一种基于重定位信息的二次反汇编算法

曾 鸣[1] 赵荣彩[1] 姚京松[1] 王小芹[2]
(中国人民解放军信息工程大学计算机科学与技术系 郑州 450002)[1]
(清华大学计算机科学与技术系 北京 100084)[2]

摘 要 反汇编技术是静态分析二进制程序的基础,目前广为采用的反汇编算法是线性扫描和递归行进算法。前者无法正确处理代码段中嵌入数据的情况,后者则必须解决间接跳转目的地址的预测问题。本文介绍了这两种算法的原理,分析了其存在的问题,并利用二进制文件中的重定位信息对它们进行了优化。将优化后的两种方法结合起来,给出了一种新颖的二次反汇编算法,这种算法能够捕获反汇编过程中出错的情况,从而控制错误传播,并使得基于反汇编代码的应用可以针对出错情况进行相应的处理。

关键词 反汇编,二进制代码,线性扫描算法,递归行进算法

A Relocation Information-based Revisited Method for Disassembly

ZENG Ming[1] ZHAO Rong-Cai[1] YAO Jing-Song[1] WANG Xiao-Qin[2]
(Department of Computer Science, the PLA Information and Engineering University, Zhengzhou 450002)[1]
(Department of Computer Science, Tsinghua University, Beijing 100084)[2]

Abstract Binary disassembly routines form a fundamental component of software systems that statically analyze or modify executable programs. Linear Sweep algorithm and Rrecursive Traversal are two popular methods used by many systems that analyze or modify executable file. The former has the disadvantage that any data that is embedded in the instruction stream is misinterpreted while the latter has difficulty in dealing with indirect jumps. This article examines these two algorithms and illustrates their shortcomings. Relocation Information is used to improve them. A novel revisited method is given by combining the two improved algorithm which can detect situations where the disassembly may be incorrect and limit the extent of such disassembly errors.

Keywords Disassembly, Binary, Linear sweep, Recursive traversal

1 引言

很多场合下需要分析程序的二进制代码。例如,对遗产软件的再利用,其源码已经缺失,需要通过对可执行程序的分析来导出其接口。如果要将其移植到新的体系结构上,还需要对代码进行修改。对于闭源软件的分析,也只能依赖于二进制代码。而利用二进制代码来分析程序的逻辑功能是十分困难的,如果能够利用反汇编技术将其转换成可理解程度更高的汇编代码,可以给程序分析工作带来很大便利。近些年来,连接优化技术、二进制代码重写技术都非常受关注,它们都依赖于反汇编技术。可以说,任何针对二进制代码进行静态分析或者修改的应用系统都是建立在对机器码正确反汇编的基础上的。

在反汇编的过程中,存在着几个关键的问题,其中之一是数据与代码的区分问题。在 c/c++ 的二进制代码中,代码段有可能包含一些非执行的数据,例如用来实现间接跳转的跳转表、对齐字节以及面向对象程序中的虚函数表等。反汇编算法必须正确区分它们,才能生成正确的结果。此外,手工嵌入的汇编码、变长指令、多种多样的间接跳转实现形式,都增加了反汇编的难度。反汇编算法必须对这些情况做出恰当的处理,保证反汇编结果的正确性,这样才能保证基于其上的应用系统的正确性。

目前主要的两类反汇编算法是线性扫描算法和递归行进算法,它们都有着广泛的应用。采用线性扫描算法的系统包括 GNU 的 objdump 程序[1]、软件剖分工具 qpt[2] 和 EEL[3]、优化连接器 Spike 和 OM 等。采用行进递归算法的系统包括 Queensland 开发的 UQBT[4]。H. Theiling 博士在他的论文[5]中对实时系统分析时,采用了这种算法来重构二进制代码控制流图。

线性扫描算法和递归行进算法各有优点和不足。本文在第 2 节中介绍了两种算法的原理,分析了它们存在的问题。第 3 节中利用二进制文件的重定位信息针对两种算法面临的一些问题给出了解决方法。第 4 节中将利用重定位信息优化过的两种方法结合起来,给出了一种二次反汇编算法,此算法能够捕获到反汇编过程中出错的情况,给出报告,从而可以限制错误的传播。基于反汇编结果上的应用系统也因此能够获得对错误的"知情权",从而可以根据实际的需求进行相应的处理。

