Understanding Collateral Evolution in Linux Device Drivers

Yoann Padioleau
OBASCO Group
Ecole des Mines de Nantes-INRIA, LINA
44307 Nantes cedex 3, France
Yoann.Padioleau@emn.fr

Julia L. Lawall
DIKU
University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

Gilles Muller
OBASCO Group
Ecole des Mines de Nantes-INRIA, LINA
44307 Nantes cedex 3, France
Gilles.Muller@emn.fr

"Greg Kroah-Hartman has gotten [Linux] 2.6.13 off to a good start with a massive set of driver core patches. There are a fair number of API changes that come with this patch set, so the whole thing is worth a look. In-tree code has been fixed to use the new API, but, as always, maintainers of external code are on their own."
`http://lwn.net/Articles/140002/`, June 23, 2005.

ABSTRACT

In a modern operating system (OS), device drivers can make up over 70% of the source code. Driver code is also heavily dependent on the rest of the OS, for functions and data structures defined in the kernel and driver support libraries. These properties pose a significant problem for OS evolution, as any changes in the interfaces exported by the kernel and driver support libraries can trigger a large number of adjustments in dependent drivers. These adjustments, which we refer to as *collateral evolutions*, may be complex, entailing substantial code reorganizations. As to our knowledge there exist no tools to help in this process, collateral evolution is thus time consuming and error prone.

In this paper, we present a qualitative and quantitative assessment of collateral evolution in Linux device driver code. We provide a taxonomy of evolutions and collateral evolutions, and use an automated patch-analysis tool that we have developed to measure the number of evolutions and collateral evolutions that affect device drivers between Linux versions 2.2 and 2.6. In particular, we find that from one version of Linux to the next, collateral evolutions can account for up to 35% of the lines modified in such code.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design; K.6.3 [**Management of Computing and Information Systems**]: Software Management; D.2.8 [**Software Engineering**]: Metrics

General Terms

Measurement

Keywords

Linux, device drivers, software evolution

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'06, April 18–21, 2006, Leuven, Belgium.
Copyright 2006 ACM 1-59593-322-0/06/0004 ...$5.00.

1. INTRODUCTION

One of the biggest problems in operating system (OS) development today is keeping device drivers up to date with evolutions in the rest of the OS. Device driver code can make up over 70% of a modern OS [3], and is heavily dependent on the kernel and driver support libraries for functions and data structures. Accordingly, any changes in the interfaces of the kernel or driver support libraries are likely to entail modifications in the device-specific code to restore its correct behavior. We refer to these modifications as *collateral evolutions.* Collateral evolutions may entail substantial code reorganizations and affect many code sites, making it time-consuming and error-prone. The need for collateral evolutions may thus hinder OS evolution in practice.

We examine the issues raised by collateral evolution in the context of Linux. Linux is currently undergoing rapid evolution, with even the so-called stable version 2.6 introducing more and more interface changes [17]. Furthermore, the size of the Linux driver code has more than doubled in the last five years. These factors suggest that driver modifications due to collateral evolutions are increasingly becoming necessary. Indeed, in our experience, a single collateral evolution may affect hundreds of code sites spread across many different files.

The problem of collateral evolution in Linux is further complicated by the difficulty of communicating precise information about the required modifications to driver maintainers. As Linux is an open source OS, many kinds of programmers contribute to its development. Indeed, driver maintainers are often not kernel experts, but instead experts in a given device or even ordinary users who find that their hardware is not adequately supported. Because the developers who update the kernel and driver support libraries often do not share a common language and expertise with device maintainers, documentation about complex collateral evolutions, if any, is often incomplete. As a result, we have observed that an evolution and dependent collateral evolutions may take several years to complete and may introduce bugs into previously mature code.

This paper

While substantial attention has been paid to how to design an OS, there has been little consideration of its subsequent evolution. In the case of device driver collateral evolution, the magnitude and complexity of the problem call for increased attention. In this paper, we analyze the collateral evolutions in Linux to obtain a better understanding of the range and scope of the problem. This analysis can lead to increased awareness of the problem by OS developers and can motivate the design of automated tools to assist in the collateral evolution process. Overall, we make the following contributions:

- We identify and name the collateral evolution phenomenon and characterize its presence as part of the lifecycle of Linux device drivers.
- We clarify the structure of Linux as it relates to collateral evolution, focusing attention on the interfaces of the kernel and driver support libraries, and provide a taxonomy of the main evolutions that occur in these interfaces.
- We identify a variety of collateral evolutions that evolutions in this taxonomy can entail, and present in detail three examples from Linux 2.5 out of the 72 that we have studied. These examples illustrate both the complexity of performing collateral evolutions and the bugs that can be introduced in mature code.
- Based on the identification of interface elements as potential triggers of collateral evolutions, we show that the likelihood of collateral evolutions is increasing, as not only the driver code size but also the complexity of the driver interfaces has doubled in the last five years.
- Using a tool we have developed that analyzes Linux patch files, we measure the number of changes in the interfaces of the kernel and driver support libraries that require collateral evolutions in device-specific code in versions of Linux from 2.2 to 2.6. We find that the number of these changes has been steadily increasing, with the so-called stable Linux 2.6 showing almost as many changes as its unstable predecessor, Linux 2.5.
- We then give an estimate of the work required for collateral evolution, based on the amount of modification performed in device-specific code. Using our tool, we find that up to 35% of the lines modified in Linux device-specific code from one version to the next are due to collateral evolutions. This amount of modification is particularly significant because collateral evolutions generally serve only to maintain the current behavior, rather than improve it.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 provides an assessment of the kinds of changes that occur in interfaces that affect device drivers and the collateral evolutions that these changes entail. Section 4 describes some of these collateral evolutions in detail. Section 5 quantifies various aspects of Linux evolution and collateral evolution. Finally, Section 6 presents some conclusions and ideas for future work.

2. RELATED WORK

In this paper, we study the effect of evolution in Linux code from the point of view of the collateral evolutions that are entailed. The evolution of open-source operating systems has also been studied from other perspectives. Godfrey and Tu have studied the changes in the size of Linux code between 1993 and 2001 [11]. Although we consider a later time period, our analysis of the increasing size and complexity of Linux driver code is consistent with their results. Li *et al.* have studied the amount of copy-pasted code in Linux versions up to 2004. They find that such code often occurs in device drivers [16]. Finally, Hassan has developed a tool that automatically extracts evolution information from versioning repositories and has been applied to variants of BSD [12]. This approach, however, only associates code fragments to the enclosing functions, and thus cannot determine direct relationships between individual old and new code fragments, as is needed to reliably detect collateral evolutions.

Our analysis of collateral evolutions requires identifying the interfaces between driver support libraries and device-specific code. If Linux were structured using components, as done in OSes such as Think [7], OSKIT [8] and K42 [1], it would be easier to identify such interfaces and detect their evolution. Collateral evolution, however, affects not only the objects mentioned in the interface, but also the context in which these objects are used, and thus the problem of collateral evolution remains in component-based OSes.

As a result of our analysis, we classify the driver support library evolutions and ensuing collateral evolutions that affect device-specific code. More general forms of evolution have been classified as *refactorings*, which are a fixed collection of general-purpose transformations that reorganize the structure of a program without changing its semantics [10]. Refactorings, however, apply to the whole program, requiring access to both the evolving code and to all usage sites of affected definitions. In the case of Linux, however, the entire code base is not available, as many device drivers are developed outside the Linux source tree. Thus, is it important to study collateral evolutions as a separate entity.

