文章编号:1001 - 9081(2009)01 - 0242 - 03

跟踪式智能反汇编算法研究

张龙杰,谢晓方,袁胜智,李洪周
(海军航空工程学院 兵器科学与技术系,山东 烟台 264001)
(zljcx1983@yahoo.cn)

摘 要:研究了嵌入式系统文件反汇编过程中存在的主要问题,给出了进行跟踪式智能反汇编的关键算法。首次采用二叉树结构对代码扫描过程进行跟踪处理,给出了生长二叉树的递归遍历算法,克服了传统反汇编过程中建立大量数组、链表以及图表的缺点;通过对二叉树的逆向浏览,解决了间接转移指令的寻址问题,并给出了寻址算法;最后,给出了数据区边界校验算法。对于进行程序反解及软件逆向工程具有较重要的参考价值。

关键词:嵌入式系统;智能反汇编;反汇编算法;二叉树;软件逆向工程
中图分类号: TP314 **文献标志码**:A

Study on algorithm of intelligent disassembling in tracking mode

ZHANG Long-jie, XIE Xiao-fang, YUAN Sheng-zhi, LI Hong-zhou
(*Department of Science and Technology of Weapons, Navy Aeronautical Engineering Academy, Yantai Shandong* 264001, *China*)

Abstract: This paper made a research on the issues in disassembling embedded system files, and brought about three core algorithms to do intelligent disassembling in tracking mode. It adopted binary tree to dispose the process of scanning for the first time, and offered a recursion traverse algorithm to grow the binary tree, which overcame the disadvantage of establishing many arrays, tables and diagrams in traditional ways. By browsing the binary tree in a reverse direction, it resolved the problem of searching indicate addresses, and provided a correlation algorithm. In the end the paper brought forward an algorithm to checkout the data block boundary. The work of this paper has great reference value to program disassembling as well as software reverse engineering.

Key words: embedded system; intelligent disassembling; disassembling algorithm; binary tree; software reverse engineering

0 引言

分析一个软件系统,反汇编是经常用到的手段之一。但现成的反汇编软件大多只适合 MS DOS 和 MS Windows 标准的文件格式,对于嵌入式系统中使用的许多文件,由于其格式与标准的文件格式不同,这些反汇编软件大多不能直接使用。

本文研究了对嵌入式系统文件进行智能反汇编的一些关键算法,首次采用二叉树的方式对代码扫描过程进行跟踪处理,克服了传统反汇编算法要建立大量数组、链表以及图表的缺点[1-2],同时也很好的解决了逐码反汇编方法不能正确区分指令区与数据区的问题[3]。

1 程序反解流程

要想对文件进行反汇编处理,首先要获取程序的入口地址。可以采取以下几种方式来处理:

1)搜寻程序起始标志,根据起始标志的机器码尝试反汇编。

2)通过系统命令查看系统内部信息,例如查看程序状态字,了解进行反汇编的代码程序所在的存储区地址,从而确定入口地址。

3)设法找出代码中的地址常量(绝对地址)及转移指令通过此常量控制达到的指令,由此指令的相对地址求得入口地址:

入口地址 = 绝对地址 - 相对地址 (1)

