计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2018,27(7):211−218 [doi: 10.15888/j.cnki.csa.006405]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

Linux 设备驱动程序自动更新及辅助工具①

任艳艳[1], 翟高寿[1], 张俊红[2]

[1](北京交通大学 计算机与信息技术学院, 北京 100044)
[2](北京建筑大学 电气与信息工程学院, 北京 100044)
通讯作者: 翟高寿, E-mail: gszhai@bjtu.edu.cn

摘　要: 研究了 Linux 设备驱动程序的自动更新方法, 建立了基于源码的 Linux 设备驱动辅助更新模型, 设计并实现了一组相应的自动更新支撑工具, 包括 Linux 设备驱动程序对内核依赖接口的分析工具、内核依赖接口差异性分析工具以及设备驱动更新辅助信息的生成工具. 相关原型经测试验证表明, 可以有效改善设备驱动开发和维护工作. 另外, 还提出了用于评估设备驱动辅助更新工作的量化指标即关于辅助更新提示信息的误报率和漏报率的概念及计算方法.

关键词: 设备驱动; 自动更新; 内核依赖接口; 源码; Linux

引用格式: 任艳艳,翟高寿,张俊红.Linux 设备驱动程序自动更新及辅助工具.计算机系统应用,2018,27(7):211–218. http://www.c-s-a.org.cn/1003-3254/6405.html

Automatic Updating and Auxiliary Tools of Linux Device Drivers

REN Yan-Yan[1], ZHAI Gao-Shou[1], ZHANG Jun-Hong[2]

[1](School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044, China)
[2](School of Electrical and Information Engineering, Beijing University of Civil Engineering and Architecture, Beijing 100044, China)

Abstract: The automatic updating method of Linux device drivers is studied and the corresponding assisted updating model based on source codes is built while a set of tools to support automatic updating of drivers are designed and implemented. This set of tools includes a tool to extract kernel-dependent interfaces for the Linux device drivers, a tool to analyze differences between kernel-dependent interfaces for two versions of kernels, and a tool to generate helpful information for updating of device drivers. Related prototypes have been tested and the results show that it can effectively improve the development and maintenance of device drivers. In addition, the concepts and calculation methods for the rate of false reporting and the rate of missing reporting about helpful information for updating are put forward, which are to be used for the quantitative indicators to evaluate assisted updating for device drivers.

Key words: device drivers; automatic updating; kernel-dependent interfaces; source codes; Linux

设备驱动程序通常会占到 70% 以上份额的操作系统内核源码, 且设备驱动程序的更新维护往往会牵涉到超过 35% 的源码修改, 故而保持设备驱动程序与操作系统内核不断变化的其余部分的一致性是目前操作系统内核开发的一项难题[1]. 伴随 Linux 内核版本更新升级速度的提升, 其内核稳定版本的发布周期已缩短至不到 30 天. 对于比较依赖内核接口的驱动程序来说, 由内核更新变化而引发的设备驱动的更新操作亦即设备驱动的协同演化将会更加频繁, 所以实现设备驱动程序的自动更新就显得尤为迫切. 换句话说, 若是设备驱动程序能够最大化地实现自动更新, 就可在更大程度上减少协同演化所耗费的时间, 进而提高设备

① 收稿时间: 2017-10-26; 修改时间: 2017-11-14; 采用时间: 2017-11-17; csa 在线出版时间: 2018-06-27

驱动程序更新效率. 作为占据内核源码较大比重且是操作系统内核漏洞主要来源的驱动代码, 设备驱动程序自动更新还可以提高驱动代码的正确性, 从而进一步提高操作系统的安全性和健壮性.

现在尚未有工具可以完整实现设备驱动程序的自动更新, 不过已有相关的差异分析工具和差异应用工具. 常见的差异分析工具有 Unix diff, 还有 Python dfflib 等, 都是基于文本的差异比较工具, 也有以函数为粒度的内核差异比较工具 DIFFE[2], 还有基于语法树的内核差异分析工具[3,4], 虽然都可以进行驱动代码差异比较, 但也仅仅是差异比较, 而且是基于已知两版驱动代码的比较. 差异应用工具如 Coccinelle[5–9], 此项目最初被设计于解决 Linux 内核与驱动协同演化问题, 设计理念是内核开发人员编写区别于传统补丁的语义补丁, 说明两个版本的变动信息, 然后由驱动开发人员使用 Coccinelle 应用此补丁, 完成内核驱动移植工作, 但这没有被 Linux 内核接受, 如今 Coccinelle 转而专注于在内核中发现并修复漏洞.