Previous work has suggested to avoid the need for collateral evolutions by using wrapper functions, as in the case of OSKit [8], or virtual machines as in the work of LeVasseur et al. [15], thus leaving the driver code unchanged. In these approaches, however, driver code does not benefit from improvements in the overall software architecture of the OS that could ease its future maintenance, *e.g.* to address new device requirements. The wrapper approach is furthermore not always sufficient, as some collateral evolutions such as the updating of calls to `usb_submit_urb` described in Section 4.1, depend on information that is only apparent in the driver code. When collateral evolution is actually needed, the chaos of coding styles induced by the wrapper approach makes the collateral evolution all the more difficult. Finally, we have observed the introduction and subsequent removal of wrapper functions in Linux code, suggesting that the Linux development community does not see them as a viable solution.

Recent years have seen a surge of interest in the automatic analysis of operating system source code, in order to detect bugs [2, 5, 6, 9, 16]. These approaches rely on a collection of required kernel API usage patterns and detect code fragments that are inconsistent with these patterns. Nevertheless, an incorrectly done or overlooked collateral evolution may satisfy expected patterns without actually correctly restoring the behavior of the device driver. Detection of the error would require combining information about the original implementation of the driver with analysis of the new version. Other work related to improving the safety of drivers includes Nooks, which provides a framework for protecting operating systems and applications from driver failures [22, 23].

3. THE COLLATERAL EVOLUTION PROBLEM

Collateral evolution is required when evolutions in the kernel and driver support libraries induce changes in the interface with device-specific code. To characterize the collateral evolution problem, we first examine the structure of Linux support for devices and identify the kinds of changes that can occur in its interfaces. We then consider the range of collateral evolutions that these changes can entail.

In the following, we avoid the term "device driver," which can be interpreted either as including only the code that interacts directly with the device or as additionally including the relevant support libraries. Instead, we refer to the former as *device-specific code* and the latter as *driver support libraries*. Furthermore, we consider the kernel to be a driver support library except where specified otherwise.

3.1 Linux support for devices

Linux support for devices is provided by a combination of generic services that are provided by the kernel, services generic to a device family that are provided by the driver support libraries, and services specific to a device that are provided by the device-specific code. As illustrated in Figure 1, these services are organized hierarchically, with all services depending on the kernel, specialized driver support libraries such as USB-serial depending on generic ones such as USB, and device-specific files such as `rtl8150.c` depending on one or more driver support libraries. Typically, these hierarchical relationships are reflected in the Linux directory structure. Nevertheless, as some directory hierarchies represent the bus a device is on, *e.g.* USB, and others represent the functionality of a device, *e.g.*, net, cross-directory references are possible. Such references are illustrated in Figure 1 by *e.g.*, `kaweth.c` and `rtl8150.c`, which are on the USB bus and are network devices.

The various driver support libraries (including the kernel) communicate with the device-specific code via interfaces. As Linux interfaces (*i.e.*, header files) do not distinguish what is shared between the driver support libraries and the device-specific code from what is shared within the driver support libraries, we infer these interfaces from the external references made by the device-specific code. The main points of interaction between the driver support libraries and the device-specific code are functions and data structures. Functions include both generic functions exported by the driver support libraries and callback functions provided by the device-specific code. Data structures include structures instantiated by the device-specific code and then used by the driver support library to maintain the state of each relevant device. These structures are typically exchanged by the device-specific code and the driver support libraries on the invocation of interface functions. Finally, the interface of a driver support library includes a protocol for using the exported functions and data types. This protocol can specify features such as function-call sequencing and error-handling requirements.

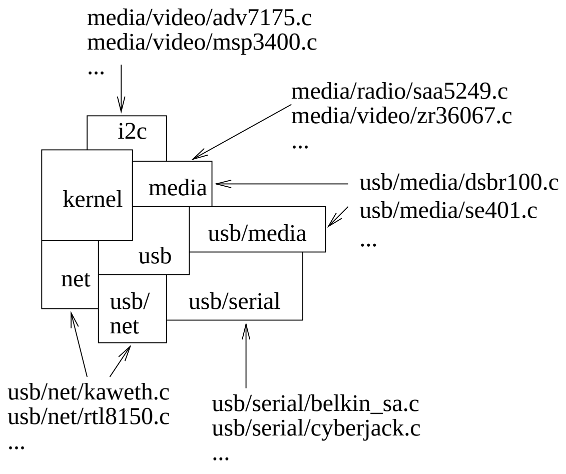


Figure 1: Hierarchical organization of services and associated dependencies

Figure 2 shows extracts of the rtl8150 device-specific code, which has a typical structure. This code depends on the USB and network device support libraries. The initialization function, `usb_rtl8150_init` (lines 46-49), registers the device with the USB support library, providing it with a structure, `rtl8150_driver` (lines 1-5), that contains a number of callback functions required by this support library. One of these functions is `rtl8150_probe` (lines 31-44), which when invoked by the USB library registers the device with the network device support library in a similar manner. Figure 2 also shows the function `rtl8150_start_xmit` (lines 7-29), which is one of the callback functions provided to the network device support library. This function illustrates the use of a variety of functions exported by both libraries, the exchange of data structures, and the instantiation of protocols, *e.g.*, data initialization and usage (lines 16-19). The relevant extracts of the USB and network device interfaces are shown in Figure 3.

3.2 Taxonomy of interface changes and collateral evolutions that affect device-specific code

When an evolution in a driver support library affects its interface, collateral evolutions must be made in all dependent device-specific files. For example, when a library function `f` gains a new argument, device-specific code that uses `f` must be modified to construct an appropriate argument value. In this section, we first provide a taxonomy of changes that are possible in driver support library interfaces and then consider the range of collateral evolutions that these changes can entail.

3.2.1 Interface changes

As motivated in Section 3.1, the interface of a driver support library includes exported functions, imported device-specific callback functions, data structures, and protocols. Evolutions in the driver support library can have arbitrary effects on one or more of these elements. Figure 4 provides

```
static struct usb_driver rtl8150_driver = {
  ...
  .probe = rtl8150_probe,
  ...
};

static int rtl8150_start_xmit(struct sk_buff *skb,
                              struct net_device *netdev) {
  rtl8150_t *dev = netdev_priv(netdev);
  int count, res;

  netif_stop_queue(netdev);
  count = (skb->len < 60) ? 60 : skb->len;
  count = (count & 0x3f) ? count : count + 1;
  dev->tx_skb = skb;
  usb_fill_bulk_urb(dev->tx_urb, dev->udev,
             usb_sndbulkpipe(dev->udev, 2),
             skb->data, count, write_bulk_callback, dev);
  if ((res=usb_submit_urb(dev->tx_urb,GFP_ATOMIC))){
    warn("failed tx˙urb %d“n", res);
    dev->stats.tx_errors++;
    netif_start_queue(netdev);
  } else {
    dev->stats.tx_packets++;
    dev->stats.tx_bytes += skb->len;
    netdev->trans_start = jiffies;
  }
  return 0;
}
...
static int rtl8150_probe(struct usb_interface *intf,
                         const struct usb_device_id *id) {
  struct usb_device *udev = interface_to_usbdev(intf);
  rtl8150_t *dev;
  struct net_device *netdev;

  netdev = alloc_etherdev(sizeof(rtl8150_t));
  if (!netdev) { ... }
  ...
  netdev->hard_start_xmit = rtl8150_start_xmit;
  ...
  if (register_netdev(netdev) != 0) { ... }
  ...
}
...
static int __init usb_rtl8150_init(void) {
  info(DRIVER_DESC " " DRIVER_VERSION);
  return usb_register(&rtl8150_driver);
}

static void __exit usb_rtl8150_exit(void) {
  usb_deregister(&rtl8150_driver);
}

module_init(usb_rtl8150_init);
module_exit(usb_rtl8150_exit);
```

