(19)中华人民共和国国家知识产权局

(12)发明专利

(10)授权公告号 CN 106295258 B
(45)授权公告日 2019.03.26

(21)申请号 201610638311.X

(22)申请日 2016.08.04

(65)同一申请的已公布的文献号
申请公布号 CN 106295258 A

(43)申请公布日 2017.01.04

(73)专利权人 南京大学
地址 210000 江苏省南京市汉口路22号

(72)发明人 曾庆凯 谢志宇

(74)专利代理机构 南京钟山专利代理有限公司 32252
代理人 戴朝荣

(51)Int.Cl.
G06F 21/12(2013.01)
G06F 21/56(2013.01)

(56)对比文件
US 2009320129 A1,2009.12.24,
CN 105488397 A,2016.04.13,
CN 102662830 A,2012.09.12,
王明华等.二进制代码块: 面向二进制程序的细粒度.《信息安全学报》.2016,第1卷(第2期),61-71.
Lucas Davi等.MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones.《http://www.trust.rub.de/media/emma/veroeffentlichungen/2012/04/11/Davi_MoCFI.pdf》.2012,2-17.

审查员 罗捷

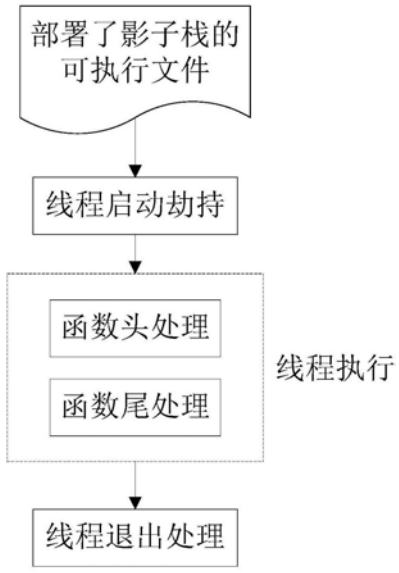
权利要求书2页 说明书7页 附图8页

(54)发明名称

用于多线程后向控制流完整性保护的影子栈实现方法

(57)摘要

本发明提供了一种用于多线程后向控制流完整性保护的影子栈实现方法。所述用于多线程后向控制流完整性保护的影子栈实现方法包括如下步骤:a、对待保护程序的源码进行编译时插装,生成部署了影子栈的可执行文件;b、启动部署了影子栈的可执行文件,由动态共享库劫持线程的创建和启动,完成创建影子栈;c、插装后的函数头和函数尾实现所述影子栈的栈顶指针的维护和后向控制流完整性的检查;d、进行所述线程的退出处理,并通过所述动态共享库中注册的析构函数对所述影子栈进行销毁。本发明的有益效果在于:所述用于多线程后向控制流完整性保护的影子栈实现方法可提高多线程C/C++程序的控制流完整性保护能力,以降低受到代码复用攻击的威胁。



CN 106295258 B

1.一种用于多线程后向控制流完整性保护的影子栈实现方法,其特征在于:包括如下步骤:

a、对待保护程序的源码进行编译时插装,生成部署了影子栈的可执行文件;

b、启动部署了影子栈的可执行文件,由动态共享库劫持线程的创建和启动,完成创建影子栈;

c、插装后的函数头和函数尾实现所述影子栈的栈顶指针的维护和后向控制流完整性的检查;

d、进行所述线程的退出处理,通过所述动态共享库中注册的析构函数对所述影子栈进行销毁;

其中,步骤b具体包括如下步骤:

调用所述动态共享库内的构造函数,并在所述构造函数中创建主线程的影子栈;

将所述主线程的影子栈的栈顶指针保存到线程控制模块中;

调用主程序的构造函数,并启动主线程;

调用伪线程的创建函数,并在所述伪线程的创建函数中创建所述伪线程的影子栈;

在所述伪线程的创建函数中调用原线程的创建函数,并执行包装例程;

将所述伪线程的影子栈的栈顶指针保存到所述线程控制模块中,并执行原始例程。

2.根据权利要求1所述的用于多线程后向控制流完整性保护的影子栈实现方法,其特征在于:步骤a具体包括如下步骤:

目标源代码通过LLVM编译器的前端Clang转换成中间语言IR;

所述中间语言IR经中间语言层的优化后进入编译器后端;

所述编译器后端为所述中间语言IR选择目标平台的具体指令,并将所述中间语言IR线性化,然后在PEI pass生成所述函数头和所述函数尾时进行插装,接着发射为目标文件,最后和动态共享库共同链接成部署了影子栈的可执行文件。

3.根据权利要求1所述的用于多线程后向控制流完整性保护的影子栈实现方法,其特征在于:在步骤c中,在LLVM编译器的后端生成所述函数头和所述函数尾时进行插装,PEI pass负责生成所述函数头和所述函数尾,并在runOnMachineFunction函数中调用insertPrologEpilogCode函数;且所述insertPrologEpilogCode函数调用emitPrologue函数生成所述函数头,并调用emitEpilogue函数生成所述函数尾。

4.根据权利要求3所述的用于多线程后向控制流完整性保护的影子栈实现方法,其特征在于:在步骤c中,所述函数头的处理流程包括如下步骤:

进入被调函数,并将保存在TCB中的影子栈的栈顶指针加四;

将保存在所述TCB中的影子栈的栈顶指针加载到第一通用寄存器中;

将所述影子栈的返回地址保存到第二通用寄存器中;

将所述第二通用寄存器的值保存到所述第一通用寄存器所指向的内存中;

清除所述第一通用寄存器中的值。

5.根据权利要求4所述的用于多线程后向控制流完整性保护的影子栈实现方法,其特征在于:在步骤c中,所述函数尾的处理流程包括如下步骤:

将保存在TCB中的影子栈的栈顶指针加载到所述第一通用寄存器中;

将所述影子栈的返回地址保存到第二通用寄存器中;

判断所述第一通用寄存器中的值是否为零，如果是，则调用abort函数终止运行，如果否，则执行下一步骤；

将所述第一通用寄存器中的值减四；

比较所述第二通用寄存器中的值和所述第一通用寄存器加四后所指向内存中的值是否相等，如果否，则返回判断所述第一通用寄存器中的值是否为零的步骤，如果是，则将所述第一通用寄存器中的值保存在所述TCB中，并返回主调函数。

6.根据权利要求1所述的用于多线程后向控制流完整性保护的影子栈实现方法，其特征在于：在步骤d中，所述线程退出处理的流程具体包括如下步骤：

启动所述线程的退出程序；

调用和所述动态共享库中TSD变量thread_cleanup_key相关联的析构函数；

判断所述析构函数的参数是否小于销毁所述TSD变量的最大尝试次数，如果是，则将所述析构函数的参数加1，并返回调用和所述动态共享库中TSD变量thread_cleanup_key相关联的析构函数步骤；如果否，则执行销毁所述线程的影子栈步骤。

