第 35 卷 第 2 期 计 算 机 工 程 2009 年 1 月
Vol.35 No.2 Computer Engineering January 2009
·软件技术与数据库· 文章编号：1000—3428(2009)02—0015—03 文献标识码：A 中图分类号：TP301.6

可执行文件中子程序异常返回的识别

张一弛，庞建民，赵荣彩，韩小素
(解放军信息工程大学信息工程学院，郑州 450002)

摘 要：针对子程序异常返回对反汇编操作的干扰，提出一种能够有效对抗该技术的反汇编算法。该算法通过 2 遍解码流程对目标可执行程序进行扫描，模拟代码执行过程中对内存栈的操作，从而正确解码出经过混淆处理的可执行程序。通过与 2 款常用反汇编器 IDAPro 和 OBJDump 的反汇编结果进行比较，证明该算法能够有效地识别出子程序异常返回的情况，从而有效提高反汇编的正确率。

关键词：反汇编；代码混淆；恶意程序

Identification of Exception Return in Subroutine of Executable File

ZHANG Yi-chi, PANG Jian-min, ZHAO Rong-cai, HAN Xiao-su
(Institute of Information Engineering, PLA Information Engineering University, Zhengzhou 450002)

【Abstract】 Malware writers make use of exception return of subroutine to evade detecting by malware detectors. To crack the technique, this paper proposes a novel disassembly algorithm. This algorithm decodes an executable file twice and emulates the operations on memory stack. Through this twice-decoding and emulation process, this algorithm can be used to recognize exception returns and thus ensure the correctness of a decoding process. Compared with two commonly used disassemblers IDAPro and OBJDump, this algorithm is better at identifying this kind of exception and improves the rate of disassembly.

【Key words】 disassembly; code obfuscation; malware

人们将可执行程序按其行为分为良性(benign)和恶意(malicious)2 类。为了保证系统的安全，对可执行程序的检测显得十分重要。通常对可执行程序的分析首先是对其进行反汇编，但恶意代码的编写者通过混淆技术来躲避逆向分析。目前，IDAPro 和 OBJDump 是比较常用的 2 种反汇编器，但它们对子程序异常返回的情况都不能得到正确的汇编代码。因此，需要针对此种情况给出解决办法，从而提高反汇编的正确率。

1 子程序异常返回

子程序调用通常使用 CALL 指令，在执行 CALL 指令时会将当前指令(也就是本条 CALL 指令)的下一条指令的地址(Addr1)放入栈顶，以便在子程序完成时继续向下执行。

在子程序执行结束时，会用 RET 指令返回主程序，这时是将当前栈的栈顶取出(正常应该是 Addr1)，继续执行主程序的操作。

“混淆(obfuscation)”用来指保护程序语义和功能的方法，同时也会给分析代码、了解程序结构带来困难。在反汇编中，“混淆”使得对二进制文件中的指令分析更加困难[1]。

其实混淆就是一种转换技术，Collberg给出了混淆的定义：P->P’，表示一种由源程序P到目标程序P’的转换，使得P’在功能上与P等价，并且这 2 段程序在运行时具有相同的行为，而混淆后的程序较混淆前更加难以理解。根据混淆原理和对象不同，可以将代码混淆分为外形(layout)混淆、控制结构(control)混淆、数据(data)混淆和预防(preventive)混淆等 4 种[2-3]。

利用子程序异常返回来干扰反汇编器工作就属于控制结构混淆，这种方法修改了程序的控制结构，但不会改变程序实际执行的内容。

许多汇编器在调用指令后面紧跟的是一条有效指令。但是，假设若干字节的无用数据被插入在此处。在X86 指令集中CALL指令是由跳转和压栈动作组成，RET指令的返回地址是当前的栈顶。但是通过分支过程能够修改栈顶，子程序异常返回的混淆技术[4]就是利用这个特点。

通过子程序异常返回来干扰反汇编的一种典型情况如下：

```
        正常情况                混淆情况
sub:                    sub:
        …                       …
        …                       push exit (调整堆栈)
ret                     ret
main:                   main:
        …                       …
        call   sub              call   sub
        …                       db   0E8h (垃圾数据)
exit:                   exit:
        …                       …
```

