针对内核模块访存错误的内存检测方法①

纪 程[1], 陈香兰[1,2], 李 曦[1,2]

[1](中国科学技术大学 计算机科学与技术学院, 合肥 230027)
[2](中国科学技术大学 苏州研究院, 苏州 215123)

摘 要: 分析了 Linux 内核模块特点, 针对内核模块中二进制指令执行时带来的访存错误, 设计了一种针对内核模块的静态检测方法. 通过模拟内核模块中指令的执行, 并比较访存指令请求与相关内存区域信息, 静态检测方法目标是找出代码对内存的非法访问, 并对可疑的访存行为发出警告. 针对 ARM 处理器平台, 给出了静态检测方法的具体实现, 并对内核模块中的访存错误就行了检测验证. 实验表明, 静态检测方法能够有效找出包括地址越界访问、读未初始化内存、访问已释放内存等访存错误, 本文的静态检测方法达到了预期的检测效果.

关键词: Linux 内核模块; 内存访问; 静态检测; 指令模拟

Memory Detecting Method for Access Errors Inside Kernel Modules

JI Cheng[1], CHEN Xiang-Lan[1,2], LI Xi[1,2]

[1](College of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
[2](Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123, China)

Abstract: This paper studies the memory access characteristics of Linux kernel module, and then suggests a static approach to detect memory access errors when running the code inside kernel module. Through simulation of instructions inside kernel module and comparison between access request and record of memory region, our suggested method is able to find a variety of memory access errors, such as overflow, accessing arbitrary address, invalid pointer, which are the most common threats. At last, based on ARM Processor platform, we give an implementation of our method for detecting errors induced by kernel module, and the experiment result shows it can efficiently work as we expect.

Key words: Linux kernel module; memory access; static detection method; instruction simulation

1 引文

Linux 系统中使用内核模块以无缝的方式, 在需要时被链接入内核. 这些模块可以被动态加载进入内核成为可加载内核模块 LKM(loadable kernel module), 如设备驱动模块[2]. 如果用户想添加一些新的功能到内核中去, 用户希望编译新的代码成为内核模块, 只需要加载这些新的模块到内核中去. 这样, 既可以减少内核的体积, 又能够不用和其他众多的模块再次重新编译降低效率, 如图 1.

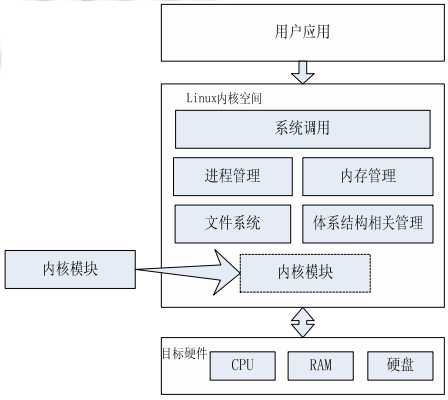

图 1 内核模块和内核关系

虽然内核模块为用户提供了方便, 但内核模块常以二进制文件发布, 用户难以得到其源代码, 因此, 如何保证其可靠性是一个关键的问题. 相比与普通用户程序, 运行在特权模式下的内核模块, 拥有 Linux 操作系统最高特权级, 可以访问内核数据结构和内核系统调用. 一旦内核模块出现问题, 更容易造成整个系

① 收稿时间:2014-04-01;收到修改稿时间:2014-04-25

统的崩溃, 威胁操作系统的可靠性. 由于 C 语言、汇编语言支持显示的内存分配操作, Linux 操作系统中始终存在着非法的内存访问风险. 严重的非法地址访问, 会使整个操作系统崩溃[10].

针对上述问题, 在可靠性要求较高嵌入式环境中, 本文提出了一种用于检测内核模块(二进制文件)的内存非法访问方法, 随后, 在 ARM 处理器平台进行了实现和验证.

