静态二进制翻译中动态地址解析恢复技术研究

丁松阳[1,2],赵荣彩[1],崔平非[1]
DING Song-yang[1,2],ZHAO Rong-cai[1],CUI Ping-fei[1]

1.解放军信息工程学院 计算机科学与技术系,郑州 450002
2.河南财经学院 计算机科学系,郑州 450002
1.Dept. of Computer Science and Technology,Information Engineering College of PLA,Zhengzhou 450002,China
2.Dept. of Computer Science and Technology,Henan University of Finance and Economics,Zhengzhou 450002,China
E-mail:dingsongyang@yahoo.com.cn

DING Song-yang,ZHAO Rong-cai,CUI Ping-fei.Recovery of dynamic address resolving for static binary translation. Computer Engineering and Applications,2008,44(22):59-61.

Abstract: The entry pointer of static binary translation generally is the function of main,so the dynamic address resolving before program loading can't be recovered.Based on the analysis of ELF dynamic address resolving mechanism,this paper puts forward a method to simulate dynamic address resolving by inserting codes into the target program.This method has been implemented in the project of static binary translation,experimental results demonstrate that it is efficient.

Key words: static binary translation;ELF file;dynamic address resolving

摘 要:静态二进制翻译的入口点通常为 main 函数,所以 main 函数执行之前的动态地址解析部分就无法在目标机上恢复。通过分析基于 ELF 文件的动态地址解析机制,提出了在目标代码中插入代码模拟动态地址解析的方法来解决该问题。该方法已在静态二进制翻译项目中实现,测试结果表明该方法是有效的。

关键词:静态二进制翻译;ELF 文件;动态地址解析
DOI:10.3778/j.issn.1002-8331.2008.22.017 **文章编号**:1002-8331(2008)22-0059-03 **文献标识码**:A **中图分类号**:TP311

1 引言

二进制翻译技术是一种在某种体系结构上运行另一种体系结构代码的技术[1,2],它将一种体系结构的二进制指令代码翻译成另一种体系结构的指令。国际上对二进制翻译技术的需求背景主要分如下两种情况:第一种是将代码从遗产平台向新平台移植,代表性的系统主要有 HP 公司的一个商用二进制翻译系统(将 HP 3000 的客户转移到新的 PA-RISC 结构)、Tandem 公司的 Accelerator(将 TNS/CISC 代码移植到 TNS/RISC)以及 Digital 公司的 VEST(将 VAX 代码移植到 Alpha)。第二种情况是为了运行竞争对手的二进制代码, 例如 AT&T 公司的 Flashport(将 Mac 68k 的代码翻译到 PowerPC)以及 Digital 公司的 FX! 32(将 x86/WinNT 代码翻译到 Alpha/WinNT)。目前二进制翻译的研究主要包括如 IBM 公司的 Daisy、Transmeta 公司的 code morphing、HP 公司的 Aries 等等。以上翻译器均为定源定目标的翻译器,同机器特性高度相关。1999 年 Queensland 开发的 UQBT[3]是一个可变源、可变目标的静态二进制翻译框架,UQBT 将机器无关的部分分离出来,用描述语言如 SLED、SSL、PAL 来支持机器相关的部分,从而提高了框架的可重用性。

按照实现机制的不同,二进制翻译技术可以分为动态翻译和静态翻译两种方式。动态翻译器对输入的源二进制文件做动态分析,运行时翻译成目标机器码,程序运行阶段没有执行到的程序段不会被翻译。动态翻译可以获取程序运行信息,有利于运行优化。静态二进制翻译在翻译过程中并不执行被翻译的程序,它是从程序的入口点开始,沿着所有可到达路径进行翻译。静态二进制翻译的优点是产生的代码执行效率比较高。但其主要的缺点是难以处理间接跳转、间接过程调用,无法处理自修改代码。

笔者参与研发的是 IA-64 到 Alpha 的静态二进制翻译系统。IA-64 体系结构是 Intel 与 HP 联合开发的新一代体系结构,其特点在于使用 64 位指令集体系(ISA),该指令集应用了 EPIC(显式并行指令计算)技术,代表了当今高性能计算机的发展方向之一。

