一种基于中断处理机制的动态反汇编算法

杨慕晗

(中国科学院研究生院信息安全国家重点实验室　北京 100049)

摘　要　传统的反汇编是静态进行的，它难以处理逻辑陷阱、花指令、代码中的数据、动态控制流等问题，这就导致其最终的反汇编结果可能不完整，存在错误。一段特意设计的代码，可以用动态控制流替代静态控制流；一段特意设计的花指令，可以让静态控制流关系任意复杂，这都将使静态反汇编分析难以进行。让代码运行起来，动态地进行反汇编，可以解决上述静态反汇编遇到的问题。结合计算机的中断处理机制，使得代码的实际执行路径是可知的。针对实际执行到的计算机指令进行反汇编，确保了反汇编结果的准确性；反复执行代码，在时间上进行积累，可以确保反汇编结果的完整性不断增加并趋于完整。

关键词　反汇编，逆向工程，信息安全

Algorithm for Dynamic Disassembly Based on Interrupt Scheme

YANG Mu-han

(State Key Laboratory of Information Security, SKLOIS, Beijing 100049, China)

Abstract　Disassembly is done statically, usually. It is hard to deal with logic trap, deceiving-instruction, data-in-code, dynamic control-flow-instruction, etc. by static disassembly. On the other hand, the code can be specially designed using dynamic control-flow-instruction instead, or with logic trap and deceiving-instruction, which both can make the static disassembly and analyzing impossible. Working with the interrupt scheme, disassembly can be done dynamically. The instructions executed exactly can be traced and disassembled. Following this method, all instructions disassembled are correct. Meanwhile, code can be run repeatly, and more code, finally almost all, will be processed.

Keywords　Disassembly, Reverse engineering, Information security

1　引言

随着时间的积累，计算机的功能越来越丰富，应用的领域越来越广，代码量也越来越大。同时，不可避免的，如此海量的计算机代码必然包含着大量的代码瑕疵。这些代码瑕疵，可能会因意外而发作，也可能被特意安排并在关键时刻被利用，从而导致不可预料的、巨大的损失。

当前，存在多种高层计算机语言：C/C++，Fortran，C#，Java 等等。因此，从高层源代码的视角，代码瑕疵的表现五花八门，难以下手去检测。

有句俗话：以不变应万变。

对于计算机代码而言，无论是用哪种高层计算机语言编写的，其最终都必将转换为 CPU 能执行的机器指令；而对于某一特定的计算机操作系统而言，所有能执行的代码，都必须首先封装成几种非常有限的文件格式，之后才能被该操作系统调度执行。

例如，运行 Intel 32 位架构上的 Windows 系统，其计算机代码属于 IA-32 指令集，其可执行文件格式限于 PE 文件格式；运行于 ARM 处理器上的 Linux 系统，其计算机代码属于 ARM 指令集，其可执行文件格式则限于 ELF 文件格式。

从这些不变的元素入手，以反汇编为手段，我们可以了解计算机代码的行为特点，分析之，进而处理一些潜在的代码瑕疵。下文提供的正是这样的一种动态反汇编算法。

杨慕晗　硕士研究生，主要研究方向为计算机系统安全、逆向工程。

2　静态反汇编

通常，反汇编可以静态地进行[6-9]。常用的静态反汇编算法有：线性遍历算法、递归遍历算法。

由于代码语义的知识只有在代码实际运行中才有可能展现，静态反汇编不可能得到这类信息，因此，一个简单的逻辑陷阱和花指令就能让它失败。

静态反汇编是基于对控制流的分析的。控制转移指令有 2 大类：静态控制转移指令(3. 3. i，3. 3. ii，3. 3. iv)，动态控制转移指令(3. 3. iii，3. 3. v，3. 3. vi)。由于静态反汇编的固有缺陷：静态性，它只能处理静态控制转移指令，不可能处理动态控制转移指令。这从根本上限制了静态反汇编的成功机率，无法逾越。

图 1、图 2 是一个简单的逻辑陷阱和花指令的示意图。

```
_start:
        pushad              ;60
                            ;逻辑陷阱
        xor eax, eax        ;31 c0
        cmp eax, 0          ;83 f8 00
        je _target          ;74 01
        ; db B8             ;花指令:
                            ;逻辑陷阱的后果
_target:
        mov eax, 0x11223344
                            ;b8 44 33 22 11
        popad               ;61
```