2 相关工作

为了检测内存访问错误, 动态的内存检测方法(在线检测)是可选的方法之一. 动态内存检测在程序实际运行时被执行. 动态检测工具的例子包括 Purify[7]、CCured[6]、Stack-Guard[9]等. 例如, Stack Guard[9]只关注栈溢出错误, 而不去关注其他类型的内存访问错误. 动态检测方法可以实施在真实的系统中, 因此, 它对于可疑错误的验证非常有效. 但是, 这种方法会带来很大的开销. 比如, Purify[7]需要截获每条访存指令, 这些额外的操作无疑会带来巨大的时间开销.

另一种方法为静态检测. 静态检测方法试图通过静态分析的方法找到访存错误. 例如, LCLint[8]是一种基于注视辅助的轻量级静态检测工具, 它利用针对源代码的语法标记来检测可能的缓冲区溢出. ARCHER[4]是一种用于检测大型源码工程的静态检测工具, 但是, 它不能用于检测二进制文件. 针对源代码的访存检测相比于针对二进制文件的检测, 需要花费更多时间用于解析代码. MLAB[1]方法将二进制代码恢复为与机器无关的编译中间表示形式, 进而恢复控制流和数据流信息, 并基于模型检测算法检测内存泄露问题, 但其难以检测其他内存非法访问问题.

本文结合 Linux 内核特点, 设计了一种针对内核模块的访存检测方法. 这种检测方法, 通过模拟模块中二进制指令行为, 试图检测出确定的访存错误并对疑似错误发出警告.

3 静态检测方法设计与实现

本节介绍了内核模块的静态检测方法, 并给出了嵌入式 ARM 平台上静态检测方法的实现.

3.1 静态检测分析

静态检测的目标是通过模拟内核模块内指令行为, 进而分析每条访存指令的正确性, 以确定这些指令属于错误、正确、警告三种状态的哪一种. 为了检测出内核模块中的非法访存指令, 我们需要得到模块中、操作系统中的关键数据, 并对其进行记录和处理. 静态分析主要检测出以下访存错误:

① 内存的边界溢出;

② 读未初始化内存;

③ 读、写已释放内存.

3.1.1 静态检测概要过程

图 2 给出了静态检测的概要过程, 其主要包括以下几个步骤:

① 反汇编

利用二进制代码和汇编语言对应关系, 进行反汇编操作.

② 信息分析

记录二进制文件中的全局符号; 利用得到的汇编语言进行内部函数基本块的划分, 为后续控制流分析做准备;

③ 执行路径恢复

通过计算下一条指令的位置, 恢复程序的控制流.

④ 程序模拟

从指令寄存器中得到当前指令, 模拟执行每条指令, 并对上下文进行更新.

⑤ 访存检测

如果当前指令为访存指令, 利用访存检测算法, 检测访存指令是否合法.

⑥ 结果分析

分析检测结果, 帮助用户定位, 错误(error)或者警告(warning)的访存指令位置.

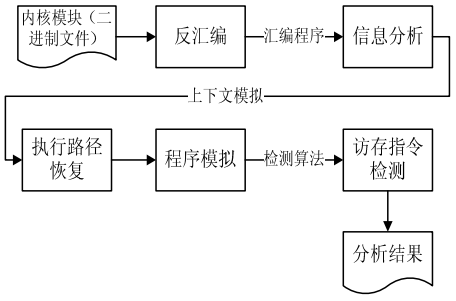

图 2 内核模块和内核关系

3.1.2 二进制文件分析

内核模块文件(*.ko)属于二进制文件, 通过分析二进制文件结构, 能够获得必要的信息. 二进制文件中

的所有段被存储在文件头和段头表中间的位置. 通过分析段信息, 得到:

① 被访问的函数

我们把要分析的函数分为两种类型: 内部函数和外部函数. 内部函数即在模块内被定义的函数, 这些函数通常是全局函数, 包括模块初始化函数、模块退出函数以及其他全局函数.

对于外部函数, 需要关注和内存访问相关的关键性函数, 如 kmem_cache_alloc, 因为类似的函数会给整个内存分配带来影响.

② 被访问的变量、常量

在检查是否出现内存上溢出、下溢出时, 有时需要找到表示内存地址、内存偏移量外部变量.

