一种面向内核接口的顺序依赖规则挖掘与违例检测方法

刘虎球 白家驹 王瑀屏
(清华大学计算机科学与技术系 北京 100084)

摘 要 内核扩展函数以接口的形式提供给驱动,用于管理设备和申请相关的资源.这些接口中存在大量的顺序依赖规则,如自旋锁必须经过初始化才能加锁,然后才能解锁;驱动在加载时申请的内存,卸载时必须予以释放等.然而,驱动开发者常常不熟悉或疏忽内核接口的使用规则,导致驱动中存在大量的接口使用违例,影响驱动及系统的可靠运行.文中提出了一种面向内核接口的顺序依赖规则挖掘与违例检测方法(SD-Miner).该方法结合驱动源码的结构特征,对驱动代码使用的内核接口进行统计分析,挖掘并提取内核接口的顺序依赖规则,并利用提取的规则检测现有的驱动源码中的使用违例.SD-Miner对Linux 3.10.10和2.6.38的驱动源码分别进行了规则挖掘和违例检测.对比检测结果发现,在2.6.38中检测出的错误中,有64处在3.10.10中得到了修正.SD-Miner检测和分析Linux 3.10.10的3781款驱动的过程仅耗费5 min,共计提取出了220个顺序依赖相关的接口使用规则,并检测到了756个使用违例,作者将其中50个提交给了开发者,累计有25个回复者对20个使用违例进行了确认.实验结果表明,SD-Miner能够有效地挖掘出内核接口的顺序依赖规则,并检测出使用违例,进而辅助开发人员对驱动进行修正来提高驱动可靠性.此外,规则的挖掘是基于驱动的结构信息和统计信息,不需要开发者在源码中提供额外的注释及标注.

关键词 内核扩展函数;规则挖掘;违例检测;顺序依赖规则
中图法分类号 TP312 **DOI号** 10.3724/SP.J.1016.2015.01007

A Method to Mine Sequence Dependent Rules and Detect Violations for Kernel Extension Interfaces

LIU Hu-Qiu BAI Jia-Ju WANG Yu-Ping
(*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084)

Abstract Kernel extension functions are provided to drivers in the form of interfaces, to manage devices and resources. There are many sequence dependent rules in these interfaces, like the spin lock must be initialized before being locked, and then unlocked; the memory allocated for probing the device should be released when removing the device. However, due to driver developers' ignorance or unawareness of using these rules of the interfaces, a lot of violations may occur, which harms the reliability of drivers. This paper presents a method to mine the sequence dependent rules and detect violations for the kernel extension interfaces (SD-Miner). SD-Miner associates the structure of drivers, and exploits the statistic method to mine and extract sequence dependent rules, and it detects violations on the source code of drivers with extracted rules. SD-Miner has checked two versions of Linux 2.6.38 and 3.10.10, and 64 bugs detected in 2.6.38 have been

收稿日期:2014-05-17;最终修改稿收到日期:2014-08-24.本课题得到国家"八六三"高技术研究发展计划重大课题项目以支撑公众与企业服务为主的网络操作系统研制(2011AA01A203)资助.**刘虎球**,男,1989年生,博士研究生,中国计算机学会(CCF)学生会员,主要研究方向为操作系统、内核扩展可靠性与安全性、驱动可靠性与安全性.E-mail:liuhq11@mails.tsinghua.edu.cn.**白家驹**,男,1990年生,硕士,中国计算机学会(CCF)学生会员,主要研究方向为操作系统、驱动可靠性与安全性.**王瑀屏**,男,1984年生,助理研究员,主要研究方向为操作系统可靠性与安全性.

fixed in 3.10.10. 220 sequence dependent rules have been extracted from 3.10.10, and 756 violations have been detected as well. It only costs about 5 minutes to mine rules and detect violations for Linux 3.10.10. The selected 50 bugs have been reported to the developers, and 20 bugs (among 25 replies) have been confirmed. All the experimental evaluation results show that SD-Miner can mine sequence dependent rules for kernel extension interfaces effectively, and it also can detect real violations from the source code of drivers. Besides, with the help of SD-Miner, programmers can improve the reliability of drivers, without adding special notations to drivers.

Keywords kernel extension functions; rules mining; violation detection; sequence dependent rules

1 引 言

驱动的可靠性对操作系统的稳定运行具有重要意义.统计结果表明,驱动所引发的错误是内核的3～7倍[1],即使在成熟的商业操作系统XP中,也有85%的崩溃与驱动相关[2].由于驱动是由不同的组织和机构编写,这些错误在大多数情况是由于接口规则过于复杂,开发者对内核提供的接口缺乏足够的了解,常常忘记或者疏忽接口的使用规范和调用规则,从而造成大量的使用违例,使得大量错误隐藏在驱动源码中.

如图1所示,Linux驱动8250(版本3.10.10,driver/tty/serial/8250/8250_dw.c)在加载时将调用函数dw8250_probe.该加载函数调用devm_ioremap、devm_kzalloc、clk_prepare_enable等接口函数,并且上述部分函数在调用时存在直接的调用顺序依赖关系.如devm_kzalloc必须在clk_prepare_enable之前被调用,否则将出现使用违例.同样,在驱动卸载时,需要调用配对的接口函数将加载时申请的资源进行释放.如调用pm_runtime_disable来关闭

```
231 static int dw8250_probe(struct platform_device *pdev){
253     xxx=devm_ioremap(&pdev->dev, regs->start…);
258     data=devm_kzalloc(&pdev->dev, sizeof(*data) …);
263     if (!IS_ERR(data->clk))
264         clk_prepare_enable(data->clk);
287     data->line=serial8250_register_8250_port(&uart);
293     pm_runtime_set_active(&pdev->dev);
294     pm_runtime_enable(&pdev->dev);
297 }
```

```
299 static int dw8250_remove(struct platform_device *pdev){
305     serial8250_unregister_port(data->line);
307     if (!IS_ERR(data->clk))
308         clk_disable_unprepare(data->clk);
310     pm_runtime_disable(&pdev->dev);
311     pm_runtime_put_noidle(&pdev->dev);
314 }
```

图1 驱动8250的加载和卸载函数

在加载时通过pm_runtime_enable激活的设备,本文中将此类函数称为配对函数.另外,在卸载时执行的接口函数调用顺序与加载时调用的函数顺序同样存在关联,如加载时先执行pm_runtime_set_active,则在卸载时后执行其配对函数pm_runtime_put_noidle.值得一提的是,在图中,卸载时没有调用devm_kfree来释放在加载时通过devm_kzalloc申请的内存,因此存在使用违例,执行到此处时会引发内存泄漏.

另一方面,驱动中存在大量的互斥或互补性操作[2],操作时需要遵守许多与使用顺序相关的接口使用规则,我们称之为顺序依赖规则.为便于描述,将互斥或互补操作使用符号P(Probe,加载)、R(Remove,卸载)表示,并称P/R为对偶操作.与之关联的顺序依赖规则主要涵盖以下3个方面:(1)P操作调用的接口函数需要遵守顺序依赖规则,如图1中的devm_kzalloc必须在clk_prepare_enable之前被调用;(2)R操作需要调用配对的接口函数来释放P操作中申请的资源等;(3)R操作中调用的配对的接口函数需要遵守P操作中逆向的顺序依赖规则.

上述3类顺序依赖规则普遍存在于驱动中,如在P操作中,使用自旋锁需要遵守初始化spin_lock_init→spin_lock→spin_unlock的使用顺序,对设备的初始化一般遵循申请内存kmalloc→地址映射ioremap→设备激活pci_enable_device等顺序.而在P中调用一些接口函数后,在R操作中则需要调用这些接口的配对函数完成资源的回收操作,如kmalloc与kfree、ioremap与iounmap、pci_enable_device与pci_disable_device等.在R操作中使用的这些配对函数的序列同样存在顺序依赖,依然针对上述调用序列来说,在卸载时需要遵守的顺序依赖规则为:失效设备pci_disable_device→取消地址映

射 iounmap→释放申请的内存 kfree. 然而，由于操作系统中缺乏上述顺序依赖规则相关的规范文档，加上开发者对文档的查阅热情较低，导致驱动开发者容易出现使用违例. 因此对驱动源码进行检测并自动提取其中的使用规则，将有利于规范接口函数的使用.

