第 33 卷 第 6 期 计 算 机 工 程 2007 年 3 月
Vol.33 No.6 Computer Engineering March 2007
·软件技术与数据库· 文章编号：1000—3428(2007)06—0089—03 文献标识码：A 中图分类号：TP314

基于加载机制分析的ELF文件main函数定位技术

齐 宁，丁松阳，孙维新，赵荣彩
（解放军信息工程大学信息工程学院，郑州 450002）

摘 要：当前二进制翻译中通用的 main 函数定位方法依赖于符号表，随着 strip 工具的普遍应用，二进制可执行文件中往往不存在符号表。该文描述了 strip 工具的应用目的，分析了其应用对二进制翻译的影响，基于 ELF 文件加载机制的分析，提出了一种新的 main 函数定位技术，通过对 IA-32 及 IA-64 下 ELF 格式二进制文件的翻译，证明该技术是有效的。

关键词：二进制翻译；符号表；ELF；Strip

Addressing Technology of ELF File's main Entry Point Based on Loading Mechanism Analysis

QI Ning, DING Songyang, SUN Weixin, ZHAO Rongcai
(School of Information Engineering, PLA Information Engineering University, Zhengzhou 450002)

【Abstract】 Nowadays, the common method of addressing the entry point of main relies on symbol table. With the widely use of strip utility, the binary file does not contain symbol table anymore. The paper describes the purpose of strip utility first, then analyzes the effects of using strip on binary translation, basing on the analysis of ELF loading mechanism, puts forward a new technique to addressing main entry point. The technique is proved to be effective by the translation of ELF binary files in IA-32 and IA-64.

【Key words】 Binary translation; Symbol table; ELF; Strip

二进制翻译过程的首要任务就是确定被翻译的二进制可执行文件的入口点，通常为main函数的地址。在当前较流行的二进制翻译工具中，采用的方法是通过查找符号表来确定main函数的地址。strip是GNU二进制工具，用来去除对象文件中所有的符号信息[1]。由于应用strip工具后的二进制可执行文件不再有符号表，因此无法通过查找符号表的方法来确定main函数的地址。

为了解决这一问题，本文在对 ELF 文件加载机制的研究和分析基础上，提出了一种新的 main 函数定位技术，应用该技术可以确定应用 strip 工具后的二进制可执行文件的 main 函数地址。通过对 IA-32 及 IA-64 下 ELF 格式二进制文件的翻译，证明该技术是有效的。

1 相关知识简介

1.1 二进制翻译

二进制翻译是一种编译技术，它与传统编译器的差别在于其编译处理对象不同。二进制翻译处理的是某种机器的目标二进制代码，该目标代码是经过传统编译器生成的，经过二进制翻译处理后生成另一种机器的目标二进制代码，传统编译器处理的是某一种高级语言，经过编译处理生成某种机器的目标代码。按照传统编译程序前端、中端和后端的划分，二进制翻译可以理解为拥有特殊前端的编译器[2]。

1.2 二进制工具 strip

二进制工具 strip 用来去除对象文件中所有的符号信息。应用 strip 工具的目的可能是为了减小可执行文件的大小，以 spec2000 测试集中的 bzip2 为例，在 IA-64 下编译得到的 bzip2 可执行文件大小为 126.9KB，应用 strip 工具处理后大小为 77.1KB。应用 strip 工具的另外一个目的可能是为了增加程序的安全性，因为去除了可执行文件中的符号信息，所以对程序进行逆向工程分析时可利用的信息较少，增加了分析的难度。目前软件的发布版本往往是已经进行了 strip 处理。

1.3 ELF 文件格式

ELF(Executable and Linking Format)是由Unix系统实验室(USL)开发和发布的二进制格式。它是SVR4 及Solaris 2.x 可执行文件使用的默认二进制格式。ELF同a.out以及COFF二进制格式相比更加强大而且更加复杂。同适当的工具联合使用，程序员可以通过ELF在运行时控制执行的流程[3]。