2 目前广泛采用的两种反汇编算法

线性扫描算法和递归行进算法是目前广为使用的两种反汇编算法,本节对这两种算法的原理加以介绍,并分析其存在的问题。

2.1 线性扫描算法

曾 鸣 博士,主要研究方向:逆向工程、软件安全;赵荣彩 教授,博导;姚京松 高工;王小芹 高工。

线性扫描算法找到程序的入口点，然后对位于入口点和代码结束之间的所有机器代码一个个顺序进行直接变换，转化为汇编语言。这种算法最大的优点是简单，但是很明显，它没有对所反汇编的内容进行任何判断，而是笼统地将遇到的所有机器码都作为代码来处理。这样，它将无法正确地将代码和数据区分开，数据也将被作为代码来进行解码，从而导致反汇编出现错误。而这种错误将一直持续到解码出错，导致反汇编过程无法继续（例如反汇编算法解码得出非法操作码），才能够被发现。

图 1 给出了一个线性扫描算法导致的反汇编错误示例。错误成因是代码中插入的对齐字节（数据，图中阴影所示部分）被作为代码来处理了，从而翻译成了指令，导致其后的解码出现明显错误而无法进行。例子里的二进制代码片段是 libc 中的 strrchr 函数在 Pentium Ⅲ处理器、RedHat Linux 下的机器码片段。从地址 0x809ef47 开始，插入了三个空字节（0x00，图中阴影所示），其目的是为了对齐，使得循环开始于地址 0x809ef4a 处。在利用 objdump 工具对图 1 中的机器码进行反汇编时，由于它采用线性扫描算法，扫描到地址 0x809ef47 到 0x809ef49 处的三个 0x00 字节时，把它们误认为是代码来处理，从而将其翻译成 add 指令。之后，线性扫描算法对错误浑然不觉，继续顺序处理机器码，生成连续的错误汇编代码。直到处理到地址 0x809efaa 处，反汇编出一条跳转指令，其目的指向 0x809ef4a，这个地址是位于 0x809ef49 处的 add 指令的中间！很显然，汇编结果是错误的。此外，位于 0x809ef49 的 add 指令，引用了一个绝对地址 0xee8304ee，此地址根本就不在执行程序的地址空间内！这些错误正是线性扫描算法无法区分代码和数据造成的。

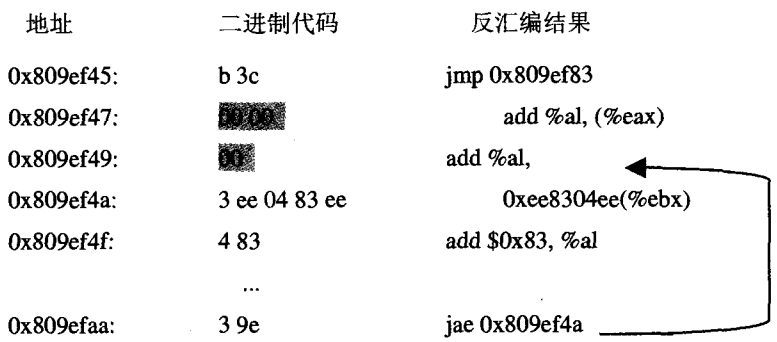

图 1　线性扫描算法存在的问题示意

2.2　递归行进算法

图 1 中出现的问题是由于线性扫描算法没有利用程序的控制流信息。为了对齐而插入的 3 个 0x00 字节前面是一条 jmp 指令，它使得程序在执行时直接跳过这三个空字节，而不会执行它们。如果反汇编算法在扫描程序的过程中，能够利用此类控制流信息，就可以确定二进制程序中哪些部分需要进行反汇编，哪些部分不需要，这就避免了图 1 中出现的错误。递归行进算法就是这样一种方法，它按照代码可能的执行顺序来反汇编程序，对每条可能的路径都进行扫描。当解码出分支指令后，利用静态分析技术确定程序可能跳转的地址集合，从这些地址继续进行反汇编。采用这种算法可以避免将代码中的数据作为指令来解码，也可以部分解决代码混淆技术中的花指令问题。采用递归行进算法的系统包括许多二进制翻译和优化系统[4~6]，它的优点在于实现比较简单，并且可以有效处理代码中包含数据的情况。