用于多线程后向控制流完整性保护的影子栈实现方法

技术领域

[0001] 本发明属于软件的代码复用攻击防御技术领域，具体地涉及一种用于多线程后向控制流完整性保护的影子栈实现方法。

背景技术

[0002] 代码复用攻击可以绕过主流操作系统上部署的数据执行保护等安全防御机制，对计算机系统的安全性构成巨大威胁。控制流完整性保护是一种较为有效的代码复用攻击防御手段。其中，后向控制流完整性是指程序中ret指令相关控制流的正确性。利用影子栈技术可以实施后向控制流完整性保护。但是，现有的影子栈保护方案不适用于多线程，实现过程中需要通过修改源代码为影子栈创建空间，不便于部署。而且，影子栈自身安全性不够完善，存在被泄露和篡改的威胁。

[0003] 因此，有必要提出一种能够解决目前影子栈实现方案不适用于多线程、需要修改源代码和隐藏机制不完善等问题的适用于多线程的后向控制流完整性保护的影子栈实现方法。

发明内容

[0004] 本发明的目的在于提供一种能够解决目前影子栈实现方案不适用于多线程、需要修改源代码和隐藏机制不完善等问题的用于多线程后向控制流完整性保护的影子栈实现方法。

[0005] 本发明的技术方案如下：一种用于多线程后向控制流完整性保护的影子栈实现方法，包括如下步骤：

[0006] a、对待保护程序的源码进行编译时插装，生成部署了影子栈的可执行文件；

[0007] b、启动部署了影子栈的可执行文件，由动态共享库劫持线程的创建和启动，完成创建影子栈；

[0008] c、插装后的函数头和函数尾实现所述影子栈的栈顶指针的维护和后向控制流完整性的检查；

[0009] d、进行所述线程的退出处理，并通过所述动态共享库中注册的析构函数对所述影子栈进行销毁。

[0010] 优选地，步骤a具体包括如下步骤：

[0011] 目标源代码通过LLVM编译器的前端Clang转换成中间语言IR；

[0012] 所述中间语言IR经中间语言层的优化后进入编译器后端；

[0013] 所述编译器后端为所述中间语言IR选择目标平台的具体指令，并将所述中间语言IR线性化，然后在PEI pass生成所述函数头和所述函数尾时进行插装，接着发射为目标文件，最后和动态共享库共同链接成部署了影子栈的可执行文件。

[0014] 优选地，步骤b中具体包括如下步骤：

[0015] 调用所述动态共享库内的构造函数，并在所述构造函数中创建主线程的影子栈；

[0016] 将所述主线程的影子栈的栈顶指针保存到线程控制模块中；

[0017] 调用主程序的构造函数，并启动主线程；

[0018] 调用伪线程的创建函数，并在所述伪线程的创建函数中创建所述伪线程的影子栈；

[0019] 在所述伪线程的创建函数中调用原线程的创建函数，并执行包装例程函数；

[0020] 将所述伪线程的影子栈的栈顶指针保存到所述线程控制模块中，并执行原始例程。

[0021] 优选地，在步骤c中，在LLVM编译器的后端生成所述函数头和所述函数尾时进行插装，PEI pass负责生成所述函数头和所述函数尾，并在runOnMachineFunction函数中调用insertPrologEpilogCode函数；且所述insertPrologEpilogCode函数调用emitPrologue函数生成所述函数头，并调用emitEpilogue函数生成所述函数尾。

[0022] 优选地，在步骤c中，所述函数头的处理流程包括如下步骤：

[0023] 进入被调函数，并将保存在TCB中的影子栈的栈顶指针加四；

[0024] 将保存在所述TCB中的影子栈的栈顶指针加载到第一通用寄存器中；

[0025] 将所述影子栈的返回地址保存到第二通用寄存器中；

[0026] 将所述第二通用寄存器的值保存到所述第一通用寄存器所指向的内存中；

[0027] 清除所述第一通用寄存器中的值。

[0028] 优选地，在步骤c中，所述函数尾的处理流程包括如下步骤：

[0029] 将保存在所述TCB中的影子栈的栈顶指针加载到所述第一通用寄存器中；

[0030] 将所述影子栈的返回地址保存到第二通用寄存器中；

[0031] 判断所述第一通用寄存器中的值是否为零，如果是，则调用abort函数终止运行，如果否，则执行下一步骤；

[0032] 将所述第一通用寄存器中的值减四；

[0033] 比较所述第二通用寄存器中的值和所述第一通用寄存器加四后所指向内存中的值是否相等，如果否，则返回判断所述第一通用寄存器中的值是否为零的步骤，如果是，则将所述第一通用寄存器中的值保存在所述TCB中，并返回主调函数。

[0034] 优选地，在步骤d中，所述线程退出处理的流程具体包括如下步骤：

[0035] 启动所述线程的退出程序；

[0036] 调用和所述动态共享库中TSD变量thread_cleanup_key相关联的析构函数；

[0037] 判断所述析构函数的参数是否小于销毁所述TSD变量的最大尝试次数，如果是，则将所述析构函数的参数加1，并返回调用和所述动态共享库中TSD变量thread_cleanup_key相关联的析构函数步骤；如果否，则执行销毁所述线程的影子栈步骤。

[0038] 本发明的有益效果在于：所述用于多线程后向控制流完整性保护的影子栈实现方法中，将影子栈技术扩展到多线程情况，而且，不需要修改源代码，方便部署。通过在线程粒度实现影子栈，将影子栈的栈顶指针存储在线程控制块中，改进了影子栈的隐蔽性，既有利于方便快捷地访问，又避免了被泄露和篡改的威胁。因此，本方法可提高多线程C/C++程序的控制流完整性保护能力，以降低受到代码复用攻击的威胁。

附图说明

[0039] 图1是本发明实施例提供的用于多线程后向控制流完整性保护的影子栈实现方法的流程图；

[0040] 图2是图1所示的用于多线程后向控制流完整性保护的影子栈实现方法中影子栈部署流程图；

[0041] 图3是图1所示的用于多线程后向控制流完整性保护的影子栈实现方法中线程启动劫持流程图；

[0042] 图4是图1所示的用于多线程后向控制流完整性保护的影子栈实现方法中线程退出处理流程图；

[0043] 图5是图1所示的用于多线程后向控制流完整性保护的影子栈实现方法中函数头处理的插装流程图；

[0044] 图6是图1所示的用于多线程后向控制流完整性保护的影子栈实现方法中函数尾处理的插装流程图；

[0045] 图7是图1所示的用于多线程后向控制流完整性保护的影子栈实现方法中函数头处理流程图；

[0046] 图8是图1所示的用于多线程后向控制流完整性保护的影子栈实现方法中函数尾处理流程图。

具体实施方式

[0047] 为了使本发明的目的、技术方案及优点更加清楚明白，以下结合附图及实施例，对本发明进行进一步详细说明。应当理解，此处所描述的具体实施例仅仅用以解释本发明，并不用于限定本发明。