目前，直接针对驱动源码进行错误查找和分析的研究方法主要集中在检测代码中的类型、指针等错误[3]，或者针对具体的单一的资源的使用规则进行违例检测[4]；也有一些研究者使用数据挖掘的手段尝试提取大规模软件中的一些使用规范[5]. 但是目前的这些研究方法都局限在一种操作内部或者单个函数内的调用序列上，不能针对上述跨函数之间的顺序依赖规则进行解析.

本文提出一种面向内核接口的顺序依赖规则挖掘与违例检测方法(SD-Miner). 该方法结合驱动源码的结构特征对驱动代码使用的内核接口进行统计分析，利用统计信息挖掘并提取内核接口的顺序依赖规则，并利用提取的规则检测现有驱动源码中的使用违例，辅助提高驱动可靠性，并且不需要开发者提供额外的注释及标注. 相比传统方法具备以下优势：(1) 不需要用户定义具体的接口使用规则，根据统计信息进行挖掘并检测序列化规则；(2) 跨函数直接进行接口的使用规范提取；(3) 结合驱动的结构信息和统计信息，无需程序提供额外的标注.

本文第 2 节主要介绍国内外关于驱动相关的规则检测和分析的研究现状；第 3 节阐述了 SD-Miner 的具体设计与实现过程；第 4 节对 SD-Miner 进行了评测；第 5 节对全文进行了简要总结.

2　相关工作

尽管有许多提高驱动可靠性的方法，如 FGFT[6]、SymDrive[7] 等，但对驱动代码的使用规范检查，仍旧集中在传统的错误检测、驱动开发规范和自动合成等方面，如指针错误检测[8]、整型数据检测 KINT[3] 等. 而在规则检测与挖掘方面，主要包括配对规则的挖掘[9] 和 API 规则挖掘[10] 等. 下面将结合具体的研究方法进行简单阐述.

LXFI[11] 根据程序员的标注，将接口的使用规则内嵌在修改后的接口中，使得驱动模块在发生错误时不会扩散成安全漏洞，用以抵御特权攻击. KFUR[4] 同样是通过修改原有的内核扩展函数，将需要遵守的规则内嵌到接口函数中，在执行接口函数之前自动执行规则检查.

Dingo[12] 针对文档的不规范和语义混淆问题，提出了一种基于协议状态自动机的设备描述协议 Tingu，可以直接辅助开发人员开发设备驱动，但是该方法需要重写所有驱动. Termite[13] 可以在上述基础上自动合成设备驱动，但生成的驱动代码量较大，效率较低.

以上方法所使用的接口使用规范都需要开发人员定义并书写，并且仍基于锁等单一资源，规则也相对较为单一. 所以部分研究者提出了使用挖掘的方法提取源码中的规则.

Engler 等人[9] 通过分析提取调用的 API，并根据调用的上下文反推与之关联的变量和函数等，如 spin_unlock 函数的调用前提是已经成功执行了 spin_lock 函数、指针在使用前需要初始化等. MAPO[14] 则利用已有的代码根据当前输入的代码对所使用的 API 给出相应的提示，其中包括关联函数，如 fopen 与 fclose 等. 以上规则虽然都与配对函数的挖掘相关，但是不能直接提取跨函数之间的接口函数使用时需要遵循的顺序依赖规则.

PR-Miner[5] 利用数据挖掘的方法，对一个函数内部可能调用的函数进行统计分析，提取潜在的最长的顺序调用序列，然后将提取的调用序列进行进一步分析并提取出使用规则，最后结合提取的规则对驱动源码进行违例检测. 但该方法要求代码的统计与重复度较大，无法提取一些低频的顺序依赖相关的接口使用规则.

CAR-Miner[15] 利用 Java 和 C++的异常机制，在异常处理时对已经申请的资源和与之关联的序列规则进行检测，进而发现其中的资源未被释放等问题. WN-Miner[16] 提出了一种使用规则的挖掘方法，基于程序的控制流对异常路径中的一些使用规则进行检查，定位其中的使用异常.

iComment[17] 利用程序中的注释，利于自然语言学习的方法，结合机器学习和统计方法，从注释中提取规则，进而结合源码的上下文进行分析，发现程序中注释与源码的上下文不一致之处，并分析和报告给开发人员. aComment[18] 在注释的基础上，结合了代码中的中断敏感函数，通过推导得到进入和离开函数时的中断开闭状态，进而发现源码与注释中关于中断开闭的约定与实际不一致的问题.

还有部分方法[19-20] 针对大规模软件，通过数据分析和挖掘的方法，提取其中的 API 使用规则或使用规范等，部分方法也利用提取的规范进行违例检测[21] 和辅助进行程序开发[22].

上述方法在提取使用规范的过程中，常常局限

于特定的一种规则或针对单一的资源，而基于统计的挖掘方法则要求重复的频度较高. 实际上，驱动具有较好的结构信息[23]. 根据结构信息推导，在降低统计阈值的同时，还可以大大提高提取的规则的准确性. SD-Miner 结合驱动的结构特征，利用统计方法对文中提及的三类接口调用顺序依赖规则进行提取和分析，在提升驱动可靠性的同时，无需程序员在源码中添加特殊的标注信息，也不再局限于特定的单一资源.

3 规则提取与违例检测

SD-Miner 结合驱动源码，利用统计方法对驱动中的顺序依赖规则进行提取. 在本节中，3.1 节简要介绍了 SD-Miner 的总体框架，3.2 节简要介绍了 SD-Miner 所需的理论模型，并在 3.3 节中给出了 SD-Miner 的规则挖掘的实现原理，第 3.4 节给出了基于挖掘出的规则实施的违例检测，第 3.5 节对系统的误报与漏报的优化技术进行了分析.

3.1 总体框架

为了完成顺序依赖规则的提取，SD-Miner 首先需要定位驱动中的 P、R 操作，然后分别提取 P、R 操作中接口函数的调用序列，再结合函数的返回值和参数信息，分析调用序列的内部依赖关系和 P、R 操作中的配对函数. 根据依赖关系来统计并提取出 P 操作中的顺序依赖规则，然后根据该顺序依赖规则对 R 操作中的逆向操作进行检查和分析，定位 R 操作中的使用违例，将违例进行误报和漏报过滤后以潜在的软件错误形式提交. 为完成上述目标，SD-Miner 需要执行图 2 中的主体执行流程，具体的细分步骤在下文中进行阐述.

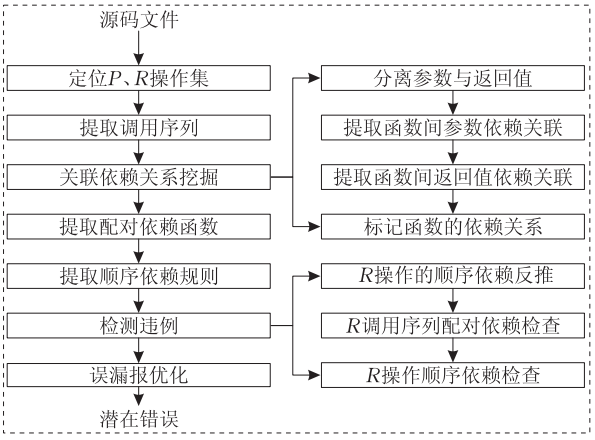


图 2 SD-Miner 主体框架

定位 P、R 操作集. SD-Miner 面向驱动源码对顺序依赖规则进行挖掘和检测，检测驱动对象是否存在关联依赖的 P、R 操作. 根据之前的定义，P、R 操作针对驱动设备执行相反的功能，如加载与卸载、休眠与唤醒等. 通过定位成对的 P、R 操作对应的函数，从而形成 P、R 操作集.

提取调用序列. 驱动执行 P、R 操作时需要大量调用内核提供给驱动模块使用的接口函数，本文提取的顺序依赖规则主要面向内核的接口函数. 因此在该步骤中，主要根据 P、R 操作集提取其中的潜在调用序列.

关联依赖关系挖掘. 针对提取的调用序列，通过分析各个接口函数的参数和返回值提取序列内接口函数之间的参数、返回值依赖，为后续的结构推导和统计提供支持. 根据提取的依赖关系对调用序列的顺序依赖进行初步标定.