为了改变设备驱动程序更新操作现状, 本文研究了 Linux 设备驱动程序自动更新方法, 设计并实现了一组相应的自动更新支撑工具, 包括设备驱动程序对内核依赖接口分析工具、依赖接口差异性分析工具以及设备驱动更新辅助信息生成工具, 这组工具统称为设备驱动自动更新辅助工具 (Device Driver Assistant Updater, DDAU), 以便辅助用户解决随着内核更新带来的驱动更新问题, 源代码可在 GitHub 上获取. DDAU 实现设备驱动的更新过程, 是通过 A、B 两版 Linux 内核源码, A 版更新后是 B 版. 其中, A 版是完整版内核源码, B 版在全文中没有特别说明的情况下默认都不含有设备驱动代码. 设备驱动更新就是, 根据 A 版内核设备驱动接口依赖分析和设备驱动依赖接口在 A、B 两版本之间的差异分析, 得到 A 版内核设备驱动依赖接口差异信息, 然后根据此差异信息就可以得到设备驱动程序必须要做出哪些改动才能和内核变化保持一致, 这些必须要改动的信息或者可能需要改动的信息就是 DDAU 生成的设备驱动更新辅助信息. 设备驱动依赖接口分析就是分析设备驱动程序对内核接口的依赖关系, 提取设备驱动依赖的内核接口, 包括内核数据、内核函数、宏以及驱动对它们的调用关系. Linux 设备驱动更新问题模型如图 1 所示.

图 1 Linux 设备驱动更新问题模型

1 Linux 设备驱动与驱动更新

1.1 Linux 设备驱动程序

Linux 设备驱动程序在操作系统中扮演着特殊的角色, 其完全隐藏了设备工作的细节, 只为操作系统提供定义好的内部编程接口, 操作系统通过这些接口调用驱动以达到操作实际硬件的目的. 因此设备驱动程序是操作系统和硬件的沟通桥梁, 它位于操作系统和硬件之间, 是直接和硬件打交道的软件程序, 设备驱动与操作系统内核关系如图 2 所示. 设备驱动依赖于操作系统内核接口, 因而随着 Linux 内核升级带来的内核接口变化, 设备驱动也必须协同更新, 以便设备驱动能和操作系统内核保持一致性, 才能和内核进行正确的信息交互.

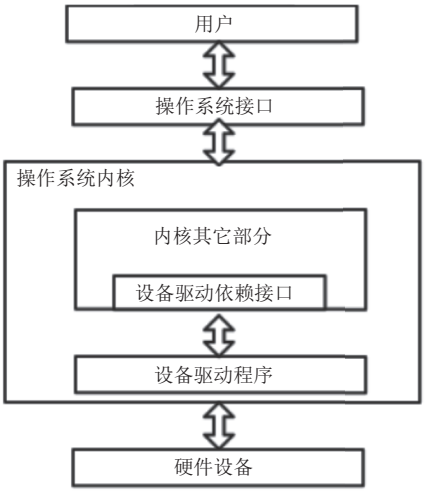

图 2 设备驱动与操作系统内核关系示意图

在 Linux 内核中, 设备驱动程序一般以内核可加载模块 (LKM) 的方式存在, 可以在内核编译的时候静态加载, 因此单个设备驱动的更新只需要更改相对应的驱动目录里的相关文件. 内核和设备驱动的信息交互通过宏和函数, 数据的传递一般是通过结构体等数据类型, 因此在分析接口依赖关系的时候, 着重在函数、宏, 以及数据类型如结构体等接口分析上, 设备驱动接口分析是图 2 中操作系统内核部分的设备驱动内核依赖接口部分的分析.

1.2 Linux 设备驱动更新

对于应用程序来说, 它因实现某个功能需要而调

用系统提供的函数时, 它并不关心这个函数内部的实现细节, 只需要知道要调用的函数名字, 要传入的参数以及返回值类型. 同样地, 对于驱动来说, 驱动更新需要了解内核接口的名字、参数以及返回值的类型发生了哪些改变. 接口改变主要包括函数、宏, 以及数据类型的变化[10].