[0048] 除非上下文另有特定清楚的描述，本发明中的元件和组件，数量既可以单个的形式存在，也可以多个的形式存在，本发明并不对此进行限定。本发明中的步骤虽然用标号进行了排列，但并不用于限定步骤的先后次序，除非明确说明了步骤的次序或者某步骤的执行需要其他步骤作为基础，否则步骤的相对次序是可以调整的。可以理解，本文中所使用的术语“和/或”涉及且涵盖相关联的所列项目中的一者或一者以上的任何和所有可能的组合。

[0049] 请同时参阅图1、图2、图3、图4、图5、图6、图7和图8，本发明实施例提供的用于多线程后向控制流完整性保护的影子栈实现方法将影子栈的栈顶指针定义为隐式局部存储变量，方便为每个线程创建和销毁影子栈。在创建影子栈后，将栈顶指针存入线程控制模块(pthread)中，方便插装代码访问影子栈，同时又避免影子栈被泄露和篡改。每次更新影子栈时，同时也要更新线程控制模块中的影子栈栈顶指针。

[0050] 而且，所述用于多线程后向控制流完整性保护的影子栈实现方法主要包括：线程劫持的动态共享库实现和后向控制流完整性检查的编译器插装。

[0051] 所述线程劫持的动态共享库实现，是对动态共享库中线程创建函数的改进。在原来的线程创建函数工作之前执行创建影子栈的工作；同时，对线程的初始例程进行包装，在包装函数中利用线程局部存储机制隐藏影子栈的位置。从而，在线程创建时，实现了劫持线程的创建和启动。其中的关键操作为线程启动劫持和线程退出处理；

[0052] 所述后向控制流完整性检查的编译器插装，通过修改LLVM后端中与代码生成相关的函数来实现。分别在emitPrologue函数的开头和emitEpilogue函数的末尾，通过LLVM提供的API插入相关基本块和指令，以使得插装的函数头处理和函数尾处理在函数执行时实现对代码的后向控制流完整性检查和保护。其中的关键操作为函数头处理的插装、函数尾处理的插装，以及被插装的运行时完成检查保护的函数头处理功能和函数尾处理功能。

[0053] 具体地，所述用于多线程后向控制流完整性保护的影子栈实现方法具体包括如下步骤：

[0054] 一、对待保护程序的源码进行编译时插装，生成部署了影子栈的可执行文件。

[0055] 如图2所示，具体地，步骤一具体包括如下步骤：

[0056] 目标源代码通过LLVM编译器的前端Clang转换成中间语言IR；

[0057] 所述中间语言IR经中间语言层的优化后进入编译器后端；

[0058] 所述编译器后端为所述中间语言IR选择目标平台的具体指令，并将所述中间语言IR线性化，然后在PEI pass生成所述函数头和所述函数尾时进行插装，接着发射为目标文件，最后和动态共享库共同链接成部署了影子栈的可执行文件。

[0059] 需要说明的是，在本实施例中，选择在生成函数头和函数尾时进行插装，并利用新的函数头和函数尾负责影子栈的维护和后向控制流完整性的检查。为了使影子栈技术在不需要修改源代码的情况下扩展至多线程，而且，本发明实施例实现了一个动态共享库，在其中使用线程劫持技术在线程启动之前创建影子栈，同时利用线程局部存储机制隐藏影子栈的栈顶指针。最终通过链接器将目标文件和线程劫持所述动态共享库链接成可执行文件。

[0060] 二、启动部署了影子栈的可执行文件，由动态共享库劫持线程的创建和启动，完成创建影子栈。

[0061] 在所述步骤二中，部署了影子栈的可执行文件启动后，由所述动态共享库劫持线程的创建和启动，并完成创建所述影子栈的工作。

[0062] 而且，所述步骤二具体包括如下步骤：

[0063] 调用所述动态共享库内的构造函数，并在所述构造函数中创建主线程的影子栈；

[0064] 将所述主线程的影子栈的栈顶指针保存到线程控制模块中；

[0065] 调用主程序的构造函数，并启动主线程；

[0066] 调用伪线程的创建函数，并在所述伪线程的创建函数中创建所述伪线程的影子栈；

[0067] 在所述伪线程的创建函数中调用原线程的创建函数，并执行包装例程；

[0068] 将所述伪线程的影子栈的栈顶指针保存到所述线程控制模块中，并执行原始例程。

[0069] 如图3所示，具体地，所述线程启动劫持流程中，步骤30是起始状态，表示内核执行完execve系统调用后回到用户态的动态链接器。步骤31调用线程劫持共享库中的构造函数。步骤32在上一步调用的构造函数中创建主线程的影子栈。接着步骤33将影子栈栈顶指针保存到线程控制模块中。所述线程控制模块的第一个成员是一个union元素，一共24*4个字节，其中开头的是tcbhead_t结构体，一共14*4个字节，剩余部分是用于填充的填充部分。因此可以将所述影子栈的栈顶指针的值放在剩余的所述填充部分内。在本实施例中，将所述影子栈的栈顶指针的值选择放在16*4＝0x40处。步骤34调用主程序的构造函数。步骤35

启动主线程。步骤36在程序创建线程时，劫持真正的线程创建函数，先进入伪线程创建函数。步骤37在伪线程创建函数中创建影子栈。步骤38在伪线程创建函数中调用原线程创建函数并执行包装例程函数。在本实施例中，将包装之后的start_routine和start_routine_arg参数传给原线程创建函数，其中start_routine被包装成thread_start函数，start_routine_arg被包装成一个结构体{shadow_stack_start,start_routine,start_routine_arg}。shadow_stack_start是该线程影子栈的起始地址，start_routine_arg是原来传给start_routine的参数。步骤39在包装例程thread_start函数中将影子栈的栈顶指针保存到线程控制模块中以便于插装代码访问。步骤3a接受所述结构体作为输入参数，通过start_routine(start_routine_arg)调用程序原本想要执行的函数。步骤3b为结束状态，表示开始运行新线程。

[0070] 三、插装后的函数头和函数尾实现所述影子栈的栈顶指针的维护和后向控制流完整性的检查。

[0071] 具体地，在所述步骤三中，在所述LLVM编译器的后端生成所述函数头和所述函数尾时进行插装，PEI pass负责生成所述函数头和所述函数尾，并在runOnMachineFunction函数中调用insertPrologEpilogCode函数；且所述insertPrologEpilogCode函数调用emitPrologue函数生成所述函数头，并调用emitEpilogue函数生成所述函数尾。

