基于动态二进制翻译和插桩的函数调用跟踪

卢帅兵 张 明 林哲超 李 虎 况晓辉 赵 刚

(信息系统安全技术国家重点实验室(军事科学院) 北京 100101)

(datadancer@163.com)

Dynamic Binary Translation and Instrumentation Based Function Call Tracing

Lu Shuaibing, Zhang Ming, Lin Zhechao, Li Hu, Kuang Xiaohui, and Zhao Gang

(*National Key Laboratory of Science and Technology on Information System Security* (*Academy of Military Sciences*), *Beijing* 100101)

Abstract Dynamic function call tracing is one of the most important techniques for Linux kernel analysis. Existing tools suffer from the problems of insufficiently supporting instruction set architectures(ISA) and low efficiency. We design and implement a function call tracing tool to support multiple ISAs with high efficiency. Firstly, we use the binary translation system to load the kernel image and recognize the branch instruction types. Secondly, we design different instrumentation code based on different kinds of ISAs and insert instrumentation code during the translation stage to get timestamps, process IDs, thread IDs and function addresses during the kernel booting and runtime. Finally, when the kernel boots up and the shell appears, we process all the information and generate function call maps. Based on binary translation, we analyze the text, symbol and string sections of the binary image, without any source code. Enriched intermediate code and extra target code are compatible with optimization algorithms like block chain, redundant code elimination and hot path optimization, which reduces the performance overhead. The core algorithm is to design the instrumentation code and get corresponding information based on different ISAs. It is easy to implement and to migrate to multiple ISAs. Experiments on QEMU and Linux 4.9 kernel show that the traced information is accordance with the source code while instrumentation code brings about 15.24% and information processing generates 165.59% overhead of original QEMU, which is much faster than existing tools.

Key words dynamic binary translation; instrumentation code; function call tracing; Linux kernel analysis; cross platform

摘 要 动态函数调用跟踪技术是调试Linux内核的重要手段.针对现有动态跟踪工具存在支持平台有限、运行效率低的问题,基于二进制翻译,设计并实现支持多种指令集的动态函数调用跟踪工具.首先,使用二进制翻译进行系统加载、分析内核镜像,识别基本块的分支指令类型.然后,根据不同平台指令集,设计桩代码并在函数调用与返回指令翻译时插入桩指令,进而在程序执行和内核启动时实时获取时间戳、进程标识、线程标识、函数地址等信息.最后,内核加载完毕后,处理获取的信息,生成过程函数调用图.只需要根据平台指令集特点设计对应的信息获取桩代码并插入到函数调用指令翻译代码中,实现简单,易于移植支持多种平台.该方法基于二进制翻译,直接对程序或内核镜像中的指令段、代码段、符号表进行分析,不依赖源码.拓展的中间代码和额外的目标码,不影响基本块连接、冗余代码消除、热路径分析等二进制翻译的优化方法,降低了开销.基于QEMU的实验结果表明:跟踪分析结果与源代码

收稿日期:2017-09-08;**修回日期**:2018-06-05

通信作者:李虎(lihu_nudt@163.com)

行为一致,桩代码执行信息记录产生了15.24%的时间开销,而信息处理并输出到磁盘文件产生了165.59%的时间开销,与现有工具相比,性能有较大提升.

关键词 动态二进制翻译;代码插桩;函数调用跟踪;Linux内核分析;跨平台

中图法分类号 TP391

函数调用分析在软件安全[1]、程序逻辑[2]、漏洞挖掘[3]等领域有着广泛的应用.特别是Linux内核的开发与调试需要处理大量复杂的函数,其调用关系对内核分析与调试有很大帮助[4].函数调用分析包括静态分析与动态分析2种方法:

1) 静态分析.根据源代码进行代码审计,得到从入口函数到退出函数的整个执行路径.静态分析面向特定的编程语言源码,进行词法语法分析,技术成熟.但是间接分支指令、间接函数调用和动态生成代码在静态条件下很难获取执行路径,因此静态调用关系并不能反映出真实的调用序列信息.

2) 动态分析.为了能获取程序执行时真实的函数调用关系,需要在程序运行时记录函数调用信息,动态跟踪函数执行路径.动态分析方法解决了间接分支目标地址的不确定性问题,可得到软件实时调用序列、各函数的运行时间、调用次数、前后依赖等信息.

现有函数调用动态分析工具包括GNU binutils工具集的gprof[5],ftrace,systemtap,dtrace等.上述工具需要特定的使用条件,不能完全满足当前的开发调试需要.gprof仅支持启动用户态可执行程序,不能用于内核分析;ftrace需要编译内核时打开编译选项,造成多种编译优化手段无法进行;systemtap与dtrace需要操作系统运行时提供接口,操作系统启动阶段无法使用.

为弥补上述工具的不足,基于模拟器的函数跟踪技术直接分析二进制镜像,不需额外的编译选项,成为内核分析技术的重要方面.使用模拟器对函数调用关系进行动态跟踪,避免对内核源码的插桩和编译过程,减少对内核的影响.S2E(selective symbolic execution)[6]基于QEMU[7]和符号执行技术,提供插件接口获取操作系统运行时状态、函数调用关系等,支持x86,x86-64,arm平台.但是S2E运行速度慢,缺乏对其他平台如MIPS,Alpha等的支持.向勇等人[8]提出基于QEMU的动态函数调用跟踪框架,通过关闭基本块链接功能,迫使每个基本块代码执行结束后进行基本块切换操作,切换过程中统计函数信息,并写入日志.相比S2E提高了分析性能和支持的CPU平台种类,但是该方法依赖于QEMU的日志并必须关闭QEMU的基本块链接功能,造成3.65倍的性能开销.