图 1　逻辑陷阱与花指令——汇编指令

```
60 31 C0 83 F8 00 74 01
B8 B8 44 33 22 11 61 C3
```

图 2 逻辑陷阱与花指令——机器指令

图 1 中的代码在实际执行中可以正常通过。但是,如果对图 2 所示的机器指令做静态的反汇编,则会认为"B8"处是一条指令的开始,静态反汇编受到欺骗。

图 1、图 2 所展示的仅仅是一种极端简单的情况,且给静态反汇编留下了指令边界冲突(见 3.10.5 节)的提示信息,使其在运气不错的情况下,有可能通过其他静态信息(如重定位信息[5]等)来确认出花指令。

但不幸的是,一个存心制造的逻辑陷阱和花指令是不会留下这种提示信息的。更进一步,它可以为花指令任意构造相关的重定位信息,并且,可以在花指令中构造任意复杂的控制流关系图,使得静态反汇编对控制流的分析在计算上陷于不可行。

所有上述问题,都难以在静态反汇编下解决。

3 动态反汇编

3.1 概述

动态反汇编的基本思路就是要在代码的运行过程中对其进行反汇编。让代码跑起来,让代码的实际执行控制流来引导反汇编,确保了反汇编的准确性、正确性。

从信息的角度来看,容易理解,动态情况下的代码会反应出一定的语义。而随着代码的不断运行,在时间上进行积累,其展现出来的语义信息量是时间 t 的不减函数;极限情况下,动态运行中代码会展现出其全部的语义。

从计算机技术理论的角度来看,本文提供的动态反汇编算法建立在"中断处理机制"上,而中断机制是当代计算机技术的核心机制。因此,本文所展示的动态反汇编方法具有普遍意义,可以移植应用到更广泛的计算机平台上。

3.2 实验环境

反汇编必然是针对某一具体的指令集的,也有其相应的具体操作系统环境。

本文针对的是 IA-32 指令集。它使用于 Intel 公司的 32 位 CPU,以及 AMD 的 32 位处理器上。

操作系统是 Windows 视窗 32 位操作系统上进行。该系统当前使用的可执行文件格式是 PE 文件格式。

实验中使用了自行开发的 IA-32 指令集反汇编单元。

3.3 假设

本文所讨论的被分析代码基于以下假设:

i. 代码不存在自变换

从完成基地址重定位、第一次将控制传给代码入口点开始,直到代码运行结束的整个过程中,代码不会发生任何形式的变化,比如,自解压、自解密等。

代码在运行过程中,重新申请一块内存、在其中写入代码并将控制传递到这块新申请的内存空间的行为,是允许的。但这块新申请内存中的代码一旦开始执行后,是不允许发生自变换的。

对于需要进行自变换的代码,需事先展开。

ii. 代码中可以有逻辑陷阱和花指令

代码中存在的逻辑陷阱和花指令,不会影响到动态反汇编过程,可以存在。

iii. 代码可以正确地执行

3.4 控制转移指令

通常情况下,CPU 是按照内存地址顺序的取指令并执行之的。当遇到控制转移指令时,CPU 根据实时的情况,其执行顺序可能发生跳转,而非顺序的执行下一条指令。

IA-32 指令集中的控制转移指令分类如下,这是一个完备的控制转移指令集:

i. 2 分支,非返回式,静态地址

包括(括号内为指令操作码,16 进制表示):Jcc(70-7f),Jcc(0f80-0f8f),jcxz(e3),Loopcc(e0-e1)。

ii. 2 分支,可返回式,静态地址

包括:call near(e8),int3(cc),int(cd),into(ce),call far(9a)。

iii. 2 分支,可返回式,动态地址

包括:calln(ff/010),callf(ff/011)。

iv. 1 分支,静态地址

包括:jmp short(eb),jmp near(e9),loop(e2),jmp far(ea)。

v. 1 分支,动态地址

包括:jmpn(ff/100),jmpf(ff/101)。

vi. 1 分支,Ret 终点,动态地址

包括:ret(c2),ret(c3),retf(ca),retf(cb),iret(cf)。

3.5 控制转移点

控制转移指令所在位置,就是控制转移点。

从代码流的观点看,控制转移点可能导致代码的分支增加(3.4.i,3.4.ii,3.4.iii)、跳转(3.4.iv,3.4.v)和终结(3.4.vi)。

3.6 控制入口点

代码执行顺序发生跳转(非顺序递增)后,执行的第一条指令所在地址,是控制入口点。

可执行(EXE 文件)模块的入口点、输出代码入口点,动态链接(DLL 文件)模块的输出代码入口点,是控制入口点;控制转移点的目标指令地址,也是控制入口点。

3.7 代码块

从一个控制入口点开始,顺序执行代码,直到第一个控制转移点或者下一个代码入口点为止,所有执行到的所有指令,构成这个代码块。

特别指出的是:控制入口点的指令是代码块的第一条指令;以控制转移指令作为结束的代码块,控制转移指令包括在这个代码块中;以下一个代码入口点作为结束的代码块,下一个代码入口点的指令的顺序上一条指令是这个代码块的最后一条指令。

```
_start2:
        pushad
        cmp esi, edi
        jne _target3
_target2:
        xor eax, ebx
_target3:
        add eax, 4
        popad
        ret
```

图 3 代码块

如图 3 所示,*_start2*、*_target2*、*_target3* 是 3 个代码块的入口点。这 3 个代码分别包括 3 条、1 条、3 条指令。

一个代码块由(代码块的入口点地址,代码块大小)唯一表示。

代码块在执行上是原子性的。就是说，一旦控制传到了代码块的入口点，那么，整个代码块必定被完整的执行一次。诚然，中断机制可以打断这个执行过程，但这并不影响我们在代码执行的概念上来建立并理解这个原子特性，因为中断机制对于代码块来说是透明的，也不会影响到我们的分析。

3.8 标记点

用 int3 指令替换一条指令的第一个字节，当代码执行到此处后，将会进入到终端处理程式中。在中断处理程式中，我们可以获得被中断代码的所有执行环境：中断返回地址、栈环境、寄存器环境、等等。

这样一条被 int3 指令替换过的指令，称为标记点。

进行 int3 替换这一操作过程，称为标记点的建立(或建立标记点)。

在建立标记点的时候，原指令的第一个字节的原始内容将被保存。在中断处理程式返回的时候，或者其它必要的时候，原指令的第一个字节将被还原，以使被分析代码可以继续正确的往下执行。这一还原过程称为标记点的删除(或删除标记点)。

3.9 标记方法

不同的控制转移类型，需要不同的标记方法，才能更好地观察被分析代码。下面分别说明各类控制转移点的标记方法。

3.9.1 “2 分支，非返回式，静态地址”的标记方法

这类控制转移点的标记点分别建立在 2 个分支的控制入口点。以控制的传入来确认其某个分支的内容的确是代码。

如图 4 所示。*_ChkPnt*41 和 *_ChkPnt*42 分别是“*je _ChkPnt*42”的 2 个分支的控制入口点。在这 2 处建立标记点：用 0xCC 替换 *_ChkPnt*41 处的 0xB8，用 0xCC 替换 *_ChkPnt*42 处的 0x61。(int3 指令的机器指令是 0xCC。)

```
_start4:
        pushad          ; 60
        cmp esi, edi    ; 39 FE
        je _ChkPnt42    ; 74 05
_ChkPnt41:
        mov eax, 1      ; B8 01 00 00 00
_ChkPnt42:
        popad           ; 61
        ret             ; C3
```

图 4 “2 分支，非返回式，静态地址”的标记方法——汇编指令

图 5 是标记点建立前机器指令的状态；图 6 是标记点建立后机器指令的状态。

```
60 39 FE 74 05 B8 01 00
00 00 61 C3
```

图 5 “2 分支，非返回式，静态地址”的标记方法——机器指令(建立标记点之前)

```
60 39 FE 74 05 CC 01 00
00 00 CC C3
```

图 6 “2 分支，非返回式，静态地址”的标记方法——机器指令(建立标记点之后)