配对函数提取. 对 P、R 操作使用的接口函数进行统计分离，结合驱动中的路径和结构推导的信息，将 P、R 操作间成对使用的接口函数进行统计，进而提取出配对函数.

顺序依赖提取. 对 P 操作中使用的函数调用序列进行分析，根据关联依赖关系提取调用序列之间接口函数的顺序依赖关系，并结合统计信息提取顺序依赖规则，便于后续的违例检测.

违例检测. 根据提取的顺序依赖规则，对驱动的源码再次进行检查，从而定位驱动在调用接口时违反顺序依赖的位置，通过一系列误报与漏报优化分析技术，分离出违例中真正的代码错误.

3.2 理论模型

为了完成配对依赖规则的挖掘，并根据提取的规则进行违例检测，我们首先提出了一个理论模型对挖掘的过程进行分析. SD-Miner 基于以下理论模型进行实现，为了清晰地对模型进行说明，首先对模型需要使用的符号集进行定义并介绍.

P、R 操作集. 该集合存储了一系列成对的 P、R 操作. 我们使用符号 O 来表示该操作的集合，其中 O_p 和 O_r 分别表示 P 操作集和 R 操作集. 由于 P、R 是成对存在的，因此 O 的每一个元素即为 O_p 和 O_r 的元素对：

$$O_i=(O_p^i,O_r^i) \tag{1}$$

假定共有 N 对 P、R 操作，则

$$O=\{O_1,O_2,\cdots,O_N\} \tag{2}$$

调用序列. 在每个操作中均调用了一系列的接口函数，使用符号 C_p 和 C_r 分别表示 P 操作和 R 操作中使用的接口调用序列. 对于第 i 对 (O_p^i,O_r^i)，调用序列表示为 C_p^i 和 C_r^i.

$$C_p=\{C_p^1,C_p^2,\cdots,C_p^N\} \tag{3}$$

$$C_r=\{C_r^1,C_r^2,\cdots,C_r^N\} \tag{4}$$

由于每个序列中包含多个函数，因此使用 $C_p^i[j]$ 和 $C_r^i[j]$ 分别表示调用序列中的第 j 个元素，被调用的函数的相关属性均被存储在 $C_p^i[j]$ 中，如函数名称(FuncName)、返回值类型(RetType)、返回值名称(RetName)、参数类型(ParType)、参数名称(ParName)等，并引入操作 GetValue 对上述属性进行获取，如通过 $GetValue(C_p^i[j],FuncName)$ 获得函数名称. 如果函数的返回值类型为空，则 $GetValue(C_p^i[j],RetType)=\varnothing$.

决策函数. 决策函数用于结合结构推导信息，辅助判定两个潜在的配对函数之间存在的配对可能性. 该函数输入分别为 C_p^i 和 C_r^i 中的一个元素，返回两者存在的配对可能性，使用符号 F_d 来表示. 经过决策函数后，可以定位出一些潜在的配对函数，使用符号 P_p 表示.

选择函数. 选择函数是指根据决策函数的结果，结合统计信息，从潜在的配对函数中选择出可能性比较大的配对函数，使用符号 F_s 来表示. 选择函数和决策函数参考了文献[24]已有的研究工作.

配对函数. 在 P、R 操作中调用的接口函数中存在大量的需要成对使用的函数，完成互补或相反的功能. 使用符号 P_f 来表示配对函数的集合.

$$P_f=\{P_f^1,P_f^2,\cdots,P_f^i,\cdots\} \tag{5}$$

同样 P_f 中的每一个元素包含一对具体的配对函数，分别表示为 $P_f^i[1]$ 和 $P_f^i[2]$. 一般来说，P_f 需要满足如下约束条件，即

$$\begin{aligned}&P_f^i[1]\in\{C_p^{ii}[j]\mid C_p^{ii}\in C_p\},\\&P_f^i[2]\in\{C_r^{ii}[j]\mid C_r^{ii}\in C_r\}\end{aligned} \tag{6}$$

为描述方便，引入函数 $f_p(P_f^i[1])$ 来获得 $P_f^i[1]$ 的配对函数，一般情况下：$P_f^i[2]=f_p(P_f^i[1])$. 当一个函数存在的配对函数多于一个时，$f_p(P_f^i[1])$ 返回配对集合，并默认使用配对频数最高的配对函数.

顺序依赖规则集. 对 P 操作中的调用序列 C_p^i 进行统计分析，获得调用序列，结合关联依赖关系对调用序列进行挖掘分析，提取顺序相关的调用序列，使用符号 S_r 来表示，规则集中的第 i 个元素以 S_r^i 来表示. 并且每个元素包含了一条 P 操作中的顺序依赖规则，每一条规则所包含的接口函数数目是不确定的，我们引入 len 函数用于描述一个序列的长度，如 $len(S_r^i)$ 表示的是第 i 条顺序依赖规则中所含接口函数的个数.

可信度估计函数. 可信度估计函数用于评估检测到的违例在统计信息中的可信度，使用 $p(P_f^i[1]\rightarrow P_f^i[2])$ 来表示. 当函数 $P_f^i[1]$ 被 P 操作调用时，函数 $P_f^i[2]$ 没有被 R 操作调用，可以用可信度估计函数估算到该违例可以被信任的可信度.

为了便于后续描述，用符号 F_t 来表示统计的频数，其输入为两个函数，输出为两者共同出现频数. 如 $F_t(P_f^i[1],P_f^i[2])$ 表示一对配对的函数同时出现的频数. 而 $P_f^i[1]$ 累计出现的频数可以通过如下方式进行统计，即

$$F_t(P_f^i[1],*)=\sum_{i=1}^{N}\begin{cases}1, & P_f^i[1]\in C_p^i\\0, & \text{其他}\end{cases} \tag{7}$$

由于一个函数可能存在多个配对函数，因此 $P_f^i[1]$ 累计出现的配对频数可以表示为

$$\begin{aligned}F_t(P_f^i[1],f_p(P_f^i[1]))=&\sum_{j=1}^{M}F_t(P_f^i[1],x)\\&\mid P_f^i[1]=P_f^j[1],\ x=P_f^j[2]\end{aligned} \tag{8}$$

结合 F_t，根据统计原理，将违例的可信度 $p(P_f^i[1]\rightarrow P_f^i[2])$ 表示为

$$p(P_f^i[1]\rightarrow P_f^i[2])=\frac{F_t(P_f^i[1],P_f^i[2])}{F_t(P_f^i[1],*)} \tag{9}$$

通过上述公式可以得出，当配对的频数越高，则发生违例时，该违例的可信度越大，但是存在局部统计信息与全局不对称的问题. 即假定当一对配对函数本身出现的频数较小时，根据式(9)将同样可以得出可信度较大的结论. 事实上这不是绝对的，因此将式(9)进行变换，综合考虑全局的配对可能性，假定配对频数最大值为 $F_t^{\max}$，则

$$\begin{aligned}&p(P_f^i[1]\rightarrow P_f^i[2])=\\&\frac{F_t(P_f^i[1],P_f^i[2])}{F_t(P_f^i[1],P_f)}\times\lambda_1+\frac{F_t(P_f^i[1],*)}{F_t^{\max}}\times\lambda_2\end{aligned} \tag{10}$$

其中，λ_1 和 λ_2 的值默认为 0.5，通过式(10)综合权衡了配对函数自身发生违例的局部可信性，以及在配对函数自身在全局中出现的可能性.

结合 P 操作中的顺序依赖规则与 R 操作中的配对依赖规则，得出 R 操作中的使用违例主要包括以下 3 种情况：

(1) P 操作调用 $P_f^i[1]$ 的次序违反顺序依赖的使用规则；

(2) P 操作中调用了 $P_f^i[1]$，但是对应的 R 操作中没有调用 $P_f^i[2]$；

(3) 在 R 操作中调用了 $P_f^i[2]$，但是违背了 P 操作顺序依赖的逆向使用规则.

