基于语义补丁的Linux驱动程序后向移植技术

朱丽华 文艳军 董 威
(国防科技大学计算机学院 长沙 410073)

摘 要 使用语义补丁技术对Linux的网卡驱动程序的后向移植方法进行研究。通过分析多个不同内核版本的驱动程序代码,在兼容库代码的支持下,提出了一种使用语义补丁进行驱动程序后向移植的方法。针对符号和函数分别开发了语义补丁,解决了后向移植过程中代码冗余以及补丁文件过多的问题,提高了后向移植的效率。实验分析表明,所使用的语义补丁的代码行数比普通补丁减少了很多,使得移植过程更加高效。在后向移植的过程中,对兼容库的构造方法进行了总结,使得移植后的代码可读性更强。所提方法对实现其他驱动程序的后向移植提供了借鉴意义。

关键词 语义补丁,驱动程序,兼容库,后向移植
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.11.010

Backporting of Linux Device Drivers Using Semantic Patch

ZHU Li-hua WEN Yan-jun DONG Wei
(College of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract The paper studied the backporting of network device drivers in Linux kernels using semantic patch technology. After analyzing the code of device drivers in many different kernel versions, this paper proposed an approach to backport device drivers using semantic patch technology with compatibility library. Semantic patches are developed separately according to symbols and functions, which reduces the redundancy of code and improves the efficiency of backporting. The experiment shows that the lines of code of semantic patches are much less than normal patches. Some experiences about building compatibility library were also proposed in the paper. The results show that this approach is also valuable for backporting other device drivers.

Keywords Semantic patch, Device driver, Compatibility library, Backport

1 引言

目前,开源操作系统Linux在桌面、服务器、嵌入式系统、移动网络等众多领域应用广泛,并且很多基于Linux内核开发的操作系统也在迅猛发展。Linux内核自1991年发布第一个版本到现在已经有上百个不同的版本,现在大约每2.5个月就有一个新的主线版本被发布。这种快速的更新使得Linux内核不断地修复错误,并且提供新的特性和功能。尽管新的内核可以提供更好的体验,但是还有很多用户依旧在使用低版本的稳定内核。稳定内核迁移到新内核并不容易,需要对各类核外的程序进行大量测试,甚至可能需要进行修改。但是新的应用程序一般会直接面向新的操作系统内核进行开发并采用新内核中的接口或者数据结构,从而导致无法在旧内核中正常运行,这一问题在驱动程序中尤为明显。如今新硬件不断出现,很多设备只支持当时的最新内核,导致在低版本内核上并没有相应的驱动程序可供使用。因此本文在构造兼容库的基础之上使用语义补丁技术对驱动程序进行后向移植,通过对Intel网卡驱动程序的后向移植过程进行优化,研究了使用语义补丁技术的优势,这对于其他驱动程序的移植具有指导意义。

本文第2节介绍相关研究工作;第3节介绍语义补丁的基本概念和使用方法;第4节针对Intel网卡驱动程序进行后向移植,包括兼容库的构造过程和具体所使用的语义补丁编写;第5节给出相关的实验结果,分析语义补丁技术的优势,并且对兼容库代码的构造进行优化;最后总结全文,并对下一步工作进行展望。

2 相关研究

驱动程序的后向移植需求一直存在,主要表现为:1)在较早的内核上并没有相应的驱动程序;2)驱动程序本身也在不断更新,早期的驱动中可能存在的错误会在新版的驱动中得到修复;3)旧版的驱动程序支持的同类型的硬件型号较少,新

到稿日期:2016-10-07 返修日期:2016-12-01 本文受国家自然科学基金(91318301),国家重点基础研究发展计划(973计划)(2014CB340703)资助。

朱丽华(1992-),男,硕士生,主要研究方向为高可信软件技术,E-mail:yjwen@nudt.edu.cn;文艳军(1975-),男,博士,副教授,硕士生导师,CCF会员,主要研究方向为可信软件构造、程序分析与验证;董 威(1976-),男,博士,教授,博士生导师,CCF会员,主要研究方向为软件分析与验证、软件智能化开发。