3.9.2 “2 分支，可返回式，静态地址”的标记方法

```
_start5:
        pushad          ; 60
        push esi        ; 56
        push edi        ; 57
_ChkPnt51:
        call _start4    ; E8 EC FF FF FF
_ChkPnt52:
        add esp, 8      ; 83 C4 08
        popad           ; 61
        ret             ; C3
```

图 7 “2 分支，可返回式，静态地址”的标记方法——汇编指令

图 17 是标记点建立前机器指令的状态；图 18 是标记点建立后机器指令的状态。

这类控制转移点的标记点分别建立在控制转移后的控制入口点、调用返回点。具体地说，就是在调用指令的目标地址和调用指令的下一条指令上建立标记点。以控制的传入来确认某个分支的内容的确是代码。

图 8 是标记点建立前机器指令的状态；图 9 是标记点建立后机器指令的状态。

```
60 56 57 E8 EC FF FF FF
83 C4 08 61 C3
```

图 8 “2 分支，可返回式，静态地址”的标记方法——机器指令(建立标记点之前)

```
60 56 57 E8 EC FF FF FF
CC C4 08 61 C3
```

图 9 “2 分支，可返回式，静态地址”的标记方法——机器指令(建立标记点之后)

3.9.3 “2 分支，可返回式，动态地址”的标记方法

这类控制转移点的标记点分别建立在控制转移点、控制转移返回点。具体地说，就是在调用指令和调用指令的下一条指令上建立标记点。

建立在控制转移点 *_ChkPnt*61 处的标记点一旦触发，将在其实时的控制转移目标地址处建立标记点，并删除 *_ChkPnt*61 处的标记点。紧接着，控制转移目标地址处的标记点被触发，它将在 *_ChkPnt*61 处再次建立标记点，以便下一次抵达 *_ChkPnt*61 处的执行。

另一种方案是，*_ChkPnt*61 处的标记点触发后，由中断处理程序来完成模拟控制转移，而不删除这个标记点。

控制转移目标地址处的标记点是静态的，它以控制的传入来确认其内容的确是代码。

控制转移返回点的标记点是静态的，它以控制的传入来确认其内容的确是代码。

如图 10 所示。*_ChkPnt*61 是调用指令，*_ChkPnt*62 是调用指令的下一条指令。在这 2 处建立标记点：用 0xCC 替换 *_ChkPnt*61处的 0xFF，用 0xCC 替换 *_ChkPnt*62 处的 0x83。(int3 指令的机器指令是 0xCC。)

```
_start6:
        pushad              ; 60
        mov eax, _start4    ; B8 1B 10 40 00
        push esi            ; 56
        push edi            ; 57
_ChkPnt61:
        call eax            ; FF D0
_ChkPnt62:
        add esp, 8          ; 83 C4 08
        popad               ; 61
        ret                 ; C3
```

图 10 “2 分支，可返回式，动态地址”的标记方法——汇编指令

图 11 是标记点建立前机器指令的状态；图 12 是标记点建立后机器指令的状态。

```
60 B8 1B 10 40 00 56 57
FF D0 83 C4 08 61 C3
```

图 11 “2 分支，可返回式，动态地址”的标记方法——机器指令(建立标记点之前)

```
60 B8 1B 10 40 00 56 57
CC D0 CC C4 08 61 C3
```

图 12 “2 分支，可返回式，动态地址”的标记方法——机器指令(建立标记点之后)

3.9.4 “1 分支，静态地址”的标记方法

这类控制转移点的标记点建立在控制转移后的控制入口点上。也就是控制转移指令的目标地址的第一条指令处。

```
_start7:
        pushad          ; 60
_ChkPnt71:
        jmp _ChkPnt62; EB E4
        nop             ; 90
_ChkPnt72:
        popad           ; 61
        ret             ; C3
```

图 13 “1 分支,静态地址”的标记方法——汇编指令

```
60 EB 01 90 61 C3
```

图 14 “1 分支,静态地址”的标记方法——机器指令(建立标记点之前)

```
60 EB 01 90 CC C3
```

图 15 “1 分支,静态地址”的标记方法——机器指令(建立标记点之后)

如图 10 所示。*_ChkPnt71* 是 1 分支控制转移指令,*_ChkPnt72* 是其目标地址的第一条指令。在*_ChkPnt72* 处建立标记点:用 0xCC 替换*_ChkPnt72* 处的 0x61。(int3 指令的机器指令是 0xCC。)