本文所研究的内容针对 IA-32 及 IA-64 体系结构下的 ELF 文件格式，具体格式规范参见文献[4, 5]。

1.4 应用 strip 工具对二进制翻译的影响

二进制翻译的第 1 步是对源二进制可执行文件进行解码。解码时采用的标准方法是从入口点开始，沿着所有可到达的路径进行解码，其入口点的定位方法一般为在符号表中寻找符号 main 所对应的地址，从该地址开始进行解码。一旦对二进制可执行文件应用 strip 工具，则无法在符号表中寻找到入口点的地址，解码过程无法正常开始，翻译过程会异常终止。

另外，即使假定可以进行翻译，由于用户自定义的过程名会被 strip 工具去除，因此一些有意义的过程名，如 sub、add，也无法恢复出来。

虽然 main 函数的地址无法得到，但仍可以得到.text 节（section）的地址，那么是否可以从.text 节开始进行解码和

基金项目：国防重点科研基金资助项目
作者简介：齐 宁(1978－)，男，博士生，主研方向：逆向工程，先进编译技术；丁松阳，博士生；孙维新，硕士生；赵荣彩，教授、博导
收稿日期：2006-03-31 **E-mail：**qi_ning@126.com

翻译？答案是否定的。因为在.text 节开始的内容通常是_start 的一个例程，其内容同体系结构紧密相关，且涉及参数设置及对__libc_start_main 库函数的调用。所以对该例程进行翻译，一方面降低了二进制翻译整体的效率，另外也增加了二进制翻译的难度。笔者对 ELF 加载机制进行了分析，目的是寻找一种方法，能够从应用 strip 工具后的二进制代码出发，得到 main 函数的地址。

2 ELF 加载机制分析

ELF 的可执行文件与共享库在结构上非常类似，它们具有一张程序段表，用来描述这些段如何映射到进程空间。对于可执行文件而言，段的加载位置是固定的，程序段表中如实反映了段的加载地址。对于共享库而言，段的加载位置是浮动的、位置无关的，程序段表反映的是以 0 作为基准地址的相对加载地址。尽管共享库的连接是不充分的，为了便于测试动态链接器，Linux 允许直接加载共享库运行。如果应用程序具有动态链接器的描述段，内核在完成程序段加载后，紧接着加载动态链接器，并且启动动态链接器的入口。如果没有动态链接器的描述段，就直接交给用户程序入口。

在控制权交给动态链接器的入口后，首先调用_dl_start 函数获得真实的程序入口。该入口地址不是main的地址，也就是说一般程序的入口不是main，然后循环调用每个共享对象的初始化函数，接着跳转到真实的程序入口，一般为_start（程序中的_start）的一个例程，该例程压入一些参数到堆栈，就调用__libc_start_main函数。在__libc_start_main函数中替动态连接器和自身程序安排析构器，并运行程序的初始化函数，最后才把控制权交给main函数[6]。

3 定位 main 函数地址的新思路

基于 ELF 加载机制的分析，可以看到在 main 函数执行之前，经过了一系列的函数调用，最终由函数__libc_start_main 调用了 main 函数，那么函数__libc_start_main 必然有着 main 函数的地址信息。实际情况也是如此，以下为函数__libc_start_main 的相关定义：

```
int BP_SYM(__libc_start_main)(
int(*main)(int,char**,char**),
int argc,
char*__unbounded*__unbounded ubp_av,
void(*init)(void),
void(*fini)(void),
void(*rtld_fini)(void),
void*__ unbounded stack_end)
```

而__libc_start_main 是在_start 中调用的，在_start 中对__libc_start_main 的参数进行了设置。GLIBC 中对 ELF 文件在各种体系结构下的_start 例程均有清楚明确的定义。例如，对于 IA-32，_start 例程的相关内容为：

```
_start:
  xorl%ebp,%ebp
  popl%esi
  movl%esp, %ecx
  andl$0xfffffff0, %esp
  pushl%eax
  pushl%esp
  pushl%edx
  pushl$_fini
  pushl$_init
  pushl%ecx
  pushl%esi
  pushl $BP_SYM(main)
  call BP_SYM(__libc_start_main)
  hlt
```

对于 IA-64，__libc_start_main 参数设置约定为：

```
* Arguments for __libc_start_main:
*       out0:  main
*       out1:  argc
*       out2:  argv
*       out3:  init
*       out4:  fini
*       out5:  rtld_fini
*       out6:  stack_end
```