[0072] 如图5所示，所述函数头处理的插装流程中，步骤50是起始状态。步骤51取SaveBlocks中的第一个基本块。步骤52取得add32mi8指令的操作码。由于是内存间接寻址，故需要步骤53设置该指令的目标操作数，其中段前缀为gs，偏移量为0x40，基址寄存器为空，index＝0，scale＝1。步骤54设置作为源操作数的立即数为4。步骤55调用BuildMI函数将该指令插入到原基本块的第一条指令之前。步骤56取得mov32rm指令的操作码。步骤57设置源操作数的段前缀为gs，偏移为0x40，基址寄存器为空，index＝0，scale＝1。步骤58设置目标操作数为eax，接着执行步骤59，作用和步骤55相同。步骤5a取得mov32rm指令的操作码。步骤5b设置源操作数的段前缀为空，偏移为0，基址寄存器为esp，index＝0，scale＝1。步骤5c设置目标操作数为ecx。接着执行步骤5d，作用同步骤55。步骤5e取得mov32mr指令的操作码。步骤5f设置目标操作数的段前缀为空，偏移为0，基址寄存器为eax，index＝0，scale＝1。步骤5g设置源操作数为ecx。接着执行步骤5h，作用同步骤55。步骤5i为结束状态，表示开始执行emitPrologue函数中的其他原有工作。

[0073] 如图6所示，所述函数尾处理的的插装流程中，步骤60是起始状态。步骤61取RestoreBlocks中的第一个基本块。步骤62判断该基本块是否为RestoreBlocks中的最后一个基本块。若不是，跳转到步骤63，取RestoreBlocks中的下一个基本块，并再次回到步骤62。若是，则跳转到步骤64，创建ABORT基本块并设置序号。步骤65将ABORT基本块设置为自身的后继。步骤66将ABORT基本块插入到当前函数的最后。步骤67创建no_match基本块并设置序号。步骤68将no_match添加到自身的后继中。步骤69在原基本块中插入相关指令，并取得分割点。步骤6a创建一个新的空基本块。步骤6b将空基本块插入到原基本块之后。步骤6c将no_match基本块插入到空基本块之前。步骤6d将原基本块分割点之后的指令转移到空基本块中。步骤6e将原基本块的后继全部转移给空基本块。步骤6f将no_match基本块添加到原基本块的后继中。步骤6g将空基本块添加到no_match基本块的后继中。步骤6h为结束状态。

[0074] 而且，在本实施例中，在步骤三中，所述函数头的处理流程包括如下步骤：

[0075] 进入被调函数，并将保存在TCB中的影子栈的栈顶指针加四；

[0076] 将保存在所述TCB中的影子栈的栈顶指针加载到第一通用寄存器中；

[0077] 将所述影子栈的返回地址保存到第二通用寄存器中；

[0078] 将所述第二通用寄存器的值保存到所述第一通用寄存器所指向的内存中；

[0079] 清除所述第一通用寄存器中的值。

[0080] 如图7所示，所述函数头的处理流程中，步骤70是起始状态。步骤71进入被调函数。步骤72将保存在TCB(即pthread结构体)中的影子栈栈顶指针加四。步骤73将保存在TCB中的影子栈栈顶指针加载到通用寄存器A(即所述第一通用寄存器)中。步骤74将栈上的返回地址保存到通用寄存器B中。步骤75将寄存器B(即所述第二通用寄存器)中的值保存到寄存器A所指向的内存中。步骤76清除寄存器A中的值。步骤77为结束状态，表示即将开始原有函数头中的其他工作。

[0081] 而且，所述函数尾的处理流程包括如下步骤：

[0082] 将保存在所述TCB中的影子栈的栈顶指针加载到所述第一通用寄存器中；

[0083] 将所述影子栈的返回地址保存到第二通用寄存器中；

[0084] 判断所述第一通用寄存器中的值是否为零，如果是，则调用abort函数终止运行，如果否，则执行下一步骤；

[0085] 将所述第一通用寄存器中的值减四；

[0086] 比较所述第二通用寄存器中的值和所述第一通用寄存器加四后所指向内存中的值是否相等，如果否，则返回判断所述第一通用寄存器中的值是否为零的步骤，如果是，则将所述第一通用寄存器中的值保存在所述TCB中，并返回主调函数。

[0087] 如图8所示，所述函数尾的处理流程中，步骤80为起始状态。步骤81将保存在TCB中的影子栈栈顶指针加载到通用寄存器A(即所述第一通用寄存器)中。步骤82将栈上的返回地址保存到另一个通用寄存器B(即所述第二通用寄存器)中。步骤83判断寄存器A中的值是否为0。如果是，跳转到步骤84，调用abort函数终止运行，然后跳转到结束状态8a。如果不是，则跳转到步骤85，将A的值减四。步骤86比较B中的值和A+4所指内存中的值。步骤87判断两者是否相等，如果不相等，跳转到步骤83。否则，跳转到步骤88，将A中的值保存到TCB中。步骤89返回主调函数。步骤8a为结束状态。

[0088] 四、进行所述线程的退出处理，并通过所述动态共享库中注册的析构函数对所述影子栈进行销毁。

[0089] 具体地，在步骤四中，所述线程退出处理的流程具体包括如下步骤：

[0090] 启动所述线程的退出程序；

[0091] 调用和所述动态共享库中TSD变量thread_cleanup_key相关联的析构函数；

[0092] 判断所述析构函数的参数是否小于销毁所述TSD变量的最大尝试次数，如果是，则将所述析构函数的参数加1，并返回调用和所述动态共享库中TSD变量thread_cleanup_key相关联的析构函数步骤；如果否，则执行销毁所述线程的影子栈步骤。

[0093] 如图4所示，步骤40是起始状态。步骤41表示线程执行完毕，即将退出。步骤42调用和动态共享库中TSD变量thread_cleanup_key相关联的析构函数。步骤43判断该析构函数的参数是否小于销毁TSD变量的最大尝试次数。若是，则跳转到步骤44，将参数加1，并返回

至步骤42。否则跳转到步骤45，销毁该线程的影子栈。步骤46为结束状态。

[0094] 相较于现有技术，本发明提供的用于多线程后向控制流完整性保护的影子栈实现方法中，将影子栈技术扩展到多线程情况，而且，不需要修改源代码，方便部署。通过在线程粒度实现影子栈，将影子栈的栈顶指针存储在线程控制块中，改进了影子栈的隐蔽性，既有利于方便快捷地访问，又避免了被泄露和篡改的威胁。因此，本方法可提高多线程C/C++程序的控制流完整性保护能力，以降低受到代码复用攻击的威胁。

[0095] 对于本领域技术人员而言，显然本发明不限于上述示范性实施例的细节，而且在不背离本发明的精神或基本特征的情况下，能够以其他的具体形式实现本发明。因此，无论从哪一点来看，均应将实施例看作是示范性的，而且是非限制性的，本发明的范围由所附权利要求而不是上述说明限定，因此旨在将落在权利要求的等同要件的含义和范围内的所有变化囊括在本发明内。不应将权利要求中的任何附图标记视为限制所涉及的权利要求。