递归式行进算法的基本原理描述如下：

```
proc Disassemble(Addr, instrList)
{
  if(Addr has already been visited)
    return;
  do
  {
    instr = DecodeInstr(Addr);
    Addr. visited = true;
    add instr to instrList;
    if(instr is a branch or function call)
    {
      T = set of possible control flow successors of instr;
      for each (target T)
      {
        Disassemble(target, instrList);
      }
    }
    else
      Addr += instr. length; /* addr of next instruction */
  }while Addr is a valid instruction address;
}
```

每个可执行文件都包含一个入口，入口信息通常在文件头中指定。Disassem-ble()程序就从这个入口地址开始执行。假设对于任意分支指令和函数调用操作，都可以静态分析出它的后继指令集，那么递归行进算法能够将所有从入口点可达的指令正确地反汇编。对于图 1 的例子，递归行进算法在解码出 0x809ef45 的 jmp 指令后，将从 jmp 的目的地址 0x809ef83 处继续进行反汇编。当反汇编进行到 ox809ef4a 时，递归行进算法将返回到 0x809ef4a 处，继续进行解码。三个为对齐而插入的空字节不会被作为代码来反汇编，因为程序的任意一条执行路径都不能到达它们。

在递归行进算法中，一个十分重要的假设是对于任一条控制转移指令，其后继即转移的目的地址都能够确定。对于大部分情况，这都是可以做到的。但对于 n 元跳转，目的地址的确定有时是十分困难的。例如通过位于代码段的跳转表来实现间接跳转的情况，要确定其所有后继指令，必须准确地估计跳转表的大小。如果对跳转表大小估计过大，则会导致部分代码被作为数据处理而无法反汇编。反之，则会导致数据被作为代码错误的反汇编。可以看出，递归行进算法面临的一个重要问题是如何预测间接跳转的目的地址。在现实情况中，n 元跳转语句的实现方式非常多，并且是和编译器以及体系架构相关的，要想识别它们，也是具有相当难度的。

定位间接跳转的目的地址，目前主要采用的方法是程序切片[7]、常量传播[8]等技术。这些技术都是基于函数的控制流图进行的。然而，在函数尚可能存在部分指令没有被反汇编的情况下，其函数控制流图的生成本身就存在很多问题。本文 3.2 节将利用重定位信息给出一种定位间接跳转目的地址的方法。

3 利用重定位信息优化反汇编算法

3.1 重定位信息介绍

连接器在生成一个 PE 文件的时候，假设这个文件执行时会被装载到默认的基地址处，并且把. code 和. data 的相关地址都写入 PE 文件。如果装入时按默认的值作为基地址装入，则不需要重定位。但如果可执行文件被装载到虚拟内存的另一个地址，连接器所登记的那个地址就是错误的，这时就需要用重定位表来调整，在 PE 文件中它往往单独分成一块，用“. reloc”表示。

在 32 位代码中，涉及到直接寻址的指令都是需要重定位的。例如这样一条指令：

00401020：8B 0D 34 D4 40 00 mov ecx，dword ptr [0x0040D434]。

此指令地址在 0x00401020 处，6 个字节长。前两个字节(0x8B 0x0D)组成指令的操作码，后 4 个字节用来保存一个 DWORD 大小的地址(0x0040D434)。指令来自一个默认基地址为 0x00400000 的可执行文件。如果可执行文件确实在 0x00400000 处装入，那么指令能够按照现在的样子正确进行。但是，假设由于某种原因可执行文件从 0x00500000 处加载了，那么指令中的直接寻址地址就需要重新计算，加载器比较基址和实际装入地址的差。在本例子中是 0x00100000，将其加到 DWORD 大小的地址值里，形成新地址 0x0050D434。