上述工具基于QEMU模拟器进行函数调用跟踪,并实现对系统状态的监测与分析,但是监测数据获得的方法破坏了QEMU的模拟和加速机制,造成较大的性能开销.如何精确记录系统的行为并且不破坏QEMU模拟器的加速机制是亟待解决的问题.

通过分析QEMU的运行机制,我们发现QEMU使用平台无关的中间表示进行不同平台指令集的支持,而源指令到中间表示的翻译过程和中间表示到宿主机平台指令的过程都可以在中间表示层进行插桩获取翻译记录.据此,本文提出基于动态二进制翻译和代码插桩的函数调用跟踪框架,在二进制翻译的中间代码阶段,针对特定函数调用和返回指令进行特殊处理,插入性能分析和信息获取桩指令块,从而在运行时获取系统启动与函数调用顺序.基于二进制翻译技术,避免了对源码的依赖,并提供跨平台支持;在中间代码阶段的代码插入技术,减少了对源平台和目标平台的依赖,降低了系统复杂度,性能开销.

本文的主要贡献如下:

1) 提出一种基于二进制翻译的支持多平台的内核函数调用跟踪框架,在二进制翻译的中间代码阶段,生成用于特定信息获取的指令,而不需重编译内核镜像文件;

2) 基于二进制翻译的方法,大大扩展了可支持平台的种类,便于支持新型平台;

3) 不影响二进制翻译技术中基本块链接、冗余代码消除、热路径分析等优化技术,运行效率高.

1 基于动态二进制翻译和代码插桩的函数调用跟踪框架

1.1 动态二进制翻译

二进制翻译是软件安全分析[9]、系统分析[10]、软件优化[11]等领域的关键技术.主流的二进制翻译分为静态二进制翻译和动态二进制翻译.动态二进制翻译在执行时根据程序执行路径以基本块为单位

进行实时加载、翻译、生成目标代码、执行、缓存管理、代码优化等任务,解决了间接分支指令目的地址不确定性、动态生成代码的翻译问题,具有较好的完备性,是二进制翻译的主流技术,其工作原理如图1所示:

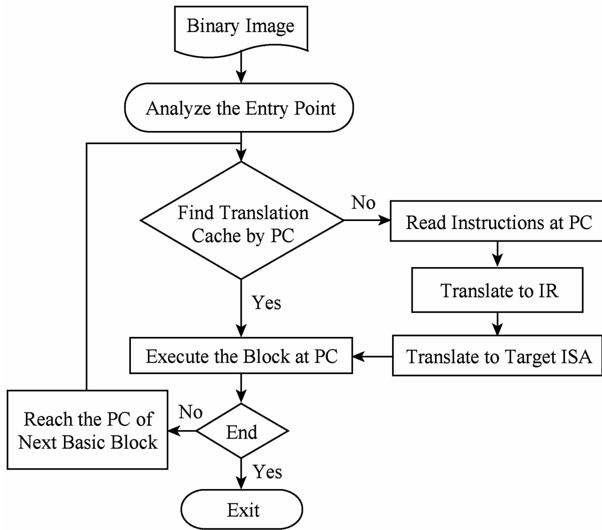


Fig. 1 Dynamic binary translation

图1 动态二进制翻译

首先,加载二进制镜像文件,二进制文件通常为ELF,PE或者Linux内核格式,分析指令入口点PC(program counter),完成内存映射、CPU初始化、虚拟设备创建等工作,并进入翻译过程.由于程序运行的局部性,对已翻译的基本块进行缓存可大大节省翻译开销.动态二进制翻译在执行某一基本块前,首先从缓存中查找,若该基本块已经翻译,则切换环境后进入执行阶段;否则,需要根据当前指令基本块首地址读取指令直到下一条分支指令,转化为中间代码(intermediate representation,IR),最后生成目标指令集(instruction set architecture,ISA)代码并执行.执行完当前基本块后进入下一个基本块并重复上述查找、翻译、执行过程,直到程序结束.从动态二进制翻译的翻译执行流程可以看出,以基本块为单位,翻译过程与执行过程交替进行,能够对不同平台的代码进行翻译转换,解决了多平台指令集支持的问题,极大地增大了软件的适用性.

本文基于动态二进制翻译的系统调用跟踪,方便地支持多种指令集,并提供了指令和基本块粒度的信息统计和代码插入方法,可获取的信息量大大增加.

1.2 代码插桩

代码插桩技术是软件调试中采用的代码修改机值,包括目标代码插桩和源代码插桩,通过插桩点实时获取程序状态,广泛应用于性能分析、覆盖测试、软件调试等领域[12-13].

源代码插桩技术通过对程序源代码的词法分析、语法分析、语义分析等,确定插桩位置,依赖于特定的编程语言,工作量大.例如gprof,ftrace,systemtap都使用到了源代码插桩技术.

目标代码插桩技术通过对目标代码进行相应的分析,分析确定插桩点,与具体的指令集相关,不依赖程序源码,与具体的编程语言和版本无关.本文针对目标代码,在指令粒度进行插桩,提升了检测信息的精确度.