[0096] 此外，应当理解，虽然本说明书按照实施方式加以描述，但并非每个实施方式仅包含一个独立的技术方案，说明书的这种叙述方式仅仅是为清楚起见，本领域技术人员应当将说明书作为一个整体，各实施例中的技术方案也可以经适当组合，形成本领域技术人员可以理解的其他实施方式。

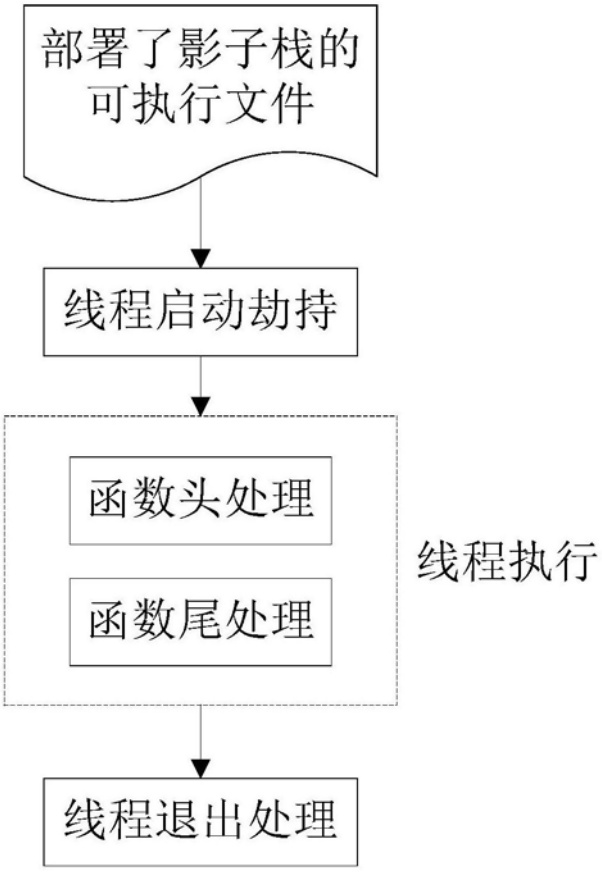


图1

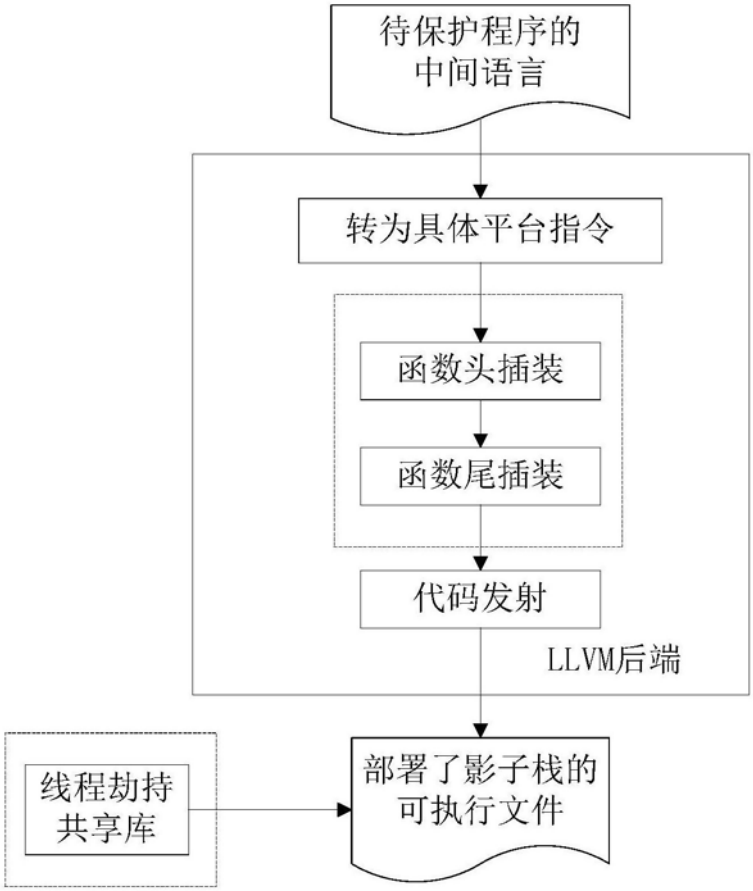

待保护程序的
中间语言
转为具体平台指令
函数头插装
函数尾插装
代码发射
LLVM后端
线程劫持
共享库
部署了影子栈的
可执行文件

图2

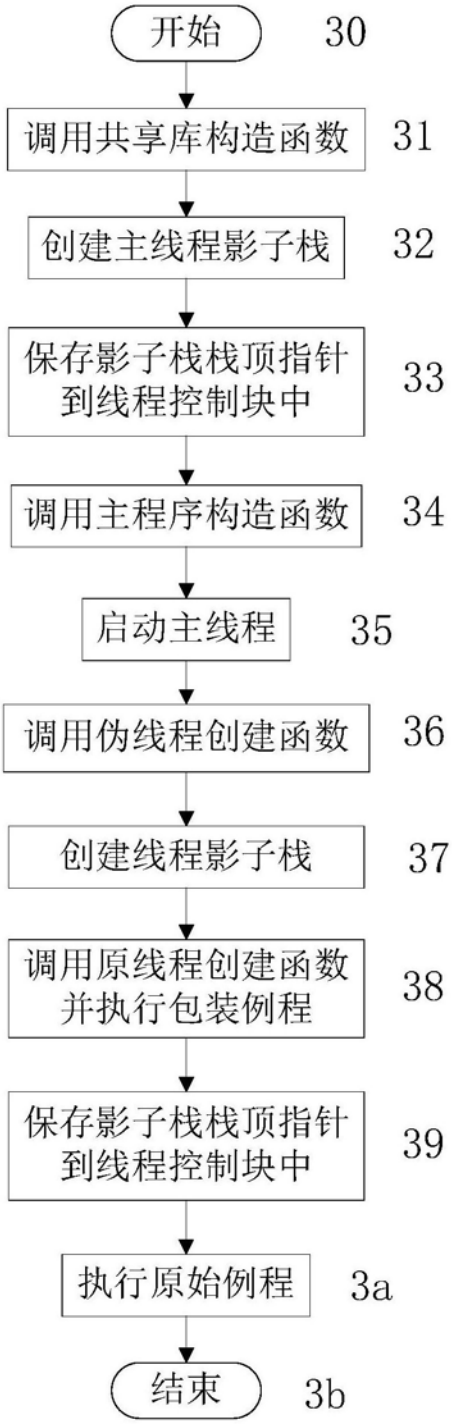

开始
30
调用共享库构造函数
31
创建主线程影子栈
32
保存影子栈栈顶指针到线程控制块中
33
调用主程序构造函数
34
启动主线程
35
调用伪线程创建函数
36
创建线程影子栈
37
调用原线程创建函数并执行包装例程
38
保存影子栈栈顶指针到线程控制块中
39
执行原始例程
3a
结束
3b