内核函数变化类型主要有: (1) 函数被删除或添加; (2) 函数添加或删除参数, 即函数的参数个数发生变化; (3) 函数参数类型改变; (4) 函数返回类型变化, 如添加错误处理机制由 void 变为返回错误码.

内核中宏的类型主要包括有参宏和无参宏. 无参宏的变化类型有: (1) 宏被删除或添加; (2) 宏的值改变, 即宏名没有变化但是宏的值发生了变化, 它的调用方式不变, 但可能会影响使用宏值的变量或其它数据类型的结果. 对于有参宏来说, 变化类型同函数类似, 有参宏可以说是某种意义上的另一种形式的函数.

数据类型主要包括结构体, 联合体和枚举, 结构体和联合体的变化主要有: (1) 删除或者重新命名结构体成员或结构体; (2) 更改结构体成员类型; (3) 更改结构体成员顺序; (4) 增加或删除结构体别名. 枚举类型的变化一般就是枚举成员的增减或者整个枚举结构被删除或添加.

接口变化还包括接口类型的改变. 例如接口在 A 版内核中声明为函数, 在 B 版内核中可能会更改为宏, 或者反之. 接口类型的改变对于设备驱动来说, 有的影响不大, 有的却必须跟随内核接口的变化提供相应的改变. 本文根据对设备驱动更新操作产生的影响将差异分析结果进行等级划分, 再根据等级划分对用户做出不同的辅助更新提示信息的输出. 设备驱动的自动更新就是将引言中图 1 所示的研究过程, 最大化地实现自动化, 研发出一个插件或者独立的小工具嵌入到编辑器或者大型集成开发工具中帮助内核开发人员或者驱动维护人员更新相关程序的源代码.

2 设备驱动自动更新方法设计

设备驱动自动更新的方法设计主体是考虑从驱动源代码调用的内核接口入手, 有针对性的分析驱动源代码接口依赖, 根据其调用的依赖接口差异分析, 提示用户做出什么样的协同演化操作.

设备驱动自动更新的方法设计具体是: 首先加载编译器插件编译内核导出特定格式的编译中间结果, 分析编译中间结果提取待处理设备驱动程序源码的接口依赖关系; 然后分析驱动源码提取待处理驱动程序所依赖的头文件, 根据头文件列表提取两版本内核接口原型声明信息, 进而根据提取的接口原型声明信息获取两版本内核中的接口差异信息, 并将差异信息筛选分类; 在此基础上, 根据提取的接口依赖关系获取设备驱动程序依赖接口差异信息及分类规则得出驱动更新的辅助信息, 最终向用户展示结果.

图 3 为设备驱动更新模型模块设计, 主要包括预处理模块, 设备驱动接口依赖关系提取模块, 设备驱动头文件依赖列表提取模块, 驱动依赖接口原型声明信息提取模块, 驱动依赖接口差异分析和驱动更新辅助信息生成模块. 设备驱动更新模型每个模块既可以单独运行也可以作为集成工具一起按规则自动执行, 自动执行脚本可以将各个模块集成为一组工具集.

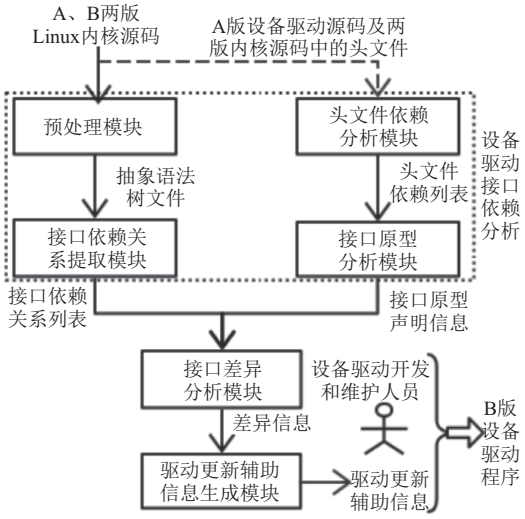

图 3 设备驱动更新模型

预处理模块. 此模块主要是编译 A 版 Linux 内核源码获取编译中间结果. GCC 编译器自 4.5.0 版本后开始支持插件技术[11], 为用户提供了丰富的 API 接口, 方便用户获取 GCC 编译中间结果, 因此可通过 GCC Plugin 技术获取编译过程中的抽象语法树然后导出特定格式的文件, 以便提取设备驱动程序接口依赖关系.