1.3 框架设计思路

为支持多种平台二进制镜像,实现指令粒度的

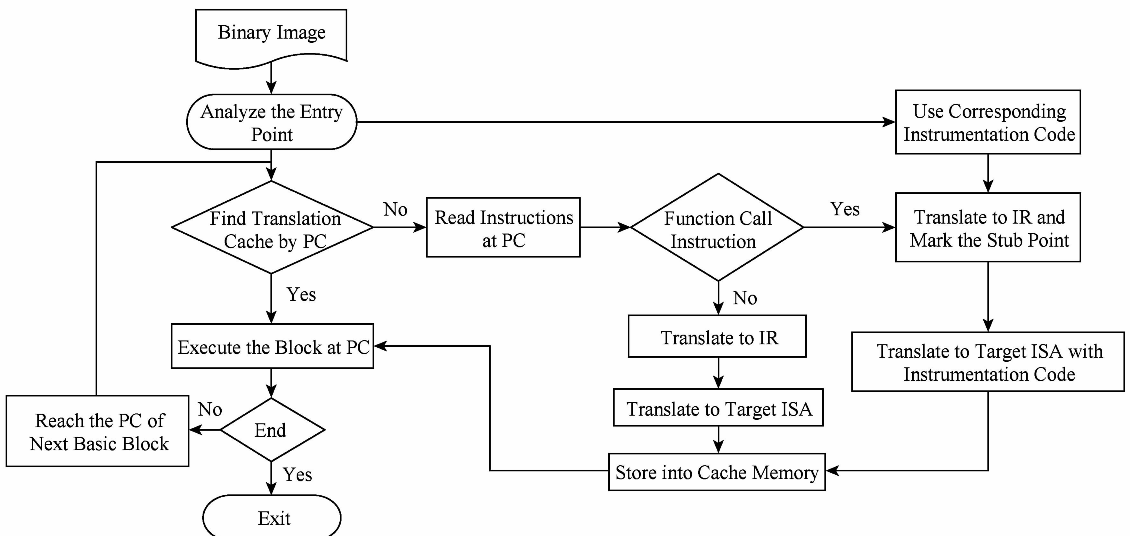


Fig. 2 Dynamic binary translation and code instrumentation based function call tracing framework

图2 基于动态二进制翻译和代码插桩的函数调用跟踪框架

代码插桩,实时获取执行信息、内核加载过程信息、函数调用关系信息等,提出基于动态二进制翻译和代码插桩的函数调用跟踪框架,框架模块如图2所示.

1) 通过二进制翻译系统,载入内核镜像;

2) 对操作系统内核进行分析并确定指令集类型、入口点;

3) 对需要执行的基本块进行二进制翻译,在中间代码阶段,针对特殊的函数调用与返回指令插入桩指令;

4) 最后在执行阶段,每次执行函数调用和返回指令,都会执行插入的桩代码,获取模拟时钟、进程地址、函数地址等.

该框架充分利用了动态二进制翻译系统的整体流程,以指令粒度分析插桩点,动态将桩代码插入到生成的目标码中,实现了实时信息获取功能.

2 基于QEMU和代码插桩的函数调用跟踪系统实现

根据第1节函数调用跟踪框架的原理,基于开源的二进制翻译系统QEMU,进行桩代码设计、中间代码桩标记码和目标代码的进入桩插入、监测信息处理等开发工作,实现了基于QEMU和代码插桩的函数调用跟踪系统.

2.1 快速的处理器模拟器QEMU

QEMU是一个快速的处理器模拟器,支持多种源平台和多种目标平台[14].利用平台无关的中间表示形式TCG(tiny code generator),实现将x86,arm,Alpha,PowerPC等指令集转换为TCG中间表示,然后翻译为宿主机的指令集,具备快速模拟和跨平台支持特性.作为模拟器可模拟多种设备、硬件,支持全系统运行,可在宿主机运行不同指令集的客户机.

QEMU以基本块为单位对源指令进行翻译,提供了代码缓存管理、基本块链接、冗余代码消除等优化,提升了模拟器速度.QEMU的执行流程如图3所示.首先,QEMU加载源指令集的可执行文件或者内核镜像,完成空间申请、地址映射、入口点分析等工作后从第1条指令开始进入翻译执行的过程.以基本块为单位,读取源指令集的指令序列,生成对应的TCG中间表示,然后分析优化后生产宿主机指令集的代码,并将动态生成的代码存入缓存,实现一次翻译多次使用.在基本块执行完毕后,模拟器试图查找下一个基本块对应的代码,如果在缓存代码区找到需要的代码块,则直接跳转执行,否则需要启动翻译过程.由于每次基本块执行完毕后进行查找下一个基本块引起较大开销,QEMU实现基本块链接功能,把下一个基本块和当前块直接使用跳转指令连接起来,避免了查找消耗.基本块链接技术极大地提高了模拟器的运行效率,是QEMU模拟器的高效率关键技术.

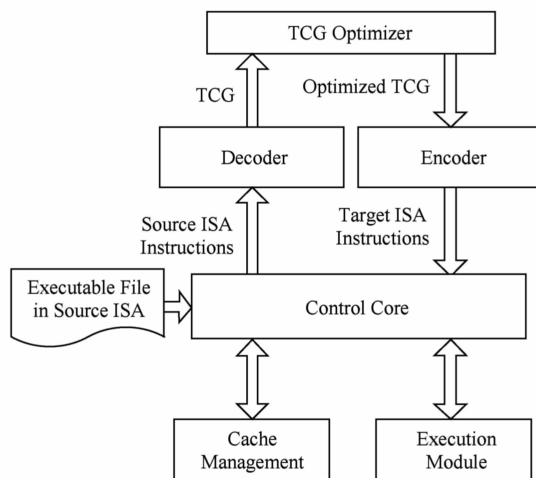


Fig. 3 Framework of QEMU

图3 QEMU框架结构

从QEMU的运行原理可以看出,QEMU使用了平台无关中间表示TCG,是典型的动态二进制翻译系统.可在TCG阶段对需要检测的指令进行操作,插入所需统计信息桩代码,在运行时获取函数调用信息.