版的驱动可能会增加对新型号的设备的支持。早在 2007 年，compat-wireless[1]项目就开始移植一些无线设备的驱动程序，直到近年才更名为 backports，并且现在仍针对 Linux 稳定内核和日常的内核进行代码修改。目前该项目支持将 Linux v4.2.6 上的 830 多个驱动程序移植到最低 Linux v3.0 上。

常用的驱动程序的后向移植方法有两种：1）在源代码中加入大量的预处理语句，针对不同版本的内核条件编译不同的目标文件，但是这种方法存在不足，因为一些驱动程序中很多代码都是相同的，简单的条件编译会使得代码中存在较多冗余，不利于今后持续的修改；2）采用兼容库的方法将一些函数或者数据结构放在额外的源代码文件中单独编译，并且在驱动中使用，其同样需要在代码中加入预处理的语句，但是由于共用了兼容库代码，因此代码量得以减少。backports 项目采用了第二种方法。

修改系统源码之后，可以通过 diff 程序或者手工编写生成一个表示前后文件区别的补丁文件（Patch）[2]，作用在源码上就可以进行修改。与普通的补丁不同，语义补丁（Semantic Patch）[3]除了考虑程序中的固定位置的匹配之外，还考虑了程序的控制流信息和变量类型等程序语义方面的内容。另外，普通补丁不具备通用性，每一次修改都需要重新生成新的 patch，这使得程序在维护上变得更加困难。

在 Coccinelle[4]程序匹配和转换工具的基础之上，编写适当规则的语义补丁可以对源码进行自动匹配和修改。在功能上语义补丁和普通补丁最后的作用都是修改代码，但是语义补丁在这个过程中可以匹配文件中符合特定规则的所有代码段，并且依照预先设定好的规则进行转换。通过引入元变量的概念，语义补丁对程序中变量的类型匹配更加准确，转换过程也更加可靠。

目前，Coccinelle 工具已经集成在较新版本的 Linux 内核中，很多发行版的 Linux 中也安装了 Coccinelle 工具（即 spatch 命令）。Coccinelle 主要有两个作用：1）内核不断演化过程中的 API 的并行演化[5-7]：当一个 API 发生变化（名称、参数个数、参数类型等）时，所有使用该 API 的地方都需要进行更改，这时语义补丁就可以发挥很大的作用，避免很多人工的修改过程；2）查找内核存在的错误[8-10]：通过编写适当的语义补丁，描述一些错误范式，并在程序中自动匹配相应的代码段，就可以认为匹配到的代码中很可能存在相应的错误。通过 Coccinelle 的错误检测功能能够发现内核中的很多错误，并且其中一些错误已经被主线内核采纳。

本文旨在通过研究语义补丁的编写方式，使用 Coccinelle 程序匹配和转换工具来对网卡驱动程序的后向移植过程进行优化，提高移植过程的自动化程度，同时对兼容库的构造方法进行研究，进而对其他驱动程序的后向移植起到指导作用。

3　SmPL 语义补丁

语义补丁的作用虽然与普通的补丁类似，但是其在编写上有特殊定义的语言，即 SmPL（Semantic Patch Language）。语义补丁具有更加灵活多样的语法规则，语义补丁程序可以看成是一系列相关的匹配规则或者模式的定义，每一个规则都包括以下几部分。

规则名称（Rulename）：通常每个规则都有自己的名称，在声明名称的同时，还可以说明规则的性质，如规则之间的依赖关系或者同构规则以及量词的限制。

元变量（Metavariable）：这是相对普通补丁而言比较大的改进，在规则名称声明之后可以声明一系列的元变量，用于匹配程序中任意术语的变量。元变量的类型很多，包括标识符（Identifier）、表达式（Expression）、语句（Statement）、常量（Constant）等。元变量的类型众多，一方面说明了语义补丁程序的匹配的准确性，另一方面体现了语义补丁考虑了程序中变量类型信息的特点，提高了程序匹配和转换的精确程度。