4)搜寻一般函数起始代码的机器码,通过机器码尝试反汇编。得到了程序的启动地址以后,就可以对其进行反解处理,如图 1 所示。

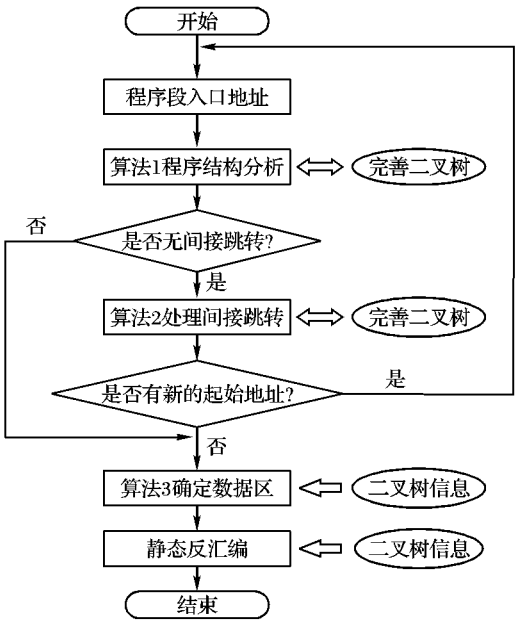

图 1 程序反解流程

2 程序结构分析

定义 1 代码基本块:在汇编代码中,只有一个入口地址、一个出口地址,且不存在跳转、子程序调用等操作的代码

收稿日期:2008 - 07 - 29;**修回日期**:2008 - 09 - 22。

作者简介:张龙杰(1983 -),男,山东莱阳人,硕士研究生,主要研究方向:计算机软件、软件逆向工程; 谢晓方(1962 -),男,河北承德人,教授,博士生导师,主要研究方向:武器系统建模与仿真、火力指挥与控制; 袁胜智(1977 -),男,湖北武穴人,讲师,博士研究生,主要研究方向:武器系统建模与仿真、图像压缩、目标识别; 李洪周(1981 -),男,山东济宁人,硕士研究生,主要研究方向:计算机软件、图像压缩。

模块称为代码基本块。

对程序进行结构分析的目的就是遍历所有确定的程序执行路径,获取相应代码基本块的起始、结束地址,同时记录间接跳转指令的相关信息。

引理 1　条件转移指令(jz、jbe、jnl 等)的转移地址位于当前段内,并且只能使用段内直接寻址方式进行寻址;无条件转移指令(jmp)和子程序调用指令(call)的转移地址可以是段内也可以是段间的,寻址方式不受限制。

条件转移指令对应的是双分支跳转,根据引理 1,只要是条件转移指令的程序分支,就只能使用直接寻址方式进行寻址,这样在对程序进行第一遍扫描时,就能确定它们的执行路径。

根据以上分析,在对程序反汇编的过程中,采用二叉树结构对程序的执行流程信息进行记录。

二叉树节点的定义:

```
struct treeNode( )
{
  long l_cAddr;                    //代码基本块的起始地址
  long l_eAddr;                    //代码基本块的结束地址
  long l_nAddr;                    //下一条指令的地址
  long l_lAddr;                    //目的代码的地址,左子树
  long l_rAddr;                    //目的代码的地址,右子树
  bool b_isLeave;                  //不可再生长的叶子节点标志
  bool b_isingleJmp;               //单跳转标志
  bool b_islongJmp;                //段间远跳转标志
  bool b_iscan;                    //扫描标志
  node *n_leftChild;               //左子树
  node *n_rightChild;              //右子树
} node;
```

二叉树的生长按照左子树优先的策略进行。这样,当 node. b_isingleJmp = true 时,右子树为空,置 node. l_rAddr = null;当 node. b_isingleJmp = false 时,按照 true 跳转生长左支树,按照 false 跳转生长右支树。

在 node 数组结构体中,每个 node 对应一个代码基本块。其中,node. l_cAddr 既是节点的起始地址也是代码基本块的起始地址,node. l_eAddr 既是节点的结束地址也是代码基本块的结束地址。

第一次扫描遍历所有确定的程序执行路径,得到代码基本块的起止信息,扫描算法如下。

算法 1

```
输入:程序入口地址
输出:代码基本块的起始、结束地址
begin:
  node. l_cAddr = 程序入口地址;
  loop:
  long exAddr = node. l_cAddr;          //代码区起始地址
  扫描程序代码;
  case 遇到单跳转:
  { node. l_rAddr = null; node. b_isingleJmp = true;
    if(跳转到的代码基本块已经被扫描过)
    { node. b_isLeave = true;
      node. b_iscan = true; break; }
    生长左支树;
    if(为间接转移)                      //设置间接调用标志
    { node. b_isLeave = true;           //将间接调用暂时封存
      node. b_islongJmp = true;  goto scan;  }
    else
    获取 node 相关信息;  goto loop;
  }
  case 遇到双跳转:
  { if(按照 true 跳转的代码基本块已经被扫描过)
    { 生长左支树; node. b_isLeave = true;
      node. b_iscan = true; }
    if(按照 false 跳转的代码基本块已经被扫描过)
    { 生长右子树; node. b_isLeave = true;
      node. b_iscan = true;  }
    if (双跳转的目的代码基本块均被扫描过)
      goto scan;
    if( n_leftChild  -> node. b_iscan = false)
    //将右子树暂时封存
    { n_rightChild. node. b_isLeave = true;
      获取 node 相关信息;  }
    else 生长右子树;  goto loop;
  }
  case 遇到结束或返回指令:
  {
    获取 node 相关信息; goto scan;
  }
  scan:
  遍历二叉树叶子节点;
  if(遍历结束)  goto end;
  if( node. l_eAddr = null&& node. b_iscan = false)
  { node. b_isLeave = false;                    //生长叶子节点
    goto loop;  }
  else goto scan;
end:  退出;
```