2.2 函数调用指令翻译

QEMU进行二进制翻译采用的平台无关中间表示是TCG,曾作为支持多平台的C语言交叉编译器的后端,是一种类似于RISC的指令集.TCG仅支持32b和64b整型,指针类型是根据宿主机的位数按照32b或64b整型定义的.QEMU把源指令变换为TCG操作,进行活性分析、常量计算优化,然后变换为对应宿主机指令集的指令序列.例如指令add_i32 t_0,t_1,t_2表示t_1+t_2的和放到t_0寄存器中,操作数t_0,t_1,t_2都是32b整型.

函数调用指令会更改程序执行顺序,并修改栈内容,TCG针对函数调用遵循以下规则:

1) 参数和返回值的类型仅支持32b,64b整型和指针.

2) 栈向下填充.

3) 前N个参数通过寄存器传递,超过N个参数的以字为单位放入栈传递.N在文件中可根据具体平台自定义.

4) 一些寄存器在函数调用过程中会被重复使用.

5) 函数可以使用寄存器返回 0 或 1. 在 32 b 宿主机上运行 64 b 系统时,为了能够返回 64 b 值,必须使用寄存器返回 2 个 32 b 值.

这些规则限定了函数调用指令处理的方法,一条函数调用指令会被分解为参数传递操作、栈修改操作、分支跳转操作 3 部分.

为了插入用于信息统计的桩代码,添加新的 TCG 中间表示,在分支跳转操作前插入 gen_stubc TCG 指令,用以生成调用桩代码的指令序列. 例如指令 call 0x7fac6ee 插入 gen_stubc 后的翻译过程如图 4 所示,可以看到生成了 call stubc 的中间代码指令和对应的目标平台指令"mov $0x56333978b660,%r10;callq * %r10",其中地址 $0x56333978b660 为桩代码 helper_stubc 在内存中的位置.

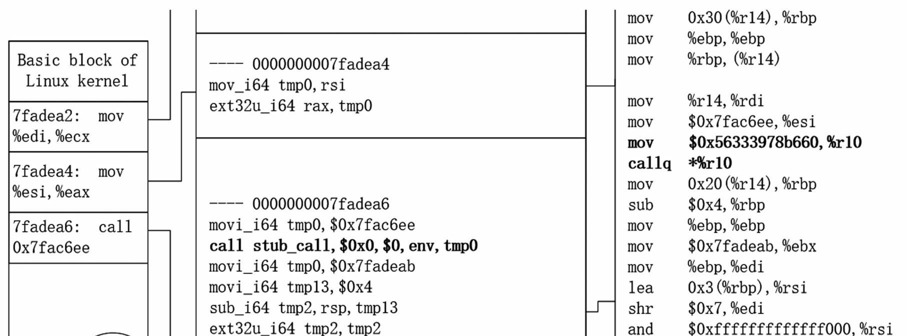


Fig. 4 Translation of call instruction with instrumentation code

图 4 插桩后的 call 指令翻译过程

2.3 桩代码设计

桩代码是在动态翻译的过程中,插入到函数调用和返回指令对应目标代码中,完成信息获取的代码段. 桩代码需要完成信息监测功能,同时不能对原有代码的语义造成影响.

每次函数调用和函数返回都调用桩代码,记录实时操作信息,使得获取从第 1 条指令到最后启动完成全局的调用信息;需要完成信息监测功能,同时不能对原有代码的语义造成影响;执行的次数非常多,尽量保持简短.

考虑以上因素,结合 QEMU 具体实现,采用 helper 机制调用桩代码. QEMU 在进行指令翻译的过程中,会遇到语义复杂的指令,例如,这类指令如果使用宿主平台汇编指令实现其功能非常复杂,以至于生成的目标代码极其庞大,易出错. 为了兼顾效率与灵活性,QEMU 使用特定的 helper 函数实现此类功能复杂型指令的模拟操作. 例如使用 *tcg_gen_helper_x_y* 可以生成调用参数为 32 b,64 b 或指针的函数. 默认情况下,在调用 helper 函数时,所有的全局变量将会保存到对应 *env* 的位置,以防 helper 修改某些寄存器的值. 一个 helper 函数是可以访问模拟器 CPU 状态变量 *env* 的. 利用 helper 机制,可方便的在函数调用和返回处调用 helper 的 stub 桩代码,访问全局 CPU 状态变量 env,完成信息记录功能.