在上述地址重定位计算过程需要的三个参数中，默认基地址在 PE 文件头中定义，实际装入地址由装载器确定。需要保留在重定位表中的信息仅仅是需要修正的代码的地址。重定位表由一系列的重定位项组成，不同的文件格式中重定位项的组织是不同的。最核心的数据是那些需要重定位的机器码的地址，其他的数据都是为了便于存取，或者节省空间等目的引入的。例如 PE 文件中采用类似按页分割的方法存储重定位项。在 PE 文件中，重定位表的地址可以从 PE 文件头得到，它指向顺序排列的许多重定位块，每一块描述一个内存页中所有的重定位项。PE 文件下的重定位表格式可以参考文[9]。图 2 给出一个 PE 文件格式重定位表实例，其中重定位表数据的高四位代表了重定位项的种类，低 12 位代表了需要重定位数据的地址。

重定位表偏移	数据	说明
0000h	00001000h	第一个块：页面起始地址是 00401000h
0004h	00000010h	重定位块长度是 10h
0008h	3012h	16 位重定位项，重定位位置：00401012h
000ah	3040h	16 位重定位项，重定位位置：00401040h
000ch	306fh	16 位重定位项，重定位位置：0040106fh
000eh	0000h	用于对齐的空白数据
0010h	00002000h	第二个块：页面起始地址是 00402000h
0014h	0000000ch	重定位块长度是 0ch
0018h	3080h	16 位重定位项，重定位位置：00402080h
001ah	30f0h	16 位重定位项，重定位位置：004020f0h
001ch	00000000h	重定位数据块结束

图 2 PE 文件中重定位表实例

利用重定位信息，可以对线性扫描算法和递归行进算法进行一些优化处理，下面两节的内容都是基于可执行文件中存在重定位信息的情况进行的。事实上，可执行文件中也有不包含重定位信息的情况，这种情况下文件必须被加载在默认基地址。但是这种情况在现实中并不特别多，因此假设重定位信息在许多应用场合是可行的。

3.2 利用重定位信息优化线性扫描算法

正如 2.1 节中所讨论的，线性扫描算法不能正确处理代码段中嵌入数据的情况。跳转表是代码段中经常嵌入的一类数据，它是实现间接跳转的一种方法，其表项是一个个地址。例如在 switch 语句中，存储各个分支的地址。本节利用重定位信息对线性扫描算法进行优化，使其能够处理指令流中嵌入的跳转表。

优化的主要原理是利用重定位信息来识别代码段中的跳转表(数据段中的跳转表不会导致反汇编问题)。线性扫描算法在处理一个地址时，首先判断它是不是跳转表项。如果是，则跳过，继续处理下一个地址。图 3 给出奔腾处理器上的一段汇编代码，实现了一个简单的 switch 语句，之前的汇编代码对 case 语句索引值进行判断，根据索引值 jmp 指令选择跳转表中的一项，从中取出目的地址。

```
movl -8(%ebp),%eax
subl $0x2,%eax
cmpl $0x5,%eax
ja 0xffffffd9 <80489dc>
jmp *0x8048a0c(,%eax,4)
8048a0c: 64 89 04 08
8048a10: 78 89 04 08
8048a14: 8c 89 04 08
8048a18: a0 89 04 08
```

图 3 代码段中嵌入的跳转表

重定位信息可以用来判断地址 a_i 是不是跳转表的表项。a_i 是跳转表表项的必要条件是：

①包含 a_i 的所有地址均为重定位项；

② 地址 a_i 处自身又包含指向代码段的重定位项。

这两个条件是 a_i 为跳转表项的必要但是非充分条件。这是因为满足这两个条件的重定位项，也可能是被指令作为立即数操作数使用的，而并不是跳转表的表项。幸运的是，在现代体系结构中，限制在一条指令中可以连续出现的立即数操作数的最大个数 K_{max}(在 Intel x86 中，$K_{max}=2$)。这样，如果一个代码段中出现 n 个连续的指向代码段的重定位项，那么最多前 K_{max} 项是指令的操作数，剩余的 $n-K_{max}$ 项一定是数据，是跳转表项。至于前面 K_{max} 项的情况，可以在随后的分析中进一步将它们区分开来。