图3

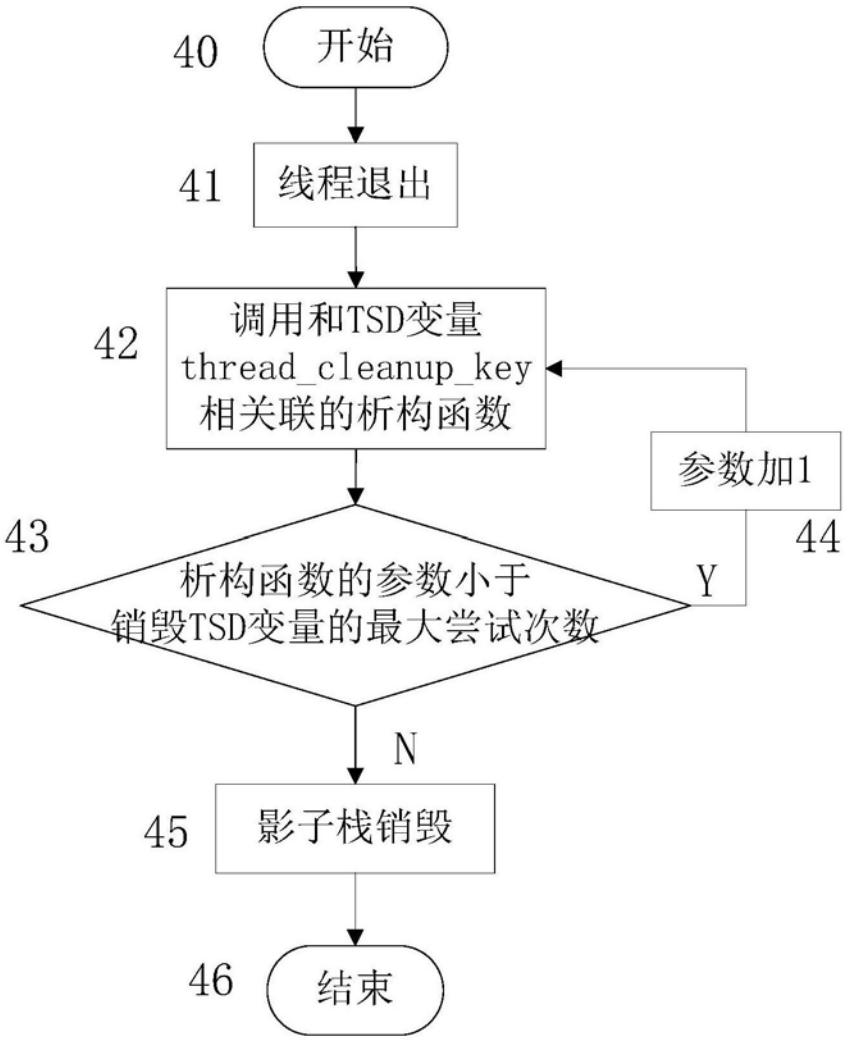

40
开始
41
线程退出
42
调用和TSD变量
thread_cleanup_key
相关联的析构函数
参数加1
44
43
析构函数的参数小于
销毁TSD变量的最大尝试次数
Y
N
45
影子栈销毁
46
结束