转化/修改（Transformation）：这是语义补丁编写中最关键的部分，转换的内容位于元变量声明之后，对代码中符合某种模式或规则的地方进行相应的修改。需要转换的代码使用“+”“-”“*”中的任意一个符号标注，分别表示需要增加、删除和强调的部分。其中“*”不能与“+”“-”一起使用。为了考虑程序中的控制流信息，还引入了符号“…”，表示 0 个或多个控制流节点可能会在其之前的语句和其之后的语句之间产生，“…”会匹配满足条件的最短路径，但是对具体的内容并不关心，因此程序的匹配并不是固定位置的，可能跨越了多个无关语句。

以一个简单的语义补丁程序为例，原本的代码如图 1 所示，语义补丁的代码如图 2 所示。该语义补丁的意义很简单，即将所有对 foo()函数调用的地方替换成对 bar()函数的调用，这个语义补丁中并未指明规则名称和元变量，只描述了代码转换的内容。图 3 给出了转换之后的代码。

```
1. int main(){
2.    int k = foo();
3.    if(1){
4.       foo();
5.    } else {
6.       foo();
7.    }
8.    return 0;
9. }
```

图 1　原代码

```
1. @@
2. @@
3. -foo()
4. +bar()
```

图 2　语义补丁

```
1. int main(){
2. int k = bar();
3. if(1){
4. bar();
5. } else {
6. bar();
7. }
8. return 0;
9. }
```

图 3　最终代码

4　使用语义补丁后向移植驱动程序

当需要移植某个驱动程序时，必然涉及到对该驱动源代码进行修改，在其中加入大量的条件编译语句，使得修改之后的驱动程序可以在更低版本的内核上正常使用。

一个比较好的解决方法是将所有相关的预处理语句都集中存储在一个头文件或源文件中[11-12]，这就是介于不同版本内核之间的兼容库代码，兼容库是按照内核版本逐一构造的，从高版本到低版本要考虑其之间的每一个主线版本的差异，因此后向移植并不是一个点对点的移植过程，在移植的过程中要考虑到中间的所有版本。以此为基础，既可以使用普通的补丁进行驱动后向移植，也可以使用更加高效自动的语义补丁进行后向移植。因此整个后向移植的过程可以用图 4 来描述。

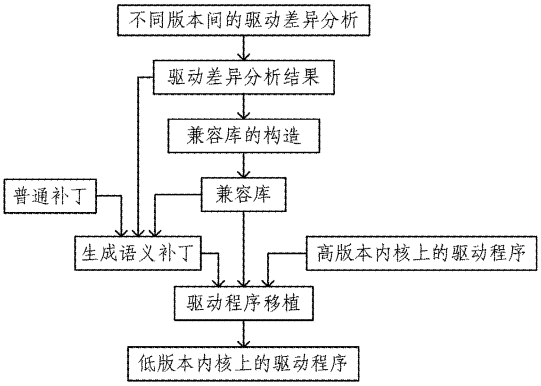

图 4 后向移植过程

通过分析不同版本驱动的差异,生成相应的语义补丁,构造出合适的兼容库,在语义补丁和普通补丁的共同作用下,对高版本内核上的驱动程序进行移植,生成低版本内核上的驱动程序。

下面介绍兼容库的构造方法以及在兼容库基础之上如何编写适当的语义补丁,以提高移植过程的自动化程度。

4.1 兼容库的构造

兼容库可以看作是驱动程序和内核代码之间的中间代码。当驱动需要调用某一个函数时,如果内核中有该函数的声明和定义,则可以调用成功,但是如果没有,则需要在兼容库中进行额外的定义。因此在构造兼容库时需要充分考虑版本的更新情况。对于特定的驱动而言,并不需要考虑所有的变化,只需要考虑本驱动中使用的内核函数或者数据结构是否有变动。另外,内核中定义了很多符号,有些是在内核编译时设置的,一些内核的功能是否完全支持都通过检查特定的符号在当前版本上是否有定义来确定,因此兼容库中也包含很多符号的相关定义。