上述方法分析出的跳转表将代码段分成了许多不连续的部分，为方便描述，称每一部分为“块”。每一块的开头可能是一个函数，也可能是前一个跳转表的结尾。针对每一块，利用 2.1 节中介绍的线性扫描算法对其进行反汇编。每一块反汇编结束后，对反汇编结果中的最后一条指令进行分析。假设这条指令有 $m(0<m<K_{max})$个立即数操作数，就推断出在连续的 n 个重定位项中，m 个是指令的操作数，另外 $n-m$ 个则是跳转表项。这样，进一步区分了 K_{max} 项中的跳转表项和指令操作数。

综合以上的讨论，利用重定位信息对线性扫描算法进行优化的方法如下：

(1)对于程序中 N 个连续的重定位项($N>K_{max}$)，标记后 $N-K_{max}$ 项为数据，是嵌入在代码段中的跳转表项。

(2)对未标记的每一段地址序列，即每一块，进行如下操作：

a) 对每一块用 2.1 节中的算法进行反汇编，直到到达一

个已标记过的地址。

b)如果反汇编出的最后一条指令不完整，则丢弃它。

c)假设最后一条正确反汇编的指令有 m 个立即数操作数($0<m<K_{max}$)，将剩下的 $K_{max}-m$ 个地址标识为数据，它们也属于跳转表项。

值得注意的是，本节对线性扫描算法的完善是建立在重定位信息的基础上的，优化后的线性扫描算法可以正确处理代码段中嵌入跳转表的情况。但是，对于其他一些和重定位信息无关的数据，如插入的 NULL 字节，此种方法并不能处理。

3.3 利用重定位信息预测递归行进算法中的非直接跳转目标地址

本节介绍一种利用重定位信息来预测递归行进算法中间接跳转目的地址的方法。在反汇编函数 f 的过程中，定义 R_f 为这样的地址集合，它包含了位于函数范围内的所有重定位项。定义 J_f 为这样的集合，它里面的地址，其自身是重定位项，指向的内容也是一个重定位项。通常来说，假设非直接跳转的目的地址是 ε，实现时通常都是先将其加载到某个寄存器中，然后跳转到这个寄存器。这样，ε 这个地址一定是可以重定位的，因此 $\varepsilon \in R_f$。根据这样的现实，可以认为所有的非直接跳转目的地址都属于 R_f。而 J_f，可以认为它代表的是跳转表项，跳转表项是数据，不可能是间接跳转的目的地址。根据以上的分析，可以粗略地认为 R_f-J_f 代表了间接跳转可能的目的地址集合。

上述的方法在实际中是可行的，但关于目的地址集合的计算并不精确。真实跳转地址一定包含在 R_f-J_f 中，但 R_f-J_f 也可能包含一些实际上不是目的地址的项，这会导致反汇编结果出现错误。图 4 给出了一个例子，代码片段来自 C 函数——mpn_add_n，运行在 linux 系统上。

```
地址              二进制代码          反汇编结果
0x80b1d8b:        8d 84 c0 95 1d 0b 08    lea 0x80b1d95(%eax,%eax,8),%eax
0x80b1d92:        ff e0               jmp *%eax
0x80b1d94:        8d                  lea
0x80b1d95:        74 26 00                0x0(%esi,1),%esi
0x80b1d98:        8b 06               mov (%esi),%eax
0x80b1d9a:        13 02               adc (%edx), %eax
0x80b1d9c:        89 07               mov %eax, (%edi)
                  ...
```

图 4　目的地址预测中的错误示例

图 4 中，0x80b1d8b 处的指令 lea 为寄存器 eax 计算出一个新值。根据 lea 指令的含义，eax 的新值＝0x80b1d95 ＋ eax 旧值 ＋ 8 * eax 旧值。手工检查程序，可以确定从 0x80b1d98 处开始了一个循环。在实际的执行过程中，lea 指令计算出来的值总是一个位于循环体的中间的合法地址。然而，静态分析 0x80b1d92 处的跳转目的时，并不能确定 eax 的值会不会为 0。按照上文预测目的地址的方法，$0x80b1d95 \in R_f-J_f$（对应 eax 值取 0 的情况）。因此，反汇编从 0x80b1d95 处继续进行，而这显然是错误的，因为这个位置是指令 lea 0x0(%esi,1),esi 的中间。实际上，例子中这条 lea 指令存在的目的仅仅是使得下面的循环开始在 8 字节边界，并不具有功能性的语义。

