软件防反汇编技术研究*

尚 涛, 谷大武
(上海交通大学 计算机科学与工程系, 上海 200240)

摘 要: 为了保护软件所有权,根据一般的反汇编算法的特征,提出代码重叠、跳转地址重定向和控制流混淆等几种代码混淆技术。这些技术能使反汇编结果出现混淆,误导攻击者对程序理解,从而提高软件防反汇编的能力,有效地阻止对软件的逆向分析,保护了软件的知识产权。

关键词: 软件保护; 逆向分析; 代码混淆; 程序理解; 反汇编
中图分类号: TP311 **文献标志码:** A **文章编号:** 1001-3695(2009)12-4553-05
doi:10.3969/j.issn.1001-3695.2009.12.043

Research on resistance to disassembly of software

SHANG Tao, GU Da-wu
(Dept. of Computer Science & Engineering, Shanghai Jiaotong University, Shanghai 200240, China)

Abstract: In order to protect the property of software, according to the characteristics of general disassembly algorithms, this paper provided code overlap, jumping address redirection and obfuscation of control flow etc. several approaches in code obfuscation technologies. These approaches are able to make the disassembly process go away, and misdirect the adversaries' comprehension to programs, consequently enhance the software's resistance to disassembly, thus preventing the software from reverse analysing and protecting its property effectively.

Key words: software protection; reverse analysis; code obfuscation; program comprehension; disassembly

0 引言

随着计算机技术普及与应用,计算机软件产业迅速发展起来,针对软件的各种攻击和未授权使用以及盗版复制等行为越来越多,软件安全成为保护知识产权的关键。目前的计算机软件基本上是以二进制代码形式发布的,攻击者通常利用静态反汇编工具或动态调试工具等逆向分析技术对软件可执行版本进行分析破解,通过寻找软件漏洞或抽取其核心算法等方式,对软件进行窜改进而窃取软件知识产权。软件逆向分析技术包括针对软件的反汇编[1,2]和反编译[3]两个部分。反汇编技术是把可执行二进制机器码反汇编成为基本可读的汇编语言程序代码的方法,一般包含静态反汇编技术和动态反汇编技术[4]。静态反汇编是把二进制代码一次性全部翻译为汇编代码,它的耗时与二进制文件大小成正比;动态反汇编是通过人为分析载入到反汇编器的二进制程序,捕捉运行特征指令翻译为可读的汇编代码。反编译技术是把汇编程序进一步反编译为可读性更强的高级语言代码。为了维护知识产权,目前常用的办法是通过法律打击非法行为,另外就是开发出软件防窜改技术来抵抗各种非法使用。

软件防窜改技术[5]是目前主要的软件保护技术,主要包括许可认证方式、水印技术[6]、窜改验证技术、软件加密等,或让软件通过网络运行在服务器端,这样防止敌手获取软件可执行文件进行逆向分析。这些技术总体上是软件开发的过程利用加密算法保护软件核心技术,在软件运行时对其进行解密后执行,的确对软件起到一定保护作用,但是一般软件运行需要先解密,且很难保证处理器能够快速解密软件;另外软件对运行环境的硬件要求较高,而且利用反汇编技术同样可以对加密技术的核心算法进行抽取分析[4],进而达到窜改水印或提取并破解相关验证算法。

软件代码混淆技术是从软件的反汇编层面进行操作的保护技术,从而加强了软件的抗逆向分析能力,对软件起到更强的保护作用。代码混淆技术早期多在软件反汇编和反编译研究中被粗略提到,Barak 等人[7]在 2001 年理论证明代码混淆技术不能完全彻底保护软件安全,但是大部分情况下,代码混淆技术不需要为软件提供绝对保护,只要能够有效地拖延针对软件的逆向分析就达到了保护的目的。随着软件逆向分析技术发展使得通过密码算法保护软件能力变得脆弱,最近几年代码混淆技术也逐渐被重视。目前代码混淆技术多数针对静态反汇编[8,9]技术或某特定语言开发平台[10]相关的混淆技巧。一方面,这种反静态反汇编方法无法对付动态反汇编技术对软件的逆向调试,由于动态反汇编技术通过调试软件读取堆栈内存中的参数,可以进一步确定软件的控制流程;另一方面,随着编程语言的发展,代码混淆技术应该应用于程序结构而不仅仅针对特定语言平台,其研究范围也将会越来越广泛。本文从物理层二进制代码混淆到高级语言的结构混淆,从不同层次研究代码混淆技术,最终达到防止软件反汇编。