实践证明，以上 3 种情况在实际中都存在，分别对应 SD-Miner 的 3 类顺序依赖规则. 其中，第 1 种仅仅与 P 操作相关，通过转换，和配对函数的统计

与检测存在较大相似性，将在后面章节进行详细介绍；第 2 种情况可以等同于配对函数规则的使用违例；而第 3 种情况中，违例可信的可信度应当综合考虑以下几个方面的因素：

(1) P 操作中被调用函数之间的顺序依赖可信度；

(2) P、R 之间操作的配对可信度；

(3) R 操作中被调用函数自身需要配对使用的可信度.

综合考虑以上因素，顺序依赖规则 S_r 的违例可以通过式(11)～(13)进行描述：

$$P_f^{i1}[1]\in S_r^i,\ P_f^{i2}[1]\in S_r^i,$$
$$(P_f^{i1}[1],P_f^{i1}[2])\in P_f,(P_f^{i2}[1],P_f^{i2}[2])\in P_f \tag{11}$$

R 操作将调用与 P 操作中对应的配对函数，并满足顺序依赖规则，该顺序依赖规则的应用场景表现为如下形式：

$$\exists j_1<j_2,P_f^{i1}[1]=S_r^i[j_1]\&P_f^{i2}[1]=S_r^i[j_2]\&$$
$$\exists S_r^i\cap C_p^i\neq\varnothing,\ P_f^{i1}[1]\in C_p^i,P_f^{i2}[1]\in C_p^i\&$$
$$P_f^{i1}[2]\in C_r^i,\ P_f^{i2}[2]\in C_r^i \tag{12}$$

如果在 R 中，$P_f^{i1}[2]$ 先于 $P_f^{i2}[2]$ 被 R 操作调用，则在 C_r^i 中出现关于顺序依赖规则 S_r^i 的违例，即

$$\exists k_1<k_2,P_f^{i1}[2]=C_r^i[k_1]\&P_f^{i2}[2]=C_r^i[k_2] \tag{13}$$

如果式(13)成立，则定位到了一处关于顺序依赖规则 S_r^i 的违例，使用符号 V 来表示使用违例的集合. 结合以上理论模型，下面将按照总体框架图 2 的主体流程对 SD-Miner 进行实现.

3.3 规则挖掘的实现原理

本节将详细介绍上述一些关键模块的实现流程. 3.3.1 节将对接口参数和接口函数的模糊定位方法进行介绍，3.3.2 节将介绍 P 操作内部的顺序关联规则的统计方法，而在 3.3.3 节中，将给出在对偶操作中使用到的配对依赖规则.

3.3.1 接口参数和接口函数的模糊定位

本文针对的检测和处理对象主体是执行相反或互补功能的 P、R 操作，因此 SD-Miner 首先要求正确地定位到驱动中的 P、R 操作集 O_p 和 O_r.

驱动自身具有较好的结构性特征，根据总线要求，提供正确的交互接口. 该交互接口中，包含有大量的满足 SD-Miner 要求的 P、R 操作. Kadav 等人[25]指出，PCI 驱动占据了 Linux 驱动的 36%，USB 驱动占据了 35%，并且与驱动加载和初始化等配置相关的代码占据了每款驱动 51%的代码，该部分代码正是 SD-Miner 的重点检测对象.

PCI 驱动和 USB 驱动都需要依赖总线挂载到系统中，总线对每款设备驱动都要求实现一个统一的接口，并在接口中规定了一些 P 操作和 R 操作. 如图 3 所示，每款 PCI 设备驱动都需要实现一个 pci_driver 驱动结构，内部包含加载(probe)P 函数指针、卸载(remove)R 函数指针.

```
1  struct pci_driver {
2      …
3      int (*probe) (struct pci_dev *dev, …);
4      void (*remove) (struct pci_dev *dev);
5      int (*suspend) (struct pci_dev *dev, …);
6      int (*resume) (struct pci_dev *dev);
7      …
8  }
```

图 3 PCI 总线驱动结构

SD-Miner 根据总线结构对驱动的结构进行分析和推导，定位到驱动中对应的 P 操作和 R 操作函数. SD-Miner 根据 pci_driver 的结构特征，提取其中的 probe 和 remove 函数，将函数所在的文件和函数名称记录到 O_p 和 O_r 操作集中. USB 驱动同样存在较好的结构特征，如图 4 所示，每个 USB 设备需要支持热插拔，因而需要实现动态的加载和卸载功能，对应 usb_driver 的 probe 和 disconnect 函数指针. 当驱动对上述函数指针完成赋值时，所赋值的函数即作为 SD-Miner 检测的对象.

```
1  struct usb_driver {
2      …
3      int (*probe) (struct usb_interface *intf, …);
4      void (*disconnect) (struct usb_interface *intf);
5      int (*suspend) (struct usb_interface *intf, …);
6      int (*resume) (struct usb_interface *intf);
7      …
8  }
```

图 4 USB 总线驱动结构

除了 USB、PCI 驱动，还有许多驱动存在共同的结构体，如 platform_driver 类驱动. 通过分析驱动的源码，结合驱动的结构特征，以函数指针名中赋值关键字匹配并定位操作集. 在 SD-Miner 中，通过指定执行相反或互斥功能的函数指针，如识别“probe”、“remove”和“disconnect”等关键字，并辅以函数指针的赋值特征和驱动的总线接口类型来定位驱动加载和卸载函数，并将成对的操作函数记录到 P、R 操作集.

一对 P、R 操作会调用一系列的接口函数来完成指定的功能，如在 P 操作中，常见操作包括申请内存、创建设备文件、绑定端口、激活设备等. 而在 R 操作中，常见操作包括：关闭设备、解除端口、移除设备文件、释放内存等. 两者调用的接口函数互为对偶函数，由于接口函数较多，功能各异，加之文档缺失，

因此使用规则无法直接获得，SD-Miner 尝试结合结构特征和统计信息提取接口函数中的使用规则.

P 操作和 R 操作的功能相反，因此在调用接口函数时，如果存在关联依赖，则在使用次序上要满足以下约束.

约束 1. 在 P 操作中先执行的顺序依赖相关的接口函数，其对偶函数在 R 操作中反而后执行.

由于 P 操作调用的接口序列并非全部存在顺序依赖，因此在 R 操作中，对偶函数的整个调用序列并不唯一，只有存在顺序依赖的接口函数在直接使用时需要满足约束 1. 3.3.2 节将对 P、R 操作之间的配对函数挖掘进行介绍，而在第 3.3.3 节中将对函数内部的顺序规则的统计和提取方法予以介绍.

3.3.2　统计 P/R 操作之间的配对依赖规则

大部分接口函数都是由内核开发者编写，因此接口函数的命名较为规范，结合函数的名字和参数可以对函数的功能进行识别. SD-Miner 通过提取函数名字中的关键字段特征，对决策函数 F_d 和选择函数 F_s 予以具体实现.

对配对的接口函数的识别关键在于定位出执行相反功能的潜在配对函数，参考一些已有的研究工作[24]，结合本文的具体情况，将配对挖掘的方法可以直接予以使用. P 操作和 R 操作中如果出现一对互斥关键字，且作用在同类对象上，该对函数将被加入到 P_p 中，并且，如果该对函数已经存在，则将其频数累加 1 次.

同时，在不同的 P、R 操作之间，还可以通过结构推导选择出潜在的配对函数. 如果 P 操作、R 操作中分别仅调用了一个接口函数，则两个函数自成配对函数，即

$$\exists len(C_p^i)=1 \& len(C_r^i)=1 \Rightarrow$$
$$P_p \leftarrow P_p \cup \{(C_p^i[1], C_r^i[1])\} \tag{14}$$

同样，当存在两个同类操作集之间的差值为一个函数，以 C_p^i、C_p^k 和 C_r^i、C_r^k 为例，引入临时集合 $C_{tp}^1=C_p^i-C_p^k$、$C_{tr}^1=C_r^i-C_r^k$，则

$$\exists len(C_t p^1)=1 \& len(C_t r^1)=1 \Rightarrow$$
$$P_p \leftarrow P_p \cup \{(C_{tp}^1[1], C_{tr}^1[1])\} \tag{15}$$

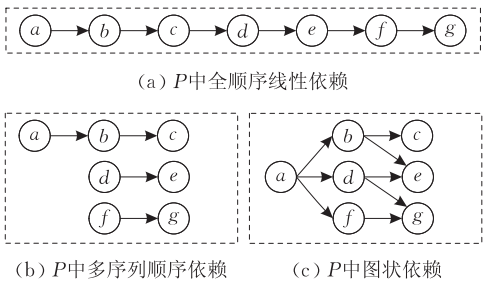
结合式(14)和(15)，对操作集之间任意个集合之间进行差值运算，运算结果满足式(15)函数的同样会被加入潜在的配对函数集中.