在调用__libc_start_main 之前同 out0 相关的语句为：

```
addl out0 = @ltoff(@fptr(main)), gp
ld8 out0 = [out0]
```

基于对加载机制的分析以及对 GLIBC 源码的阅读，提出以下的 main 函数定位思路：在基于符号表查找 main 函数地址失败的情况下，从.text 节起始位置（即_start 例程）开始寻找，按照特定体系结构下_start 例程的规范来确定函数__libc_start_main 同 main 函数地址相关的参数，依据该参数获取 main 函数地址。

4 在 IA-32 及 IA-64 下的具体实现

4.1 IA-32 下的实现

IA-32 下_start 例程见第 3 节 IA-32 下的_start 例程的相关内容。通过读取某 IA-32 二进制可执行文件的符号表，得到如下内容：

```
080483f0 g    F .text 00000055    main
```

即该二进制可执行文件的 main 函数地址为 0x80483f0。

进行 strip 处理后，符号表为空，经 dump 后得到的.text 节中的代码如下：

```
08048300<.text>:
8048300:xor    %ebp,%ebp
8048302:pop    %esi
8048303:mov    %esp, %ecx
8048305:and    $0xfffffff0, %esp
8048308:push   %eax
8048309:push   %esp
804830a:push   %edx
804830b:push   $0x8048494
8048310:push   $0x8048298
8048315:push   %ecx
8048316:push   %esi
8048317:push   $0x80483f0
804831c:call   0x80482e0
8048321:hlt
```

笔者采用的寻找方式是使用模板匹配的方法，当然，只要能够定位到地址 0x8048317 处的 push 指令并提取其操作数，其他方法亦可。

对照经 dump 后得到的.text 节中的代码及第 3 节 IA-32 下的_start 例程相关内容，可以判定 main 函数的地址为 0x80483f0，同读取符号表所得到的结果一致。

4.2 IA-64 下的实现

IA-64 下__libc_start_main 的参数约定如第 3 节 IA-64，__libc_start_main 参数设置约定。通过读取某 IA-64 二进制可执行文件的符号表，得到如下内容：

```
4000000000000640 .text    main
```

即该二进制可执行文件的 main 函数地址为 0x4000000000000640。

经过 strip 处理后，符号表为空，经 dump 后得到.text 节中代码如下(IA-64 下的_start 代码实例)：

```
4000000000000440<.text>:
4000000000000440: alloc r2=ar.pfs,7,0,0
4000000000000446: movl r3=0x9804c0270033f
4000000000000450: adds r34=16,r12
4000000000000456: movl r1=0xdffffffffffff6b0;;
4000000000000460: ld8 r33=[r34],8
4000000000000466: mov.m r10=ar.bsp
400000000000046c: mov r9=ip;;
4000000000000470: mov.m ar.fpsr=r3
4000000000000476: sub r1=r9,r1
400000000000047c: adds r38=16,r12;;
4000000000000480: addl r11=48,r1
4000000000000486: addl r32=40,r1
400000000000048c: addl r35=32,r1;;
4000000000000490: ld8 r3=[r11]
4000000000000496: ld8r32=[r32]
400000000000049c: addl r36=24,r1;;
40000000000004a0: ld8 r35=[r35]
40000000000004a6: ld8 r36=[r36]
40000000000004b0: st8[r3]=r10
40000000000004b6: mov r37=r8
40000000000004bc: br.call.sptk.few
                      b0=0x4000000000000420
40000000000004c0: break.m 0x0
```

依据 IA-64 调用约定，out0 为 r32 通用寄存器，因此第 3 节__libc_start_main 之前同 out0 相关的语句中的同 out0 相关的语句为 0x4000000000000486 和 0x4000000000000496 处的语句。笔者采用的方法是按照 IA-64 的指令束格式对从.text 节开始的指令流依次进行读取，比较并定位到如下格式语句：

```
addl r32 = imm22, r1
```