收稿日期: 2009-02-26; 修回日期: 2009-03-23 基金项目: 国家"863"计划资助项目(2006AA01Z405)
作者简介: 尚涛(1985-),男,陕西安康人,硕士研究生,主要研究方向为软件安全(shangtaohit@yahoo.com.cn);谷大武(1970-),男,教授,博导,主要研究方向为密码学与信息安全.

1 二进制可执行文件格式(PE)

二进制可执行文件又称做 PE 文件, 具有 PE 文件格式[11] (portable executable file format) 。如图 1 所示, PE 文件根据内容分为不同的区块(section) , 每个区块拥有相同属性的数据。当 PE 文件被执行时, PE 装载器首先检查 DOS 直接跳转到 PE header。PE header 中包含有代码段起始点和程序入口点, 如果 PE header 有效, 执行程序将跳转到 PE header 后面的块表, 然后 PE 装载器读取块表中区块的信息, 并采用文件映射方法将这些区块映射到内存, 同时附上块表里所指定的各个区块的属性。PE 文件被映射入内存后, 将会从 PE header 里面 Address-of-Entry-Point 结构项所记录的 RVA(相对虚拟地址) 处开始执行。

PE 文件在执行时需要把文件从磁盘映射到内存(图 2) , 而且这种映射不是把 PE 文件的所有节段(section 或称区段) 全部映射到内存中去, 而只把一些需要执行的代码映射到内存, 如. text、. data 段等。由于可执行文件从磁盘区块映射到内存的过程中必须保持文件对齐和内存对齐, 每个代码区段映射到内存后以内存某个页边界为开始, 即每个区段的第一个字节对应磁盘的某内存页起始位置。在 x86 系统中, 内存页的大小是 4 KB, 即内存页对齐地址为 0x1000, 则每个区块必须以 0x1000 的整数倍数开始; 而硬盘的对齐值一般是 512 Byte, 即在磁盘一般按 0x200 取偏移地址, 每个区块是按 0x200 的整数倍数的文件偏移开始的。这就造成了文件在内存中的相对虚拟地址(相对虚拟地址(RVA) = 虚拟地址(VA) - 基地址(image base)) 与文件在磁盘的偏移地址(又称物理地址(RAW offset)) 不一致。

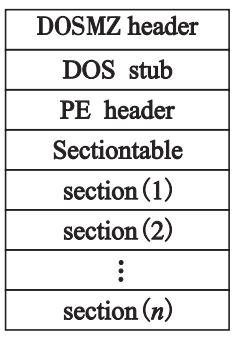

图1 PE文件格式

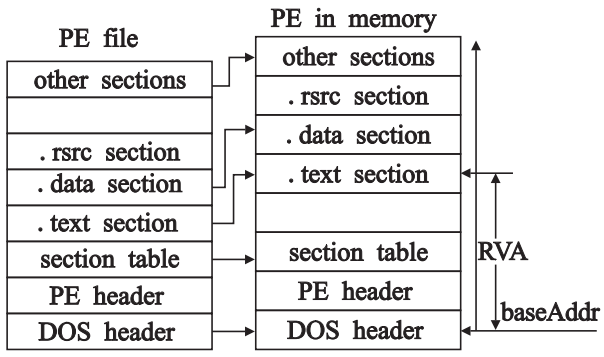

图2 PE文件内存映射

另外磁盘文件中每个区块大小是 512 Byte, 但 PE 文件的每个区段的实际代码或数据并不一定刚好是 512 Byte, 所以可执行文件在磁盘上的存储都存在空隙, 一般用 0x00 填补或在这里添加相应的恶意混淆代码, 可以利用这些间隙实现代码混淆技术。为了保护软件可以利用二进制重写工具, 如 Etch[12], 可重写 Win32/Intel 系统下 PE 可执行文件, 可用于重写 Linux/x86 的二进制文件, UQBT[13] 等。PE 文件的这些特征是软件能够通过对二进制代码修改、添加混淆代码, 起到保护软件的原因。