由模块内部定义的全局量称为内部量. 这些变量可以被内核以及其他的内核模块访问. 大多数时候, 内部变量被本模块之外的函数访问时, 其可靠性由访问它的函数保证. 但是, 如果检测模块内包含未初始化的变量, 仍然可以对其发出警告即使当前模块并未访问此未初始化变量.

3.2 执行路径恢复

在静态分析中, 为了模拟程序的运行, 需要获得程序的执行路径, 即获得程序的控制流.

3.2.1 基本块划分

基本块是除了最后一个跳转指令之外, 内部无任何分支指令的代码块. 每个模块内部函数将被分为数个基本块. 对于外部函数, 为了简化, 我们认为其只有一个基本块. 在对二进制文件分析时, 要找到每个函数的基本块, 首先需要定位到函数的位置以及函数的边界. 从符号表中, 能够得到模块内函数的起始地址和大小. 图 3 为基本块划分过程的流程图.

3.2.2 控制流恢复

本文的静态检测方法通过跟踪 PC/LR 寄存器中值, 模拟程序的走向. 对程序控制流的跟踪, 也是对 PC 寄存器值的跟踪, 通过跟踪程序计数器 PC 寄存器值得变化, 能够获得程序的控制流.

① 顺序指令

在顺序执行情况下, 每执行一条指令, PC 的值会被加 4. 如果遇到分支指令, 情况将变得复杂一些.

② 分支指令

1.调用外部函数

考虑到函数会正常返回, 那么 PC 值仍然进行加 4 操作. 而如果遇到绝对跳转指令跳入外部命名函数, 如 b 跳转指令, 那么除了模拟被调用的函数功能外, 还需考虑到跳转之后返回的位置. 这时, lr 寄存器会保存函数返回的地址, PC 的值是 lr 寄存器值的一个拷贝.

2.调用内部函数

如果遇到内部函数的调用, PC 的值将会切换到内部函数第一个基本块的地址. 这时, lr 寄存器值变为 PC+4, 供控制流在从当前内部函数返回时恢复 PC 使用.

3.从栈中恢复 PC

还有一类特殊的分支指令, 即 pop 指令. 当出现 pop {PC}指令时, PC 寄存器将会变为当前运行时栈栈顶的值. 这一类分支指令的用于多层的函数调用时保存外层函数的返回地址.

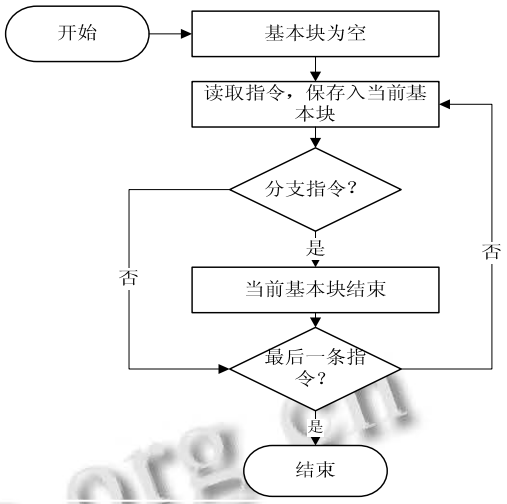

图 3　基本块划分过程

3.3 程序上下文

我们定义模拟时上下文 context 为 C=(R, M, S), 其中 R 为寄存器上下文, M 为内存区域记录, S 为栈.

① 寄存器上下文

寄存器上下文是有一组与汇编代码运行相关的寄存器组, 寄存器上下文在程序运行时用来保存局部变量, 传递参数, 保存数据等. 本文对内存模块的检测方法, 应用在单任务环境下, 即不考虑任务切换带来的上下文变化, 以简化设计. ARM 平台下, 寄存器上下文包括: R1 至 R15, 共 16 个寄存器[3]. 我们定义寄存器上下文 R={R1...R16}.