本文首先介绍了静态二进制翻译框架等相关知识,接着分析了 ELF 文件的动态地址解析过程, 最后提出了在静态二进制翻译中解决动态地址解析问题的方案。该方案的核心思想是在目标代码中插入相关代码来模拟源端的动态地址解析。

实验研究的源二进制程序操作系统平台为 Red Hat Linux AS3 for IA-64,编译器使用 GCC 3.2.3。

基金项目:国家高技术研究发展计划(863)(the National High-Tech Research and Development Plan of China under Grant No.2006AA01Z408)。
作者简介:丁松阳(1971-),男,博士研究生,研究方向:软件逆向工程;赵荣彩,男,教授,博士生导师,研究方向:先进编译技术、软件逆向工程;崔平非(1975-),男,硕士研究生,主要研究方向:软件逆向工程。
收稿日期:2007-10-10 **修回日期**:2008-01-21

2 相关背景知识

2.1 使用 C 后端的静态二进制翻译框架

早期的二进制翻译器如 Digital 公司的 VEST/mx 直接将一条源 VAX 指令译为零个或多个目标机 Alpha AXP 指令。这种结构主要适用于机器指令较少且源寄存器个数少于目标寄存器个数的情况。

目前较为通用的静态二进制翻译结构如图 1 所示。前端是机器相关模块,主要包括3 个部分:二进制文件装入,反汇编,产生中间代码。在中间是机器与操作系统无关模块,主要进行代码分析,如代码优化等操作。后端也是机器相关模块,产生目标机可执行代码。

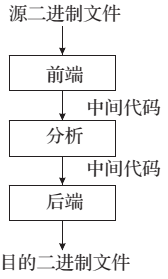

图 1 静态二进制翻译结构

在笔者参与的二进制翻译项目中,使用了图 1 所示结构的一种变化形式。其特点是对中间代码进行分析,将其提升为 C 语言,称之为使用 C 后端的静态二进制翻译框架。在该系统中,使用了两种中间表示,机器相关的寄存器传输列表(RTLs)和机器无关的高层寄存器传输语言(HRTL)[3]。在前端使用二进制文件解码器,从输入的源可执行文件中读入二进制指令流,并将指令流解码成源机器指令序列。语义映射器基于源机器的语义说明将每条源机器指令翻译成第一级中间表示寄存器传输列表(RTLs)。寄存器传输列表(RTLs)到高层寄存器传输语言(HRTL)的翻译器应用控制转换信息、过程约定以及其他的信息对前面所产生的寄存器传输列表(RTLs)流进行分析,去除同机器相关的部分,得到第二级中间表示高层寄存器传输语言(HRTL)。在此基础上即可进行机器无关的分析和转换,在后端将 HRTL 转换为低级 C。接着通过目标机器上的 C 语言编译器对低级 C 程序进行编译,并将编译结果与源程序中的数据段链接,最后得到目标机上可执行的二进制文件。

2.2 ELF 文件的重定位段

ELF 文件在链接时要进行符号、段及指令的重定位,在被装入时还需要对动态链接库函数、标准输入、标准输出及标准错误等在运行才能确定的地址进行重定位。库函数的地址解析分为两种情况:当程序中直接调用库函数,在编译时选取动态链接时,库函数的重定位采取的是在全局偏移表 PLT 段中插入桩子(stub)的方式[4]。文献[5]讨论了它的恢复算法;当间接调用动态链接库函数时,ELF 文件使用重定位段解析只有在运行时才能确定的库函数地址。针对此种情况,也提出并实现了间接调用动态链接库函数恢复方法。该方法使用伪地址来代替需要动态解析的动态链接库函数的地址,在后端构建了伪地址与目标机上动态链接库函数地址的对应表,并在后端 C 代码生成过程中插入根据伪地址获得动态链接库函数地址的查表代码,使用查表得到的库函数地址恢复对于动态链接库函数的间接调用。其具体算法将在另一篇论文中进行论述。本文提出的算法则是为了解决标准输入、标准输出及标准错误等的地址重定位问题。

ELF 文件在装入时,动态链接器从 DYNAMIC 段的内容获取动态链接需要的信息。如果程序需要进行重定位,则由 DYNAMIC 段指出重定位表 Elf64_Rela 的入口地址及其大小[6]。重定位表的结构如图 2 所示。

```
typedef struct elf64_rela {
  Elf64_Addr r_offset;      /* Location at which to apply the action */
  Elf64_Xword r_info;       /* index and type of relocation */
  Elf64_Sxword r_addend;    /* Constant addend used to compute value */
} Elf64_Rela;
```