2.4 针对 x86 指令集的桩代码设计

使用 QEMU 的 helper 机制,针对 x86 指令集进行代码插桩的方法如下,首先在 target/i386/helper. h 中添加定义 *DEF_HELPER_2*(*stub_call*, void,*env*,*tl*)其中 *stub_call* 是 helper 函数名,void 是返回值,*env*,*tl* 是参数类型,表示函数 void *helper_stub_call*(*CPUx86State* * *env*,target_ulong *val*)的声明,类似的 *DEF_HELPER_2*(*stub_ret*,void,*env*,*tl*)表示 ret 指令的返回嵌入桩代码. 然后,在 translate. c 中对 call,ret 指令翻译的响应位置,插入 *gen_helper* 生成调用 *helper_stub_call*,*helper_stub_ret* 的 TCG 指令,其中使用 *gen_helper* 在翻译时首先会把模拟 CPU 的寄存器值保存到内存中,防止内部代码执行影响寄存器内容;最后,在 target/i386/misc_helper. c 中实现 2 个函数. 桩代码的功能如图 5 所示,使用全局的信息结构体 env 保存包括时间戳、进程 ID、调用地址、当前函数地址、函数名称等信息,并使用全局的 *call_info_arr* 数组把所有函数调用和返回的信息进行保存. 然后调用 QEMU 内置函数 *QEMU_clock_get_ns*()获取模拟时钟,通过 *CPUx86State* 结构体的寄存器值,分别获取 ESP 值、进程 ID、调用函数地址、当前函数地址并存入全局数组;最后桩代码执行完毕,回到基本块,继续执行后续代码.

```
/* Information: timestamps, process ID, thread ID, * address of
   callee */
  void helper_stub_call(CPUx86State * env, target_ulong next_
    eip){
  /* get qemu virtual clock */
  qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL);
  /* The upper 20 bits of CR3 of x86 is the physical address * of
     the first page directory entry */
  env->cr[2];
  /* ESP register points to the top of current thread's stack */
  env->regs[R_ESP];
  /* Next_eip is assigned when translating */
  next_eip;
  }
```

Fig. 5 Instrumentation code of x86

图5 针对x86平台的桩代码

2.5 针对ARM指令集的桩代码设计

ARM指令集是目前移动平台使用最广泛的指令集,ARM属于精简指令集,有31个32b的通用寄存器,其中有3个寄存器SP(stack pointer)、LR(link register)、PC有特殊的用途.

通常R13作为栈指针寄存器存储栈指针SP,pop和push指令使用R13的值访问栈空间.

R14作为链接寄存器LR.当使用分支链接指令(BL,BLX)指令时,该寄存器保存当前分支链接指令的下一条指令的地址.在函数调用结束返回时,R14作为返回的地址,其他时候R14可作为通用寄存器使用.因此在进行函数调用跟踪时,可通过访问R14获得返回地址.

R15作为程序计数器PC,由于ARM体系结构采用了多级流水线技术,对于ARM指令集而言,PC总是指向当前指令的下2条指令的地址,即PC的值为当前指令的地址值加8B程序状态寄存器.

页表转换基寄存器(TTBR),*CP*15的第2个寄存器存放当前进程的物理基地址,在QEMU模拟ARM类型的CPU时使用*CP*15.*ttbr*0_*el*[2]表示该值.

ARM指令集使用BL,BLX进行子程序调用时,把返回地址存储到LR,但是没有专门的返回指令,而是通过把LR的值传入PC寄存器中的方法.通常使用如下4种函数返回方法:

MOV PC,LR或BX LR

该MOV指令直接把LR寄存器的值复制到PC寄存器,更改程序执行流程,使函数返回到LR存储的地址处.同样BX LR直接把LR寄存器中的返回地址作为无条件跳转目的地址,完成函数返回操作.

或者stmfd与ldmfd,push,pop指令在函数入口处执行入栈并在函数结束时执行出栈操作来更改程序执行流程.使用在函数入口处使用如下指令保存寄存器状态:

stmfd SP!,{<registers>,LR}或push{<registers>,LR},

即保存通用寄存器和LR的值到栈中,函数体执行完毕需要返回时,恢复CPU状态,

ldmfd SP!,{<registers>,PC}或pop{<registers>,PC}.

以上4种方式是常用的函数返回方法,在进行函数调用跟踪时,需要在翻译阶段判断这4种情况,并生成调用的桩代码的TCG指令.

根据ARM指令的函数调用和返回方法以及特殊寄存器的使用方式,设计ARM平台的桩代码如图6所示:

```
void helper_stub_arm(CPUARMState * env, target_ulong next_
   eip){
  /* get qemu virtual clock */
  qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL);
  /* The ttbr0_el of CP15 is the physical address of the * first
     page directory entry */
  env->CP15.ttbr0_el[2];
  /* R13 register points to the top of current thread's stack */
  env->regs[13];
  /* Next_eip is assigned when translating */
  next_eip;
}
```

Fig. 6 Instrumentation code of ARM

图6 针对ARM平台的桩代码

2.6 针对MIPS指令集的桩代码设计

MIPS指令集广泛应用在交换设备、移动设备、嵌入式微控制器等产品,属于精简指令集,有32个通用寄存器,其中寄存器$29用作栈寄存器$sp,$31寄存器用作返回地址寄存器$ra.函数调用指令使用Motorala命名法,子程序调用成为跳转并链接,助记符以al结尾,例如jal imm或jalr $reg该指令首先把C寄存器保存到$ra,然后跳转到立即数imm或$reg指向的指令块.函数返回时,使用jr $ra跳转到保存的指令地址$ra,完成函数返回操作.若被调用函数再次调用其他函数,那么$ra的值会先存入栈空间,子函数执行结束后,重新载入$ra的值,进行返回.类似的,使用bal,bgezal,bltzal进行相对PC偏移的函数调用,返回值会存储到$ra中.在翻译阶段对上述子程序调用指令和

函数返回指令生成调用桩代码的 TCG 指令,实现函数调用跟踪.