2 反汇编的基本策略

二进制机器代码一般包含代码段、数据段、只读数据段等, 这些机器代码包括有对软件描述以及软件执行的部分。反汇编是把二进制机器代码段、数据段等代码翻译为可读取、可分析的汇编指令代码。反汇编一般包括静态反汇编和动态反汇编。静态反汇编是在不运行可执行文件的情况下直接翻译出汇编代码[4]; 动态反汇编是在执行软件的同时, 把运行所得特征数据和反汇编结果相结合形成的汇编代码。由于逆向工程的反编译是建立在对软件反汇编的基础上, 防止和混淆反汇编同样为反编译过程增加了难度。目前通常使用的反汇编策略包括线性扫描策略和递归扫描策略[1]。

2.1 线性扫描策略

线性扫描算法是反编译器从可执行文件的入口开始读取软件二进制机器指令, 逐条反汇编成为汇编语言, 直到 PE 文件的所有代码段二进制指令被反汇编完毕为止。算法如下:

```
global startAddr, endAddr;
procedure DisasmLinear( addr)
while ( startAddr < addr < endAddr) {
I = decode instruction at address addr;
//对当前地址的指令进行反汇编
addr + = length ( I) ; //修改 addr 的值令 I 指向下一条指令地址
}
```

线性扫描策略应用在很多反汇编软件中, 大部分静态反汇编器采取这种策略, 如 GNU utility objdump、GNU 调试器(GDB) 、Microsoft 的 WinDug 调试器和 objdump 工具等多采用的是线性扫描策略。线性扫描策略能够把每条二进制指令反汇编成汇编语言, 但是它不能正确地区分指令中的数据和代码指令, 通常会把嵌入在指令中的数据错误地翻译成为指令。线性扫描算法在一些情况下会与递归扫描算法相结合应用在反汇编过程中。下文将详细叙述这点并利用线性扫描算法的这一缺点提出相应的防止反汇编策略。

2.2 递归扫描策略

线性扫描策略是逐条翻译机器指令, 无法正确地区分二进制代码中的指令和数据, 这将不能保证反汇编结果从整体上是正确的。相比线性扫描策略, 递归扫描提取二进制代码中的跳转、调用分支指令等关键的控制指令, 根据所得到的控制流中的调用指令地址、跳转目的地址决定相继的需要反汇编的指令位置, 能够区分程序中的代码和数据, 从而有效地绕开了程序中的无关数据。下面是递归扫描算法:

```
global startAddr, endAddr;
procedure DisasmRec( addr, instrList)
{ if ( addr has been visited already)
    return;
  do{ I = decode instruction ( addr) ;
    //把当前反汇编的指令地址赋给 I
    addr. visited = ture; //标记 addr 处的指令访问过
    instrList + = I; //把已经反汇编的指令加入 instrList 列表
    if ( I is a branch or function call) {
      T = set of each possible target t of I do;
      //把指令 I 所要跳转的所有可能地址放入到集合 T 中
      for each( target∈T) {
      DisasmRec( target, instrList) ;
      }
      }
    else addr + = length( I) ;
    //把当前指令的下一条指令地址传递给 addr
  } while ( startAddr < addr < endAddr)
}
```

递归扫描算法能够提取程序的控制流程, 防止反汇编器错误地把数据翻译成指令, 极大地减少了反汇编中的错误, 它被很多二进制翻译或优化系统利用, 如 IDA Pro、OllyDbg、PE-Browse Profession 等反汇编分析工具。递归扫描策略的优势是很好地把握了整个程序的各个模块, 防止翻译指令反汇编插入