其中:对二叉树的扫描采用中序遍历的方法[4]130。此外,当要反解的文件规模较大时,为提高遍历二叉树的效率,可以采用构造穿线二叉树的方法进行处理[4]137。

3　定位间接转移地址

经过第一遍扫描之后,所有的间接转移模块都在二叉树的叶子节点上,通过遍历二叉树的叶子节点就可以得到所有的间接转移代码模块。

本节的目的在于查找计算间接转移指令的目的地址,获取二叉树中存放间接跳转指令的叶子节点的相关信息,以叶子节点为父节点继续生长二叉树。

引理 2　间接转移指令的寻址方式分为两种,一种方式通过寄存器进行寻址,另一种方式通过存储器进行寻址。

为了对间接转移指令的转移地址进行准确定位,根据引理 2,构造一维结构数组:

```
struct indicateJmp
{
  long l_cAddr;                    //间接转移指令地址
  long l_regValue;                 //寄存器寻址的目的地址值
  long l_memValue;                 //存储器寻址的目的地址值
  long l_isReg;                    //用于寻址的寄存器变量
  long l_isMem;                    //用于寻址的存储器变量
  bool b_isReg;                    //地址选择方式标志
  bool b_useReg                    //存储器借助寄存器寻址标志
} indJmp;
```

一维结构数组 indJmp 通过扫描二叉树的间接转移地址而不断生长。其中,indJmp. l_regValue 或 indJmp. l_memValue 中存放要寻找的间接跳转指令的目标地址值。

寻址过程中首先遍历二叉树,将所有的间接跳转指令依次添加到结构数组 indJmp[ ]中,根据间接跳转指令的寻址方式在二叉树中逆向追踪相关变量的处理流程,最后求取 indJmp. l_regValue 或 indJmp. l_memValue 的数值,如下所示。

算法 2

```
输入:间接转移指令的跳转地址
输出:间接转移代码基本块的起始
loop:
```

```
扫描二叉树叶子节点;
if(不存在远跳转节点)
  goto end;
if(不是间接转移模块)
  goto loop;
将叶子节点信息添加到 indJmp[]中;
node.b_isLeave = false;                    //取消叶子标志
if(!b_isReg)                               //存储器寻址
{ if(为直接寻址方式)
  { 将该数值写入 indJmp.l_memValue;
    goto loop; }
  else(借助寄存器进行寻址)
  { indJmp.b_isReg = true;
    indJmp.b_useReg = true; }
}
if((b_isReg)&&(!b_useReg))                 //寄存器寻址
{ 将数值写入 indJmp.l_regvalue 中;
  goto loop;
}
//利用寄存器进行的存储器寻址方式
else if((b_isReg)&&(b_useReg))
{
  if(在代码基本块内能够直接获取寄存器的值)
  { 计算目的代码的起始地址 l_jmpAddr;
    l_memValue = l_jmpAddr; goto loop; }
  else
  { 根据二叉树节点信息逆向扫描程序代码;
    追踪寻址寄存器的相关信息;
    获取到寻址寄存器的数值;
    计算目的代码的起始地址 l_jmpAddr;
    l_memValue = l_jmpAddr; goto loop;}
}
end: 退出;
```

在得到所有间接转移代码基本块的起始地址后,通过indJmp.l_cAddr继续生长二叉树,并以新的起始地址为输入重复算法1。在生长二叉树的过程中,对新出现的间接转移指令,采用算法2进行交叉处理,直到再没有新的跳转地址出现为止。