我们使用当前进程的物理地址作为唯一标识,以区分不同的进程. MIPS 为每个进程分配了地址空间 ID,称为 ASID(address space id),在进行 TLB 转换时,使用 ASID 和虚拟页号(virtual page number, VPN)拼接成唯一的页地址,映射到物理地址,因此可以使用 ASID 拼接 VPN 作为当前进程的标识. QEMU 实现了 MIPS 的 ASID,VPN 保存在 *env*→*CP0_EntryHi* 中. 根据 MIPS 指令集和寄存器特点,针对 MIPS 指令集的桩代码设计如图 7 所示:

```
void helper_stub_mips(CPUMIPSState * env, target_ulong next_
  eip){
  /* get qemu virtual clock */
  qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL);
  /* The ASID and VPN is used By TLB to get the * physical
     address of the first page directory entry */
  /* Get ASID from CP0_EntryHi by xor the mask */
  env→CP0_EntryHi | env→CP0_EntryHi_ASID_mask
  /* Get VPN from CP0_EntryHi and use VPN link * with ASID
     as the process identifiers */
  env→CP0_EntryHi & (PAGE_SIZE-1)
  /* R31 register points to the top of current thread's stack */
  env→regs[31];
  /* Next_eip is assigned when translating */
  next_eip;
}
```

Fig. 7 Instrumentation code of MIPS

图 7 针对 MIPS 平台的桩代码

以上给出了 x86,ARM,MIPS 指令集的桩代码设计方法,可以看出桩代码从全局环境结构体 env 中直接读取相应信息,代码简短、直接,可很快根据具体平台要求,具有多平台支持的特点,其他平台可根据寄存器的使用特点,很方便地移植到需要支持的指令集构架平台上.

2.7 日志记录与函数调用图

日志记录包括 call 记录和 ret 记录,call 记录需保存调用时间、进程标识、线程标识、被调用函数地址、被调用函数名称;而 ret 记录包括调用时间、进程标识、线程标识即可. 例如在时刻 592092451ns 函数 *start_kernel* 被调用,接着函数 *start_kernel* 调用了 *set_task_stack_end_magic*, *smp_setup_processor_id* 记录如图 8 所示:

```
59 2092451 ffff8800000140a0 ffff81e02000 ffffffff81f4db11 start_
  kernel
59 2104850 ffff8800000140a0 ffff81e02000 ffffffff81057010 set_
  task_stack_end_magic
592118045 ffff8800000140a0 ffff81e02000
59 2130081 ffff8800000140a0 ffff81e02000 ffffffff81f4db05 smp_
  setup_processor_id
592137020 ffff8800000140a0 ffff81e02000
59 2141513 ffff8800000140a0 ffff81e02000 ffffffff81f6cb5d cgroup_
  init_early
59 2157318 ffff8800000140a0 ffff81e02000 ffffffff810d8af0 init_
  cgroup_root
…
```

Fig. 8 Example of the logs

图 8 日志记录示例

通过桩代码的统计,获取了对应信息,需结合分析二进制镜像获取符号表中函数名称,即可把 call_info 结构体中的 *function_name* 确定,从而形成完整的统计,按照相关工具进行输出,生成函数调用图,本文使用 graphviz[15] 工具生成函数调用图,如图 9 所示 start_kernel 执行时的函数调用关系. 该可视化工作已有相关研究,可结合已有相关解析工具和算法[16],不作为本文的主要工作.

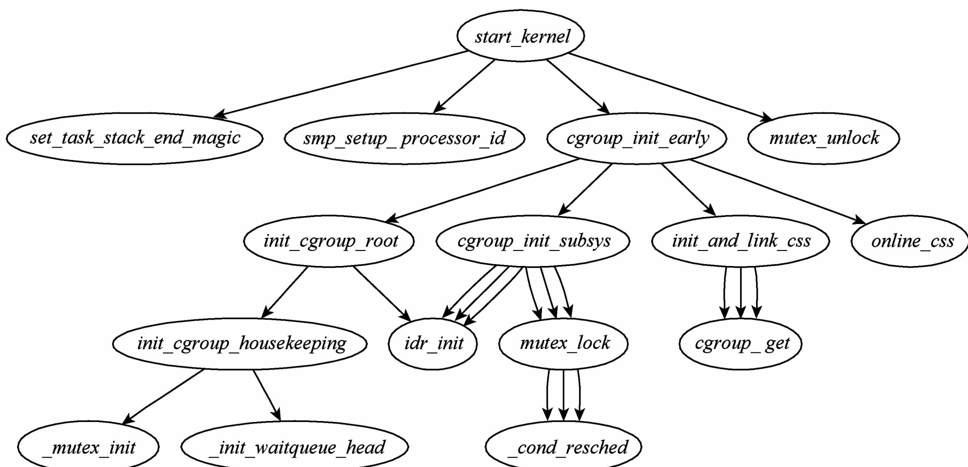


Fig. 9 Example of function call map of Linux 4.9 kernel

图 9 函数调用跟踪 Linux 4.9 内核绘图示例

3 与现有工作的比较

现有的基于模拟器的函数调用跟踪工具有S2E和向勇所提出的基于QEMU的动态函数调用跟踪框架不妨记为QEMU-DFCT(QEMU-based dynamic function call tracing).下面将本文所提出的方案分别与S2E,QEMU-DFCT进行对比.

3.1 与S2E对比

3.1.1 功能性比较

S2E结合了符号执行和QEMU,并提供插件发布、订阅、处理事件.可通过编写回调函数,注册某类型事件即可完成订阅.当所注册时间类型触发如函数调用,则回调函数会接收到调用信息、执行回调函数定义的功能.

由于S2E对QEMU本身做了较大量的修改,目前可支持x86,ARM指令集,未能对QEMU所支持的多种平台进行继承,丢弃原始QEMU的诸多特性.

3.1.2 性能比较

S2E的符号执行模块会对分析对象的每一个执行路径进行分析,对指定模块的路径属性、条件进行监测和操纵,功能复杂,导致性能很低.本文提出的方案直接对QEMU的中间表示TCG,helper机制进行利用,实现了函数调用与返回跟踪技术,继承了QEMU原生的多平台、速度快的特性.