Figure 2: Extracts of the `rtl8150.c` in Linux 2.6.13

USB interface	
Exported library functions	`usb_fill_bulk_urb`, `usb_sndbulkpipe`, `usb_submit_urb`, `interface_to_usbdev` `usb_register`, `usb_deregister`
Imported device-specific callback functions	`rtl8150_probe`
Data structures	`rtl8150_driver` of type `struct usb_driver`, `dev->udev` of type `struct usb_device`
Protocol	`usb_fill_bulk_urb` if used precedes `usb_submit_urb`

Network device interface	
Exported library functions	`netif_stop_queue`, `netif_start_queue`, `alloc_etherdev` `register_netdev`
Imported device-specific callback functions	`rtl8150_start_xmit`
Data structures	`netdev` of type `struct net_device`
Protocol	`netif_start_queue` follows `netif_stop_queue` on error.

Figure 3: Extracts of the USB and network device interfaces

Exported library functions	– add/drop arguments – change function name – change return type
Imported device-specific callback functions	– add/drop required parameters – change required return type
Data structures	– split or merge structures – introduce layers of indirection – convert a structure field reference to a getter/setter function call
Protocols	– add or drop required calls – change sequencing – change locking requirements – add required error checking

Figure 4: Changes that can occur in a driver support library interface

a taxonomy of these effects, obtained by considering systematically the information included in each case and the changes that can occur in this information. In a study of Linux 2.5, described below, we have observed all of these changes in driver support library interfaces. We thus expect this taxonomy to apply to other Linux versions as well.

3.2.2 Collateral evolutions

A collateral evolution represents a side effect on the context in which an interface element is used. While interface elements themselves are intrinsically restricted and fixed, their usage context consists of arbitrary code, and can vary widely from one device-specific file to another. It is thus not possible to develop an exhaustive taxonomy of all possible collateral evolutions. Instead, we have made a careful study of 72 collateral evolutions in the various subversions of Linux 2.5, based on several hundred potential collateral evolutions that we have identified. The studied collateral evolutions affect over one thousand files. We furthermore believe that these examples are representative of the range and scope of collateral evolution in Linux because Linux 2.5 is an unstable version, in which many evolutions and collateral evolutions occur, and because they cover all of the identified interface changes.

In the rest of this section, we provide an overview of the kinds of collateral evolutions that we have identified in our study. Representative examples among the 72 collateral evolutions that we have studied are shown in Figure 5. The line numbers in the text below refer to the lines in this figure.

Library functions. We first consider the collateral evolutions required in response to changes in the signature of a library function, including its arguments, name, and return type.

Adding an argument to a library function or changing the type of an existing argument requires that the device-specific code construct a new value. In simple cases, the new value is a constant (line 1), a variable already bound in the current function (line 3), or a fixed transformation of the current argument, such as adding a structure field reference (line 4). In many cases, however, the addition of a new argument or change in an argument type requires substantial code rewriting, possibly depending on control and data flow information or external knowledge. For example, when the type of the argument changes to a new structure type it is necessary to create and initialize a new structure value as well as modifying the function call (line 6).

Dropping an argument is generally straightforward, since the argument has no impact on the context (line 8). It may, however, be desirable to remove any code involved in computing the argument's value.

Changing the name of a library function is straightforward if the change is performed uniformly. It is more difficult, however, if the choice of the new function depends on the context. This is best illustrated by the case of `bus_to_virt` and related functions (line 14). In Linux 2.5.4, these functions were replaced by a functions having names beginning with "`isa_`," apparently to force programmers to consider whether the functionality of `bus_to_virt` was appropriate for the given context [18]. The treatment of calls to this the function was, however, not performed uniformly across the driver source code, leading to a flood of complaints in the Linux mailing lists [14, 24, 25]. As a result, in Linux 2.5.8, wrapper functions were introduced defining `bus_to_virt` etc. to their "`isa_`" counterparts, thus nullifying the benefit of the evolution.

Function names also change when a collection of functions defined by the device-specific code is unified into a single library function (line 15). The collateral evolution requires both recognizing and eliminating the device-specific definitions, which may exhibit inessential syntactic variations, as well as updating the call sites.

Changes in function return types often derive from changes in error handling. When the return type of a library function changes from `void` to a type such as `int` that indicates an error condition, device-specific code has to be modified to introduce appropriate error handling (line 16). Often the device-specific code responds to the error by itself returning prematurely with an error code. In this case, the collateral evolution must take care to release any locally allocated resources (line 16). In other cases, it is the semantics, not the type of the return value that changes. Some library functions use 0 or 1 to indicate an error and 1 or 0 to indicate success, while others use more informative error codes, such as `-EIO` and `-ENODEV`. When a library function adopts the more informative error reporting strategy, collateral evolution may be required at the call sites to invert the sense of 0 and to introduce specialized error handling depending on the kind of error that is indicated by the return value. Similar issues occur when a library-specific return type is introduced.

Device-specific callback functions. We next consider the collateral evolutions required when changes in the interface exported by a driver support library entail modifications to the signature of an imported device-specific callback function.

In some cases, a parameter is added to a device-specific callback function to accommodate some new instance of the function that needs some additional information (line 17). In such cases, no further collateral evolution is required. On the other hand, a new parameter may supersede information previously computed by the function (line 18). In that case, careful analysis is needed to eliminate only the computation of this value. Finally, a new parameter can be added because the function will directly or indirectly call a library function that takes this information as a new argument (line 19). In this case, the new parameter has to be transmitted to all intervening function calls.

Dropping a parameter or changing its type means that the original value must be reconstructed if it is needed by the function (line 20). This can require pervasive changes in the function definition.

Collateral evolutions related to function return types again typically concern error return values. When a device-specific callback function that returns 0 or 1 is required to use more informative values, the collateral evolution entails identifying the existing values that indicate error or success, and choosing an appropriate value in each error case (line 22).

Data structures. Evolutions in data structures typically involve adding, removing, or reorganizing fields, which trigger collateral evolutions similar to the changes in arguments and parameters described above. The collateral evolution may, however, be complicated by the use of local variables to name substructures, requiring a careful dataflow analysis to identify the affected code (line 24). In some cases, the reorganization of a structure is accompanied by the introduction of getter and setter functions, abstracting over the new access path (line 25). In the case of read accesses to the affected field, the collateral evolution may introduce a local variable to store the result of calling the getter function, rather than replacing every read access by a function call.

Protocols. A driver support library protocol specifies the order in which various operations related to the functions and data structures exported by the library should be carried out. Such a protocol may for example specify a required sequence of function calls or a context in which error checking is needed. The instantiation of a protocol in device-specific code is often determined by the device-specific code structure. For example, when a protocol requires error checking, the actual code used to clean up in the case of an error may depend on the set of resources allocated by the device-specific code.