寄存器中的数据无法预先确定是普通数据或程序地址, 随着指令的模拟执行, 我们记录寄存器值、值类

型的变化. 对于每一个寄存器, 其上下文包括寄存器值、值类型、指向内存区域. 对于特殊寄存器, 如 PC、LR 寄存器, 还需记录其代码所属的程序段位置. 随着程序的执行, 寄存器的值随之发生改变. 由于访存地址会由寄存器给出, 我们需要知道寄存器中数值的类型, 即判断寄存器值是否为一合法地址或其他值. 寄存器值可以是代码类型、数据类型、地址类型或 BSS 类型, $M=\{m_1...m_n\}$.

② 关联内存区域

在静态检测过程中, 为了检测访存指令的正确与否, 我们需要记录并更新与之相关联的内存区域信息. 内存区域信息包括区域其实地址、大小、状态等. 这里, 采用静态链表的方法, 保存每条内存区域信息, 这里, 定义内存区域上下文 M={m_1, m_2, ... m_n} 这里的内存区域, 既包括从符号表中得到的全局内存, 也包括模块内管理的内存区域. 内存区域的状态有: 未初始化、已写入、已释放、未知状态等.

我们用 Region 结构体来记录一个在模块内开辟的内存区域, 其结构如图 4 所示.

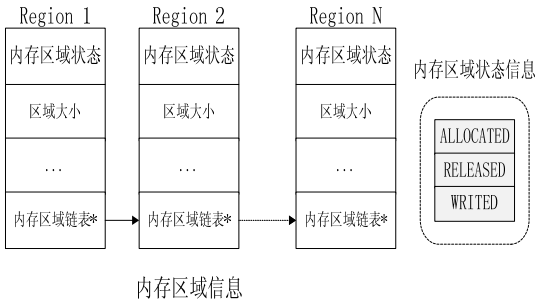

图 4 内存区域上下文

③ 模拟栈

本文的内核模块设定为运行在单任务环境下, 即只有一个栈 Stack. 静态模拟时, 函数会将局部变量保存到栈中去. 所有暂时保存入栈中的寄存器上下文, 会在后续从栈中恢复. 编译器会把一些寄存器保存到栈中, 例如, 对于 PC 寄存器的压栈与出栈, 如果无法模拟栈上下文, 将难以得到程序位置, 进而模拟程序失败.

3.4 指令模拟

本文的检测方法, 通过模拟指令行为, 采用类似于“沙盒”方法, 在程序加载到真实系统之前, 检测访存指令是否出错.

① 汇编指令模拟

我们给每一条 ARM 汇编提供一个处理例程, 用于模拟指令的执行以及更新对上下文 C. 值得注意的是, 相比与其他复杂的指令集, 用于嵌入式环境下 ARM 指令种类相对精简, 检测时对其行为的模拟时可行的. 模拟每条汇编指令, 除了函数调用指令以外, 在 ARM 处理器上都有一个预定义的输入和输出. 例如, 输入的的指令

STRB r3, [R4,＃10]

输入寄存器为 r4, 立即数 10, 以及存储器, 而输出是寄存器 r3 中.

② 外部函数模拟

对于函数调用指令, 其输入和输出的上下文取决于被调用函数的功能. 静态分析关心的外部函数是和内存分配、内存释放、内存读写相关的函数, 如果是其他不相关函数, 我们直接跳过以降低模拟复杂性.

3.5 静态检测实现

模块中的各种入口点会被内核在加载成功后自动执行, 入口点是我们分析内核模块的起始点. 一旦模块被加入内核, 模块初始化函数将被自动执行. 我们的静态检测将从模块初始化函数开始.

3.5.1 静态检测整体流程

静态检测的过程为模拟内核模块内代码的执行, 其具体的执行过程如图 5 所示.