在跳转指令之后的无关代码。

具有递归扫描策略反汇编器能够有效地区分指令和数据。但是在处理目标模块内顺序流如 add、move、push、pop 等指令时采用策略与线性扫描算法一样,逐条翻译二进制指令为汇编代码。在遇到条件跳转指令等间接转移或间接跳转时,由于无法及时确定跳转条件,反汇编器会推迟反汇编时间并在最后利用跳转表恢复机制进行处理;在遇到无条件跳转指令时,反汇编器无法确定跳转目标地址的合法性,反汇编器自动跳转到目标地址继续进行反汇编;在遇到分支函数调用指令时,反汇编器在不考虑目标地址的合法性的前提下,对所有的目标地址进行汇编,但对于无法及时确定调用函数目标地址的情况,反汇编器会推迟反汇编时间并利用条件跳转表恢复机制进行处理。针对递归扫描策略存在的这些问题,本文将详细讨论防止反汇编技术。

程序编译过程中,编译器针对代码中的分支调用函数 switch 会产生一个跳转表[14],保存在可执行文件内。跳转表是包括一系列函数跳转目标地址的数据段(图 3),在进行反汇编时,为了找到分支调用函数的跳转目标地址,反汇编器通常采取跳转表恢复机制来获取跳转目标地址。对于一般的 switch 模式,如代码段和相对应的跳转表如图 3 所示。

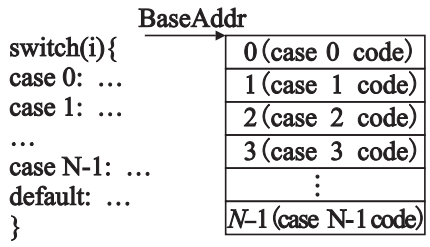

图3　跳转表

跳转表恢复机制通过切片和表达式置换技术可以得到跳转表首地址、分支数目、下界、上界等信息,跳转表恢复机制根据所得到的这些信息利用算法恢复出所有的跳转目标地址。跳转表恢复机制为指令代码相应的指令跳转地址,反汇编器再根据这些跳转地址构成了程序的控制流,这是保证反汇编器区分指令和数据、绕过添加混淆代码的关键。其算法如下:

```
r = evaluate i;  //读取到的跳转第几个地址
if r  > = N goto default;
//如果跳转个数大于最大次数时,使得跳转转向 default
r*  = 4;  // 一般一条跳转指令占据 4 Byte 数据位
r + = baseAddr; //计算出第 r 条跳转的地址
jmp r;
```

虽然递归扫描策略能够获取控制指令,有效地区分一些指令中代码和数据,但它最明显的弱点是在提取程序控制流程的时候无法确定跳转地址的合法性。处理间接调用或间接跳转指令非常困难,只要对跳转指令进行稍微修改,使得跳转地址指向混淆的无关代码,通过跳转表恢复机制得到的跳转地址将会不准确。

3　防止反汇编技术

反汇编器所采用的策略基本上是由上面所述的两种算法独立或者相结合构成,大部分已经提出的防止反汇编技术都是基于指令系统结构和这两种策略所存在的缺点设计的。针对以上讨论,本文根据反汇编几个方面的特征提出防止逆向分析策略如下:

a) 混淆二进制代码中的数据流和指令流,即把数据嵌入到代码段区域。

b) 根据二进制指令变长的特征,设计程序使得汇编出具有混淆功能的二进制代码,利用反汇编入口地址的随机性造成反汇编结果出错。

c) 改变跳转指令为间接跳转或调用指令。

d) 隐藏混淆分支跳转、调用函数的目标地址。

3.1　代码重叠技术

根据上述两种反汇编策略,特别是线性扫描策略对于代码中的数据和指令不加区分否认的特点,本文结合两种代码重叠技术[15]来对反汇编进行混淆。代码重叠技术可以使代码共享一条指令代码或整段代码,使得共享段被多个不同的指令控制流所调用。代码重叠技术可以利用在高级语言的源代码层次,也可利用在二进制代码存储的物理层。代码重叠技术可以混淆程序的控制流程,或者使得机器指令反汇编出错。这种方法主要是利用了 IA32、x86 等系统采用灵活的变长指令特征,反汇编器会随时对任意地址作为程序入口地址读取的机器码进行翻译,可以从上述两个层次进行重叠。