驱动接口依赖关系提取模块. 根据 C 语言的语法规则, 接口应该包括函数、宏、结构体、枚举结构等主要方面. 此模块就是分析预处理模块得到的编译中间结果即序列化的抽象语法树文件, 提取 A 版内核中设备驱动的内核接口依赖关系, 生成包括设备驱动的

自身接口声明和去掉自身接口调用关系的外部接口调用关系文件, 调用关系包括驱动所调用的函数, 引用的宏, 使用的数据类型, 还有这些接口的调用位置和主调函数.

驱动头文件依赖提取模块. 提取 A 版内核中待处理驱动程序源码所直接引用的头文件, 然后根据此头文件信息进一步提取 A、B 两版本内核中头文件中引用的头文件, 称之为二级头文件即间接引用的头文件, 然后由二级头文件以同样的方法再提取三级头文件, 生成设备驱动对两个内核版本的头文件依赖列表, 这个头文件依赖列表基本包含了驱动程序依赖接口所涉及的头文件. 而为了新增功能或者代码重构而添加头文件导致一级头文件列表的变化情况, 由于添加的头文件对此时的用户来说是透明的, 所以这种情况不予考虑.

接口原型声明信息提取模块. 此模块是获取驱动头文件依赖提取模块生成的两个内核版本的头文件依赖列表中所有头文件的内容, 包括宏、函数以及 extern 原型的声明, 结构体、枚举等数据类型的声明, 声明信息包括接口名称、接口类型, 所在文件, 起始行号, 原型声明语句. 此模块生成两个版本内核头文件中的接口原型声明文件.

接口差异分析和驱动更新辅助信息生成模块. 此模块是分析这些依赖关系在两版内核中的差异信息, 例如函数调用, 这些差异信息就包括函数名称、返回值的改变, 参数列表的改变, 如参数个数或者参数类型发生变化等. 再如宏常量的使用, 大多数宏常量名称没有变化, 但是其值却发生了变化, 虽然值发生了变化但由于调用宏时用的是名字, 所以宏常量的值变化不影响编译却可能会影响后续的使用结果. 因此, 所有这些差异信息都要提取出来, 并加以分析处理, 然后根据对驱动更新操作的影响进行等级划分, 最后把差异信息的分级处理结果转化为驱动更新的辅助信息.

3 原型设计与实现

根据设备驱动自动更新方法设计, 设计并实现设备驱动更新辅助工具 DDAU, 开发平台是虚拟机 32 位操作系统 Ubuntu 14.04 LTS, GCC 编译器版本 4.8.4, Linux 内核版本 3.5.6 和 3.8.13.

设备驱动更新辅助工具 DDAU 操作主要步骤包括: (1) 预处理, 加载 GCC 插件编译 A 版 Linux 内核源码得到编译中间结果--序列化的抽象语法树; (2) 分析编译中间结果提取待处理设备驱动程序源码的接口依赖关系; (3) 分析源码提取待处理设备驱动程序所依赖的头文件列表, 包含两个内核版本头文件依赖列表; (4) 根据头文件列表提取两版本内核头文件的接口的原型声明信息; (5) 根据提取的接口依赖关系和接口的原型声明信息获取接口差异信息, 再由差异信息得出驱动更新的辅助信息, 然后向用户展示结果等 5 个主要步骤. 各步骤之间以及生成的文件的关系如图 4 所示.