4　二次扫描算法

上面讨论的递归行进算法和线性扫描算法在反汇编中都不可避免地会产生错误。即使利用重定位信息对其进行优化，错误仍然是不能完全避免的。反汇编结果的错误必然影响到基于其上的应用系统的准确性。此外，在这两种算法中，许多错误都被隐藏起来了，算法并不给出报告，应用这些结果的系统对此浑然不觉。如果有一种方法可以识别出这些错误，就可以在后期工作中进行相应的修正，最起码也保证应用系统对错误的“知情权”，便于根据具体情况进行处理。

将递归行进算法和优化后的线性扫描算法结合起来，本文给出了一种二次反汇编算法，它可以在一定程度上识别反汇编过程中的错误。这种方法首先利用优化后的线性扫描算法对可执行程序进行反汇编。每一个函数的反汇编代码再用递归行进算法进行验证，判断两种方法的反汇编结果是否有矛盾的地方，诸如是否存在转到某条指令中间的现象。如果有，则说明反汇编结果存在错误，要留待进一步检查和优化。具体描述此方法如下：

(1)利用递归行进算法对函数中的每一条指令进行反汇编。

(2)对于地址 a_I 处的指令 I，检查线性扫描算法是否也在 a_I 处解码出了指令 I。如果不是，则报告验证失败。

(3)如果函数中所有的指令都通过了验证，则报告验证通过。

在算法的实现中，使用递归行进算法进行验证时，不必再一次为函数生成完整的反汇编代码，只需要解码一条指令，对比一条指令，这样可以节约存储空间。

当两种反汇编算法生成的结果不一致时，有问题的函数被标识出来，留待检查和修改。对上述算法也可以进行一些简单的扩展。比如，在两者不一致时，尝试判断哪一种是正确的。例如，如果函数不包含间接跳转，就可以认定递归行进算法的分析结果是正确的。这样，比简单的放弃这个函数或者标识其为错误更为合理。

总结　对可执行代码的正确反汇编是逆向工程应用如二进制代码重写、修改、比较等技术的基础。本文对目前广泛使用的线性扫描算法和递归行进算法进行了讨论，分析了它们在反汇编过程中出错的原因，并利用重定位信息给出了各自改进的方法。现存的这两种方法在反汇编结果出错时没有任何的警告信息，这必将影响基于反汇编结果的众多应用系统的准确性。本文将利用重定位信息优化之后的线性扫描算法和递归行进算法结合起来，给出了一种二次反汇编算法，此算

(下转第 292 页)

入 TSMQ 队列时，路由服务器会自动地从 TSMQ 队列中将消息转发出去，直至目的地。通信实现流程如图 5 所示。

结束语 与用 SSL 协议开发的面向消息传输中间件不同的是，TSMQ 只是在普通 MOM 上增加模块实现功能的扩展，并不是在中间件与传输层之间增加一个新的安全套接层，即可以简化系统的复杂度，提高系统运行效率，也可以避免增加安全套接层后带来应用层数据安全路由的问题。因为增加安全套接层以后密钥的协商和数据通信的加密工作就放在安全套接层了，无论是客户端与路由服务器，还是路由服务器与路由服务器之间的连接，路由服务器都要参与会话密钥的协商，也就是说路由服务器的安全直接决定整个通信的安全性，这就带来了巨大的安全隐患，路由服务器一旦被攻击者控制，攻击者就可以轻易地取得消息明文。而在整个 TSMQ 通信过程中，路由服务器只知道消息的目的地而不知道传输的具体明文，这只有知道会话密钥的会话双方客户端才知道，从而消除了安全隐患。

参 考 文 献

1 Blahut R，Clancy T，Hua X，et al. Secure Middleware for Infrastructure Systems [R]. University of Illinois，Urbana-Champaign，January 2004. http://www.ifp.uiuc.edu/～kiyavash/papers/secureMW_04_TR.pdf