图 11 是标记点建立前机器指令的状态;图 12 是标记点建立后机器指令的状态。

特别地,由于这类控制转移点只有 1 个分支,当这类控制转移指令一旦执行,其必然会执行到其目标地址——控制入口点,所以,这类标记是虚标记。在实际操作中,可以不必实际的建立标记点,而仅仅需要记录其目标的控制入口点代表了一个代码块的入口。

3.9.5 “1 分支,动态地址”的标记方法

这类控制转移点的标记点建立在控制转移点上。即,控制指令上。

建立在控制转移点*_ChkPnt81* 处的标记点一旦触发,将在其实时的控制转移目标地址处建立标记点,并删除*_ChkPnt81* 处的标记点。紧接着,控制转移目标地址处的标记点被触发,它将在*_ChkPnt81* 处再次建立标记点,以便下一次抵达*_ChkPnt81* 处的执行。

另一种方案是,*_ChkPnt81* 处的标记点触发后,由中断处理程序来完成模拟控制转移,而不删除这个标记点。

控制转移目标地址处的标记点是静态的,它以控制的传入来确认其内容的确是代码。

如图 16 所示。_ChkPnt81 是调用指令。在_ChkPnt81 处建立标记点:用 *0xCC* 替换_ChkPnt81 处的 *0xFF*。(*int3* 指令的机器指令是 *0xCC*。)

```
_start8:
        pushad              ; 60
        mov eax, _ChkPnt82
                            ; B8 55 10 40 00
_ChkPnt81:
        jmp eax             ; FF E0
        nop                 ; 90
_ChkPnt82:
        popad               ; 61
        ret                 ; C3
```

图 16 “1 分支,动态地址”的标记方法——汇编指令

图 17是标记点建立前机器指令的状态;图18是标记点建立后机器指令的状态。

```
60 B8 55 10 40 00 FF E0
90 61 C3
```

图 17 “1 分支,动态地址”的标记方法——机器指令(建立标记点之前)

```
60 B8 55 10 40 00 CC E0
90 61 C3
```

图 18 “1 分支,动态地址”的标记方法——机器指令(建立标记点之后)

3.9.6 “1 分支,Ret 终点,动态地址”的标记方法

这类控制转移点的标记点建立在控制上。

如图 19 所示。*_ChkPnt91* 是控制转移指令。在*_ChkPnt91* 处建立标记点:用 0xCC 替换*_ChkPnt91* 处的 0xC3。(int3 指令的机器指令是 0xCC。)

```
_start9:
        pushad ; 60
        popad  ; 61
_ChkPnt91:
        ret    ; C3
```

图 19 “1 分支,Ret 终点,动态地址”的标记方法——汇编指令

图 20 是标记点建立前机器指令的状态;图 21 是标记点建立后机器指令的状态。

```
60 61 C3
```

图 20 “1 分支,Ret 终点,动态地址”的标记方法——机器指令(建立标记点之前)

```
60 61 CC
```

图 21 “1 分支,Ret 终点,动态地址”的标记方法——机器指令(建立标记点之后)

3.10 动态反汇编算法

3.10.1 数据结构框图

图 22 是动态反汇编过程中各数据结构的关系框图。

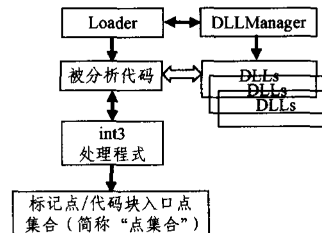

图 22 动态反汇编算法是数据结构框图

“Loader”在“DLLManager”的帮助下,负责将“被分析代码”及其相应的 DLL 模块加载到内存,并进行相应的基地址重定位等处理。

“DLLManager”负责内存中 DLL 模块的加载、卸载等维护工作,协助 Loader 的工作。

“int3 处理程式”在“被分析代码”的运行过程中,与其进行动态的交互。负责标记点的建立、删除,“标记点/代码块入口点集合”的维护。

“标记点/代码块入口点集合”是动态反汇编“被分析代码”的结果。每一个“被分析代码”与一个“点集合”一一对应。当“点集合”在“被分析代码”上的覆盖率达到设定值时,反汇编算法收敛、并结束。