4.1.1 在兼容库中定义符号

内核是在不断更新的,当配置内核时有很多选项。一些只在高版本内核编译时才会出现的选项往往会散布在内核的各个地方。除了内核编译选项产生的符号之外,我们也可以定义自己的符号。一个符号有没有定义同样也可以表明当前的内核版本是否满足条件,例如:在 Linux v2.6.29 中,将 net_device结构中的一些有关设备操作的回调函数抽取出来放在一个新的结构体 net_device_ops 中,即对于 2.6.29 之前的内核来说没有可以使用的这一数据结构。因此可以定义 HAVE_NET_DEVICE_OPS 这一符号,如图 5 所示。当内核为 2.6.29 版本或者更高版本时,在兼容库中就会定义 HAVE_NET_DEVICE_OPS 符号,其意义也很清楚。

```
1. #if (LINUX_VERSION_CODE <
        KERNEL_VERSION (2,6,29) )
2. /* code here */
3. #else
4. #ifndef HAVE_NET_DEVICE_OPS
5. #define HAVE_NET_DEVICE_OPS
6. #endif /* HAVE_NET_DEVICE_OPS */
7. #endif /* < 2.6.29 */
```

图 5 定义 HAVE_NET_DEVICE_OPS

在驱动程序中具体使用该符号时的编码如图 6 所示。

```
1. #ifdef HAVE_NET_DEVICE_OPS
2. /* have netdev_ops struct */
3. netdev->netdev_ops = &igb_netdev_ops;
4. #else
5. netdev->open = &igb_open;
6. netdev->stop = &igb_close;
7. /* code style in old kernels */
```

图 6 使用 HAVE_NET_DEVICE_OPS

从图 6 中可以看出,使用这个符号的作用等价于使用版本号判断语句,但是与直接判断版本号相比,额外定义一个符号可以使程序的可读性更强。在涉及众多版本的内核的兼容库代码编写中适当地引入符号定义会让驱动程序代码的可读性更强,更加便于维护。

4.1.2 在兼容库中定义函数

在高版本驱动程序中很有可能出现一些以前没有的函数,如图 7 所示,在 Linux v4.6 中为内存中的页面引用次数递增选项新增了一个函数 page_ref_inc()。因为在 Linux 4.6 内核中新增加了页面引用次数的调试选项,当需要调试时函数中的 if 语句才会起作用,否则条件一直为 false。

```
1. static inline void page_ref_inc(struct page *page)
2. {
3.       atomic_inc(&page->_count);
4.       if(page_ref_tracepoint_active(
                __tracepoint_page_ ref_mod))
5.       __page_ref_mod(page, -1);
6. }
```

图 7 Linux 4.6 内核中的 page_ref_inc 代码

为了将 Linux 4.6 内核中的驱动移植到低版本上,需要在兼容库中添加一个相同的函数,只有当内核版本号小于4.6 时才会被使用,但是需要去掉第 4 行的 if 语句,因为低版本的内核中没有相关的配置选项,因此相应的兼容库代码如图 8 所示。

```
1. #if LINUX_VERSION_CODE <
2.         KERNEL_VERISON (4,6,0)
3. static inline void page_ref_inc(struct page *page)
4. {
5.    atomic_inc(&page->_count);
6. }
7. #endif /* < 4.6.0 */
```

图 8 兼容库中的 page_ref_inc

在兼容库中增加函数时,可以尽量使用与原函数相同的函数名称,当然前提是两者之间的参数也要相同,而且将条件编译的语句一起放在兼容库中,使源代码中条件编译相关的代码减少。

除了直接声明相同的函数名之外,还可以采用定义宏的方式,但实际效果相同。

采用上述方式可以方便地解决函数名称变化的问题,那么根本不需要修改驱动程序的源码。但实际中内核 API 可能会出现参数类型变化或者参数个数的变化,这时就需要采

用其他策略。例如用于输出设备调试信息的 dev_err，dev_info 等函数，其所接收的第一个参数的实参类型会发生变化，从单独的 struct device 结构变成了 struct pci_device 结构体中的 dev 成员。针对这种变化，如果在源代码中进行修改，那么需要增加很多条件编译语句。这时可以在兼容库中定义 pci_dev_to_dev()函数，该函数可以依据不同内核版本分别实现，但是在实际的源码中可以省去很多重复的条件编译，其代码如图 9 所示。

```
1. #if (LINUX_VERSION_CODE <
        KERNEL_VERSION (2,5,0))
2. static inline struct device * pci_dev_to_dev(struct pci_dev * pdev){
3.      return (struct device *) pdev;
4. }
5. #else
6. static inline struct device * pci_dev_to_dev(struct pci_dev * pdev){
7.      return &pdev->dev
8. }
9. #endif /* < 2.5.0 */
```