Protocol changes involve removing the instantiation of the old protocol from the device-specific code and inserting the

Library function definitions

Add argument/change argument type

	Version	Function	New value
1	2.5.53	`pnp_activate_dev`	`NULL`
2	2.5.22	`end_request`	`CURRENT`
3	2.5.67	`LOCK_TEST_WITH_RETURN`	parameter of the enclosing function
4	2.5.16	`usb_stor_clear_halt`	field of existing argument
5	2.5.54	`dev_get(set)_drvdata`	subexpression of existing argument
6	2.5.59	`agp_(un)register_driver`	newly created and initialized global structure
7	2.5.4	`usb_submit_urb`	context-dependent constant

Drop argument

	Version	Function	Context effect
8	2.5.63	`pnp_activate_dev`	none
9	2.5.70	`acpi_hw_low_level_read(write)`	none

Change function name

	Version	Renamed function	Function selection strategy
10	2.5.69	`mem_map_(un)reserve`	uniform
11	2.5.69	`cs4x_mem_map_(un)reserve`	uniform
12	2.5.33,45	`FILL_CONTROL_URB`, etc.	uniform
13	2.5.16	`usb_clear_halt`	uniform within a given directory
14	2.5.4	`bus_to_virt`, `virt_to_bus`, and `page_to_bus`	context-dependent, does not follow directory structure
15	2.5.50	`sched_event`	function moved from device-specific code to driver support library

Change return type

	Version	Function	Effect
16	2.5.20	`acpi_hw_register_read`, etc.	add/adjust error checking using the `acpi_status` type

Driver function definitions

Add parameter

	Version	Function	Impact
17	2.5.51	USB callback functions	none
18	2.5.71	SCSI `proc_info` functions	existing computation of the same value deleted, can involve loop slicing
19	2.5.3	Video device `mmap` function	new parameter passed to library function `remap_page_range`

Drop parameter/change parameter type

	Version	Function	Effect
20	2.5.71	SCSI `proc_info` functions	reconstruct value from new argument (see above)
21	2.5.8	Video driver ioctl functions	references to a structure pointer type become references to a local structure

Change return type

	Version	Function	Effect
22	2.5.71	`attach`/`probe`.	function moves from the `attach` field of a `Scsi_Device_Template` structure to the `probe` field of a `scsi_driver` structure; `return 1` becomes `return -ENODEV`

Data structures

	Version	Structure type	Evolution
23	2.5.45	`IsdnCardState`, `BCState`	change field name
24	2.5.27	`mddev_t`	inline substructure
25	2.5.67	`i2c_client`	substructure introduced, getter/setter functions introduced

Protocols

26	2.5.52	`acpi_device_dir`	insert assignment after call to `remove_proc_entry` in some contexts
27	2.5.50	`irq_func` callback function	drop parameter test, insert locking around function body
28	2.5.36	`usb_stor_clear_halt`	introduce error checking
29	2.5.4	network ioctl functions	new ethtool cases, depending on whether the code defines a debug variable
30	2.5.45	USB callback functions	interrupt urbs must now be explicitly resubmitted
31	2.5.33	conversion from i2c-old to i2c	many changes, including new functions to define, new structures to define and initialize, and new library functions to use

Figure 5: Some collateral evolutions in Linux 2.5

appropriate instantiation of the new one. In some cases, the code to add or remove is fixed, and appears in a fixed context (line 26). In other cases, the instantiation of a protocol is context sensitive. For example, when locking is added, it must often be placed at every function return point; the positioning of function returns varies from function to function (line 27). When error checking is added, it may only be needed in code that does not already lead to an error return value (line 28). The set of new cases to be handled by an ioctl function may depend on the other features provided by the device-specific code (line 29). Finally, major reorganizations in an interface often involve a combination of these collateral evolutions (line 31).

3.2.3 Assessment

In some cases, the C compiler can help with collateral evolution, for example by detecting when a function is passed the wrong number of arguments. Nevertheless, the compiler only helps in cases where the need for the collateral evolution manifests itself as a type error; when 0/1 return values are converted to integer-typed error codes or the required sequencing of a set of function calls changes, the compiler provides no assistance.

The simplest collateral evolutions, such as renaming a function or adding a constant first argument, can be easily implemented using editor macros or shell scripts. There is indeed evidence that collateral evolution is done this way, as sometimes comments that coincidentally contain the name of an affected function are changed as well. Nevertheless, even in simple cases this approach is highly error prone, as it may modify code fragments that contain the same text but are unrelated to the collateral evolution. More complex collateral evolutions require parsing complex terms, analyzing the context, transforming multiple lines of code, and translating variable names and code patterns to those used in the affected file. The minor variations, omissions, and errors that we have observed in the updated code suggest that collateral evolution is often done by hand, which is time-consuming and error-prone.

4. CASE STUDIES

In this section, we consider three collateral evolutions in detail. These examples illustrate some of the more complex cases in three common categories: an argument added to a library function, a change in the required parameter type of a callback function, and a change in a protocol. We consider not only the modifications that the maintainer of device-specific code must make for each collateral evolution, but also the history of the collateral evolution, including a study of the bugs that were introduced.

4.1 Addition of an argument

The USB library function `usb_submit_urb` implements the passing of a message, implemented as USB Request Block (urb). This function uses the kernel memory-allocation function, `kmalloc`, which must be passed a flag indicating the circumstances in which blocking is allowed. Up through Linux 2.5.3, the flag was chosen in the implementation of `usb_submit_urb` as follows:

```
    in_interrupt () ? GFP_ATOMIC : GFP_KERNEL
```

Comments in the file `usb/hcd.c`, however, indicate that this solution is unsatisfactory:

```
// FIXME paging/swapping requests over USB should not
// use GFP_KERNEL and might even need to use GFP_NOIO ...
// that flag actually needs to be passed from the higher level.
```

Starting in Linux 2.5.4, `usb_submit_urb` takes one of the following as an extra argument: `GFP_KERNEL` (no constraints), `GFP_ATOMIC` (blocking not allowed), or `GFP_NOIO` (blocking allowed but not I/O). The programmer of device-specific code selects one of these constants according to the context of the call to `usb_submit_urb`.

Choosing the extra argument of `usb_submit_urb` requires a careful analysis of the surrounding code as well as an understanding of how this code is used by driver support libraries. Comments describing the relevant conditions are provided with the definition of `usb_submit_urb` starting in Linux 2.5.4. These comments state that `GFP_ATOMIC` is required in a completion handler, in code related to handling an interrupt, when a lock is held (including the lock taken when turning off interrupts), when the state of the running process indicates that the process may block, in certain kinds of network driver functions, and in SCSI driver queuecommand functions. Many of these situations, however, are not explicitly indicated by the code surrounding

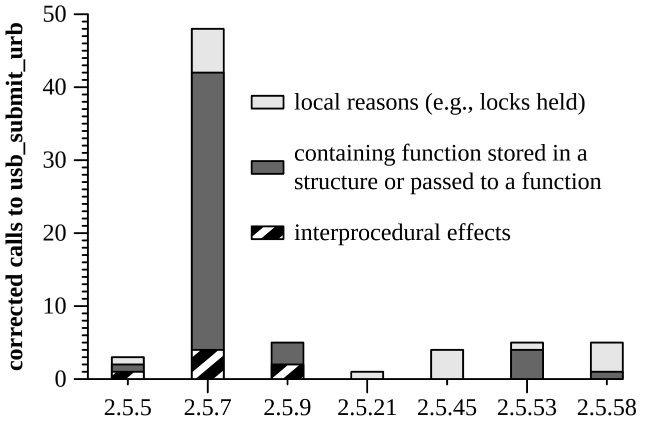


Figure 6: Linux 2.5 versions in which `GFP_KERNEL` is corrected to `GFP_ATOMIC` in a call to `usb_submit_urb`

the call to `usb_submit_urb`. Instead, they require an understanding of the contexts in which the function containing the call to `usb_submit_urb` may be applied. In practice, this function can be passed to a driver support library via a data structure or function call and used in arbitrary ways, or can be invoked directly or indirectly by a local function that has one of the above properties.