此外,在扫描二进制代码的过程中,可能会出现访问跳转表的情况[5]。对于这种情况,可以将访问跳转表的指令看作是单跳转指令,然后生长二叉树的右子树,左子树置空,这样在对二叉树进行扫描时,就可以轻松地识别出跳转表的相关信息。

至此,代码区分析完毕,生成的二叉树保存了完整的代码区的信息。

4 数据区边界校验

定义2 数据基本块:在汇编代码中,把按照数据操作对象格式的要求(十六进制字节、字、双字、四字或十进制的数字、ASCII码等)进行表示的、连续的数据存储区称为数据基本块。

定义二维数组l_DataAddr[][2]对数据区的相关信息进行存储,其中l_DataAddr[][0]中存放数据基本块可能的起始地址,l_DataAddr[][1]中存放数据基本块的结束地址。

遍历二叉树,取出所有的node.l_nAddr值(node.l_nAddr中保存了数据基本块所有可能的起始地址),按由低到高的顺序,插入到二维数组的l_DataAddr[i][0]项中,并把l_DataAddr[i][1]项初始化为null (i=0,1,2,…,n。n表示所有可能的数据基本块的个数);同时在遍历过程中取出所有非根节点的node.l_cAddr值(代码基本块的起始地址),按由低到高的顺序,插入到到一维数组l_CodeAddr[]中;最后通过二者的比对,确定数据基本块边界数值,其中数据基本块的结束地址存放在l_DataAddr[i][1]中,算法如下。

算法3

```
输入: 数据基本块所有可能的起始地址
输出: 数据基本块的起始、结束地址
for(int i=0; i<l_CodeAddr[ ].GetSize(); i++)
{
  static int m=0;
  for(int j=m; j<l_DataAddr[][0].Getsize(); j++)
  {
    if(l_DataAddr[j][0] >l_CodeAddr[i])
    continue;
    //该模块是代码基本块,删除
    if(l_DataAddr[j][0] =l_CodeAddr[i])
    {
      l_DataAddr.RemoveAt(j);
      l_DataAddr[j][1] =l_CodeAddr[i];
      break;  }
    //该模块是数据基本块,获取结束地址
    else if(l_DataAddr[j][0] <l_CodeAddr[i])
    {
      l_DataAddr[j][1] =l_CodeAddr[i] -1;
      m++; break; } }
}
```

5 代码反汇编

最后一遍扫描主要完成代码基本块和数据基本块的反汇编工作。

通过遍历二叉树就可以得到所有代码基本块的起始、结束地址,在反汇编的过程中只需要将机器码直译成指令代码,原理简单,不再赘述。

引理3 任何一个可执行的程序,无论结构多么复杂,程序的操作对象与数据块之间总是存在一一对应的关系。

在对数据基本块进行反汇编的过程中存在数据类型的判断问题。根据引理3,有的反汇编软件在扫描的过程中记录程序中访问到的所有数据的类型、存储地址等信息,然后在数据基本模块中进行反汇编,工作量比较大。

6 结语

本文在对数据基本块反汇编的过程中,首先通过线性跟踪二维数组l_DataAddr[][],将数据基本块统一反汇编为byte(根据不同的反解对象也可以设为word、dword等类型)型数据,然后遍历代码基本块,对程序中访问到的非byte型数据重新进行类型判断,可以看出,只要默认数据类型选择恰当,可以减轻相当一部分工作负担。

参考文献:

[1] 李学汇. 自动反汇编程序的一种解决方案[J]. 微型机与应用, 1997,16(10):7-9.

[2] 谷伟, 侯成君. MCS-51智能反汇编软件的设计与实现[J]. 微电子学与计算机, 1993,10(10):23-26.

[3] LINN C, DEBRAY S. Obfuscation of executable code to improve resistance to static disassembly[C]// CCS: Proceedings of the 10th ACM Conference on Computer and Communications Security. [S.l.]: ACM Press, 2003. 290-299.

[4] 徐士良, 葛兵. 实用数据结构[M]. 北京: 清华大学出版社, 2006:11.

[5] 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.