通过上述步骤，潜在的配对函数被记录在 P_p 集中. 然后遍历 C_p 和 C_r，统计 P_p 中的潜在配对函数成对出现的频数，再利用 F_s 对 P_p 进行选择.

3.3.3　统计函数内部的函数使用顺序关联

调用序列的顺序依赖关系主要体现在接口函数执行的逻辑次序. 如设备激活之前，常常需要申请内存保存设备的状态等信息，因此存在顺序依赖. 在 P、R 操作内部，都存在顺序依赖规则关联，由于 R 操作的顺序依赖规则与 P 操作中的顺序依赖关系存在关联性，因此 SD-Miner 将问题转换成了 P 操作内部的顺序依赖规则提取和 P、R 操作之间的配对函数提取.

接口函数之间的逻辑次序关系存在多种情况，如图 5 所示，图中描述了常见的接口函数之间的依赖关系，图中 $a\sim g$ 表示接口函数. 图 5(a)中描述了在 P 操作中，所有调用的接口函数存在线性依赖关系，即驱动必须使用完 a 才能使用 b，以此类推. 对于该类规则，根据约束 1 的要求，如果 $a\sim g$ 存在与之配对的函数，并假定函数名称为 $P_a\sim P_g$，即 a 与 P_a 是一对配对函数，以此类推，则在 R 操作中的调用顺序唯一，如图 6(a).

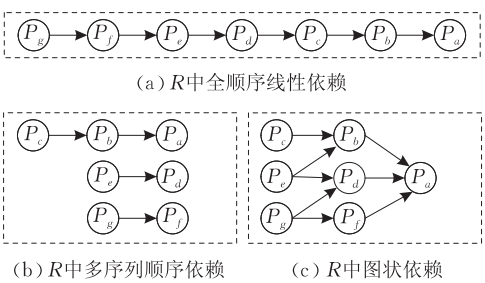


(a) P中全顺序线性依赖

(b) P中多序列顺序依赖　　(c) P中图状依赖

图 5　P 操作内部顺序依赖关系图

图 5(b)中的 P 操作同样调用了函数 $a\sim g$，但是函数之间的依赖关系分离在 3 条独立的规则之中，即 P 操作在执行时，只需分别保证每条独立规则的有序性，而规则之间没有先后关系，如函数 a 只需在函数 b 和函数 c 之前被调用，而与函数 d、e、f、g 的位置无关. 同样，根据约束 1 的要求，在 R 操作中该顺序依赖体现在，R 操作调用 $a\sim g$ 的配对的接口函数 $P_a\sim P_g$ 时只需遵守 3 条反向的依赖规则，如图 6(b)所示.



(a) R中全顺序线性依赖

(b) R中多序列顺序依赖　　(c) R中图状依赖

图 6　R 操作反向顺序依赖关系图

在图 5(c)中，P 操作调用的函数序列之间的依赖关系较为复杂，称为依赖关系图. 事实上，图 5(a)和图 5(b)是图 5(c)中图状依赖的特例，因此 SD-Miner 在实现顺序依赖挖掘时，使用了图来存储序列内部的依赖关系. 同样根据约束 1 的要求，R 操作中调用的序列需要遵守图 6(c)中的约束.

通过对比图 5(a)～(c)和图 6(a)～(c)可以发现，图 6 中的子图分别是图 5 子图的反向图，并且 R 操作调用的序列只要满足图 6 中的拓扑排序，即为一个正确的调用序列. 换言之，通过挖掘 P 操作中的顺序依赖规则，针对 R 操作，存在以下约束.

约束 2. 若 R 操作中调用的接口函数序列是 P 操作的依赖关系图的反向图的一个拓扑排序序列，则该调用序列是合法的.

至此，关键问题在于如何统计和提取出 P 操作中的顺序依赖规则. 假定调用序列中两个函数 a、b，并且 a 先于 b 被 P 操作调用，则 a、b 如果存在关联约束，主要存在以下 4 种关联：(1) 函数 a 的返回值被 b 使用；(2) 函数 a、b 的参数存在关联性；(3) 函数 a、b 的返回值存在关联；(4) 函数 a、b 的使用逻辑上存在先后. 其中，第 1～3 种可以通过提取 P 操作中接口函数的返回值和参数，进而分析返回值和参数的关联关系，辅助以统计信息来实现.

仍以图 1 为例，第 258 行调用 devm_kzalloc 申请内存空间并清零，将返回的内存空间起始地址存储在 data 指针变量中；第 264 行 clk_prepare_enable 函数使用到了 data 的成员变量，属于第 1 种关联，即 devm_kzalloc 函数必须先于 clk_prepare_enable 函数被调用. 第 287 行调用了函数 serial8250_ register_8250_port，该函数的返回值使用到了 data 的成员变量，因此该函数与 devm_kzalloc 存在上述第 3 种关联，并且同样必须先调用 devm_kzalloc 函数. 但是，尽管 clk_prepare_enable 函数与关联性 serial8250_register_8250_port 存在第 2 种关联，但是实际使用中并不要求严格的先后执行次序.

```
1607    static int twl_probe(struct pci_dev *pdev, …)
…           …
1632        host=scsi_host_alloc(…);
1679        retval=scsi_add_host(host, &pdev->dev);
1724        scsi_scan_host(host);
…           …
1754    }
```

图 7 P 操作 twl_probe 函数

而第 4 种关联并不能直接通过函数内部的调用关系予以直观推导，因此 SD-Miner 通过统计函数 a、b 之间的顺序依赖频数来提取顺序依赖规则. 如图 7 所示，函数 twl_probe 调用了 scsi_host_alloc、scsi_add_host、scsi_scan_host 等函数，其中 scsi_host_alloc 和后两者存在返回值与参数之间的直接依赖关系. 而 scsi_add_host 和 scsi_scan_host 之间属于第 4 种依赖，同时存在参数关联，SD-Miner 统计两者的先后出现次序累计出现了 39 次，并且两者拥有相同的参数约束，scsi_host_alloc、scsi_add_host、scsi_scan_host 三者构成了一条顺序依赖约束.

SD-Miner 首先定位一个 P 操作函数内部调用的所有接口函数，根据上述 4 类依赖，建立单个函数的关系依赖图，再结合统计信息，从所有的关系依赖图中提取频度较高的顺序依赖规则. 然后根据 P 操作中的顺序依赖关系图来判定 R 操作的序列是否满足是 P 操作中顺序依赖关系图的反向图的一个拓扑排序序列.

3.4 违例检测与实现

SD-Miner 的主要任务之一是定位驱动中的使用违例，即寻找驱动中违反顺序依赖规则的用例. 其中包括 P 操作中的顺序依赖规则的使用违例、P 操作和 R 操作中配对依赖规则的使用违例、R 操作中顺序依赖规则的使用违例.

3.4.1 P 操作中顺序依赖规则使用违例

通过统计分析，提取出了 P 操作内部的顺序依赖规则，即 P 操作内部需要遵守的接口函数调用顺序. 如之前提到的 scsi_host_alloc、scsi_add_host、scsi_scan_host 三者构成了一条顺序依赖约束. 当 P 操作调用 scsi_host_alloc 时，在后续流程中，一般需要调用 scsi_add_host，进而调用 scsi_scan_host，否则 SD-Miner 认定出现使用违例.