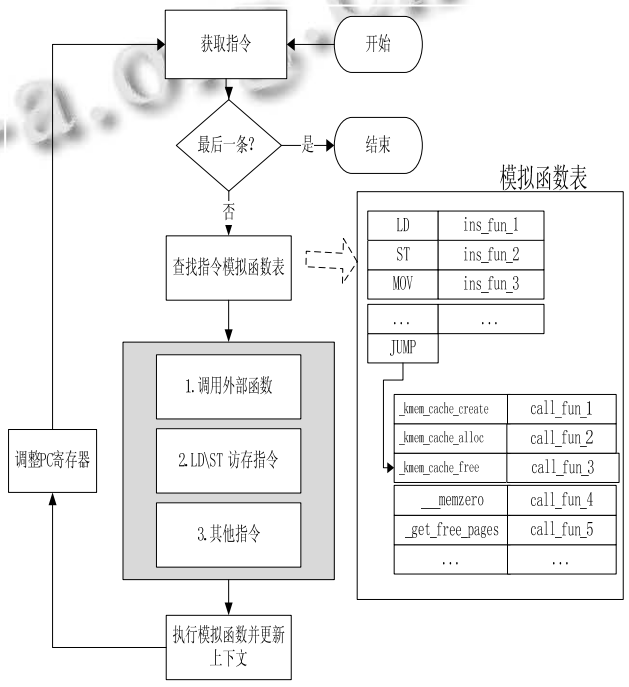

图 5 静态检测整体流程

① 获取指令

静态检测方法通过 PC 寄存器值读取二进制指令, 并转换成相应汇编指令. PC 的起始值为模块的入口函数首地址.

② 查找指令模拟函数

在模拟函数表中得到相应指令的汇编指令模拟例程. 如果是跳转指令, 还要进一步从外部函数模拟例程中得到处理函数. 这里, 我们关注的外部函数是会进行内存管理的相关内核接口, 如 slab 分配器用来管理小内存的内存创建接口_kmem_cache_create,内存分配接口_kmem_cache_alloc, 内存释放接口_kmem_cache_free 等.

③ 模拟指令运行、更新上下文

指令模拟函数会模拟指令的执行功能, 静态检测工具主要关注指令运行时对上下文的影响, 即寄存器组的变化、栈数据的压入与弹出. 如果所检测指令为访存指令(ST/LD)时, 静态检测将会比较寄存器中访存的位置是否和已经保存的内存区域(Region)的信息相符合, 并进行内存越界、无效指针判断等检查.

对于非内存访问指令, 模拟例程需要反映其对上下文的影响;

对于内存访问指令, 如 LD、ST, 通过内存检测算法, 给出检测结果. 检测结果包括: 正确, 警告和错误;

对于函数调用指令, 我们需要模拟被调用的函数. 对于内存管理相关函数, 我们需要创建或更新对于其他一些函数, 我们只需在模拟例程中反映其对上下文的影响.

④ 调整 PC

每条指令执行完, 将会更新 PC 指针, 以便后续指令的模拟.

3.5.2 内存区域记录检测

内核模块内创建的内存信息与模块访问的外部内存信息, 一起记录在内存上下文中. 每一个内存块将分配一个 Region 结构体, 如上文描述.

内核模块中代码访问外部内存区域时, 有两种情况, 存有全局名称的内存和不存在全局名称的内存. 如果是存在全局名称的内存, 我们可以从内核符号表中获得这段内存的起始地址和偏移量, 以及其他关于这段内存的内核结构信息. 即使是具有全局名称的内存, 除非是被静态分配的, 否则我们难以知道这段内存是否已被释放等关键信息; 不存在全区名称的内存在符号表中找不到其信息, 所有对其访问的操作都有可能带来错误, 因此必须发出警告提醒用户.

类似的, 对于模块内代码动态开辟的内存, 需要跟踪所有分配和释放内存的函数并记录其内存操作信息, 以便检查访存的可靠性.

为了后续访存指令的检测, 首先检测相关联的内存区域(region entry), 检测过程如图 6.