The difficulty in understanding the conditions in which `GFP_ATOMIC` is required and identifying these conditions in driver code is illustrated by the many calls to `usb_submit_-urb` that were initially transformed incorrectly. Figure 6 lists the versions in Linux 2.5 in which corrections in the use of `usb_submit_urb` occur and the reason for each correction. In each case, the error was introduced in Linux 2.5.4 or when the driver entered the kernel source tree, whichever came later. A major source of errors is the case where the function containing the call to `usb_submit_urb` is stored in a structure or passed to a function, as these cases require extra knowledge about how the structure is used or how the function uses its arguments. Indeed, in the `serial` subdirectory, all of the calls requiring `GFP_ATOMIC` fit this pattern and all were initially modified incorrectly (and corrected in Linux 2.5.7). Surprisingly, in 17 out of the 71 errors, the reason for using `GFP_ATOMIC` is locally apparent, reflecting either carelessness or insufficient understanding of the conditions in which `GFP_ATOMIC` is required. Indeed, in Linux 2.6.13, in the file `usb/class/audio.c`, `GFP_KERNEL` is still used in one function where a lock is held.

The difficulty of choosing the value of the extra argument for `usb_submit_urb` is illustrated by the case of the function `rtl8150_start_xmit` shown in Figure 2. The rtl8150 driver was introduced into the Linux source tree in Linux 2.5.8, at which point the call to `usb_submit_urb` in this function was given the argument `GFP_KERNEL`. This choice of argument is, however, incorrect, as `rtl8150_start_xmit` is one of the kinds of network driver functions that requires `GFP_ATOMIC`. The code was corrected in Linux 2.5.9.

4.2 Change in the type of a parameter

A Linux ioctl function allows user-level interaction with a device driver. Copying arguments to and from user space is a tedious but essential part of the implementation of such a function. In Linux 2.5.7, the media support library introduced a wrapper function to encapsulate this argument copying. This function was refined in Linux 2.5.8 and named `video_usercopy`. As of Linux 2.6.13, `video_usercopy` was used in 31 `media` files and 6 `usb` files.

Introducing the use of `video_usercopy` affects the type of one of the parameters of the ioctl code. In the original version, this parameter is a pointer to user space, and each ioctl command must use the functions `copy_from_user` and `copy_to_user` to access or update its value. In these cases, the data is typically copied once, and otherwise accessed via a local data structure whose type is specific to the ioctl command. After the introduction of `video_usercopy`, the parameter of the ioctl function becomes a generic pointer to kernel space, which the ioctl code can read from or write to directly. The collateral evolution thus entails modifying the treatment of each ioctl command to remove the copy functions, casting the generic pointer parameter to a pointer of the structure type used by the command, and replacing the references to the local structure by pointer dereferences. The latter transformation can be quite invasive. For example, in the ioctl function of `media/radio/radio-typhoon.c`, 61% of the lines of code changes between Linux 2.5.6 and 2.5.8.

The behavior of `video_usercopy` is not specific to media drivers, and thus there has been interest in making the function more generally available [13]. Some evidence of the difficulties this may cause are provided by the case of `i2c/other/tea575x-tuner.c` in which `video_usercopy` was introduced in Linux 2.6.3. In this file, the calls to `copy_from_user` and `copy_to_user` were not removed. The bug was never fixed. Instead, the use of `video_usercopy` was removed from this file in Linux 2.6.8.

4.3 Change in a function protocol

The function `check_region` is used in the initialization of device drivers, in determining whether a given device is installed. In early versions of Linux, the kernel initializes device drivers sequentially [20]. In this case, a driver determines whether its device is attached to a given port using the following protocol: (i) call `check_region` to find out whether the memory region associated with the port is already allocated to another driver, (ii) if not, then perform some driver-specific tests to identify the device attached to the port, and (iii) if the desired device is found, then call `request_region` to reserve the memory region for the current driver. In more recent versions of Linux, the kernel initializes device drivers concurrently [4]. In this case, between the call to `check_region` and the call to `request_region` some other driver may claim the same memory region and initialize the device. To solve this problem, starting with Linux 2.4.2, device-specific code began to be rewritten to replace the call to `check_region` in step (i) with a call to `request_region`, to actually reserve the memory region. Given this change, if in step (ii) the expected device is not found, then `release_region` must be used to release the memory region.

Eliminating a call to `check_region` requires replacing it by the associated call to `request_region` and inserting calls to `release_region` along error paths. In the first step, it is necessary to find the call to `request_region` that is associated with the given call to `check_region`. In practice, these are often not in the same function, requiring an interprocedural analysis. In the second step, it is necessary to identify code points at which it is known that the expected device has not been found and thus `release_region` is required. This condition is often indicated by the returning of an error value, but may also be indicated by going around a loop that checks successive ports until finding one with the desired device. At such code points, it may be the case that only a subset of the incoming paths contain a call to `check_region`. In these cases, the call to `release_region` must be placed under a conditional.

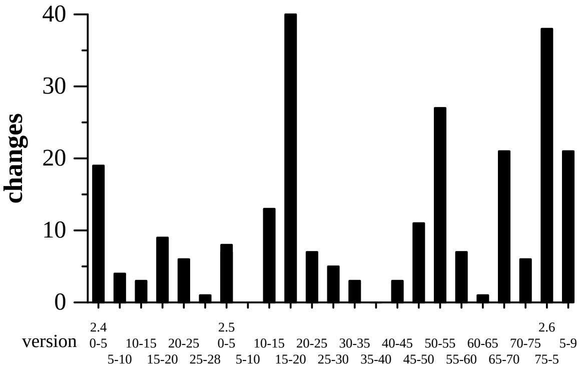


Figure 7: `check_region` elimination in Linux 2.4-2.6

The elimination of `check_region` has been a recurring topic in Linux mailing lists, including the following exchange:

```
Subject: Re: Linux-2.6.13 : __check_region is deprecated
Newsgroups: gmane.linux.kernel
Date: 2005-08-29 23:21:30 GMT

On Tue, 30 Aug 2005, S.W. wrote:

> Hi,
>
> By compiling my kernel, I can see that the
> __check_region function (in kernel/resource.c)
> is deprecated.
>...
> Is there a function to replace this deprecated
> function ?

Just restructure the code to use request_region().
```

The response in this case does not convey any of the complexity of the collateral evolution process, and highlights the need for a formal language for specifying collateral evolutions. Indeed, both steps in eliminating `check_region` are difficult and time-consuming. This difficulty has lead to the slow pace of the evolution, as shown in Figure 7. Although beginning in Linux 2.4.2, released in February 2001, the evolution is still not complete as of Linux 2.6.13.3, released in October 2005.

5. QUANTITATIVE ASSESSMENT

In this section, we present a quantitative assessment of factors related to collateral evolution in Linux. We begin by assessing the complexity of the interdependencies of driver code based on the relationship between interfaces and device-specific files. We then consider the effect of evolution in the kernel and driver support libraries on interfaces. Finally, we quantify the required collateral evolutions.