图 9　pci_dev_to_dev 的兼容代码

4.2　开发语义补丁

在驱动程序的后向移植过程中，很多实现基本功能的函数都被使用了很多次，一旦这些函数在某一个版本的内核中发生了变化，那么在驱动源码中相应需要修改的地方就有很多。通过补丁的方式可以完成这样的修改，但是一旦驱动程序的数量很多，需要维护的补丁数量也会变多，而且补丁是一层一层发挥作用的，打补丁的过程容易失败，这给程序维护带来了困难。语义补丁通过匹配源码中符合特定规则的地方进行转换，代码较短，而且可以作用在不同的文件之上，为后向移植提供了便利。下面针对 4.1 节中新定义的符号和一些函数说明进行语义补丁的编写，并且讨论如何提高语义补丁开发的效率和语义补丁使用中的注意事项。

4.2.1　使用语义补丁处理符号

一旦一个函数在声明时处于某些条件编译语句中，那么在接下来定义或者使用这个函数的地方都需要加入相应的条件编译语句。例如上文提到的 HAVE_NET_DEVICE_OPS 符号，目前出现 struct net_device_ops 结构的地方都应该加上相应的条件编译语句，使用图 10 所示的语义补丁可以达到这一效果。

```
1. @have_net_device_ops@
2. identifier s;@@
3. + #ifdef HAVE_NET_DEVICE_OPS
4. static const struct net_device_ops s={…};
5. + #endif
```

图 10　net_dev_ops 使用的语义补丁

实际上这是一个十分通用的语义补丁，因为在内核v2.6.29 之后的所有驱动程序中都有这样的变化，不仅仅局限在网卡驱动中。

使用符号语义补丁还可以处理头文件的包含问题，因为随着内核更新，内核支持的功能越来越多，模块需要依赖的模块也会增加，在代码中需要包含的内核头文件也会增多。这也可以通过语义补丁来解决。例如新增的 linux/mii.h 用于提供对介质无关接口(Media Independent Interface，MII)的支持时，可以用图 11 所示的语义补丁进行处理。

```
1. @include_mii@
2. @@
3. + #ifdef SIOCGMIIPHY
4. #include <linux/mii.h>
5. + #endif
```

图 11　mii.h 使用的语义补丁

4.2.2　使用语义补丁处理函数

上文提到了在兼容库中定义函数时可以使用与源码中相同的函数名称，从而不需要对源码进行改动。但是在一些情况下无法通过这种方式处理。例如在上文提到了 pci_dev_to_dev 函数的作用，该辅助函数，可以使源代码更加简明，但是这是一个纯粹的兼容库函数。对于这种转换，可以使用图 12 所示的语义补丁来处理。

```
1. @pci_dev_to_dev_1@expression e1, e2;@@
2. dev_err(
3. -e1
4. +pci_dev_to_dev(e1)
5. ,e2)
6. @pci_dev_to_dev_2@expression e3, e4;@@
7. dev_info(
8. -e3
9. +pci_dev_to_dev(e3)
10. ,e4)
```

图 12　pci_dev_to_dev 的语义补丁

通过这个语义补丁将源码中的 dev_err 和 dev_info 的参数进行了转化，使其使用兼容库中已经定义的函数。

还有一些函数的改动并不需要在兼容库中进行定义，如函数的返回值类型变化，或者函数的参数个数变化。例如igb_clean_rx_irq的返回值由 bool 变成 int，因为这个函数的返回状态发生变化，所以可以通过简单的语义补丁来完成自动修改。

4.2.3　提高语义补丁的开发效率

在实际应用中很多语义补丁完成的功能是类似的，例如一些与符号相关的语义补丁，其中符号与函数或者数据结构大多存在一种对应关系。源代码中存在如图 13 所示的函数声明语句。