3.10.2 步骤

i.“Loader”加载“被分析代码”及其相应的 DLL 模块;

ii. 初始化设置:“被分析代码”入口点是第一个标记点,加入“集合”;收敛条件设定;

iii. 传递控制到“被分析代码”;

iv.“被分析代码”在与“int3 处理程式”交互中执行,“int3 处理程式”维护“点集合”;计算收敛条件,不满足,继续第 iv 步,若满足,跳到第 v 步;

v. 根据“点集合”确认花指令、未知指令/数据。

vi. 完成。

3.10.3 “标记点/代码块入口点集合”的最终状态

算法结束后，在“点集合”中有多个代码块入口点。“标记点/代码块入口点”数据结构摘要如图 23 所示。

```
struct stMarkPoint
{
        unsigned __int32 Offset;
        unsigned __int32 Size;
        BOOL   Excuted; // 是否执行过

        // ......

};
```

图 23 “标记点/代码块入口点”数据结构

按 Excuted 分类：

Excuted == TRUE，表示在算法运行过程中，此标记点被实际执行过，可以确认是代码。于是，标记点在意义上转换为代码块入口点。Size 的值在此代码块第一次执行时被计算，或者在有新的标记点加入点集合时被更新。

Excuted == FLASE，表示在算法运行过程中，此标记点从未被执行过。这可能是由于代码语义逻辑关系，在运行过程中没有覆盖到与这段代码相关的功能，也可能就是由逻辑陷阱产生的花指令。

3.10.4 收敛条件

作为结束算法第 iv 步的收敛条件，计算如下 2 个参数：

i. 反汇编率 =（*Excuted* == *TRUE* 的区域总合）/（“被分析代码”所有代码区域总合） * 100%

反汇编率反映了当前已经识别出来的代码占需要识别的代码的总量。这是一个阈值，在反汇编开始时设定，也可以在反汇编过程中实时的修改。

当反汇编率达到或超过预先设定的期望时，我们认为动态反汇编收敛。

ii. 最近连续建立重复标记点的次数

每次处理完一次标记点触发后，静态的标记点将被从“被分析代码”中删除，在“点集合”中的相应标记点上做一个标记。当再次试图向“点集合”中建立此标记点时，建立不会实际上发生。这样的行为称为“建立重复标记点”。

实际上，在整个步骤 iv 中，所有的静态标记点只会被分析一次。而所有的“建立重复标记点”行为，只发生在动态地址控制转移处。

因此，当最近发生了连续的建立重复标记点行为，且次数达到一定量时（也就是说，很长时间没有新的标记点需要建立了），我们认为，“被分析代码”在当前运行环境下，已经不会执行到更多的代码了，动态反汇编可以结束了。

3.10.5 花指令的确认

Excuted == FLASE 的标记点说明：这是一个可能的控制传入点，但在已经进行的运行中，却还没有执行过这些代码。

如图 24、25、26 所示。根据 3.10.2 节的算法，_*ChkPoint*101 和_*ChkPoint*102 在“点集合”中分别有一个标记点，其 Excuted 值分别为 FALSE，TRUE。

对_*ChkPoint*101 的标记点进行静态反汇编，得到图 26 的结果。这导致了机器指令错位重叠：_*ChkPoint*101 的标记点认为 0x11 处是一个机器指令边界，这与_*ChkPoint*102 标记点的分析结果相冲突。

```
_start10:
        pushad          ;60
                        ;逻辑陷阱
        xor eax, eax    ;31 c0
        cmp eax, 0      ;83 f8 00
        je _target      ;74 01
_ChkPoint101:
        ; db B8         ;花指令:
                        ;逻辑陷阱的后果
_ChkPoint102:
        mov eax, 0x11223344
                        ;b8 44 33 22 11
        popad           ;61
        ret             ;c3
```

图 24 花指令的确认——汇编指令

```
60 31 C0 83 F8 00 74 01
B8 B8 44 33 22 11 61 C3
```

图 25 花指令的确认——机器指令边界冲突

```
_ChkPoint101:
        mov eax, 0x223344b8
                        ; B8 B8 44 33 22
                        ; 花指令
```