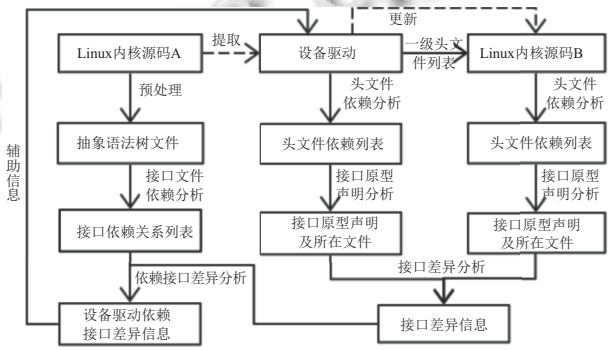

图 4 DDAU 各步骤和生成文件之间的关系图

预处理步骤主要是编译内核执行 make 时设置参数 EXTRA_CFLAGS="-fplugin=plugin_name"加载 GCC 插件导出序列化的抽象语法树文件, 头文件依赖分析步骤是得到驱动两版内核头文件依赖列表, 现在着重介绍驱动接口依赖分析、接口原型声明提取、依赖接口差异分析和驱动更新辅助信息生成的设计与实现.

3.1 驱动接口依赖关系初步提取

预处理阶段导出的序列化抽象语法树文件, 是以函数为单位构成, 树的根节点是函数声明节点 func_decl, 它的子树有返回值子树 result_decl, 参数子树 parm_decl, 以及它的函数体子树 bind_expr, 根据不同子树节点不同含义可以提取驱动自定义函数声明信息和函数、宏调用关系信息以及数据结构的使用信息等.

以函数声明节点为入口和界限, 提取驱动自身定义的接口, 提取函数体子树 bind_expr 中驱动调用的函数、宏, 以及结构体等, 对于函数、宏的调用需要记录下所有主调函数和调用位置. 函数调用信息还需要剔除设备驱动程序调用自身定义函数的调用信息. 最后将这些函数定义以及函数、宏等的调用关系即驱动接口依赖关系分别保存成文本文件供后续处理.

3.2 接口原型声明提取

根据头文件列表获取模块, 可以分析这些头文件的内容, 得到设备驱动程序依赖的头文件中声明的函数、宏、结构体等的原型声明原句, 以便在接口差异信息获取和驱动更新辅助信息生成模块比较差异和根据差异生成驱动更新辅助信息.

本文使用 Exuberant-Ctags[12](以下统称为 Ctags)作为信息获取的辅助工具. 首先修正 Ctags 的输出. 由于 Ctags 的主要目的是用于生成索引, 它并不关心接口是如何声明的, Ctags 生成各种索引的时候, 虽然列出了索引的文件名和行号, 但是在取原型声明语句的时候只是取了对应行号的那一行, 因此它的输出结果并不能满足实验需求, 需要对 Ctags 的输出结果中接口声明部分进行修改, 将函数、宏的原型声明提取完整, 对于数据类型和数据结构, 输出的结果中数据的名字和数据成员是拆分开的, 需要重新合并组合成一个完整的数据类型或数据结构的声明, 以满足实验中对接口原型声明信息的提取和后续的差异分析.

3.3 接口差异分析和驱动更新辅助信息生成

接口差异类型如图 5 所示.

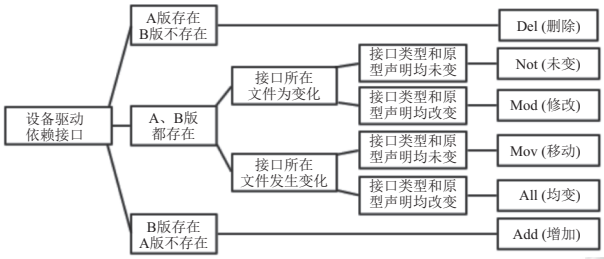

图 5 接口差异类型分类

用 v0 表示原版内核接口信息, v1 表示新版内核接口信息, 接口信息存储在 Python 语言的字典 dict 中, 用 k 表示接口名称, file 表示接口所在文件, type 表示接口类型即接口是函数还是宏或者是结构体等等, state 表示接口原型声明语句, diff_type 表示差类型, #号表示注释语句, 差异分析算法关键代码如下:

```
if k in v1_dict:    #接口在 v1 版内核存在
if v0_file == v1_file:    #接口所在文件没有变化
if v0_type == v1_type and v0_state == v1_state :
#接口类型和原型声明均未变
diff_type = 'Not'
else:    #接口类型和原型声明变化
diff_type = 'Mod'
else:    #接口所在文件变化
if v0_type == v1_type and v0_state ==v1_state:
diff_type = 'Mov'
else:    #接口类型和原型声明变化
diff_type = 'All'
else:    #接口在 v1 版内核不存在
diff_type = 'Del'
```

上述操作后剩下的 v1_dict 中的接口就是 Add 增加类型的接口.