代码重叠包括语义重叠技术和物理重叠技术。语义重叠技术是程序代码共享调用相同指令语句或者整块代码完成程序不同的功能,这种方式使得程序产生的多条控制流相互交汇,从而使得逆向分析得到的控制流混乱,攻击者很难得到正确的结果。语义重叠技术可以让有效的和无效的函数块调用一个共享代码块,前提条件是保证那些无效代码段在程序运行的时候不会被执行,但多个无效的函数块构成的控制流在反汇编时也会被翻译出来,这就极大地增加了逆向分析的难度,达到保护软件的目的。如图 4 所示,使用语义重叠技术、get 函数、send 函数以及一些混淆代码段共享调用 func_ctr 函数,有效地隐藏了软件的函数执行顺序及目的。

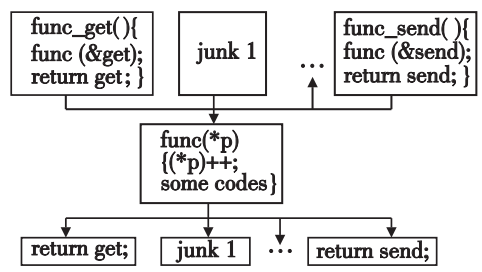

图4　语义重叠

物理重叠技术是指使不同机器指令之间可以共享相同的二进制字节段,由于采用复杂指令系统计算机(CISC)的机器指令是变长的,反汇编器无法确定指令的长度,也就无法确定一条指令结束的地址,从而可以使得反汇编器从不同的指令入口点所开始翻译的汇编代码相互不同。

3.2　插入垃圾代码

对于采用了线性扫描策略的反汇编器,一种现有的有效防止反汇编的方法[16]是在程序中加入垃圾代码,如操作数、恶意代码等,使反汇编结果出现错误。考虑到要保证程序运行时被加入这些恶意混淆代码指令不会运行,一般把这些代码放在跳转指令后面。

3.3　分支跳转地址多重定向技术

根据上文所述,反汇编器利用跳转表恢复机制可以获得分支函数跳转的目标地址,达到获取二进制代码程序的控制流程,避免反汇编被添加在一些控制指令之后的混淆代码,这种

方式能够很好地解决线性扫描策略造成的错误。针对这种策略,可以把程序中的直接跳转指令(图 5)改造成有分支函数(图 6)调用的形式,并且利用分支跳转地址重定向技术迷惑跳转表恢复机制。

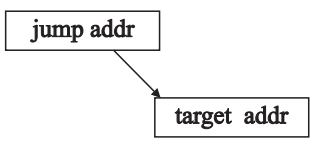

图5 直接跳转结构

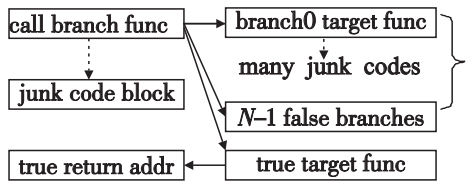

图6 多重重定向结构

首先跳转表恢复机制根据分支函数的条件判断来计算跳转的目标地址,据此可以伪造多个虚假的分支函数(图 6 中的大括弧包含的 N 个虚假调用分支);把一个正确的调用隐藏在所有目标函数中,如图 6 中 true target func 为正确函数,然后利用寄存器或内存传送数据使得判断条件保证调用函数跳转到正确的目标地址。其次,分支调用函数在调用函数指令后,通常使用跳转表恢复机制所得到的跳转目标地址执行被调函数,被调用函数执行完成任务后返回该调用 call 指令处,并顺序执行下一条指令。分支跳转多重定向技术实际是修改分支调用函数的目标地址和返回地址(图 6 中 true return addr 为正确返回地址),并在调用函数指令后加入混淆代码,如图 6 中的 junk code block,使得反汇编结果出错。利用这种方法能够有效地使反汇编程序控制流混乱并造成跳转表恢复机制产生大量错误的跳转地址,能高效地抵抗反汇编。