对于 IA-64 下的_start 代码实例，得到的 imm22 即为 40。读取 GOT 表的首地址(即为 IA-64 下的_start 代码实例的 r1 的值)，并读取 GOT 表中 imm22 为偏移处的 8B 的值，该值为指向 main 的函数描述符的指针。上述过程相当于识别并实现第 3 节 IA-64 下同 out0 相关语句中的两条语句。该二进制文件 GOT 表内容为：

```
6000000000000db0 <.got>:
  ...
6000000000000dc8: 50 0a 00 00 00 00
6000000000000dce: 00 40 60 0a
6000000000000dd2: 00 00 00 00 00 40
6000000000000dd8: 70 0a 00 00 00 00
6000000000000dde: 00 40 30 0e
6000000000000de2: 00 00 00 00 00 60
```

故上述计算得到的结果为 0x4000000000000a70，该值为指向 main 函数的函数描述符的指针。观察该地址所处的节及内容：

```
4000000000000a50 <.opd>:
4000000000000a50: 60 08 00 00 00 00
4000000000000a56: 00 40 b0 0d 00 00
4000000000000a5c: 00 00 00 60
4000000000000a60: 90 07 00 00 00 00
4000000000000a66: 00 40 b0 0d 00 00
4000000000000a6c: 00 00 00 60
4000000000000a70: 40 06 00 00 00 00
4000000000000a76: 00 40 b0 0d 00 00
4000000000000a7c: 00 00 00 60
```

opd 是 official procedure descriptors 的简写。指向函数的指针在本节中对应一个过程描述符。该描述符中的前 8 个字节的内容为函数指针所指向的函数的地址，本例中 0x4000000000000a70 处开始的描述符中的地址为 0x4000000000000640，即为 main 函数的入口地址。

5 结束语

随着 strip 工具的广泛应用，基于符号表的 main 函数定位方法已经无法满足二进制翻译及其他逆向工程技术的需要。基于对 ELF 文件加载机制的分析，提出了一种新的 main 函数定位技术，该技术基于对__libc_start_main 函数的参数分析及 ELF 文件在各体系结构下的具体_start 例程分析，这些信息是 strip 工具所不能去除的、更加基本的信息。该技术已经在某二进制翻译框架上进行了应用，证明对 IA-32 及 IA-64 下的 ELF 文件是适用的。下一步工作拟将本文提出的思路和技术拓展到其他格式文件及其他体系结构下进行检测。

参考文献

1 Wall K, Watson M, Whitis M, et al. GUN/Linux 编程指南[M]. 王 勇, 王一川, 林花军, 等, 译. 北京: 清华大学出版社, 2000.

2 马湘宁. 二进制翻译关键技术研究[D]. 北京: 中科院计算所, 2004.

3 Lu Hongjiu. ELF: From the Programmer’s Perspective[R]. NYNEX Science & Technology, Inc., 1995.

4 Alert7. Before main()分析[DB/OL]. 2001. http://www.xfocus.net/articles/200109/269.html.

5 Glibc-2.3.2[DB/OL]. 2004. ftp://ftp.gnu.org/gnu/glibc/.

(上接第 88 页)

从表 1 可以得到应用谓词消除算法可以取得相当好的效果，与自然的方法相比，平均减少基本块个数 5.1%，控制流边减少 3.2%，指令条数减少 2.6%。

6 结论

判定执行技术导致对目标代码的底层结构的深度重构，使得应用传统逆向工程技术很难恢复原程序的逻辑。本文提出的算法可以消除程序变换的效果，使得识别原程序结构的工作简化，测试数据说明谓词消除算法有效的缩减了对优化的 IA-64 可执行代码翻译中得到的控制流图的大小和复杂性，提高逆向工程的质量。

参考文献

1 Byrne E J. Software Reverse Engineering: A Case Study[J]. Software——Practice and Experience, 1991, 21(12): 1349-1364.

2 Johnson R, Schlansker M. Analysis Techniques for Predicated Code[C]//Proc. of the 29th Annual International Symposium on Microarchitecture. 1996: 100–113.

3 Snavely N, Debray S K, Andrews G R. Predicate Analysis and If-conversion in an Itanium Link-time Optimizer[C]//Proc. of Second Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology. 2002-11.