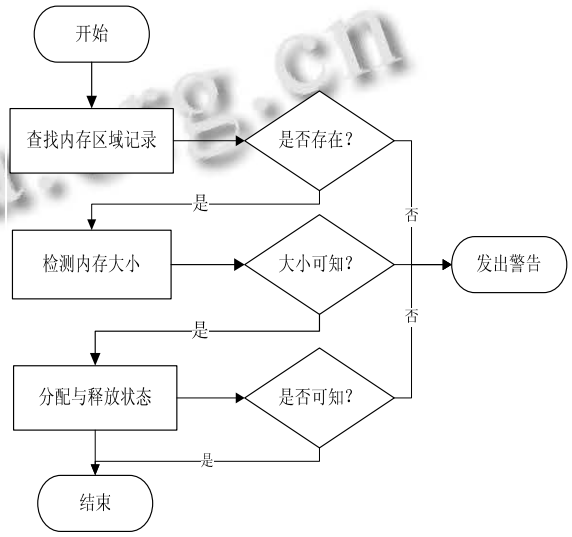

图 6　内存区域(memory region)检测

① 分析内存的大小信息, 如果不能够获得, 进行内存溢出警告报告;

② 分析内存的分配与释放状态, 如果不能够获得, 则有读未初始化、访问已经释放内存的风险, 发出警告;

③ 当内存释放函数试图对某一段内存进行操作, 如果对这段内存的是否释放状态不确定, 进行释放未分配内存的警告报告.

3.5.3 访存指令检测

在 ARM 环境中, 对于访存指令, 根据输入上下文 C = (R,M,S), 其检测算法如下,

① 查找访存指令内存信息, 如 3.4.2 节;

② 分析访存指令中起始地址和偏移量, 计算访存指令是否会对内存区域产生越界错误;

③ 分析访存指令寄存器参数的值, 判断其值类型是否为地址类型, 否则发出“访问非法地址”警告;

我们以 ST 指令, STR R, [mem_addr ,offset] 的检测为例.

开始:

1 计算访存地址;

2 从内存区域记录中查找相应 Region;

3 If Region 为空

警告“找不到寄存器 R 关联地址记录”;

4 If R.type 是非 ADDRESS 地址类型

警告“访问未知地址”;

5 If Region.type 为 released

报错“访问已释放内存”;

6 If 访存地址 mem_addr+offset 超出 region 的范围

报错“访存越界”;

7 更新 R.type 值;

8 更新 R[PC];

结束.

4 实验

4.1 实验结果

实验环境: 在 64 位 ubuntu11.04 操作系统上搭建 QEMU 模拟器, 利用 QEMU 模拟 ARM1176 处理器.

为了测试静态检测方法的有效性, 我们对模块中可能出现的几种典型的内存访问错误进行检测. 静态检测工具为 binAnalysis, 使用交叉编译工具, 编译包含内存访问错误代码的五种类型内核模块, 分别命名为access_invalid_pointer、access_uninitialized_memory、access_heap_bounds、access_freed_memory.

静态分析对内核模块中访存指令(ST/LD 指令)进行检测, 给出所有访存指令的分析结果. 实验结果如表 1 所示.

表 1 实验结果

检测结果	正确	错误	警告
access_invalid_pointer	2	0	1
access_uninitialized_memory	2	0	1
access_heap_bounds	7	0	2
access_freed_memory	3	1	0

对于访存指令, 如果其访问的地址信息在内存区域记录(Region)中能够找到, 静态检测方法可以直接判断访存指令是否正确. 否则, 静态检测方法会发出警告, 定位到可疑访存位置, 由用户进行审查.

以对访问已释放内存的检测为例, 用例中函数 func1 的源代码和汇编代码见图 7, 检测过程如下:

```
void func1(char *p2){          00000000 <func1>:
    char *p3, *p4, *p5;          0:  push  {r4, lr}
    p3 = p2;                     4:  mov   r4, r0
    p4 = p3;                     8:  bl    <kfree>
    kfree(p2);                   c:  mov   r3, #1
    p5 = p4;                    10: strb   r3, [r4, #10]
}                               14: pop    {r4, pc}
```

图 7 访问已释放内存源码、汇编码比较

① 根据“push”指令, 在上下文栈中保存 r4 和 lr 寄存器;