图 5 中的直接跳转改造成图 6 中函数调用方式,首先造出 $N-1$ 个虚假的被调用的分支函数,且在这 $N-1$ 个假函数的混淆代码内部同样设置虚假的函数返回地址,并在调用 call 指令后添加混淆代码,这种方法能够从两方面有效地提高软件抵抗反汇编能力。反汇编器根据递归扫描算法进行反汇编,一方面反汇编器会利用跳转表恢复机制产生一个含有大量虚假目标地址的跳转表,使得敌手无法正确定位真正的跳转地址,通过虚假跳转表会造成程序控制流混淆;另一方面被调用函数内部返回地址的修改,但是系统默认调用函数的返回地址紧跟在调用指令后被压栈,反汇编器会错误地返回到调用指令的下一条指令中的地址,进而把恶意攻击者引入大量的混淆代码中。这种改造方式对于源程序的执行基本没有影响,能够很好地起到保护作用。

3.4 程序控制流混淆技术

一般情况,程序的控制流程是通过简单的条件判断来确定程序的下一个跳转地址,通过循环或顺序等方式执行程序。这种程序编译得到的二进制代码能够被采用递归扫描策略的反汇编器轻松提取控制流,恶意分析人员也能够据此很容易地提取出程序的核心算法。针对这种问题本文对程序的控制流进行改造,以提高软件防逆向分析能力。

控制流拓宽技术是一种有效的控制流混淆技术,把程序的直接间接跳转转换成 switch 跳转结构,并且使程序的控制流依赖于分支函数内部对判断条件的控制变量修改,使得程序的控制流和数据流相互依赖。在这种程序的控制流数据流相互结合的基础上,向分支函数的调用子函数之间添加大量虚假的混淆子函数模块,这些虚假函数模块在形式上与真正的被调用的模块类似,并且在程序运行时不会被调用,但是在反汇编分析时,反汇编器利用跳转表恢复机制反汇编出所有的分支目标地址,进一步会把所有的调用函数(包括添加的混淆函数模块)翻译成汇编语言。这将产生大量的无用信息,使得分析者找不到真正的控制流程。例如对于一个函数进行改进,相应的控制流如图 7 所示。函数代码如下:

```
int function( int a, int b){
do{
int m = 1;
if( a < b)
call function a1;  //function1 中修改 m, a, b 值
call function a2;  //function2 中修改 m, a, b 值
} while( m < c)
return m;}
```

对这种简单的控制流程进行修改。为了加强控制流对数据的依赖性并使得敌手无法得到准确的数据,调用函数内应该通过寄存器或内存单元的传值来间接修改控制变量,并在调用函数模块间加入多个混淆函数块,如图 8 所示。

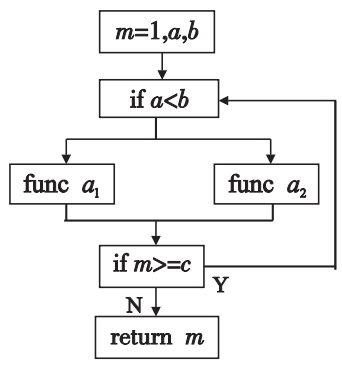

图7 混淆前的控制流

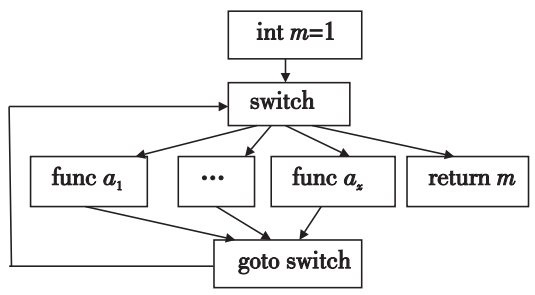

图8 混淆后的控制流

对上述程序采用控制流混淆技术后,由于控制流依赖于子函数产生的变量 m, a, b 的值,即使逆向分析者能够看清楚程序执行方式,但无法确定准确的函数调用目标模块。反汇编是根据恢复出来的跳转表提供的目标地址逐条进行翻译的,包括那些添加在函数块之间的混淆函数块,反汇编结果必然会混乱。