如图 8 所示，sbp2 _ probe (3.10.10, driver/firewire/sbp2.c)函数调用了前两者完成 host 的申请和加载工作，却并没有执行 scsi_scan_host 函数，因此判定这是一个使用违例. 值得一提的是，由于在命名上部分函数进行了私有实现，对原有的接口函数增加了前缀或后缀，对于该种情况 SD-Miner 参考参数等信息进行了模糊处理，进而减少误报.

```
1133    static int sbp2_probe(struct device *dev)
…           …
1150        shost=scsi_host_alloc(…);
1165        if(scsi_add_host_with_dma(shost,…);
…           …
1205    }
```

图 8 P 操作 sbp2_probe 函数违例用例

3.4.2　P 操作与 R 操作中配对依赖规则使用违例

R 操作依赖将 P 操作的配对函数对申请的资源进行释放，SD-Miner 根据 3.3.2 中的挖掘的配对依赖规则并结合 P 操作调用的函数列表对 R 操作进行检查.对于一对配对函数如果在 P 中调用了其中的一个，而 R 操作中没有与其对应的的配对函数，则被视为使用违例.

举例来说，P 操作函数 smtcfb_pci_probe(Linux 3.10.10，drivers/staging/sm7xxfb/sm7xxfb.c)调用了 pci_enable_device 函数.但是在对应的 R 操作函数 smtcfb_pci_remove 中，并没有调用 pci_disable_device 函数，所以存在使用违例.

对于这种配对函数相关的顺序依赖规则，SD-Miner 通过对 P 操作中的接口函数来定位，进而检查在 R 操作中是否出现了相应的配对函数.为减少误报，当检查到相应的使用违例时，SD-Miner 使用了一系列方法进行误报和漏报优化，具体在 3.5 节中进行介绍.

3.4.3　R 操作中顺序依赖规则使用违例

结合 P 操作的顺序依赖规则和 P/R 操作之间的配对依赖关系及约束 2 的要求，根据 P 操作的顺序依赖关系图构造反向图，然后参考配对依赖关系，对 R 操作中的顺序依赖规则进行检测，定位其中的使用违例.再结合拓扑排序算法理论对该检测算法按照算法 1 进行实现.

算法 1.　R 操作中违例检测算法 *SrCheck*.

输入：G_i 表示关于 O_p^i 的顺序依赖关系图，以邻接表存储，包含 *Vertex* 和 *Edge* 成员，图中每个点代表相应的接口函数，顺序依赖规则集已经嵌入在关系图中

C_r^i 表示与 O_p^i 对应的 R 操作调用序列

P_f 表示 P、R 操作之间的配对函数集

输出：V 表示规则使用违例集

```
1.  Vertex=G_i->Vertex;
2.  Edge=G_i->Edge;
3.  FLAG=new(len(Vertex));
4.  FOR v IN Vertex
5.      FLAG[v]=∅;
6.  FOR c IN C_r^i
7.      IF (pair=P_f(c))≠∅
8.          FLAG[pair]=c;
9.          DFS-visit(pair,FLAG,P_f,C_r^i,V);
10. RETURN V;
```

算法 1 按照 R 操作的调用顺序对每个接口函数进行检查，需要使用到改进的 *DFS-visit* 实现拓扑序列的验证，将验证 *pair* 在其子节点中是否存在配对依赖规则的使用违例和顺序依赖规则的使用违例，具体实现如过程 1 所示.

过程 1.　子节点遍历检查.

```
DFS-visit(pair,FLAG[],P_f,C_r^i,V)
/*输入输出的参数含义与 SrCheck 中介绍的一致*/
1.  FOR v IN pair.Vertex //访问 pair 的邻接点
2.      IF f_p(v)∩C_r^i≠∅&&FLAG[f_p(v)]=∅
3.          /*顺序依赖规则的使用违例，追加到 V*/
4.      ELSE IF f_p(v)∩C_r^i=∅&&FLAG[f_p(v)]=∅
5.          /*配对依赖规则的使用违例，追加到 V*/
6.      DFS-visit(v,FLAG,P_f,C_r^i,V);
```

过程 1 对 P、R 操作之间的配对依赖规则和 R 操作需要遵守的顺序依赖规则进行使用违例的检测，并将检测结果记录在 V 中，特别指出的是，由于可能存在重复的违例，在追加时，出现完全重复的违例将被忽略.由于 $f_p(v)$ 返回的是 v 的配对集合，因此判定 $FLAG[f_p(v)]$ 是否为空时，实际上只需集合中的任意一个元素不为空即可.

3.5　误报与漏报优化

任何静态方法都存在一定的误报和漏报，SD-Miner 结合驱动自身的结构特征，采取了一系列方法，对检测到的违例进行优化，主要包括进入被调用私有接口函数进行确认、建立白黑名单机制、实现对接口函数的模糊匹配.

部分驱动在实现 P、R 操作时并没有直接调用内核提供的接口函数，而是采用了私有接口或同类驱动的共同的私有接口.而这些私有接口的实现并没有内核接口规范，因此出现了较多 P 操作调用内核接口，而 R 操作使用私有接口的情况.针对该问题，SD-Miner 在发现使用违例时将进入 R 操作中的私有接口函数内部进行搜索和确认.但为了减少搜索和确认的复杂性，仅搜索直接被 R 操作调用的私有接口函数.

在内核提供的接口中也存在一些函数用于驱动获取内核的数据结构或获取设备状态等，如 pr_info 等.对这些接口函数的使用在理论上没有约束，为减少对规则提取的干扰，SD-Miner 同样支持白黑名单机制，对一些无需配对的函数直接去除，不加入统计范畴，同时也不进行违例检测.针对一些统计频数较低，但又较为重要的配对函数，支持将其加入白名单进行强制检查.

同时，SD-Miner 对接口函数采用模糊匹配的方式，减少误报，如图 8 中提到的 scsi_add_host 函数.另外，在配对依赖规则的挖掘上同样支持对接口

函数进行模糊匹配，如 alloc_etherdev 与 free_netdev 函数，SD-Miner 的决策函数会将 etherdev 等价于 netdev 进行处理.

4 实验分析

本节将对 SD-Miner 的实验进行分析和介绍，对 SD-Miner 的规则挖掘能力和违例检测能力进行评测. 本文采用的编程分析环境如表 1. 其中，第 4.1 小节将对工具的运算耗时和复杂性进行分析；第 4.2 小节中对从 Linux 中提取的顺序依赖相关的规则进行介绍；在第 4.3 小节中，SD-Miner 对 Linux 的高低版本分别进行评测，通过评测结果来验证其真实违例的检测能力；本节的最后对当前较新的 Linux 版本进行检测和分析.

表 1 实验分析环境

指标	CPU	内存	硬盘
参数	Intel i3-3220，3.30GHz	4G DDR3	1T，7200 rps

4.1 计算复杂性与耗时分析

SD-Miner 需要对 P、R 操作集两两函数之间进行关联性挖掘和分析，针对 P 操作内部的每一个函数需要分析与其他函数之间的顺序依赖关系，以及分析与对应的 R 操作内部的每一个函数之间的配对依赖关系. 假定共有 N 对 P、R 操作，每个操作平均调用了 M 个函数，据此对运算的复杂性进行估计，则规则分析需要的操作估计达到 $N\times M\times M$ 次函数匹配.

在违例检测时也需要再次扫描 P、R 操作中的每个函数，时间与挖掘分析时间基本相当，但是对于 M 个函数而言，假定序列规则的平均长度为 L，则 *DFS-visit* 本身时间复杂度约为 $\theta(ML)$，由于 L 为常数，因此分析的总操作次数规模在 $\theta(N\times M^3)$.

SD-Miner 共计耗时 2min 对 Linux 3.10.10 的驱动进行规则挖掘，并花费 1 min 对规则进行优化和分析，结合优化后的结果，花费 2 min 对驱动进行了违例检查，以上累计耗时 5 min.

4.2 Linux 源码中规则挖掘分析

SD-Miner 利用本文中提到的挖掘方法，针对 Linux3.10.10 驱动程序中的 3 类顺序依赖相关的规则进行了提取. 利用 probe/remove、probe/disconnect 等，累计对 3781 对 P、R 操作进行了分析，从驱动源码中共计提取出了 171 对配对依赖相关的规则，从 P 操作中提取了 49 条顺序依赖规则.

部分重要的 P 操作中的顺序依赖规则如表 2 所示，以 scsi_host_alloc、scsi_add_host、scsi_scan_host 为例，三者之间分别构成顺序依赖关系. 表中每个单元内的括号内表示统计的频数，而表中每行列出的是规则中的前 3 个函数，并且顺序依赖关系是“函数 1→函数 2→函数 3”.