5.1 Code base

Our assessment is based on Linux code from version 2.2.0, released in January 1999, to version 2.6.13, released in August 2005. This sample contains both stable versions (2.2, 2.4, and 2.6) and unstable ones (2.3 and 2.5). All files were obtained from `http://www.kernel.org`. Since 2.6.11, Linux has been released in a series of subminor versions (2.6.11.1, 2.6.11.2, etc.), which we do not consider separately. We focus on the `drivers` and `sound` directories. The `sound` directory is included because it was part of the `drivers` directory until Linux 2.5.5.

Our study distinguishes between driver support libraries and device-specific code. As there is no convention in Linux for identifying driver support libraries, we use the following heuristic. We consider that there is at most one driver support library per directory. A file is in the driver support library if it exports functions to multiple files or if it exports functions to another file in the driver support library. Other files are considered to be device-specific. We ignore both libraries that only export functions to other libraries and files that do not import any library functions, as these are not affected by the interface between driver support libraries and device-specific code that is the source of collateral evolution.

We have applied the algorithm for distinguishing between driver support libraries and device-specific code to the files in the `drivers`, `sound`, and `net` directories. The `net` directory in included as a source of driver support libraries because network device drivers typically use its code. The kernel is considered to be another driver support library, defining all of the functions for which definitions are not found in the `drivers`, `sound`, or `net` code. In Linux 2.4.0 there are 66 driver support libraries and 874 device-specific files, while in the most recent version of Linux, 2.6.13, these numbers have more than doubled to 164 and 1926, respectively.

5.2 Methodology

A key point in our quantitative assessment is to understand the changes in device-specific code from one version of Linux to the next. For this purpose, we have developed a tool, the *patch analyzer*, that detects commonalities and differences in two versions of C code. The patch analyzer starts from a patch file, in which it analyzes each of the identified difference regions. We illustrate the analysis using the difference region shown in Figure 8 that is derived from a patch file comparing the definition of the function `rtl8160_start_xmit` in Linux 2.4 (Linux 2.4.19) to the most recent version in Linux 2.6.13. The complete Linux 2.6.13 definition of `rtl8160_start_xmit` was shown in Figure 2.

The first step of the patch analyzer is to align the common parts of the two fragments and to identify the maximally different regions. In our example, the maximally different regions are as follows:

```
memcpy(dev->tx_buff, skb->data, skb->len)
```
replaced by
```
dev->tx_skb = skb
```

```
FILL_BULK_URB(ARG0,ARG2,ARG4,dev->tx_buff,
   RTL8150_MAX_MTU,ARG8,ARG10)
```
replaced by
```
usb_fill_bulk_urb(ARG0,ARG2,ARG4,skb->data,
   count,ARG8,ARG10)
```

```
dev->tx_urb->transfer_buffer_length = count
```
dropped

```
if((res = usb_submit_urb(ARG0)))
```
replaced by

```
  count = (skb->len < 60) ? 60 : skb->len;
  count = (count & 0x3f) ? count : count + 1;
- memcpy(dev->tx_buff, skb->data, skb->len);
- FILL_BULK_URB(dev->tx_urb, dev->udev,
-                    usb_sndbulkpipe(dev->udev,2),
-                    dev->tx_buff, RTL8150_MAX_MTU,
-                    write_bulk_callback, dev);
- dev->tx_urb->transfer_buffer_length = count;
- if ((res=usb_submit_urb(dev->tx_urb))){
+ dev->tx_skb = skb;
+ usb_fill_bulk_urb(dev->tx_urb, dev->udev,
+                usb_sndbulkpipe(dev->udev, 2),
+                skb->data, count, write_bulk_callback, dev);
+ if ((res=usb_submit_urb(dev->tx_urb, GFP_ATOMIC))){
        warn("failed tx˙urb %d“n", res);
        dev->stats.tx_errors++;
        netif_start_queue(netdev);
```

Figure 8: Extracts of a patch derived from the rtl8150 driver

```
if((res = usb_submit_urb(ARG0, GFP_ATOMIC)))
```

In this result, the various statements of the dropped and added regions are matched up line by line except for the assignment of the `transfer_buffer_length` field. This assignment is considered by itself because the next element in both fragments is a conditional test, and these are aligned instead. When a function call is matched with another function call, the common arguments are replaced by a term ARG*n*, where *n* is determined by the argument position. As the common arguments are not possible evolutions, this normalization improves the chance that this pair of calls will match with other calls to the same functions. As a further abstraction, for function names, we drop substrings that correspond to the name of the analyzed file, *e.g.*, `rtl8150` in `rtl8150_probe` (see Figure 2), as such substrings are often used to make function names unique across the Linux source tree. Finally, we observe that the calls to `usb_submit_urb` have a non-trivial common context, including an assignment and a conditional test. This occurs because conditionals, assignments and function calls that have a top-level difference among their subterms are considered to be different as well.

The next step is to distinguish between differences that are specific to a single device-specific file and differences that are recurrent across multiple device-specific files found in one or more Linux versions, and thus represent a collateral evolution. For this, we use a threshold: a difference is considered to be part of a collateral evolution if it occurs at least 5 times and these occurrences are distributed across at least 3 files. It may, however, be the case that a complete maximally different region does not occur often enough to satisfy the threshold, but there is some subterm that represents the collateral evolution. An example is the case of the conditional test identified in the case of `rtl8160_start_xmit`:

```
if((res = usb_submit_urb(ARG0)))
replaced by
if((res = usb_submit_urb(ARG0, GFP_ATOMIC)))
```

Here, the maximal difference contains the conditional and the assignment, but, as described in Section 4.1, the collateral evolution is only the addition of the second argument to the call to `usb_submit_urb`.

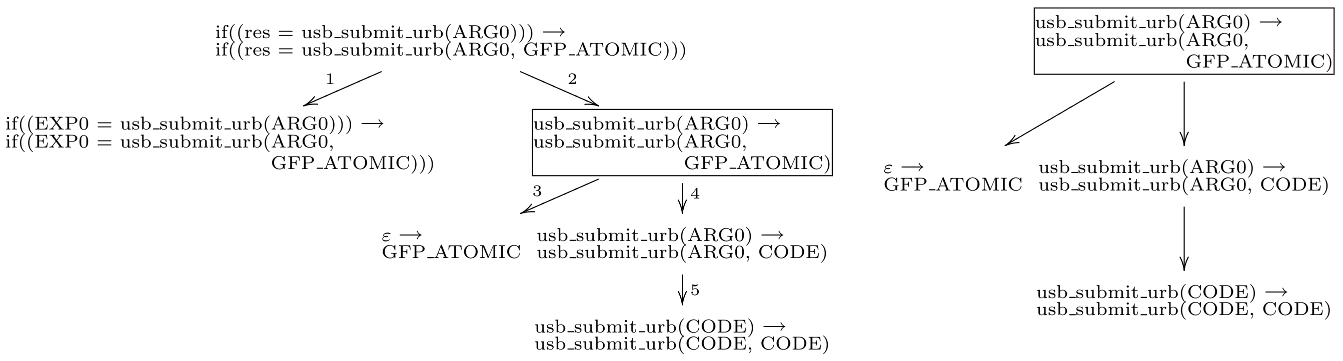

if((res = usb_submit_urb(ARG0))) →
if((res = usb_submit_urb(ARG0, GFP_ATOMIC)))
1
2
if((EXP0 = usb_submit_urb(ARG0))) →
if((EXP0 = usb_submit_urb(ARG0, GFP_ATOMIC)))
usb_submit_urb(ARG0) →
usb_submit_urb(ARG0, GFP_ATOMIC)
3
4
ε → GFP_ATOMIC
usb_submit_urb(ARG0) →
usb_submit_urb(ARG0, CODE)
5
usb_submit_urb(CODE) →
usb_submit_urb(CODE, CODE)
usb_submit_urb(ARG0) →
usb_submit_urb(ARG0, GFP_ATOMIC)
ε → GFP_ATOMIC
usb_submit_urb(ARG0) →
usb_submit_urb(ARG0, CODE)
usb_submit_urb(CODE) →
usb_submit_urb(CODE, CODE)

Figure 9: Trees used in matching modifications across multiple files. The tree on the left is from `drivers/usb/net/rtl8150.c` and the one on the right is from `drivers/usb/misc/auerswald.c`. The modifications that would be detected are boxed.