根据在接口依赖关系分析模块得到的接口调用关系信息, 可以从接口差异信息中提取设备驱动所调用的接口是否发生了变化, 以及发生了哪些变化. 利用差异筛选算法把这些差异信息又根据对设备驱动程序更新的影响程度分为表 1 所示 4 个等级.

表 1 差异信息影响等级划分

等级	描述
A	影响较大, 需要协同更改
B	接口名称未变, 但值改变, 调用方式不变, 但可能影响结果
C	接口类型发生变化, 但接口名称和参数列表等未发生变化, 影响较小
D	接口类型未发生变化或者发生的变化没有影响

辅助更新信息只输出 A、B、C 三个等级.

4 原型测试与结果分析

4.1 测试方法

首先进行模块化黑盒测试, 分析各个模块得出的结果是否正确, 确认各个模块之间衔接正确, 再对辅助工具进行整体测试, 查看最后得出的更新辅助信息, 然后根据辅助信息介入人工操作对驱动进行更新, 最后将更新的驱动复制到 B 版内核 (假设此操作之前 B 版内核还没有更新对应驱动) 的对应位置, 对 B 版内核进行编译, 测试是否编译通过, 编译通过后加载运行测试驱动模块进一步验证.

选择测试驱动时, 为了尽可能测试多种驱动类型, 选择 Linux 内核源码中 drivers 目录下不同的驱动目录进行实验测试验证, 修改配置文件中内核源代码所在路径, 然后运行批处理自动执行脚本文件 batch_run.sh, 也可以运行单处理脚本 run.sh 分析单个测试文件.

4.2 测试结果

对 Linux 内核源码中 drivers 目录下不同驱动目录

进行实验测试验证, 统计分析结果时不是单纯统计代码行数, 而是统计设备驱动依赖的内核接口变化而应当采取的设备驱动更新操作, 设备驱动更新操作修改的可能是一行代码, 也可能是相关联的多行代码, 不考虑源代码内部优化问题.

为量化实验结果, 本文提出漏报率和误报率概念. 漏报率 (Rate of Missing Report, RMR) 是指驱动程序根据设备驱动依赖接口变化应当给出但未给出修正提示的次数 (MR) 占据根据设备驱动依赖接口变化应当给出修正提示的总次数 (TAR) 的比例. 误报率 (Rate of False Report, RFR) 是指驱动程序根据设备驱动依赖接口变化所给出的错误修正提示次数 (FR) 占据根据设备驱动依赖接口变化给出修正提示的总次数 (TR) 的比例. 单个设备驱动文件的漏报率 (RMR_i) 的计算公式是 $RMR_i = MR_i/TAR_i$. 平均漏报率的计算公式是 $RMR = \sum RMR_i/N$, N 为文件总数, 平均漏报率的数值越低越好. 单个设备驱动文件的误报率 (RFR_i) 的计算公式是 $RFR_i = FR_i/TR_i$. 平均误报率的计算公式是 $RFR = \sum RFR_i/N$, N 为文件总数, 平均误报率的数值越低越好. 实验结果统计如表 2 所示.

表 2 drivers/目录测试文件的测试结果统计表

测试目录	平均漏报率 (%)	平均误报率 (%)
net/	0	0
pci/	52.3	13.3
input/	0	0
char/	0	37.5
video/	16.7	16.7
scsi/	0	26.7
parport/	0	33.3
block/	0	81
其他目录	0	38.7
平均	6.4	29.1

对于漏报原因分析: (1) 接口在 A 版内核中定义在驱动内部, 而 B 版内核中把定义移到了头文件中, 并可能更改接口名, 例如 pci-acpi.c 驱动中接口 add_pm_notifier(参数) 为自定义接口, 在 B 版内核则把此接口移动到了 acpi/acpi_bus.h 头文件里并改名为 acpi_add_pm_notifier(参数), 而驱动程序中的调用则同步替换. (2) 内核头文件中既保留原有接口函数、又新增同功能接口函数、并调用新增同功能接口函数的情况. 这两种漏报情况, 前者需要进一步优化接口差异分析算法, 后者无法避免.