表 2 部分 P 操作中的顺序依赖规则

规则	函数 1	函数 2	函数 3
host	scsi_host_alloc	scsi_add_host	scsi_scan_host
usb	usb_create_hcd	hcd_to_ehci	usb_add_hcd
sound	snd_card_create	snd_card_set_dev	snd_card_register

根据该顺序依赖规则，如果 P 操作中调用不完整或 R 操作中的配对函数的调用次序不满足此顺序依赖关系图的反向图的拓扑排序，则视为违例. 在配对依赖规则方面，共计提取了 49 对顺序依赖规则，其中频数与配对函数的数目之间的关系如图 9 所示，其中频数高于 50 的顺序依赖规则共计 3 条.

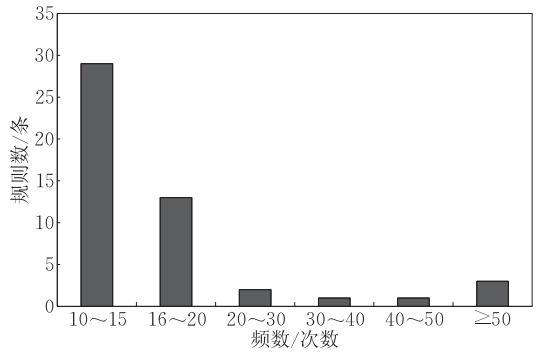


图 9 频数(横)与配对依赖规则的数目(纵)关系

4.3 不同版本潜在的修正的 bug 对比

为了验证 SD-Miner 的违例异常检测能力，通过以下方式对其进行测试和验证：

(1) 以 Linux 2.6.38 的源码作为分析对象，通过挖掘和分析提取相应的规则，进而定位其中的顺序依赖相关的使用违例；

(2) 以 Linux 3.10.10 的源码作为分析对象，通过同样的方式对系统的违例进行分析和检测；

(3) 对比两者之间的检测结果，找出在 2.6.38 中出现而在 3.10.10 中被修正的使用违例.

通过分析，共定位了 64 个在 2.6.38 中出现的违例，它们在 3.10.10 中已经被修正. 其中部分重要的违例如表 3 所示，以 P 操作 ax_probe 函数(Linux 2.6.38，drivers/net/ax88796.c)为例，在 P 操作中调用了 request_mem_region 函数，该函数存在配对使用的函数 release_mem_region，但是在 R 操作中并未调用该函数对 request_mem_region 申请的内存资源进行释放，该种情况下可能造成内存泄露. 该

问题在 Linux 3.10.10 中已经被修复，并且在 cgit 的提交日志中，提到修复的原因为“ax88796：clean up probe and remove function”.表中的其他用例修复原因在修复日志中给与了介绍.

表 3　部分 *P*/*R* 操作中的配对相关的顺序依赖规则

路径	*P* 操作函数	关键函数	*R* 操作中配对函数	修复日志
drivers/net/ax88796.c	ax_probe	request_mem_region	release_mem_region	ax88796：clean up remove function …
drivers/hid/hid-wacom.c	wacom_probe	device_create_file	device_remove_file	HID：wacom：Unregister sysfs attributes
drivers/watchdog/omap_wdt.c	omap_wdt_probe	pm_runtime_enable	pm_runtime_disable	Fix the mismatch of unmap IO memory
drivers/staging/xgifb/XGI_main_26.c	xgifb_probe	ioremap	iounmap	staging：xgifb：release and unmap …

表 3 中给出了部分已经被修正的 BUG，其中修正的日志中大都明确给出了错误的原因，和 SD-Miner 定位该类违例的初衷一致，该实验表明，SD-Miner 能够准确定位到一些比较重要的 BUG，下面将对新版本的驱动源码进行违例检测.

4.4　较新版本中存在的 bug

SD-Miner 选取了当前较新的 Linux 内核 3.10.10 作为评测版本，通过静态分析，基于提取的 49 条 *P* 操作内部的顺序依赖规则，以及 171 个 *P*、*R* 操作之间配对依赖相关的顺序依赖规则，定位到了 756 个使用违例.按照驱动的类别进行分类，违例在各类驱动的分布如图 10 所示，从图中可以看出，4 类驱动所含违例总数占总的 23.3%，人工分析以太网(ethernet)相关的 40 处违例，内含不规范使用 10 处警告，发现漏报 1 处，误报 1 处.事实上，一些关键设备驱动中仍旧存在一些使用违例，如 USB 网络设备驱动中 *P* 操作 usbnet_probe 函数调用了 pm_runtime_enable 函数，但是在 *R* 操作中没有调用 pm_runtime_disable 函数执行相反的操作.

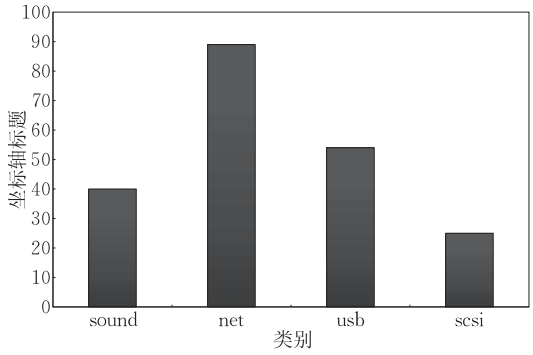


图 10　违例在各类驱动中的分布

在提取的违例检测结果中，将定位到的其中 50 个错误以潜在的 BUG 形式提交给了相关的开发人员，至今累计 25 个开发人员回复了我们的报告，其中 20 个错误被确认，部分被确认的 BUG 如表 4 所示.表 4 中给出了违例发生的关联函数和宿主函数.需要特别指出的是，SD-Miner 与 PF-Miner[24] 等作者已有工作中重合的已确认 BUG 已被直接包含在确认的统计之中.

表 4　部分被确认的驱动错误(BUG)

路径	操作函数	关键函数	备注
I82975x_edac.c	i82975x_init_one	pci_enable_device	*R* 违例
gpio-lynxpoint.c	lp_gpio_probe	pm_runtime_enable	*R* 违例
ipmi_si_intf.c	ipmi_pci_probe	pci_enable_device	*R* 违例

一般的静态工具都会存在漏报和误报等问题，主要原因包括：(1) 部分接口函数的调用出现在判定分支中；(2) 部分驱动结构性较差，通过私有的深层调用方式释放资源，而 SD-Miner 的分析深度不够时，会引起误报，但是该问题已经通过宽松的统计策略和严格的搜查策略来解决，即一旦发现违例，则进入深层次搜索，防止出现误报；(3) 驱动部分接口本身设计不规范，接口之间存在包容关系，如设备加载时，会先调用 input_allocate_device 申请资源，然后调用 input_register_device 注册，然后卸载时直接调用 input_unregister_device 即可，因为其内部已经调用了 input_free_device.

SD-Miner 通过关键字等优化方法，对检测到的违例进行了过滤，对观察到的一些违例进行了筛查，并增加一些规则到该工具中，结合 3.5 节中提到的误报和漏报优化技术，从而减少误报和漏报.目前通过人工分析现有的检测结果，其误报基本控制在 15%左右.

5　结论与展望

本文提出了一种面向内核接口的顺序依赖规则挖掘与违例检测方法(SD-Miner)，结合驱动源码的结构特征，对驱动代码使用的内核接口进行统计分析，挖掘并提取内核接口的顺序依赖规则，并利用提取的规则检测现有驱动源码中的使用违例.SD-Miner 对 Linux 3.10.10 和 2.6.38 的驱动源码进行了规则挖掘与违例检测，对比实验结果发现，在 2.6.38 中存在 64 个被检测出的错误在 3.10.10 中

得到了修正. SD-Miner 对 Linux 3.10.10 的 3781 款驱动进行的检测和分析仅耗费 5 min,共计提取出了 220 个顺序依赖相关的接口使用规则,并检测到 756 个使用违例. 实验结果表明,SD-Miner 能够有效地挖掘出内核接口的顺序依赖规则,并检测出使用违例,进而辅助提高驱动可靠性. 同时规则的挖掘是基于驱动的结构信息和统计信息,不需要开发者在源码中提供额外的注释及标注.