3.2 与QEMU-DFCT对比

本文提出的方法所实现的函数调用记录的功能与QEMU-DFCT一致.

QEMU-DFCT继承了原生QEMU的诸多优点,如跨平台、易拓展,同时以很小的修改代码实现了函数调用跟踪技术.QEMU-DFCT直接读取QEMU原始数据结构,并并行解析,提高了处理速度.

但是,QEMU-DFCT必须强制关闭基本块链接功能,在每一个基本块执行结束后进行基本块切换操作,都必须检查是否为函数调用或返回基本块,造成了3.65倍的性能开销.本文提出的方案不是在基本块切换阶段记录信息,而是将以桩代码的形式,插入到函数调用和返回指令生成的宿主机代码中,支持基本块链接功能,降低了性能开销.

4 实　　验

本文在QEMU 2.9.92版本进行实验,相关实验环境如表1所示.其中Linux内核是使用QEMU加载分析的内核,文件系统是使用QEMU加载时指定的文件系统.宿主机是指运行QEMU的计算机.

Table 1 Experiment Setup

表1 实验环境

Hardware & Software	Version
QEMU	2.9.92
S2E	2.0
Linux Kernel	4.9
File System	Busybox 1.27.0
CPU of Host	Intel Core i7-6500U CPU @ 2.50GHz×4
Memory of Host/GB	7.7
Operation Systme of Host	Ubuntu 16.04 LTS
Hard Disk of Host/GB	SSD 128
Machine Type of ARM	vexpress-a9
Machine Type of MIPS	malta

本文实现了所提出的插桩算法(QEMU helper stub,QHS)和QEMU-DFCT算法的记录函数调用信息和输出部分,并在x86,ARM,MIPS平台进行了性能对比实验,实验结果如图10所示:

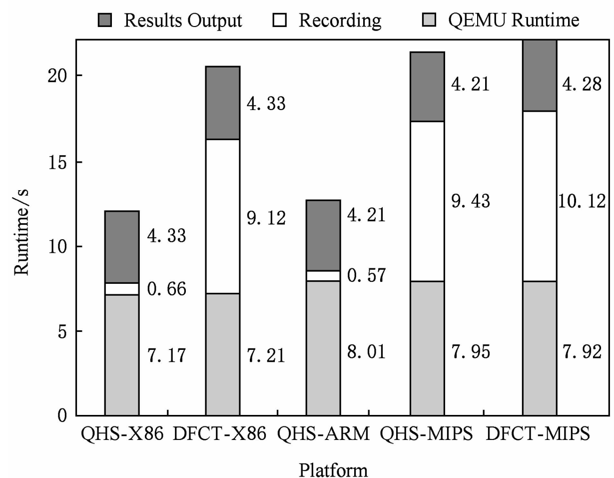


Fig. 10 Performance comparison of QHS and DFCT

图10 QHS,DFCT的性能比较

以QHS-x86为例,使用未修改的QEMU加载x86指令集Linux 4.9内核、busybox 1.27.0文件系统,从启动到出现shell提示符时间消耗是4.33 s.使用修改后的用于函数调用跟踪的版本,从启动到出现shell提示符,同时完成函数调用信息记录,并将信息处理并输出到磁盘文件中,所用时间是12.16 s,其中,桩代码执行总时间是0.66 s,信息处理并输出使用时间是7.17 s.桩代码执行信息记录增加了15.24%的开销,而信息处理并输出到磁盘文件增加

了165.59%的开销;DFCT-x86信息记录增加了210.62%的开销,记录输出开销是166.51%.由于ARM和MIPS平台寄存器数目多,DFCT进行基本块切换操作需要保存或恢复的寄存器数目多,造成调用记录开销高于x86平台,而寄存器数目并不影响QHS算法的性能.

而使用S2E对x86指令集的内核进行加载分析,总时长达3429.34s,是因为S2E会对内核使用KLEE模块进行符号分析,对内核的海量的执行路径进行路径属性、执行条件分析,并在每个路径和函数调用部分保存并检查系统状态、反馈插件调用等任务.所以造成内核启动很慢.

从实验结果可以看出,本文实现的基于动态二进制翻译和代码插桩的函数调用跟踪工具的开销要远远优于现有的S2E,QEMU-DFCT跟踪工具.

5 总 结

本文提出了基于动态二进制翻译和代码插桩的函数调用跟踪,给出了系统框架设计并基于QEMU开发了函数调用跟踪系统,可针对Linux内核跟踪所有函数调用,并在x86,ARM,MIPS平台进行了实现,验证了系统框架的有效性并评估了系统性能.针对QEMU二进制翻译过程的中间表示进行设计,插入信息搜集桩代码而不影响基本块链接、活性分析、常量计算等优化技术,以尽可能小的性能开销达到了动态函数跟踪的目的.

本工作还有一些可改进的地方,例如桩代码的实现可以使用嵌入式汇编或者不使用helper机制,直接生成完成信息记录功能的宿主机指令,进一步提高速度,降低开销.实现插件机制,例如注册指定函数调用事件的回调函数,在桩代码进行记录时,若被调用函数是指定的监测函数,调用回调函数.