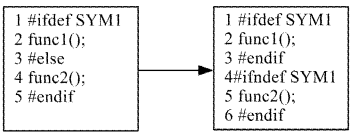

图 13　源代码预处理

通过对源码进行预处理，可以更加方便地自动提取符号和函数声明之间的对应关系，从而可以统一生成很多类似的语义补丁，避免重复的工作。这就需要我们在编写源码时注意代码的风格，尽量使符号和函数声明之间的对应关系表达清晰，容易被自动提取。

Linux 内核的编码有一定的规范，例如 igb_open()是

igb 驱动中的一个函数，在其他驱动中会同样注册 xxx_open()，这种统一的命名风格已经成为实际的标准，因此对一些针对 igb 中函数的语义补丁稍加修改也可以用于其他驱动。

4.2.4 语义补丁的不足

尽管语义补丁本身的能力很强，但也存在缺点。首先，语义补丁作用的方式是按照规则顺序处理的，这就带来一个问题：当两个语义补丁针对同一处进行修改时，需要格外注意两者的作用顺序，也可以在开发时就声明两个语义补丁的依赖关系，否则随意应用补丁可能会带来意想不到的错误。其次，对于语义补丁的执行效率问题，因为与普通补丁采用的简单的字符串匹配不同，语义补丁需要分析代码的控制流信息，需要的时间更长，因此在应用语义补丁时需要考虑补丁的实际作用而不是盲目使用所有文件，这样可以在一定程度上提高语义补丁的执行效率。最后，对于语义补丁的更新和维护，与普通的补丁不同，语义补丁无法通过简单的 diff 而生成，当内核更新时，需要对涉及到的语义补丁进行针对性的修改，因此对其进行维护的难度相比普通补丁更大。

5 实验结果

本文针对 Linux v4.6 内核中的 igb 网卡驱动程序的后向移植使用了 11 个语义补丁，使用的兼容库最低可以支持到内核 2.4 版本。实际上这些语义补丁的作用不仅限于 igb 驱动，尤其是一些关于符号的语义补丁都可以使用在其他驱动程序中。用于处理函数的语义补丁是专门针对 igb 驱动编写的，但是因为在内核代码编写过程中函数的命名都遵循一定的规范，所以很多此类补丁稍加修改函数的名称就可以用于其他驱动程序的后向移植。表 1 列出了本文使用的语义补丁及其简要说明。

表 1 语义补丁简介

类型	名称	作用描述
符号相关	HAVE_NET_DEVICE_OPS	是否定义了 net_device_ops 结构体
	HAVE_NDO_SET_FEATURES	net_device_ops 是否支持 set_features 操作
	CONFIG_NET_POLL_CONTROLLER	是否配置网卡轮询控制器
	SIOCGMIIPHY	是否支持 MII 功能
	HAVE_I2CSUPPORT	是否支持 I2C
函数相关	pci_dev_to_dev	Pci_dev 转换成 dev 结构
	igb_alloc_q_vector	分配网卡队列向量，参数类型发生变化
	E1000_READ_REG	硬件相关，读取设备寄存器
	Adapter->flags & IGB_FLAG_HAS_MISX	Igb_adapter 结构体成员变化
	igb_set_uta	函数的参数个数发生变化
	igb_clean_rx_irq	函数的返回值类型更改

针对这些语义补丁，本文分析了每个补丁的应用情况，表 2 给出了符号相关的语义补丁的实验结果，这部分语义补丁并不是专门针对 igb 驱动使用的，因此考察了这几个语义补丁在以太网卡驱动中的应用情况；表 3 给出了函数相关的语义补丁的实验结果，因为这部分是针对 igb 驱动的，所以只考虑其在 igb 驱动中的使用情况。

表 2 符号相关的语义补丁实验结果

名称	行数	应用次数	补丁行数
HAVE_NET_DEVICE_OPS	5	12	228
HAVE_NDO_SET_FEATURES	20	5	196
CONFIG_NET_POLL_CONTROLLER	13	12	364
SIOCGMIIPHY	5	3	39
HAVE_I2CSUPPORT	23	2	52
总计	76	34	879