在主程序中插入无用数据(E8)，然后在子程序返回之前调整堆栈，将真正的返回地址(exit)放入栈顶。

对于混淆情况，当程序执行到CALL指令调用子程序(sub)时，会自动将当前指令的下一条指令的地址(Addr1)放入堆栈中，在 sub 结束之前使用的 RET 指令会将当前的栈顶取出(正常情况应该是调用该过程的 CALL 指令自动压入堆栈的 Addr1)，并将取出的值(exit)放入 PC(程序计数器)中，从该地址开始执行，这样保证了程序能够跨过垃圾数据，使程序正

基金项目：国家“863”计划基金资助项目(2006AA 01Z408)
作者简介：张一弛(1983－)，男，硕士研究生，主研方向：逆向工程；庞建民、赵荣彩，教授、博士；韩小素，硕士研究生
收稿日期：2008-11-14 **E-mail：**each5595@sina.com

确执行。

通常情况，反汇编器认为CALL指令之后的内容也是有效指令，所以，会将执行不到的垃圾数据(E8)和其后面的指令进行组合识别为其他指令(call near ptr 1CA810A2h)。又因为X86 指令集是变长指令集，所以指令的开始位置可以从任意地址开始[3]。反汇编器是根据当前指令的长度来计算下一条指令的开始地址，如果当前指令识别出错，指令长度就可能不正确，导致下一条指令解码出错(xor [eax+0], al)，这样就出现解码连续出错的情况。IDAPro对例 1 的混淆情况反汇编结果如下：

```
.text:00401000      sub_401000   proc near
…
.text:00401015                   push      401034h
.text:0040101A                   retn
…
.text:0040101B      start   proc near
.text:0040102E              call      sub_401000
.text:00401033              call near ptr 1CA810A2h(反汇编出错)
.text:00401038              xor       [eax+0], al      (反汇编出错)
…
.text:0040105A      start     ends
```

2　抗混淆静态反汇编的算法设计

2.1　静态反汇编算法

静态反汇编是指在不执行或模拟执行的情况下，完成对二进制代码的反汇编工作。

静态反汇编主要有 2 种方法：线性扫描从二进制文件的代码段开始连续解码，利用这种方法的有 GNU 的 OBJDump；行进递归按照控制流图进行解码，它是递归解码，例如 IDAPro，具有一定的抗混淆能力，能够跳过一部分垃圾数据(如：jump 指令之后添加垃圾数据)，而线性解码无法做到。

这 2 种算法都不能对子程序异常返回的混淆情况进行处理，需要设计新算法解决这种情况。

2.2　抗混淆算法设计

针对压栈后返回的情况，如果利用线性扫描算法，不能对过程进行划分，从而无法跳过垃圾数据。选择行进递归算法为基础，能够对过程进行划分，在划分的同时能够记录过程的调用关系，以便后期使用。

首次对主程序解码时子程序还没解码，不能得到子程序实际返回地址，如果存在异常返回的情况，依然会将垃圾数据识别为指令，需要在全部子程序解码完成之后再次对主程序解码。

首次解码流程如图 1 所示，针对每个过程(Proc)，添加虚拟栈(T)记录对堆栈的操作。在过程开始解码之前，将调用该过程的 CALL 指令的下一条指令的地址(Addr1)压入 T；在解码过程中，在识别出堆栈操作指令时，同时在 T 上做相应的动作；在当前过程解码完成时，则以 Addr1 的值为索引填表 M，表示该调用的子程序已解码，并且记录实际的返回地址。比较 T 的栈顶是否与 Addr1 相同。若相同，实际返回地址就是 Addr1，在 M 中填“ 0 ”，表示子程序正常返回；否则，实际返回地址是 T 的栈顶值(Addr2)，将 Addr2 填入 M，将调用 Proc 过程的开始地址放入二次解码堆栈(DT)中。在调用的所有子程序解码完毕之后开始二次解码。二次解码流程如图 2 所示，从 DT 中取出地址开始解码，当发现当前指令是 CALL 指令时，用 Addr1 值查表M，若实际返回地址为“ 0 ”，继续连续解码；否则，下一条应该解码指令的开始地址就是实际返回地址(Addr2)。本过程解码完毕之后，从堆栈 DT 中取出新的过程开始地址解码，直到 T 为空，反汇编结束。