对于误报原因分析: (1) 函数参数只名字改变, 函数原型差异分析考虑了参数名, 造成误报. (2) 函数返回值发生变化, 但调用时没有使用函数返回值, 因而函数返回值改变并没有对程序造成影响, 无需给出更新提示, 造成误报. (3) 可变参数宏变化成了固定参数宏, 但由于调用时参数个数依然可以接受, 所以无需修正, 却给出修正提示, 造成误报. (4) 宏变化成函数, 返回值、参数由无类型变更为具体类型, 例如两版本接口声明语句: "-- #define con_debug_enter(vc) 函数体 ++ static inline int con_debug_enter(struct vc_data *vc)", 此情况给出更新提示误报可以谅解. (5) 对于函数参数中某个参数类型和名称均发生变化, 例如内核接口 async_synchronize_full_domain , 参数列表由 (struct list_head *list) 变为 (struct async_domain *domain), 但驱动程序源码中此接口调用语句参数部分却都是 (&scsi_sd_probe_domain), 调用不受影响, 无需给出更新提示, 此情况的误报可以谅解.

对于误报原因各种情况分析得出, 实际平均误报率要比计算得出的误报率小的多, 因而 DDAU 可以为驱动更新提供有效的辅助更新提示.

辅助更新提示信息中的 A 级差异信息生成的辅助信息如图 6 所示, 依次是接口差异信息的级别 A 级, 接口的差异类型 Del, 接口类型, 接口所在文件, 接口的原型声明信息, 这些信息方便用户了解接口的调用方式, 接下来是驱动文件调用此接口的位置, 直接定位源代码方便用户修改. 图 7 中所示是接口的差异类型 All, 辅助信息会把接口变化前后的文件和原型声明列出来, 方便用户查看以便根据需要修改代码.

根据辅助信息的提示, 对 A 版驱动进行人工辅助更新操作, 根据辅助信息提供的接口变化类型和接口原型信息进行修改, 将修改好的驱动放到 B 版内核对应位置后编译内核, 经验证可以编译通过, 证明论文的设备驱动辅助更新方法设计是可行的.

```
5 ===== A  level =====
6 total count: 3
7
8 __devinit   ( diff_type: Del type: macro file: include/linux/init.h )
9   -- #define __devinit __section(.devinit.text) __cold notrace
10
11  pcnet32.c: 1446:static void __devinit pcnet32_probe_vlbus(unsigned
    int *pcnet32_portlist)
12  pcnet32.c: 1465:static int __devinit
13  pcnet32.c: 1524:static int __devinit
14
15 __devexit   ( diff_type: Del type: macro file: include/linux/init.h )
```

图 6 设备驱动更新 A 级辅助信息部分结果示例

```
26 ===== B  level =====
27 total count: 11
28
29 KERN_DEBUG  ( diff_type: All type: macro file: include/linux/
   printk.h-->include/linux/kern_levels.h )
30  -- #define KERN_DEBUG "<7>"
31  ++ #define KERN_DEBUG KERN_SOH "7"
32
33  pcnet32_ethtool_test:
34    drivers/net/ethernet/amd/pcnet32.c:859:2
35    drivers/net/ethernet/amd/pcnet32.c:863:2
36    drivers/net/ethernet/amd/pcnet32.c:866:2
37  pcnet32_loopback_test:
38    drivers/net/ethernet/amd/pcnet32.c:914:2
```

图 7 设备驱动更新 B 级辅助信息部分结果示例

4.3 结果分析

文献[3,4]利用 GCC 插件导出已知两版内核设备驱动源码的抽象语法树文件, 提取接口名字和在抽象语法树文件中的起止行号, 传入修改了前后端的 Gumtree[13]进行差异分析, 分析驱动内部代码实现的细节差异. 但由于其是基于已知两版驱动代码比较, 且旨在比较接口实现内部细节, 因而并不能实现驱动的更新操作. 与之相比, DDAU 是利用 GCC 插件分析待更新的驱动代码提取待更新设备驱动接口依赖, 同时依据引用的头文件分析依赖接口差异, 并最终得出辅助更新提示, 在此基础上, 就可以根据提示对原有设备驱动源码进行修正而得到新版驱动源码.