表 3 函数相关的语义补丁实验结果

名称	行数	应用次数	增加行数	减少行数	补丁行数
trans_pci	11	40	65	46	410
alloc_vector	6	1	1	3	13
read_reg	4	56	81	80	618
igb_adapter	5	4	4	4	45
igb_set_uta	6	2	2	2	20
clean_rx_irq	5	2	2	2	20
总计	37	105	155	137	1126

因为表 2 中都是针对符号的语义补丁，所以源码只有增加没有减少，补丁行数是指达到与语义补丁相同的转换效果所需的补丁行数。可以发现，针对符号的语义补丁在应用于 Intel 网卡驱动时比较有优势，每个语义补丁都应用了多次，而且行数只有普通补丁的 10%左右。表 3 主要列出了针对 igb 驱动中特定函数的语义补丁，其中由于 pci_dev_to_dev 和 E1000_READ_REG 在驱动代码中反复使用，因此语义补丁的作用很明显，但实际上更多的情况类似于另外 4 个语义补丁，应用的次数并不会太多，由此看出，针对函数的语义补丁的作用总体比针对符号的语义补丁弱一些。但是由于对针对函数的语义补丁稍加修改就完全可以将其使用在其他驱动程序中，因此可以达到编写一次使用多次的目的。

结束语 本文研究了 Linux 内核中网卡驱动程序的后向移植方法。通过构造驱动程序和内核之间的兼容库代码，同时针对兼容库和源码本身进行相应的语义补丁的开发，提高了网卡驱动程序后向移植过程的自动化程度，从而可以更加高效地进行后向移植。本文总结了兼容库构造的方法，同时提出了一些可以提高语义补丁开发效率的方法，这些方法对内核中其他驱动程序的后向移植都具有较好的借鉴作用。下一步工作的重点包括：1）同类型驱动程序之间的兼容库代码中实际有很多重复的部分，因此可以考虑研究兼容库代码的融合问题，开发适用范围更大的兼容库代码以供使用。考虑到语义补丁本身的特性，一些在兼容库中定义的数据结构或者函数也可以在语义补丁中加以实现，从而进一步降低兼容库的复杂程度，将重点放在语义补丁的开发上。2）研究语义补丁的正确性验证问题。虽然通过传统的测试方法可以带来一定程度的保障，但代码的改变可能会破坏程序的语义不变式，因此可能需要对修改之后的代码进行修改影响分析[13-15]，以保证移植过程的正确性。3）进行语义补丁的推理[16-17]。除了分析代码中的条件编译语句之外，还可以对大量的补丁进行研究，通过分析普通补丁的规律来推理一些可用的语义补丁。

（下转第 79 页）

IEEE,2014:590-604.
[8] PANCHENKO, KARSTENS J, PLATTNER H, et al. Precise and scalable querying of syntactical source code patterns using sample code snippets and a database[C]// Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension. IEEE,2011:41-50.
[9] LINTON M A. Implementing relational views of programs [C] // Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. ACM,1984:132-140.
[10] CHEN Y, NISHIMOTO M Y, RAMAMOORTHY C V. The c information abstraction system[J]. IEEE Transactions on Software Engineering,1990,16(3):325.
[11] CHEN Y, GANSNER E R, KOUTSOFIOS E. A C++ data model supporting reachability analysis and dead code detection [C]// Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Springer,1997:414-431.
[12] HOLMES R, WALKER R J. Approximate structural context matching: an approach to recommend relevant examples [J]. IEEE Transactions on Software Engineering,2006,32(12):952-970.
[13] BAJRACHARYA S, NGO T, LINSTEAD E, et al. Sourcerer: a search engine for open source code supporting structure-based search[C]// Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. ACM,2006:681-682.
[14] BAJRACHARYA S, OSSHER J, OSSHER J. Sourcerer: an infrastructure for largescale collection and analysis of open-source code[J]. Science of Computer Programming,2014,79:241-259.
[15] BADROS G. Javaml: A markup language for java source code [J]. Computer Networks,2000,33(1):159-177.
[16] AGUIAR, DAVID G, BADROS G. Javaml 2.0: enriching the markup language for java source code[M]// XML: Aplicac Oese Tecnologias Associadas,2004:1-12.
[17] EICHBERG M, HAUPT M, MEZINI M, et al. Comprehensive software understanding with sextant[C]// Proceedings of the 21st IEEE International Conference on Software Maintenance. IEEE,2005:315-324.
[18] PANCHENKO, KARSTENS J, PLATTNER H, et al. Precise and scalable querying of syntactical source code patterns using sample code snippets and a database[C]// Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension. IEEE,2011:41-50.
[19] YAMAGUCHI F, GOLDE N, ARP D, et al. Modeling and discovering vulnerabilities with code property graphs[C]// Proceedings of the 2014 IEEE Symposium on Security and Privacy. IEEE,2014:590-604.
[20] URMA R, MYCROFT A. Source-code queries with graph databases-with application to programming language usage and evolution[J]. Science of Computer Programming,2015,97:127-134.
[21] ZHANG T, PAN M, ZHAO J, et al. An open framework for semantic code queries on heterogeneous repositories[C]// Proceedings of the 2015 International Symposium on Theoretical Aspects of Software Engineering. IEEE,2015:39-46.