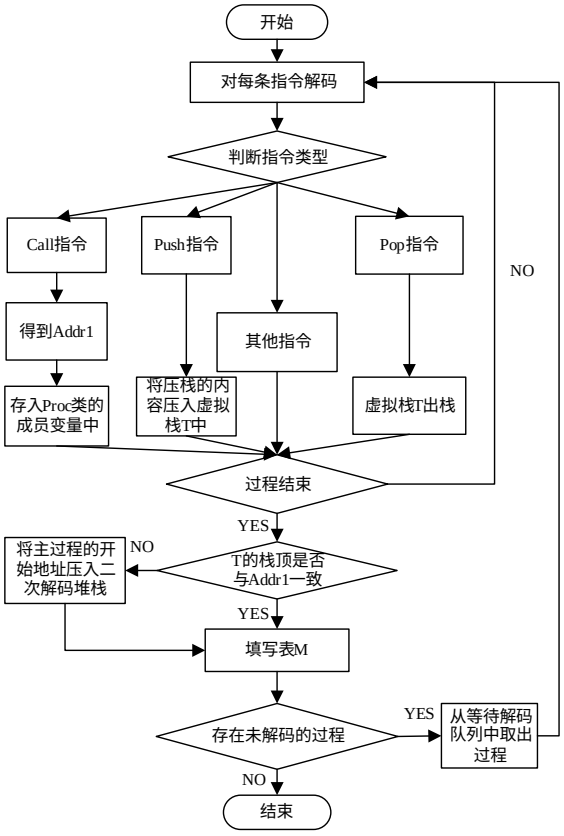

图 1　首次解码流程

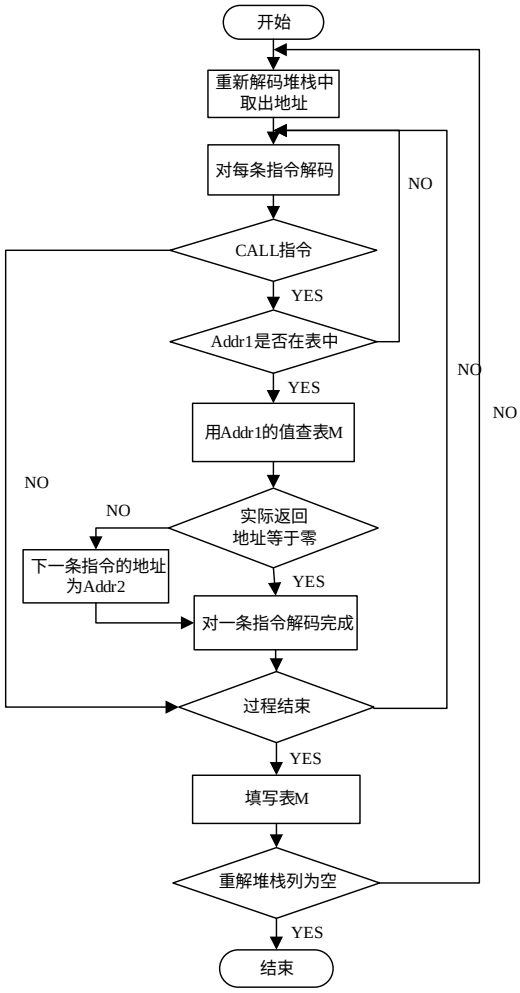

图 2　二次解码流程

2.3 特殊情况的处理

由于可能产生指令连续解码出错，不仅将垃圾数据识别为指令，而且造成垃圾数据之后的有效指令识别出错。又因为 X86 指令集具有自动恢复的特点，在对“几条”指令解码出错之后，就会开始正确的解码。绝大多数情况下，这“几条”解码出错的指令会在二次解码时得到修正，但这里有 2 种特殊情况需要进行处理：

情况 1 如下：

```
        情况 1:                IDAPro 结果:
main:                       .text:00401025 start:
        …                   …
        call sub1           .text:00401038     call sub_401000
        db 0E8h             .text:0040103D     call near ptr 3FE92Ah
exit:                       …
        call sub2
        db 0E8h             .text:00401042     dw 0E8FFh
        …                   …
end main                    .text:0040105E_text    ends
```