To find collateral evolutions within subterms, we use a matching algorithm that considers for each difference a tree consisting of the difference, its subterms, and abstractions of the subterms. The tree for the above difference is shown on the left side of Figure 9. The root of the tree is the difference itself. The various subterm and abstraction strategies illustrated by its descendents are as follows:

- Replace identifiers by `EXP`n, as illustrated by arrow 1. This abstraction eliminates the dependence on local variable names, which may vary between usage contexts.
- Extract a maximal difference from the subterms, as illustrated by arrows 2 and 3. In the difference at the target of arrow 3, $\varepsilon \rightarrow$ GFP_ATOMIC on the left side of the transformation indicates that GFP_ATOMIC is added, rather than replacing existing code.
- Replace unabstracted function arguments by CODE, as illustrated by arrow 4. In this example, this abstraction would allow detecting the case where a second argument is added to `usb_submit_urb`, but there is no discernible pattern in the choice of the new value.
- Replace abstracted function arguments (`EXP`n or `ARG`n) by CODE, as illustrated by arrow 5. In this example, this abstraction would allow detecting the case where there are changes in the existing argument as well as the addition of a new one.

Other abstractions are possible; for example, we could convert various permutations of the unabstracted function arguments to CODE rather than converting them all at once. Furthermore, we do not consider over-abstracted differences, such as $\varepsilon \rightarrow$ CODE, which would match any added device specific code, even code that has nothing to do with the difference represented here. The choices we have made limit the computational complexity of the analysis, and seem to work well in practice, based on our crosschecking with the manually studied collateral evolutions described in Section 3.2.2.

Given the collection of trees representing all of the modifications, the patch analyzer compares all of the modifications in all of the trees from the largest modification to the smallest, discarding a subtree as soon as the root matches enough other modifications to satisfy the threshold. As an example, if we assume that the only available modifications are those shown on the left and right sides of Figure 9, then the matching process would first consider the largest modification `if((res = usb_submit_urb(ARG0)))` $\rightarrow \ldots$, then the abstracted modification `if((EXP0 = usb_submit_urb(ARG0)))` $\rightarrow \ldots$, which is the next-largest one, and finally the subterm modification `usb_submit_urb(ARG0)` $\rightarrow$ `usb_submit_urb(ARG0, GFP_ATOMIC)`, which it finds to occur in both trees. Assuming that there are enough other occurrences of this difference to satisfy the threshold, this difference would be reported as the result.

We use this analysis not only to detect collateral evolution sites, as quantitatively analyzed in Section 5.5, but also to detect evolutions in interfaces themselves, as quantitatively analyzed in Section 5.4. Specifically, a collateral evolution that directly affects an interface element is also an indicator of an evolution in the interface.

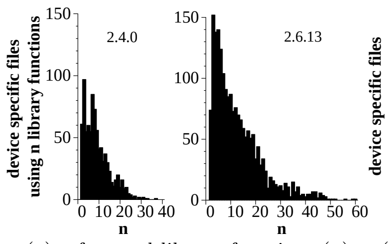
Note that our patch analyzer detects only modifications that occur at the level of individual C statements or expressions. We refer to such modifications as *micro collateral evolutions*. The collateral evolutions studied in our manual analysis, on the other hand, often amount to a collection of such modifications, and we thus refer to them as *macro collateral evolutions*. For example, the introduction of `video_usercopy`, described in Section 4.2, involves changing not only a function's parameter type, but also local variables and function calls scattered throughout its body. The measurements in the rest of this section, however, are concerned only with number of lines affected by collateral evolutions in device-specific code, rather than on the number of (macro) collateral evolutions themselves. Thus, the precision of the patch analyzer is sufficient for our purposes.

5.3 Interfaces

Because collateral evolutions are derived from interface changes, the size and distribution of interfaces is a measure of the potential difficulty of collateral evolution in device-specific code. We consider the relationship between interfaces and device-specific code from the perspective of the maintainer of a single device-specific file and from the perspective of the library developer.

Interface complexity from the perspective of the maintainer of device-specific code. The number of library functions used by device-specific code is a measure of the code's complexity, as the maintainer must understand each of these functions, including its arguments and associated protocols. Figure 10a shows the number of library functions used by each device-specific source file. This figure shows clearly that not only has the size of the driver code doubled in the last five years since Linux 2.4.0, but also the complexity. Indeed, substantially more device-specific files refer to up to 20 library functions in Linux 2.6.13 than in Linux 2.4.0. Furthermore, in Linux 2.4.0 the largest number of library function references per file is 36 while in Linux 2.6.13 this number has jumped to 59.

We may further refine the assessment of the complexity of device-specific code by taking into account the library structure. Each driver support library represents a unit of understanding, and thus code that relies on multiple driver support libraries requires more expertise to maintain than code that relies on only one. Figure 10b shows the number of libraries on which each file depends. In Linux 2.4.0 only 169 files rely on three or more libraries, while in Linux 2.6.13 this number has increased to 501.


(a) referenced library functions (n) per device-specific file

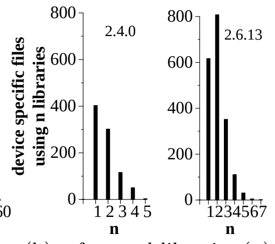
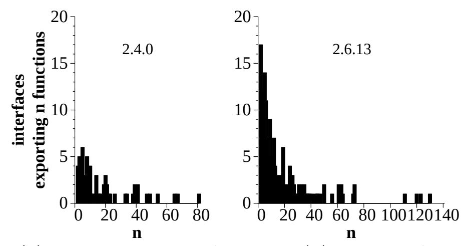

(b) referenced libraries (n) per device-specific file

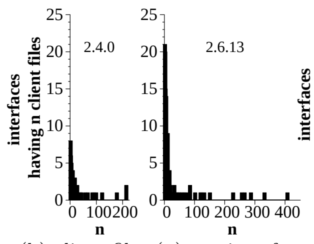
Figure 10: (a) Library function references and (b) library references in Linux 2.4.0 and Linux 2.6.13

Interface complexity from the perspective of the developer of a driver-support library. We measure the complexity of an interface in terms of the number of functions it exports, as shown in Figure 11a. While the number of library functions exported by the typical interface is under 20 in both Linux 2.4.0 and Linux 2.6.13, the maximum number of exported library functions increases significantly, from around 80 in Linux 2.4.0 to around 130 in Linux 2.6.13.