图4

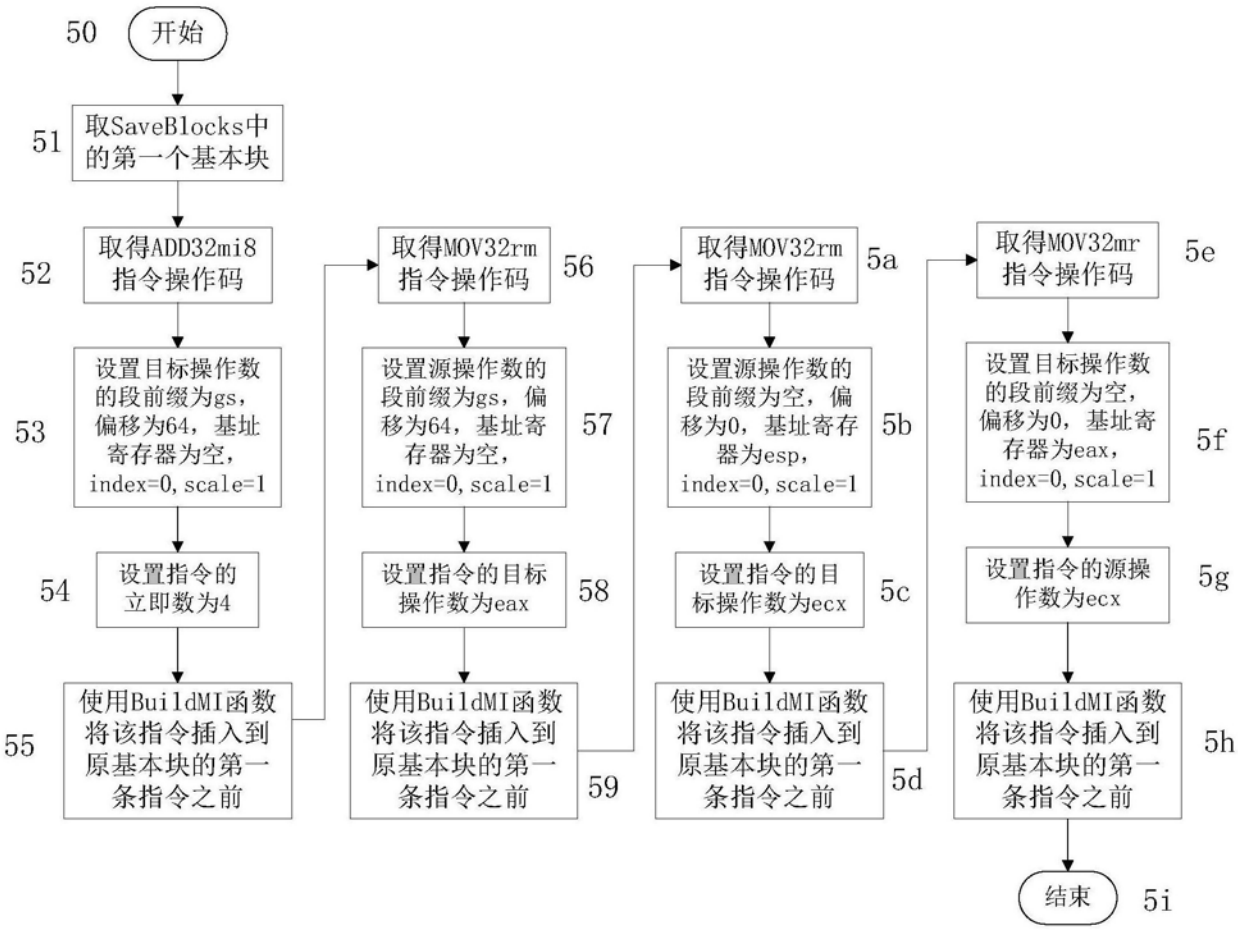

50
开始
51
取SaveBlocks中的第一个基本块
52
取得ADD32mi8指令操作码
53
设置目标操作数的段前缀为gs，偏移为64，基址寄存器为空，index=0, scale=1
54
设置指令的立即数为4
55
使用BuildMI函数将该指令插入到原基本块的第一条指令之前
56
取得MOV32rm指令操作码
57
设置源操作数的段前缀为gs，偏移为64，基址寄存器为空，index=0, scale=1
58
设置指令的目标操作数为eax
59
使用BuildMI函数将该指令插入到原基本块的第一条指令之前
5a
取得MOV32rm指令操作码
5b
设置源操作数的段前缀为空，偏移为0，基址寄存器为esp，index=0, scale=1
5c
设置指令的目标操作数为ecx
5d
使用BuildMI函数将该指令插入到原基本块的第一条指令之前
5e
取得MOV32mr指令操作码
5f
设置目标操作数的段前缀为空，偏移为0，基址寄存器为eax，index=0, scale=1
5g
设置指令的源操作数为ecx
5h
使用BuildMI函数将该指令插入到原基本块的第一条指令之前
5i
结束

图5

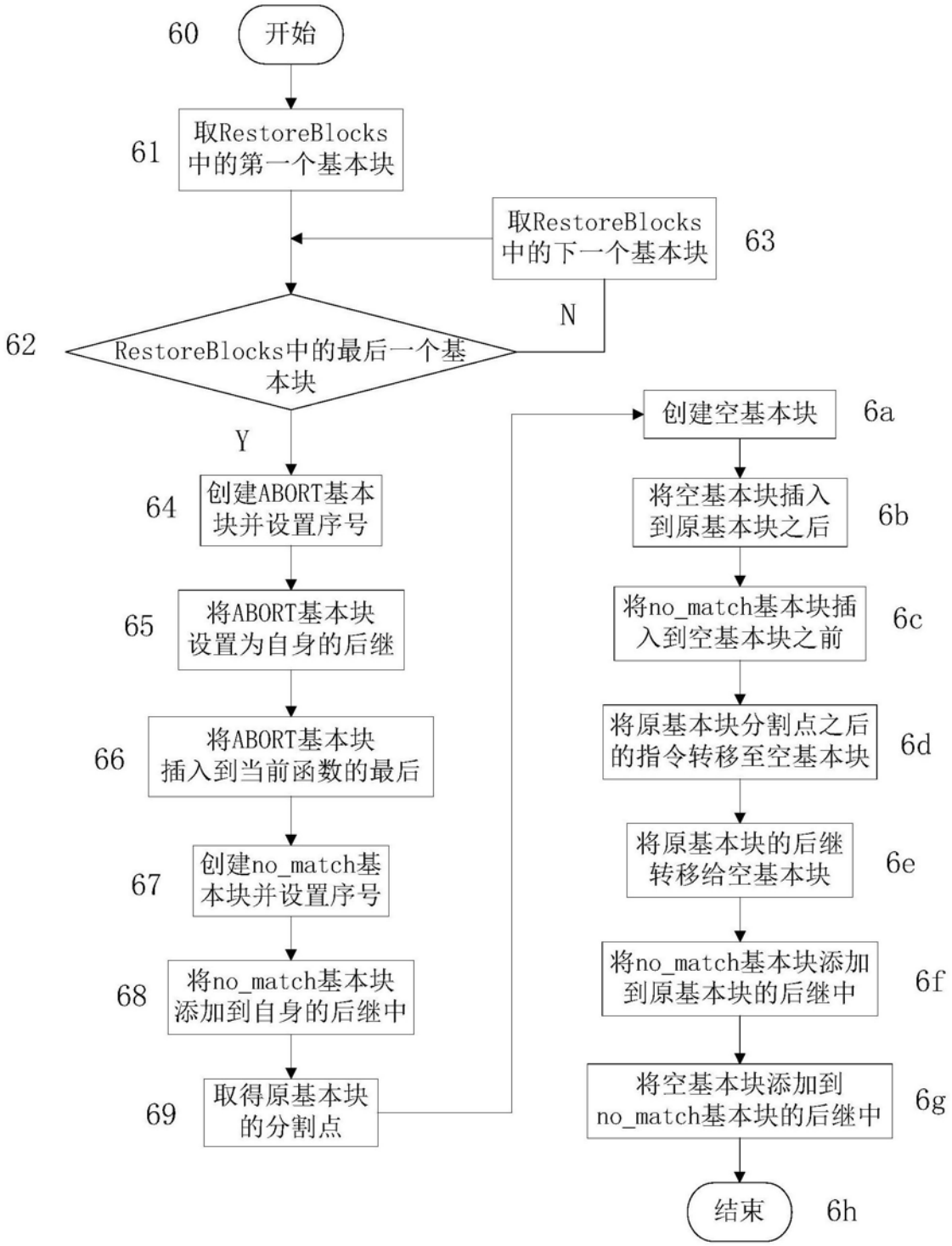

60
开始
61
取RestoreBlocks中的第一个基本块
63
取RestoreBlocks中的下一个基本块
N
62
RestoreBlocks中的最后一个基本块
Y
64
创建ABORT基本块并设置序号
65
将ABORT基本块设置为自身的后继
66
将ABORT基本块插入到当前函数的最后
67
创建no_match基本块并设置序号
68
将no_match基本块添加到自身的后继中
69
取得原基本块的分割点
6a
创建空基本块
6b
将空基本块插入到原基本块之后
6c
将no_match基本块插入到空基本块之前
6d
将原基本块分割点之后的指令转移至空基本块
6e
将原基本块的后继转移给空基本块
6f
将no_match基本块添加到原基本块的后继中
6g
将空基本块添加到no_match基本块的后继中
结束
6h