这种代码混淆方法的特点在于:

a) 添加混淆代码块增大了程序的代码量,给逆向分析工作带来麻烦;

b) 混淆了函数的控制流,反汇编分析人员无法在短时间内找到正确控制流程;

c) 通过混淆程序控制流,使动态反汇编和静态反汇编都变得非常复杂;

d) 极大降低代码的人工可读性,隐藏代码的目的意图,增加软件的复杂性,有效防止逆向分析。

上述结构变更不会对程序正常运行产生太大影响,但是攻击者要想弄清楚整个过程的控制流,必须分析所有可能被调用的模块,通过静态反汇编分析逐步找到与控制变量有关系的函数块,然后通过动态反汇编分析确定程序的控制流程。这一分析过程必须先从 switch 的变量变化预测出可能的分支模块,并再随着控制变量变化的过程不断循环测试计算出最后正确的控制流。

4 理论分析与实验验证

根据 3.4 节所述的控制流混淆技术,在对程序结构改造之前,只需要对确定的两个模块进行逆向分析就能找出正确的控制流。但是改造程序结构之后,要寻找出正确的控制流程主要是依靠变量 m 的修改情况和程序中包含有混淆代码段函数子

模块 ax 的个数 x 所决定。由于对于每次的 m 变化,变量在 x 个可能的子模块中修改所产生,则不难从推理中看出,需要攻击者对 x^m 个调用模块的具体语句进行分析才能找到可能正确的控制流,这使得程序逆向分析的复杂度呈指数级增加,有效地阻碍拖延了对软件反汇编。对于程序基于数据流问题已经被证明是一个 NPC 问题[17]。

3.1 和 3.2 节所述的代码混淆方式是针对采用线性扫描策略反汇编器从基本语句层次进行反逆向分析,通过加入混淆代码或者通过二进制代码的重叠技术能够很大程度使静态反汇编结果出现混乱。通过实验验证,把五段未添加混淆语句和五段添加了混淆语句的汇编代码分别编译成两组可执行文件,在 Windows 2000 环境下利用 SoftICE 对两组可执行代码反汇编,结果对比显示被加入混淆语句的文件反汇编结果 100% 出现错误或乱码。

3.3 和 3.4 节所述代码混淆方式是针对采用递归扫描策略反汇编器从代码结构层次进行反逆向分析,通过把简单的跳转和循环函数代码段改造成利用复杂的多分支函数调用结构,并在代码中加入不被程序执行的混淆跳转地址,使得改造后代码所编译可执行文件的反汇编速度减慢且结果出现大量难以分析的垃圾代码。在 Windows 2000 环境下利用 IDA Pro v4.17 对改造前后的跳转结构、循环结构代码和运行状况进行分析,如表 1 所示,其表中数值是每一组代码运行十次所得到的平均数。其中 size1 是程序结构混淆之前的代码量, size2 是采用相应的代码混淆技术之后的代码量,单位是 Byte; time1 是混淆前可执行文件的反汇编执行时间, time2 是采用代码混淆技术之后的反汇编执行时间,单位是 ms。CF obfuscation 是控制流混淆的缩写。从实验数据可知,代码混淆对软件逆向有明显的延缓作用,最重要的是反汇编结果产生大量不可读的垃圾代码,能有效隐藏软件的控制流。代码混淆之后程序的代码量变大,且反汇编的执行时间变长,若在代码中添加更多的混淆代码,效果会更加明显。

表 1　程序结构混淆前后状态

obfuscation style	size1 / Byte	size2 / Byte	time1 / ms	time2 / ms
jump redirection	638	2 346	28.45	243.35
CF obfuscation	344	1 468	432.40	1 452.55