2 ndersen A，Blair G，Myrvang P H，Stabell-Kulo T. Security and middleware [C]. In：Object-Oriented Real-Time Dependable Systems，2003. Proceedings of the Eighth International Workshop on，Jan. 2003. 186 ～ 190

3 Stallings W. Cryptography and Network Security Principles and Practice (ThirdEdition)[M]. PrenticeHall，2003

4 Nash A，Duane W 著. 张玉清，等译. 公钥基础设施(PKI)：实现和管理电子安全[M]. 北京：清华大学出版社，2003

5 Tanenbaum A S 著. 潘爱民译. 计算机网络(第 4 版)[M]. 北京：清华大学出版社，2004. 8

6 李婉婷. 基于 J2EE 的安全中间件的研究与实现[D]. 解放军信息工程大学，2005. 6

(上接第 283 页)

6 Ravindran B，Li Peng，et al. Proactive resource allocation for asynchronous real-time distributed systems in the presence of processor failures. Journal of Parallel and Distributed Computing，2003，63(12)：1219～1242

7 Sih G C，Lee E A. A Compile Time Scheduling Heuristic for interconnection Constrained Heterogeneous Processors Architectures. IEEE Trans. Parallel and Distributed Systems，1993，4(2)：175～187

8 王永炎，王强，等. 基于优先级别表的实时调度算法及其实现. 软件学报，2004，15(3)：360～370

9 Bajaj R，Agrawal D P. Improving Scheduling of Tasks in a Heterogeneous Environment. IEEE Transaction on Parallel and Distributed Systems，2004，15(2)：107～118

10 Paulin P G，Knight J P. Force Directed Scheduling for the Behavioral Synthesis of ASICs. IEEE Trans. Comuter-Aided Design of Integrated Circuits and Systems，1989，8(6)：661～679

11 Wang C Y，Parhi K K. Resource Constrained Loop List Scheduler for DSP Algorithms. J VLSI Signal Processing，1995，11：75～96

12 Ito K，Lucke L. ILP-Based Cost Optimal DSP Synthesis with Module Selection and Data Format Conversion. IEEE Trans. VLSI systems，1998，6：582～594

13 Shao Z，Zhuge Q F，et al. Efficient Assignment and Scheduling for Heterogeneous DSP Systems. IEEE Trans. Parallel and Distributed Systems，2005，16(6)：516～525

(上接第 287 页)

法利用两种方法分别对二进制代码进行反汇编，引入了相互验证的过程。这种二次反汇编算法能够报告出反汇编结果中的错误，从而限制错误的传播。错误的代码可以根据应用的需求进一步进行处理。在 SPECint-95 和 SPECint-2000 上的实验证明，这种方法可以有效地对二进制文件进行反汇编。

参 考 文 献

1 GNU Project - Free Software Foundation，objdump，GNU Manuals Online. http://www.gnu.org/manual/binutils-2.10.1/html chapter/binutils4.html.

2 Larus J R，Ball T. Rewriting Executable Files to Measure Program Behavior. Software-Practice and Experience，1994，24(2)：197～2181994.

3 Larus J R，Schnarr E. EEL：Machine-Independent Executable Editing. In：Proc. SIGPLAN '95 Conference on Program-ming Language Design and Implementation，June 1995. 291～300

4 Cifuentes C，Van Emmerik M，Ung D，et al. Preliminary Experiences with the UQBT Binary Translation Framework. In：Proc. Workshop on Binary Translation，Oct. 1999

5 Theiling H. Extracting Safe and Precise Control Flow from Binaries. In：Proceedings of the 7th Conference on Real-Time Computing Systems and Applications，Dec. 2000

6 Cifuentes C，Gough K J. Decompilation of Binary Programs. Software-Practice and Experience，1995，25(9)

7 Cifuentes C，Van Emmerik M. Recovery of Jump Table Case Statements from Binary Code. In：Proceedings of the Inter national Workshop on Program Comprehension，May 1999

8 De Sutter B，De Bus B，De Bosschere K，et al. On the Static Analysis of Indirect Control Transfers in Binaries. In：Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)，2000

9 罗云彬. Windows 环境下 32 位汇编语言程序设计. 北京：电子工业出版社，2002