图 2 重定位表 Elf64_Rela 结构

结构中域r_offset 指示需要进行重定向的位置。对于可执行文件来说,它是被重定位存储单元的虚地址。域 r_info 分为两部分,一部分表示需要被重定位的符号在符号表中的索引,另一部分表示的是重定位类型。域 r_addend 是一个常量,用来计算存储在重定位域中的值。

2.3 二进制翻译中的动态地址解析问题

本文研究的问题产生根源在于动态地址解析的目标地址存在于源程序的读写数据段中,且目标地址是在 main()函数执行之前的程序装入过程中通过重定位机制得到并写入读写数据段中,而静态二进制翻译通常从 main()函数入手根据程序流进行翻译,这样源程序的动态地址解析就没有在目标机上重现。所以在目标机上如果不针对目标地址进行处理,只是简单的将翻译后的代码与源程序数据进行链接,那么需要重定位的源程序读写数据中的目标地址就得不到重写。在目标机上运行翻译后的程序时,得到的是错误地址,程序就会发生段错而中止运行。需要强调的是,使用 main()函数作为翻译入口的静态二进制翻译系统要特别注意此类问题:凡是在 main()函数执行之前对数据段进行了写操作,在目标机上就要将此操作恢复。动态地址解析是此类问题中的一个典型问题。

3 动态地址解析分析

3.1 用例

在 unix 系统中,标准输入、标准输出和标准错误地址并不能在编译时确定,所以 ELF 文件就需要一种机制来动态解析它们的地址。

为了分析 ELF 文件解析标准输入、标准输出和标准错误地址的过程,使用了图 3 所示的 C 程序。该程序的功能是接收从键盘输入的 4 个浮点数,并对它们进行累加。接收用户输入使用的是库函数 fgets(),在此例中它的第三个参数是标准输入 stdin,用来接收来自键盘的输入。由于 stdin 的地址在编译时并不能确定,所以只能使用动态地址解析的方法,在程序装入时解析 stdin 的地址。

在分析该问题的过程中,采取了由易到难,由特殊到一般的方法。针对该用例,首先使用的编译器是 GCC 3.2.3,优化级别为零级,使用动态链接选项。

3.2 地址解析过程分析

标准输入 stdin 的地址只有在程序装入时才能确定,为了分析 stdin 地址重定位的过程,对图 3 所示用例编译后的可执行文件使用 objdump 命令进行了反汇编。

```
main( )
{
    char stringValue[15];
    double sum=0.0;
    int i,j;

    for (i=1;i≤4;i++) {
        printf("Enter a floating point string: ");
        fgets(stringValue,15,stdin);
        sum += atof(stringValue);
    }
    printf("\nThe total of the values is %f\n",sum);
    return 0;
}
```

图 3 使用标准输入 stdin 的例子

图 4 是调用库函数 fgets( )前的反汇编片断。其中指令"br.call.sptk.many b0=40000000000004e0"调用的就是库函数 fgets( )。库函数 fgets( )有 3 个参数,它们分别表示内存空间指针、允许读入的最大字符数及流文件指针。在本例中,fgets( )的 3 个参数分别通过寄存器 r38,r39,r40 来传递。其中寄存器 r40 正是标准输入文件 stdin 的地址。向后回溯分析可以发现:r40 的值由 r14 指向的地址载入,r14 的值又是由 r15 指向的地址载入,r15 的值等于寄存器 r1 的值加上立即数 24。在 IA64 体系结构中,寄存器 r1 被称为全局指针(GP),它指向全局偏移表(GOT)[7]。由上面的分析可知,在全局偏移表(r1+24)处存放着一个地址,这个地址指向的仍然是一个地址,它就是标准输入文件 stdin 的地址。

```
40000000000007e6:    addl r15=24,r1
40000000000007ec:    nop.i 0x0;;
40000000000007f0:    ld8 r14=[r15]
40000000000007f6:    adds r38=-16,r37
40000000000007fc:    mov r39=15;;
4000000000000800:    ld8 r40=[r14]
4000000000000806:    mov r32=r1
400000000000080c:    br.call.sptk.many b0=40000000000004e0
```