图 26 花指令的确认——反汇编出来的花指令

由于_*ChkPoint*102 标记点的 *Excuted* == *TRUE*，我们有充分的理由相信_*ChkPoint*102 标记点的信息是可靠的。因此，我们认定，_*ChkPoint*101 标记点是一处花指令。

3.10.6 未知指令/数据的标记

Excuted == FLASE 的标记点中，有一部分不会与已知代码块发生机器指令边界冲突。这有 2 个原因：一方面，这些标记点可能是由于代码语义上的关系，暂时还没有被执行到的；另一方面，可能的确是花指令，只是指令边界恰好与已确认的代码块发生了重合。

对于这些 Excuted == FLASE 且未被确认为花指令的未知代码数据，我们进行预防性标记。

对于能确认一定是代码的标记点，将未知代码全部替换为 0xCC(int3 指令)；对于其中可能含有数据的标记点区域，将其控制入口点替换为 0xCC(int3 指令)。

在以后的运行中，“被分析代码”一旦触发到这些标记区域，将会立即进入 int3 中断处理程式。int3 中断处理程式将根据实时的策略配置，或替换回原数据，让“被分析代码”继续执行；或抛出警告，交由更高层次来处理。

3.10.7 空间复杂度

记“被分析代码”所使用内存空间为 N，“Loader”使用的内存空间为 N_l，“DLLManager”使用的内存空间为 N_{dm}，加载到内存中的所有 DLL 模块所使用的内存空间为 N_{dll}，“int3 中断处理程式”所使用的内存空间为 N_{int3}。

最坏情况下，“被分析代码”所有内存空间全部是单字节的控制转移指令，此时，“点集合”中包含有 N 个标记点数据，需要的空间为 cN，c 为每个标记点数据结构占的空间。

因此，最坏情况下需要的内存空间总量为（$N_l + N_{dm} + N_{int3} + N_{dll} + N + cN$）。其中，$N_l$，$N_{dm}$，$N_{int3}$ 为常量；考虑到系统中 DLL 模块的总量也是一定的，故 N_{dll} 也为常量；N 根据每次任务的不同而不同，是变量。