在首次解码时，如果 CALL 指令因为混淆没有被正确识别指令(call sub2)，则相应的子程序没有解码。该过程存在异常返回的情况，没有被发现。

对此种情况，在二次解码查M时，若M中没有对应的项，说明该过程没有进行解码，需要对其子程序重新解码来判断是否存在异常返回的情况。

处理方法：中断本过程的二次解码，先对子程序解码，待子程序解码完毕，填写 M，再重新对主程序进行解码，这时表 M 中就会有对应该过程的项，解码正确完成。

情况 2 如下：

```
情况 2:                        IDAPro 结果:
sub1:                       .text:00401000      sub_401000    proc near
        …                   …
        call sub2           .text:00401015      call sub_401021
        db    0E8h          .text:0040101A      call near ptr 40506387h
exit2:                      .text:0040101F      add    bl, al
        push exit           …
        ret
sub2:                       .text:00401021      sub_401021    proc near
        …                   …
        push exit2          .text:00401034      push      40101Bh
        ret                 .text:00401039      retn
main:                       .text:0040103A      start     proc near
        …                   …
        call sub1           .text:0040104D      call sub_401000
        db    0E8h          .text:00401052      call near    ptr 1DA810C1h
exit:                       .text:00401057      xor [eax+0], al
        …                   …
end main                    .text:0040106D      start    endp
```

由于在首次解码时，混淆没有正确识别过程结束指令(如 RET 指令)，因此过程之间会发生过程重叠的情况(sub2 与 sub1 重叠)。

处理方法：增加条件判断，当前过程为 sub1，被调用过程为 sub2，判断 sub2 是否在 sub1 中，若是就对 sub1 进行划分，将属于 sub2 的内容从 sub1 中删除，使过程的划分正确。

3 测试

为了定义混淆之后的反汇编效果，用 *cf*(confusion factor)来衡量指令反汇编错误情况：

$$cf = \frac{|v-o|}{v}$$

其中，*v*表示程序中有效指令；*o*表示反汇编器输出的正确　结果[1]。

DA(Disassembler Accuracy)表示反汇编的正确率：

$$DA = 1 - cf$$

测试选用了 7 个含有子程序异常返回的可执行程序，GNU 的 OBJdump 和 IDAPro5.0 用于对照反汇编的正确率 *DA*，测试结果如表 1 所示。

表 1　测试结果

序号	*DA*(反汇编正确率)/(%)		
	OBJDump	IDAPro5.0	抗混淆算法
1	91.3	91.3	100
2	87.8	85.1	100
3	85.0	84.6	100
4	85.7	88.5	100
5	93.7	96.7	100
6	90.6	92.5	100
7	92.4	96.1	100

通过测试结果可以看出，不论是采用线性扫描的 OBIDump，还是采用行进递归的 IDAPro，都不能很好处理子程序异常返回的混淆技术。本文的抗混淆算法能够正确跳过垃圾数据，提高反汇编的正确率。

4 结束语

恶意代码经常使用子程序异常返回方法，导致反汇编结果的错误，隐藏其真实行为。本文提出的抗混淆算法能够有效地对抗子程序异常返回的问题，应用在可执行程序的静态分析中，能够提高识别的正确率，对恶意程序的行为识别起到重要作用。

但恶意程序的混淆方法较多，本算法有一定的适用范围，只能处理子程序中通过调整堆栈、异常返回的情况。下一步工作准备借助对 CFG(控制流图)的分析来处理其他混淆情况，使反汇编结果正确率更高，对恶意行为的识别更加有效。

参考文献

[1] Kruegel C, Robertson W, Valeur F, et al. Static Disassembly of Obfuscated Binaries[C]//Proc. of the 13th Conference on USENIX Security Symposium. San Diego, CA, USA: [s. n.], 2004: 18.

[2] Collberg C, Thomborson C, Low D. A Taxonomy of Obfuscating Transformations[EB/OL]. (1997-01-14). http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a/.

[3] 罗　宏, 蒋剑琴, 曾庆凯. 用于软件保护的代码混淆技术[J]. 计算机工程, 2006, 32(11): 177-179.

[4] Christodorescu M, Jha S, Song D, et al. Malware Detection[M]. [S. l.]: Springer, 2007.

(上接第 12 页)

[3] 王　年, 范益政, 韦　穗, 等. 基于图的Laplace谱的特征匹配[J]. 中国图象图形学报, 2006, 11(3): 332-336.

[4] Park S H, Lee K M, Lee S U. A Line Feature Matching Technique Based on an Eigenvector Approach[J]. Computer Vision and Image Understanding, 2000, 77(3): 263-283.

[5] Caelli T, Kosinov S. An Eigenspace Projection Clustering Method for Inexact Graph Matching[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2004, 26(4): 515-519.

[6] Zhu Ping, Wilson R C. Stability of the Eigenvalues of Graphs[C]// Proceedings of Conf. on Computer Analysis of Images and Pattern. [S. l.]: Springer, 2005: 371-378.