直接编译方法[14]是在新版内核中编译驱动源码, 修改错误, 重新编译, 不断重复此过程直到驱动编译不再出错, 但改错的方法是人工分析提取的错误日志, 这需要人工查找接口在两版内核中的接口原型声明信息, 对比接口原型声明差异分析调用如何更改驱动源码, 与直接编译方法不同, DDAU 为用户提示的驱动更新辅助信息, 直接提供了接口的变化类型, 接口所在文件和两版接口原型声明对比, 以及驱动代码调用此接口的位置, 直观的为用户展示出接口声明的不同和代码需要修改的位置, 为驱动开发和维护人员省去大量繁琐的人工查找和定位修改代码的时间, 大大减少了驱动更新操作所耗费的时间.

5 结论与展望

本文实现了一个 Linux 设备驱动程序自动更新辅助工具 DDAU, 以便用户解决随着内核更新带来的驱动协同演化问题. 该工具可以为内核驱动的开发和维护提供有效的辅助更新信息, 为驱动在内核空间和用户空间的拆分自动化问题[15]提供借鉴意见, 同时也为操作系统软件更新问题提供技术思路. 但此工具也存在一些不足之处: (1) 由于使用 GCC 插件所以对于驱动内核中的条件编译语句#ifdef—#endif 括起来的非本机器体系架构的代码无法处理; (2) 对于接口发生变化, 但依据调用情况的不同需要做出不同提示的情况不够灵活进行处理, 需要进一步的完善; (3) 接口差异分析算法对处理接口名称改变但功能不变的特殊情况需要进一步细化完善; (4) 对于驱动接口依赖分析中的数据类型或数据结构依赖关系的提取不够全面; (5) 更新辅助信息提示不能图形化动态展示在源码需要更新的位置, 没有将此辅助工具嵌入 vi 等编辑器或者集成开发环境中进而高效率更新代码. 这些不足之处需要进一步的研究和探索, 会不断的修正和完善.

参考文献

1 Padioleau Y, Lawall JL, Muller G. Understanding collateral evolution in Linux device drivers. Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems. Leuven, Belgium. 2006. 59–71.

2 颜世勋. 内核源代码差异分析与展示[硕士学位论文]. 北京: 北京理工大学, 2015.

3 武祥晋. 基于语法树的源码差异分析工具 gccdiff. https://github.com/taccoraw/gccdiff.

4 李雪. 基于语法树的源码差异分析软件[简称:ast-diff] V1.0. 计算机软件著作权登记号: 2015SR284101, 2015

5 Rodriguez LR, Lawall J. Increasing automation in the backporting of linux drivers using coccinelle. Proceedings of the 11th European Dependable Computing Conference. Paris, France. 2016. 132–143.

6 Padioleau Y, Lawall JL, Muller G, *et al*. SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers. Electronic Notes in Theoretical Computer Science, 2007, (166): 47–62. [doi: 10.1016/j.entcs.2006.07.022]

7 Padioleau Y, Lawall JL, Hansen RR, *et al*. Documenting and automating collateral evolutions in Linux device drivers. ACM SIGOPS Operating Systems Review, 2008, 42(4): 247–260. [doi: 10.1145/1357010]

8 Padioleau Y, Hansen RR, Lawall JL, *et al*. Semantic patches for documenting and automating collateral evolutions in Linux device drivers. Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems. San Jose, CA, USA. 2006. 10.

9 Lawall J. Coccinelle: Reducing the barriers to modularization in a large C code base. Proceedings of the Companion Publication of the 13th International Conference on

Modularity. Lugano, Switzerland. 2014. 5–6.

10 ABI Compliance Checker. https://lvc.github.io/abi-compliance-checker.

11 GCC WiKi: Plugins. https://gcc.gnu.org/wiki/plugins.

12 Exuberant Ctags. http://ctags.sourceforge.net.

13 Falleri JR, Morandat F, Blanc X, *et al*. Fine-grained and accurate source code differencing. Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. Vasteras, Sweden. 2014. 313–324.

14 王欢. Linux 内核错误追溯系统的研究与设计[硕士学位论文]. 北京: 北京工业大学, 2016.

15 谭苗, 翟高寿. 设备驱动非内核化通信架构的研究与实现. 信息网络安全, 2016, (11): 57–65. [doi: 10.3969/j.issn.1671-1122.2016.11.010]