图6

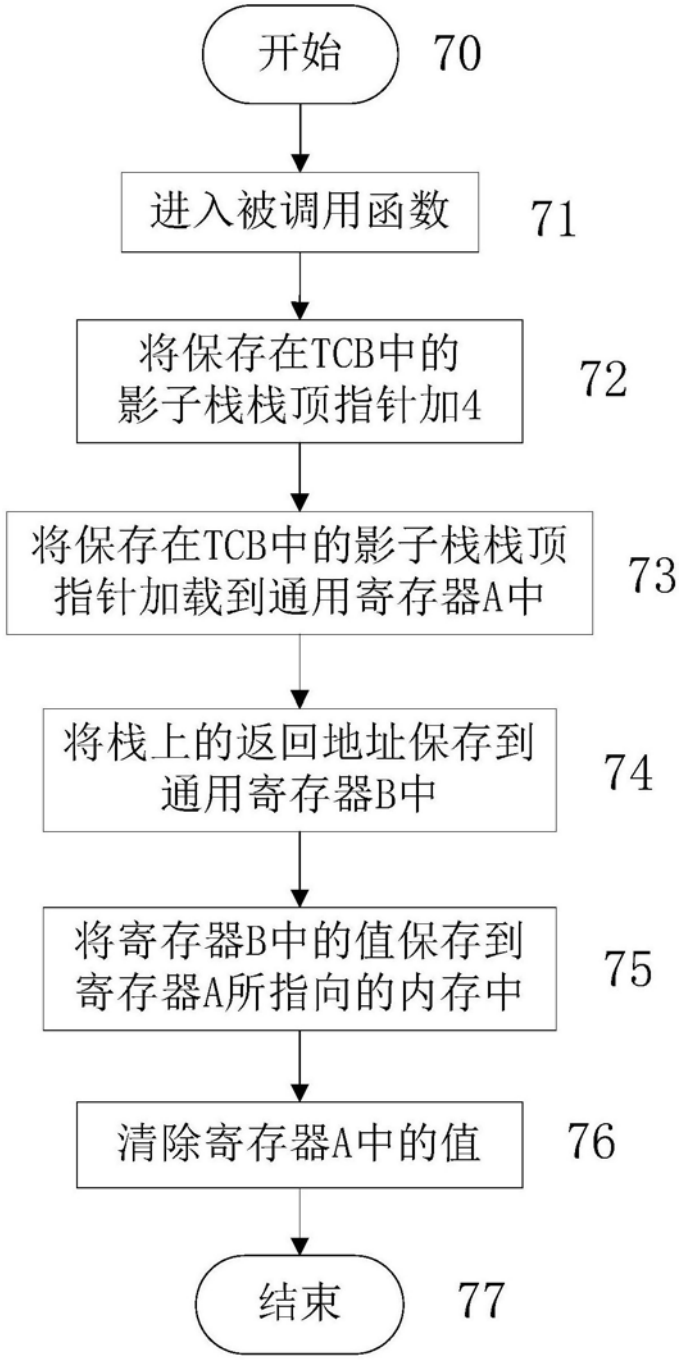

开始
70
进入被调用函数
71
将保存在TCB中的
影子栈栈顶指针加4
72
将保存在TCB中的影子栈栈顶
指针加载到通用寄存器A中
73
将栈上的返回地址保存到
通用寄存器B中
74
将寄存器B中的值保存到
寄存器A所指向的内存中
75
清除寄存器A中的值
76
结束
77

图7

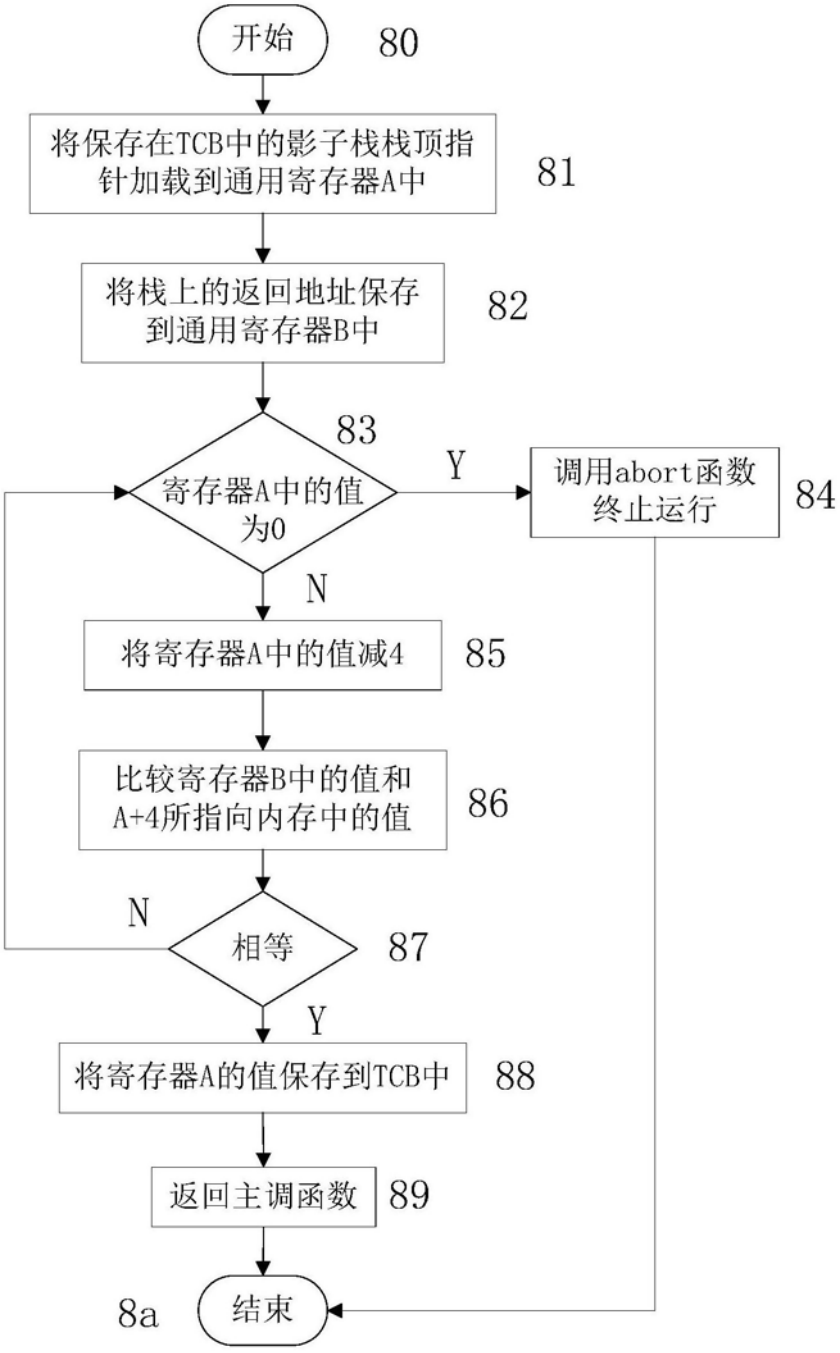

开始
80
将保存在TCB中的影子栈栈顶指针加载到通用寄存器A中
81
将栈上的返回地址保存到通用寄存器B中
82
83
寄存器A中的值为0
Y
调用abort函数终止运行
84
N
将寄存器A中的值减4
85
比较寄存器B中的值和A+4所指向内存中的值
86
N
相等
87
Y
将寄存器A的值保存到TCB中
88
返回主调函数
89
8a
结束

图8