参 考 文 献

[1] Pfoh J, Schneider C, Eckert C. Nitro: Hardware-based system call tracing for virtual machines [C] //Proc of the Advances in Information and Computer Security. Berlin: Springer, 2011: 96-112

[2] Hibbeler J D, Wang Jhychun. Dynamic CPU usage profiling and function call tracing: US, Patent 7093234 [P/OL]. (2006-08-15) [2018-05-16]. http://www.freepatentsonline.com/7093234.html

[3] Pappas V, Polychronakis M, Keromytis A D. Transparent ROP exploit mitigation using indirect branch tracing [C] //Proc of the 22nd USENIX Conf on Security. Berkeley, CA: USENIX Association, 2013: 447-462

[4] Zheng Yuhui, Mu Yongmin, Zhang Zhihua. Research on the static function call path generating automatically [C] //Proc of the 2nd IEEE Int Conf on Information Management and Engineering. Piscataway, NJ: IEEE, 2010: 405-409

[5] Graham S L, Kessler P B, McKusick M K. Gprof: A call graph execution profiler [J]. ACM SIGPLAN Notices, 2004, 39(4): 49-57

[6] Chipounov V, Kuznetsov V, Candea G. The S2E platform: Design, implementation, and applications [J]. ACM Transactions on Computer Systems, 2012, 30(1): 1-49

[7] Bellard F. QEMU, a fast and portable dynamic translator [C] //Proc of the Annual Conf on USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2005: 41-46

[8] Xiang Yong, Cao Ruidong, Mao Yingming. QEMU-based dynamic function call tracing [J]. Journal of Computer Research and Development, 2017, 54(7): 1569-1576 (in Chinese)
(向勇,曹睿东,毛英明.基于QEMU的动态函数调用跟踪[J].计算机研究与发展,2017,54(7):1569-1576)

[9] Guo Haoran, Pang Jianmin, Zhang Yichi, et al. Hero: A novel malware detection framework based on binary translation [C] //Proc of the IEEE Int Conf on Intelligent Computing and Intelligent Systems. Piscataway, NJ: IEEE, 2010: 411-415

[10] Feiner P, Brown A D, Goel A. Comprehensive kernel instrumentation via dynamic binary translation [C] //Proc of the 17th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2012: 135-146

[11] Chylek S. Collecting program execution statistics with QEMU processor emulator [C] //Proc of the Int Multi Conf on Computer Science and Information Technology. Piscataway, NJ: IEEE, 2009: 555-558

[12] Shah H, Coombes A, Raabe A, et al. Measurement based WCET analysis for multi-core architectures [C] //Proc of the 22nd Int Conf on Real-Time Networks and Systems. New York: ACM, 2014: 257-266

[13] Ofuonye E, Miller J. Securing web-clients with instrumented code and dynamic runtime monitoring [J]. Journal of Systems and Software, 2013, 86(6): 1689-1711

[14] Bartholomew D. Qemu: A multihost multitarget emulator [J]. Linux Journal, 2006, 2006(145): 68-71

[15] Ellson J, Gansner E R, Koutsofios E, et al. Graphviz and Dynagraph—Static and Dynamic Graph Drawing Tools [M]. Berlin: Springer, 2003: 127-148

[16] Jia Di, Xiang Yong, Sun Weizhen, et al. Database-based online call graph tool applications [J]. Journal of Chinese Computer Systems, 2016, 37(3): 422-427 (in Chinese)

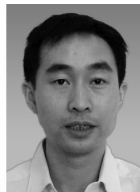
(贾荻, 向勇, 孙卫真, 等. 基于数据库的在线函数调用图工具[J]. 小型微型计算机系统, 2016, 37(3): 422-427)

Lu Shuaibing, born in 1990. Master and research assistant. His main research interests include operation system, binary translation.

Zhang Ming, born in 1990. Master and research assistant. His main research interests include artificial intelligence, machine learning and network security.

Lin Zhechao, born in 1990. Master and research assistant. His main research interests include software validation and operating system.

Li Hu, born in 1987. PhD and engineer. His main research interests include information system security, machine learning and data mining.



Kuang Xiaohui, born in 1975. PhD and professor in the National Key Laboratory of Science and Technology on Infomations System Security. His main research interests include wireless network, and information security.

Zhao Gang, born in 1969. Received his PhD degree from the Department of Computer Science and Technology, Tsinghua University in 2009. Professor in the National Key Laboratory of Science and Technology on Information System Security. His main research interests include computer network and information security.

《计算机研究与发展》征订启事

《计算机研究与发展》(Journal of Computer Research and Development)是中国科学院计算技术研究所和中国计算机学会联合主办、科学出版社出版的学术性刊物,中国计算机学会会刊. 主要刊登计算机科学技术领域高水平的学术论文、最新科研成果和重大应用成果. 读者对象为从事计算机研究与开发的研究人员、工程技术人员、各大专院校计算机相关专业的师生以及高新企业研发人员等.

《计算机研究与发展》于1958年创刊,是我国第一个计算机刊物,现已成为我国计算机领域权威性的学术期刊之一. 并历次被评为我国计算机类核心期刊,多次被评为"中国百种杰出学术期刊". 此外,还被《中国学术期刊文摘》、《中国科学引文索引》、"中国科学引文数据库"、"中国科技论文统计源数据库"、美国工程索引(Ei)检索系统、日本《科学技术文献速报》、俄罗斯《文摘杂志》、英国《科学文摘》(SA)等国内外重要检索机构收录.

国内邮发代号:2-654;国外发行代号:M603

国内统一连续出版物号:CN11-1777/TP

国际标准连续出版物号:ISSN1000-1239

联系方式:

100190 北京中关村科学院南路6号《计算机研究与发展》编辑部

电话: +86(10)62620696(兼传真); +86(10)62600350

Email: crad@ict. ac. cn

http://crad. ict. ac. cn