本文所提到的几种防反汇编策略可以容易地应用在软件开发过程中,分别从软件的局部和宏观结构保护软件。通过整体改造程序控制结构达到混淆软件的控制流,防范逆向分析;通过局部加入混淆代码或改造跳转语句结构,造成软件反汇编代码难以阅读分析,加强软件防反汇编能力。

5　结束语

通过实验数据和理论分析可以看出,本文提出的几种代码混淆技术从软件编码到结构的基本层次着手,阻止攻击者对软件的破解分析,在保护程序的核心算法方面能够有效地防止反汇编分析工具的逆向分析,可以尽最大可能拖延恶意攻击者的逆向分析时间,从而使得软件在有效时间内受到保护。通过理论和实验验证,代码混淆技术相比众多的其他软件保护方式,能够利用相对比较简单的实现方式达到较好的保护效果。在此感谢 SafeNet 公司对本文的资助。

参考文献:

[1] ENJAMIN S, DEBRAY S, GREGORY A. Disassembly of executable code revisited[C]//Proc of the 9th Working Conference on Reverse Engineering. Washington DC: IEEE Computer Society, 2002: 45.

[3] CIFUENTES C, GOUGH K J. Decompilation of binary programs[J]. Software-Practice and Experience, 1995, 25(7): 811-829.

[3] HSIEH W C, ENGLER D, BACK G. Reverse-engineering instruction encodings[C]//Proc of USENIX Annual Technical Conference. Berkeley: USENIX Association, 2001: 133-145.

[4] 段钢, 王勇. 软件加密技术内幕[M]. 北京: 电子工业出版社, 2004.

[5] CIFUENTES C, FRABOULET A. Intraprocedural static slicing of binary executables[C]//Proc of International Conference on Software Maintenance. Washington DC: IEEE Computer Society, 1997: 188.

[6] WILLIAM F Z. Concepts and techniques in software watermarking and obfuscation[D]. New Zealand: The University of Auckland, 2007.

[7] BARAK B, GOLDREICH O, IMPAGLIAZZO R, *et al*. On the (Im) possibility of obfuscating programs[C]//Proc of the 21st Annual International Cryptology Conference, California. London: Springer-Verlag, 2001: 1-18.

[8] 吴金波, 蒋烈辉. 反静态反汇编技术研究[J]. 计算机应用, 2005, 25(3): 623-625.

[9] LINN C, DEBRAR S. Obfuscation of executable code to imporve resistance to static disassembly[C]//Proc of the 10th ACM Conference on Computer and Communications Security. New York: ACM Press, 2003: 290-299.

[10] ZHANG Xue-song, HE Feng-ling, ZUO Wan-li. An inter-classes obfuscation method for Java program[C]//Proc of the 2nd International Conference on Information Security and Assurance. Washington DC: IEEE Computer Society, 2008: 360-365.

[11] PIETREK M. Peering inside the PE: a tour of the Win32 portable executable file format[CD]. [S. l.]: Microsoft MSDN Library, 1994.

[12] TED R, GEOFF V, DENNIS L, *et al*. Instrumentation and optimization of Win32/Intel executables using Etch[C]//Proc of USENIX Windows NT Workshop. Berkeley: USENX Associaeion, 1997: 1.

[13] CIFUENTES C, van EMMERIK M. UQBT: adaptable binary translation at low cost[J]. Computer, 2000, 33(3): 60-66.

[14] BERNSTEIN R L. Producing good code for the case statement[J]. Software-Practice and Experience, 1985, 15(10): 1021-1024.

[15] MATTHIAS J, RAMARATHNAM V, MARIUSZ H J. Towards integral binary execution: implementing oblivions hashing overlapped instruction encodings[C]//Proc of the 9th Workshop on Multimedia & Security. New York: ACM Press, 2007: 129-140.

[16] ELDAD E. Secrets of reverse engineering[M]. Hoboken: Wiley, 2005: 337-338.

[17] MYERS E. A precise inter-procedural data flow algorithm[C]//Proc of the 8th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York: ACM Press, 1981: 219-230.