图 4 程序反汇编指令片断

图 5 所示是程序的全局偏移表。寄存器 r15=r1+24 的值就是全局偏移表的首地址 0x6000000000000f60 加上偏移 0x18,得到的地址是 0x6000000000000f78,从图 5 中可以看出,该地址指向的 8 个字节全部为 0。这是因为标准输入 stdin 的地址是在程序装入时动态解析得到的。在程序装入过程中,动态解析得到指向存储 stdin 地址的指针,这个指针会被写入 GOT 表中,在本例中,这个指针被写在(r1+24)开始的连续 8 个字节中。程序在需要 stdin 地址时,首先从 GOT 表中取得指向存储 stdin 地址的指针(ld8 r14=[r15]),然后再根据这个指针读出 stdin 的地址(ld8 r40=[r14])。

```
6000000000000f60 00000000 00000000 00000000 00000000
6000000000000f70 00000000 00000000 00000000 00000000
6000000000000f80 000c0000 00000040 100c0000 00000040
6000000000000f90 200c0000 00000040 10100000 00000060
6000000000000fa0 00000000 00000000 00000000 00000000
6000000000000fb0 c00b0000 00000040 e00b0000 00000040
```

图 5 全局偏移表

程序在装入过程时需要知道重定位段中的信息。从重定位段中可获取需动态解析的符号、类型及重定位的位置。如图 6 是程序的重定位段 rela.dyn。

```
4000000000000330 780f0000 00000060 27000000 01000000
4000000000000340 00000000 00000000
```

图 6 重定位段 rela.dyn

图 6 中的地址 0x6000000000000f78 是重定位的位置,指向图 5 所示的全局偏移表中。0x27 表示的是重定位的类型,0x01 表示的是重定位符号在符号表中的偏移值。由该偏移值可找到符号表中该符号在字符串表中的偏移值,再根据这个偏移值从字符串表得到需要动态解析地址的符号。对于本例来说,这个符号就是 stdin。

4 解决方案

要解决动态地址解析的翻译问题,关键点是如何让目标机中的动态地址与翻译后的程序联系起来。从第 3 章的分析中可以看出,这个联系点就是全局偏移表(GOT)。由此可得到解决问题的基本思路:当程序在目标机上运行时,根据目标机上的动态地址修改全局偏移表(GOT)中相应项的内容,这样目标程序通过全局偏移表(GOT)得到的地址就是目标机上的动态地址。

在具体实现时,根据二进制翻译程序的前端、后端结构,实现过程也分为两个部分。

在前端二进制文件装入后,分析文件的重定位段 rela.dyn,判断是否需要动态地址解析,如 stdin、stdout 及 stderr 等。如果存在需要动态解析的地址,则计算出重定位位置相对于全局偏移表(GOT)首地址的偏移值。

在后端生成 C 代码时,根据计算出的重定位位置相对于全局偏移表(GOT)首地址的偏移值,插入修改全局偏移表(GOT)的代码。

对于如图 3 所示的程序,经过翻译之后,在生成的目标代码中插入修改全局偏移表(GOT)的代码。代码如图 7 所示。

```
FILE *fpstdin;
fpstdin=stdin;
*(unsigned long long*)(r1+24)=(int64)&fpstdin;
```

图 7 目标程序中插入的代码

本例中需要动态解析的地址是标准输入文件 stdin。在插入代码中声明了一个文件型的指针变量 fpstdin,并将 stdin 的值赋给这个指针变量。对于本例来来说,stdin 重定位的位置是全局偏移表(GOT)首地址加上偏移值 24,所以将指针变量 fpstdin 的地址写入该处。上述的三句代码执行后,就完成了对于标准输入文件 stdin 的重定位, 目标程序运行时取得的地址就是目标机上标准输入文件 stdin 的地址。

本文分析使用的是 GCC3.2.3 在零级优化下编译的二进制可执行文件。该算法在项目中实施后,又针对 icc 编译器在一级及二级优化级别下编译的二进制可执行文件进行了翻译,运行结果全部正确。

5 结束语

通过分析 IA-64 体系结构下 ELF 文件对于动态地址解析

(下转 67 页)