综上可知，3.10.2 节所描述算法的空间复杂度为 O(N)。

3.10.8 时间复杂度

根据 3.10.2 节的算法，算法时间主要消耗在第 iv 步上。而这一步骤的耗时与收敛条件的设定相关。

记最初设定的“最近连续建立重复标记点的次数”阈值为 C_r。

（下转第 294 页）

一种新的组密钥管理算法…………………………………（116）
一种基于移动 Agent 的网络性能管理系统及性能分析
…………………………………………………………（122）
基于小波自适应算法的传感器信号降噪研究…………（126）
计算本体映射纯语义查准率和查全率的框架…………（128）
基于二维 QoS 模型的 Web 服务组合 …………………（131）
基于概念格标尺的广义匹配研究………………………（135）
前馈过程神经网络的网络结构与泛化能力……………（137）
一种介词-动词模式的获取方法 ………………………（139）
文本分类技术在海洋信息处理领域中的应用…………（144）
基于子集类蚁群模型的属性相对约简算法……………（147）
直觉模糊逻辑算子研究…………………………………（151）
直觉模糊三角模的剩余蕴涵及其性质…………………（154）
赋范格 H 蕴涵代数和模糊格 H 蕴涵代数 ……………（156）
一种基于网格距离的融合式聚类算法…………………（160）
一种改进的 BMH 模式匹配算法 ………………………（164）
单体型组装问题计算模型的比较与分析………………（166）
基于类权重的模糊不平衡数据分类方法………………（170）
数据驱动的可变精度粗糙集噪音阈值获取方法………（174）
RCGNN：一种基于实数编码的遗传神经网络预测方法
…………………………………………………………（178）
基于邻域模型的 K-means 初始聚类中心选择算法 …（181）
一种基于最大频繁项目集的挖掘事务间关联规则方法
…………………………………………………………（185）
一种基于 Petri 网原理的数据流模型研究 ……………（189）
移动数据库中一种基于 XML 的视图增量更新算法
…………………………………………………………（192）
XML 强闭包依赖的研究 ………………………………（195）
双目标无等待流水线调度的加权混合算法……………（199）
一种基于人眼视觉特性和小波变换域的图像数字水印技术
…………………………………………………………（203）
一种新型非线性视频图像交叉加密算法………………（207）
变精度覆盖粗糙集模型的推广研究……………………（210）
基于 HVS 的自适应鲁棒视频水印算法 ………………（214）
非参量变换在彩色图像立体匹配中的应用研究………（217）
一种新的基于 TNAM 的二值图像表示方法 …………（220）
真实感植物绒毛建模和实时绘制………………………（225）
基于链表的同构化首尾排序新算法……………………（229）
工作流系统中基于场所的分布式授权模型研究………（232）
一种嵌入式软件项目估计方法…………………………（236）
软件静态结构的依赖网络建模方法与特性分析………（239）
基于 Qt 和 Open Inventor 跨平台虚拟油泥造型系统构建方法的研究与实现…………………………………（244）
两种基于分离/匹配机制的普适计算编程模型比较与研究
…………………………………………………………（248）
对象互操作的层次模型…………………………………（251）
基于 Petri 网的无死锁控制器设计 ……………………（255）
基于 RDF4S 语义服务描述模型的服务资源搜索框架
…………………………………………………………（258）
有色 Petri 网协作模型的 BPEL 代码实现 ……………（262）
一种基于 Hurst 指数的异常检测软件…………………（267）
网格环境下基于上下文和角色的访问控制……………（270）
一种新的语义对象行为语言……………………………（272）
一种基于概念格的软件过程改进算法…………………（276）
基于可靠性增长模型的软件可靠性增长测试充分性准则
…………………………………………………………（281）
类动态更新事务研究……………………………………（284）
HUNTBot—第一人称射击游戏中 NPC 的结构设计
…………………………………………………………（290）
基于本体的智能学习资源分配模型构建………………（293）

第 12 期（略）

（上接第 284 页）

最坏情况下，“被分析代码”全部是动态控制转移指令，则，标记点上限为 N。

每触发一次动态的控制转移处理，必定会有一个当前的控制转移目标点，并向“点集合”中进行一次建立标记点的操作。一次建立操作要么成功的建立了一个标记点，要么是一次“建立重复标记点”行为；2 次“建立非重复标记点”行为最多间隔（C_r-1）次“建立重复标记点”行为。

因此，最多（N（C_r-1）+ 1）次触发 int3 中断处理程式后，满足收敛条件，算法结束。

综上可知，3.10.2 节所描述算法的时间复杂度为 O(N)。

结束语 动态反汇编是了解、认识、分析代码行为、系统行为的一个重要角度。它不需要源代码的支持。在当前的计算机技术理论框架下（中断处理机制），它对所有能够实际执行的代码都有效。

本文展示的动态反汇编算法实际上也是一种了解、掌握、控制代码的行为核心手段。从安全的角度来看，将这一算法应用到计算机安全领域是大有可为的。

参考文献

[1] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual Volume 1：Basic Architecture. http://www.intel.com
[2] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual Volume 2：Instruction Set Reference. http://www.intel.com
[3] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual Volume 3：System Programming Guide. http://www.intel.com
[4] 李青山，等. 逆向工程中的反射植入的研究. 计算机学报，2004，27(4)：535
[5] 曾鸣，等. 一种基于重定位信息的二次反汇编算法. 计算机科学，2007，34(7)：284
[6] 蒋烈辉，等. 反汇编结果代码结构分析算法研究. 小型微型计算机系统，2007，28(6)：1060
[7] 许敏，等. 静态反汇编算法研究. 计算机与数字工程，2007，35(5)：13
[8] 蒋烈辉，等. 基于控制流和数据段分析的反汇编策略研究. 计算机工程，2007，33(2)：94
[9] 吴金波，等. 基于控制流的静态反汇编算法研究. 计算机工程与应用，2005(30)：89
[10] 蒋烈辉，等. 支持通用反汇编的处理器结构库设计与实现. 计算机工程与设计，2006，27(3)：500
[11] 曾鸣，等. 一种基于反汇编技术的二进制补丁分析方法. 计算机科学，2006，33(10)：283