(上接第 68 页)

参 考 文 献

[1] RODRIGUEZ L R, LAWALL J. Increasing Automation in the Backporting of Linux Drivers Using Coccinelle[C]// Dependable Computing Conference. 2016:132-143.
[2] MACKENZIE D, EGGERT P, STALLMAN R. Comparing and Merging Files With Gnu Diff and Patch. Network Theory Ltd [OL]. http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html.
[3] MULLER G, PADIOLEAU Y, LAWALL J L, et al. Semantic patches considered helpful[J]. ACM Sigops Operating Systems Review,2006,40(3):90-92.
[4] BRUNEL J, DOLIGEZ D, HANSEN R R, et al. A foundation for flow-based program matching using temporal logic and model checking[C]// POPL 2009. 2009:114-126.
[5] PADIOLEAU Y, LAWALL J, HANSEN R R, et al. Documenting and automating collateral evolutions in Linux device drivers [C]// EuroSys 2008. Glasgow, Scotland: ACM,2008:247-260.
[6] PADIOLEAU Y, LAWALL J L, MULLER G. Under standing collateral evolution in Linux device drivers[C]// EuroSys. 2006:59-71.
[7] LAWALL J L, BRUNEL J, PALIX N, et al. WYSIWIB: exploiting fine-grained program structure in a scriptable API-usage protocol-finding process[J]. Software: Practice and Experience, 2013,43(1):67-92.
[8] ENGLER D R, CHELF B, CHOU A, et al. Checking system rules using system-specific, programmer-written compiler extensions[C]// OSDI'00. San Diego, CA,2000:1-16.
[9] CHOU A, YANG J, CHELF B, et al. An empirical study of operating systems errors[C]// Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01). New York, NY, USA, ACM,2001:73-88.
[10] PALIX N, THOMAS G, SAHA S, et al. Faults in Linux: ten years later[C]// 16th International Conference on Architectural Support for Programming Languages and Operating Systems. Newport Beach, CA, USA,2011:305-318.
[11] CORBET J, RUBINI A, KROAH G. Linux Device Drivers (third edition)[M]. USA: O'Reilly,2006:35-45.
[12] BOVET D P, CESATI M. Understanding the Linux Kernel (third edition)[M]. USA: O'Reilly,2007:525-534.
[13] GALLAGHER K B, LYLE J R. Using program slicing in software maintenance[J]. Transactions on Software Engineering, 1991,17(18):751-761.
[14] WEISER M. Program slicing[C]// ICSE 1981. 1981:439-499.
[15] REN X, SHAH F, TIP F, et al. Chianti: A tool for change impact analysis of Java programs[C]// OOPSLA'04. Vancouver, BC, Canada,2004:432-448.
[16] ANDERSEN J, LAWALL J L. Generic patch inference[J]. Automated Software Engineering,2010,17(2):119-148.
[17] MENG N, KIM M, MCKINLEY K S. LASE: locating and applying systematic edits by learning from examples[C]// ICSE 2013. 2013:502-511.