论文的后续研究工作将结合动态分析技术,把静态定位到的违例,通过动态运行和错误注入技术进行确认,后续将考虑结合动态插装和检测技术,在提高驱动可靠性的同时,将检测的违例动态修复,如及时并自动释放申请的资源. 这些功能将在 SD-Miner 的后续版本中进行补充和完善.

致 谢 本文作者得到了实验室的老师和同学们的许多帮助和建议,在此表示感谢;同时也感谢审稿人对本文提出宝贵意见和建议;并对帮助我们确认和修正 bug 的开发者表示诚挚感谢!

参 考 文 献

[1] Ganapathi A, Ganapathi V, Patterson D. Windows XP kernel crash analysis//Proceedings of the 20th Large Installation System Administration. Washington, USA, 2006: 101-111

[2] Liu Hu-Qiu, Ma Chao, Bai Jia-Ju. Automatically inserting log system for driver configuration. Chinese Journal of Computers, 2013, 36(10): 1982-1992(in Chinese)
(刘虎球, 马超, 白家驹. 面向驱动配置的自动日志插入方法研究. 计算机学报, 2013, 36(10): 1982-1992)

[3] Wang X, Chen H, Jia Z, et al. Improving integer security for systems with KINT//Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. Hollywood, Canada, 2012: 163-177

[4] Ma Chao, Yin Jie, Liu Hu-Qiu, et al. KFUR: A new kernel extension security model. Chinese Journal of Computers, 2012, 35(10): 2091-2100(in Chinese)
(马超, 尹杰, 刘虎球等. KFUR: 一个新型内核扩展安全模型. 计算机学报, 2012, 35(10): 2091-2100)

[5] Li Z, Zhou Y. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. SIGSOFT Software Engineering Notes. 2005, 30(5): 306-315

[6] Kadav A, Renzelmann M J, Swift M M. Fine-grained fault tolerance using device checkpoints//Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems. Houston, USA, 2013: 473-484

[7] Renzelmann M J, Kadav A, Swift M M. SymDrive: Testing drivers without devices//Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. Hollywood, Canada, 2012: 279-292

[8] Chou A, Yang Junfeng, Chelf B, et al. An empirical study of operating systems errors//Proceedings of the 18th ACM Symposium on Operating Systems Principles. Banff, Canada, 2001: 73-88

[9] Dillig I, Thomas D, Aiken A. Static error detection using semantic inconsistency inference//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. San Diego, USA, 2007, 42(6): 435-445

[10] Pradel M, Jaspan C, Aldrich J, Gross T R. Statically checking API protocol conformance with mined multi-object specifications//Proceedings of the 34th International Conference on Software Engineering. Zurich, Switzerland, 2012: 925-935

[11] Mao Y, Chen H, Zhou D, et al. Software fault isolation with API integrity and multi-principal modules//Proceedings of the 23rd ACM Symposium on Operating Systems Principles. Cascais, Portugal, 2011: 115-128

[12] Ryzhyk L, Chubb P, Kuz I, Heiser G. Dingo: Taming device drivers//Proceedings of the 4th ACM European Conference on Computer Systems. Nuremberg, Germany, 2009: 275-288

[13] Ryzhyk L, Chubb P, Kuz I, et al. Automatic device driver synthesis with Termite//Proceedings of the 22nd Symposium on Operating Systems Principles. Montana, USA, 2009: 73-86

[14] Xie T, Pei J. MAPO: Mining API usages from open source repositories//Proceedings of the 2006 International Workshop on Mining Software Repositories. Shanghai, China, 2006: 54-57

[15] Thummalapenta S, Xie T. Mining exception-handling rules as sequence association rules//Proceedings of the 31st International Conference on Software Engineering. Vancouver, Canada, 2009: 496-506

[16] Weimer W, Necula G C. Mining temporal specifications for error detection//Halbwachs N, Zuck L D eds. Tools and Algorithms for the Construction and Analysis of Systems. Berlin Heidelberg: Springer, 2005: 461-476

[17] Tan L, Yuan D, Krishna G, Zhou Y Y. /*iComment: Bugs or bad comments?*///Proceedings of the 21th ACM Symposium on Operating Systems Principles. Washington, USA, 2007, 41(6): 145-158

[18] Tan L, Zhou Y, Padioleau Y. aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs//Proceedings of the 33rd International Conference on Software Engineering. Honolulu, USA, 2011: 11-20

[19] Dallmeier V, Knopp N, Mallon C, et al. Generating test cases for specification mining//Proceedings of the 19th International Symposium on Software Testing and Analysis. Trento, Italy, 2010: 85-96

[20] Gabel M, Su Z. Testing mined specifications//Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. Cary, USA, 2012: 1-11

[21] Lawall J L, Brunel J, Palix N, et al. WYSIWIB: Exploiting fine-grained program structure in a scriptable API-usage protocol-finding process. Software: Practice and Experience, 2013, 43(1): 67-92

[22] Xie T, Pei J. MAPO: Mining API usages from open source repositories//Proceedings of the 2006 International Workshop on Mining Software Repositories. Shanghai, China, 2006: 54-57

[23] Saha S, Lozi J P, Thomas G, et al. Hector: Detecting resource-release omission faults in error-handling code for systems software//Proceeding of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks. Budapest, Hungary, 2013: 1-12

[24] Liu Huqiu, Wang Yuping, Jiang Lingbo, Hu Shimin. PF-Miner: A new paired functions mining method for Android kernel in error paths//Proceeding of the 38th IEEE Annual Conference on Computers, Software and Application. Vasteras, Sweden, 2014: 33-42

[25] Kadav A, Swift M M. Understanding modern device drivers //Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems. London, UK, 2012: 87-98

LIU Hu-Qiu, born in 1989, Ph. D. candidate. His main research interests include operating systems, the reliability, security of kernel extensions and drivers.

BAI Jia-Ju, born in 1990, M. S. His research interests include operating systems, reliability and security of driver.

WANG Yu-Ping, born in 1984, assistant researcher. His research interests include reliability and security of operating systems.

Background

Drivers exploit the kernel extension interfaces to operate devices and manage resources, and there are lots of implicit rules between these interfaces. As drivers are developed by different organizations and institutions, lots of bugs are brought into operating system when the developers forget or ignore to call the interface correctly. Some researchers have tried to mine and extract implicit rules for large software, and lots of related papers are published in SOSP, OSDI, PLDI and other international conferences. Among these methods, most of the rules are extracted from the same function. However, drivers have their own special interfaces, and many operations use the rules across functions, like probing and removing operation between different functions.

In this paper, we present a method to mine and extract sequence dependent rules for kernel extension functions (SD-Miner), and SD-Miner can also detect violations from paired functions with extracted rules. Firstly, mining and extracting of the sequence dependent rules are discussed in theoretical modeling and analysis. And then based on the structure characteristics of the driver's source code and associating with the statistical analysis method on the kernel interfaces for drivers, SD-Miner mines lots of sequence dependent rules without the developers to provide additional comments and annotation in the source code. According to these rules, SD-Miner detects lots of violations from the source code of drivers.

SD-Miner mainly focuses on the mutually-exclusive and complementary operations for drivers, and these operations should obey the rules directly. To simplify the description, the mutually-exclusive or complementary operations are denoted by the symbol set P(Probing, loading), R(Removing, unloading). The sequence dependent rule mainly cover the following three aspects: (1) Functions in P operation call sequence dependent functions and should obey the rules, such as the function pci_ioremap generally is called after the function devm_kmalloc; (2) Functions in R operation call paired functions to release the acquired resources allocated by P, such as calling the function pci_disable_device to disable the device that are actived by pci_enable_device in P operation functions; (3) Functions in R operation call paired functions and also need to obey the sequence dependent rules, such as turning off the equipment firstly, and then releasing the memory etc.

Several research works of our team based on drivers have been published in Computer, China science, COMPSAC etc. The rules from exception paths, different functions, and full paths have been extracted. We also have reported some potential bugs to the developers, and many bugs have been confirmed. In the future work, some runtime checking information will also be imported into the system.

This work is supported by Tencent Incorporated and the National High Technology Research and Development Program (863 Program) of China (Project No. 2011AA01A203).