② 第二条指令“mov”将 r0 的值传给 r4. 随后一条指令, kfree 会被调用;

③ kfree 函数将需要被释放内存的起始地址传递给 r0. 在模拟函数行为时, r0 值应为一有效地址, 并且其关联的内存区域状态为已分配且未释放. 但是, 例子中的 r0 寄存器实际存储的值可能并不是一个有效地址. 对每一段由模块内代码开辟的内存都新建一个与之关联的 Region 结构体, 当调用 kfree 时, 如果没有找到与之对应的 Region 信息, 即发出警告, 以方便程序员排查. 在分析完 kfree 的参数 r0 后, r0 值对应的内存区域信息将被更新为已释放, 存入与之关联的 Region 结构体中;

④ 第四条指令改变 r3 寄存器的值为 1, 存入寄存器上下文中记录中;

⑤ 对于 ST 指令, 寄存器 r3 中的值将被存入到内存, 其地址为 r4 值加上偏移量 10. r4 寄存器指向的内存区域, 其状态应该为已经分配并且未被释放. 另外, r4 所指向的这段内存应该存够大(大于 10 个字节), 否则存入 r3 的值时将会发生溢出. 然而, 根据前面一条指令, r4 所指向的内存已经被释放到, 静态分析会直接发出错误报告, 检测结束.

4.2 性能分析

静态检测通过匹配已知内存区域信息和访存请求信息, 来确定访存指令的正确性. 其时间性能由查找内存区域(Region)决定. 本文的内存区域记录(Region)采用链表管理, 如图 8, 平均查找时间开销为 T/2,其中 T 为遍历头结点到尾结点的时间. 为了提高查找效率, 在后续的工作中, 将引入其他对内存记录的管理方法, 如哈希表等.

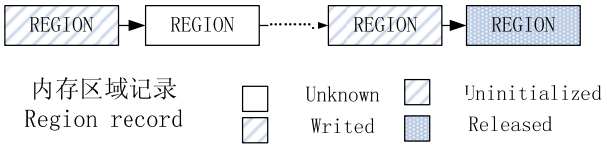

图 8 内存区域链表

5 结语

为了提高 Linux 内核模块中访存指令的可靠性, 本文的静态检测方法通过模拟指令行为, 在内核模块被实际加载前对其访存指令进行检测. 在嵌入式ARM平台, 实验表明本文静态检测方法能有效找出内核模块内非法的访存指令, 并对可疑访存指令发出警告.

参考文献

1 龚育昌,胡燕,张晔,等.一种针对可执行代码的内存泄漏静态分析方案.中国科学技术大学学报,2009,39(2):189−95.

2 刘天华,陈枭,朱宏峰,等.Linux 可加载内核模块机制的研究与应用,2007.

3 吕正,陈昊,陈峰.一种 ARM 存储模型的快速检测方法.西安交通大学学报,2013,47(6):68−72.

4 Xie Y, Chou A, Engler D. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. ACM SIGSOFT Software Engineering Notes, ACM, 2003, 28(5): 327−336.

5 Qin F, Lu S, Zhou Y. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. 11th International Symposium on High-Performance Computer Architecture (HPCA-11). IEEE. 2005. 291−302.

6 Necula GC, McPeak S, Weimer W. CCured: Type-safe retrofitting of legacy code. ACM SIGPLAN Notices. ACM. 2002, 37(1): 128−139.

7 Hastings R, Joyce B. Purify: Fast detection of memory leaks and access errors. Proc. of the Winter 1992 USENIX Conference. 1991.

8 Evans D, Guttag J, Horning J, et al. LCLint: A tool for using specifications to check code. ACM SIGSOFT Software Engineering Notes. ACM, 1994, 19(5): 87−96.

9 Wagle P, Cowan C. Stackguard: Simple stack smash protection for gcc. Proc. of the GCC Developers Summit. 2003. 243−255.

10 高海昌,冯博琴,卫鹏,等.Linux 下可执行文件的动态内存检测设计与实现.计算机工程,2007,33(1):74−76.