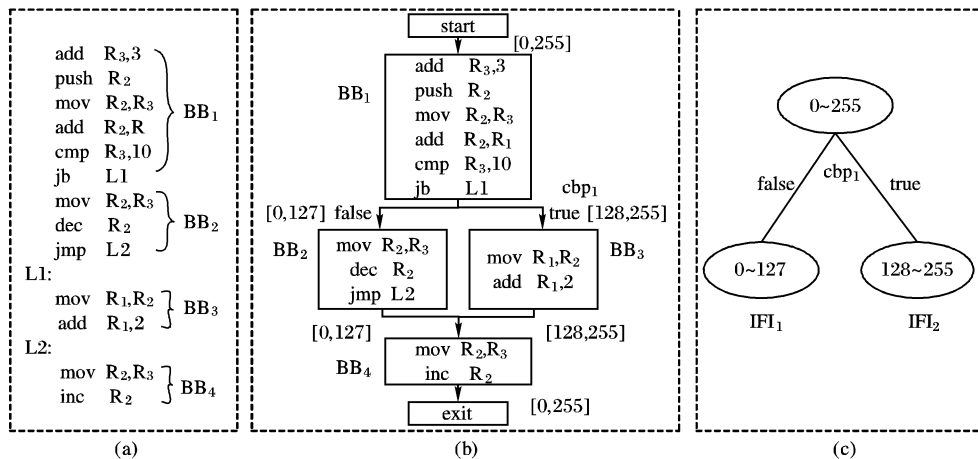


图 7 用于生成重命名表的 IFG 和 IFIs 示意图

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 <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>
IF <sub>1</sub>	R <sub>1</sub> <sup>0</sup>	R <sub>2</sub> <sup>0</sup> , R <sub>2</sub> <sup>1</sup> , R <sub>2</sub> <sup>2</sup> , R <sub>2</sub> <sup>3</sup>	R <sub>3</sub> <sup>0</sup> , R <sub>3</sub> <sup>1</sup>
IF <sub>2</sub>	R <sub>1</sub> <sup>0</sup> , R <sub>1</sub> <sup>1</sup>	R <sub>2</sub> <sup>0</sup> , R <sub>2</sub> <sup>1</sup> , R <sub>2</sub> <sup>2</sup>	R <sub>3</sub> <sup>0</sup> , R <sub>3</sub> <sup>1</sup>

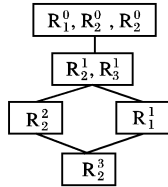


图8 寄存器变化示意图

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 $\alpha_j$ 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 $\alpha_j$ 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 $\alpha_j$ 的逆指令组。

2) 赋值引用法。通过查找 $\alpha_j$ 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 $\alpha_j$ 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<sub>4</sub>中的指令 $\alpha: R_2 = R_3 + 1$ 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 $\alpha$ 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 $\alpha$ 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<sub>4</sub>的逆基本块RBB<sub>4</sub>,如图9(b)所示。

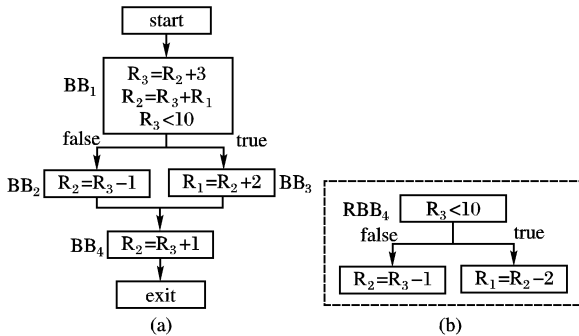


图9 重定义法生成逆指令示意图

#### 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<sub>i</sub>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<sub>4</sub>中的交汇点 $P_0$ 变成了RIFG中的分叉点Fr, BB<sub>1</sub>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

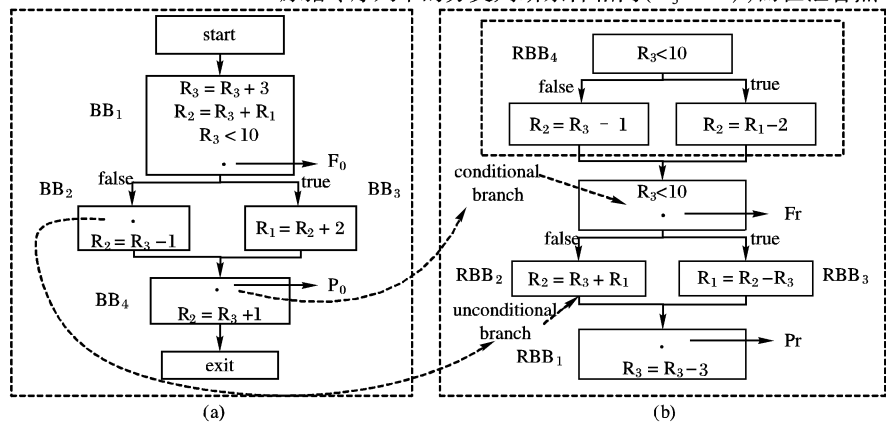


图10 指令序列的IFG与RIFG对照图

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。

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,如图(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。

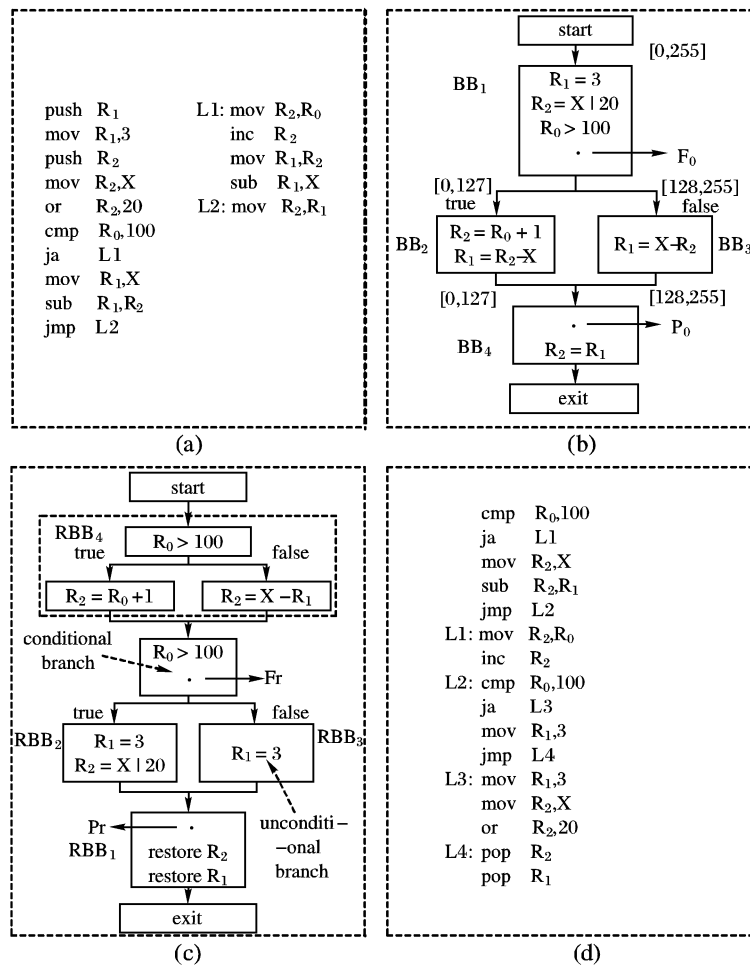


图 11 逆指令序列生成过程示例图

#### 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] GUPTA 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.