When a change occurs in the interface of a driver support library, collateral evolution is needed in all dependent device-specific code. The difficulty of performing this collateral evolution depends not only on the number of files involved, but also on the distribution of these files across different directories, as files in other directories may not be known to the library developer and may exhibit unique code patterns. For example, Figure 12 shows the distribution of the use of the USB driver support library across the various driver directories. While most of the uses are in the `usb` directory, there are a few uses in each of 9 other directories.

More generally, Figures 11b and 11c show the number of device-specific files depending on each interface and the number of directories containing at least one device-specific file with such a dependency. Again, the maximum number has doubled between Linux 2.4.0 and Linux 2.6.13. In Linux 2.6.13, the most widely used library is PCI, which is the basic bus used on PCs. The network device and ethernet support libraries are the next most widely used. The USB support library is also among the most widely used, being used by 131 files.


(a) exported library functions (n) per interface


(b) client files (n) per interface

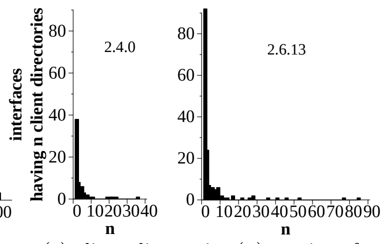

(c) client directories (n) per interface

Figure 11: (a) Interface size and (b,c) interface usage in Linux 2.4.0 and Linux 2.6.13

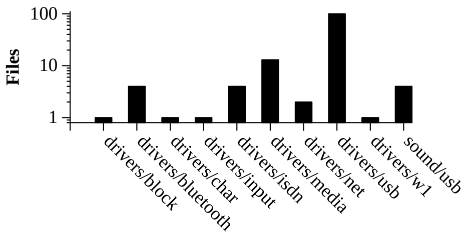


Figure 12: References to the USB driver support library

5.4 Quantitative assessment of evolution

Figure 13 shows the number of evolutions that have occurred in library functions, device-specific callback functions, data structures, and protocols in the versions of Linux between 2.2 and 2.6. In the current state of our patch analyzer, protocols are detected only as the addition or deletion of single function calls. The most significant number of evolutions occurs in library functions, with an increase across the versions of Linux that roughly mirrors the increase in code size.

It is interesting to compare the number of evolutions in the unstable versions 2.3 and 2.5 and their stable derivatives 2.4 and 2.6. The stable Linux 2.4 had slightly more evolutions than the unstable version 2.3, but was the main version of Linux for almost three years, while Linux 2.3 was only under development for one year. In the case of Linux 2.5 and Linux 2.6, the so-called stable Linux 2.6 has had almost as many evolutions as the unstable Linux 2.5. The current age of

Linux 2.6 is about the same as the time in which Linux 2.5 was under development, but we can expect Linux 2.6 to be the main version for some time longer, and thus to accumulate even more evolutions.

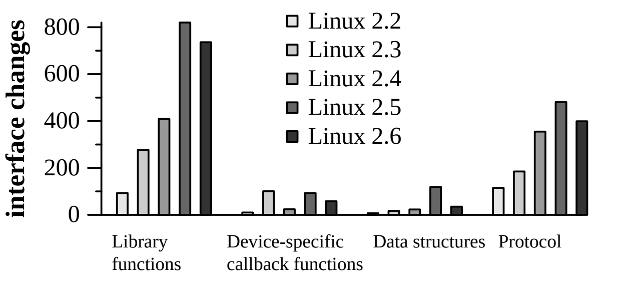


Figure 13: Number of evolutions in interface elements in Linux 2.2 to Linux 2.6

5.5 Quantitative assessment of collateral evolution

Figure 14 assesses the number of lines modified due to collateral evolutions from the first patch file for Linux 2.2 to the patch file for Linux 2.6.13. We observe that while the number of lines affected by collateral evolutions varies from one version to the next, there is a general increasing trend, with a significant increase in Linux 2.6. In terms of the percentage of lines modified due to collateral evolution as compared to the the number of lines modified overall in device-specific code, we see that the biggest spikes, of up to 35%, occur in the unstable versions. We conjecture that OS developers postpone evolutions that may induce many collateral evolutions until these versions, due to the amount of work that they entail.

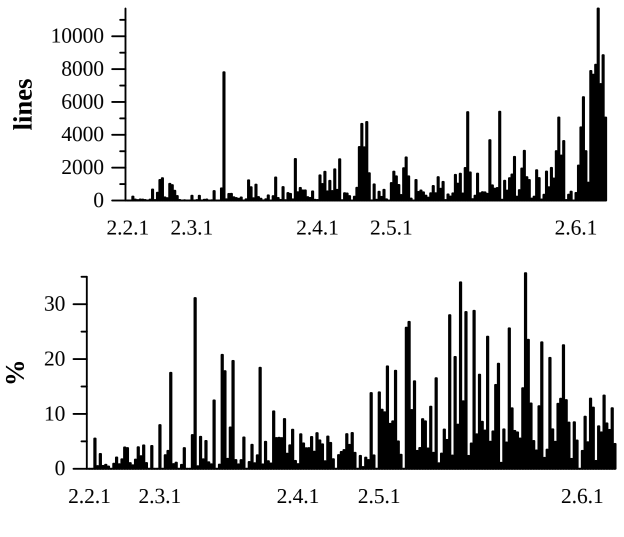


Figure 14: Patch file lines and percentage of patch file lines derived from device-specific code and containing collateral evolutions in Linux 2.2 to Linux 2.6

Figure 15 measures the magnitude of individual collateral evolutions in terms of the number of sites affected and terms of the number of files affected. On average, a collateral evolution is required at around 14 sites and in a total of 10 files. Nevertheless, many collateral evolutions are much more pervasive, with one change in library function affecting around 1000 sites in Linux 2.6. Overall, collateral evolutions in library functions and protocols affect both the most sites and files. Collateral evolutions in library functions vary from simple textual replacement to cases that involve careful analysis of the source code. Protocol changes that involve adding new functions, on the other hand, are often difficult, as they require situating new code within a context whose precise structure can vary.

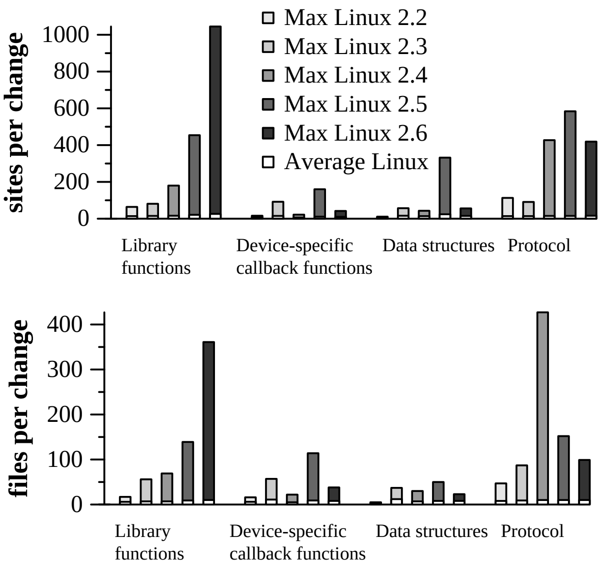


Figure 15: Maximum number of sites and files affected by a single collateral evolution

6. CONCLUSION

In this paper, we have shown that the evolution of driver support libraries is a critical issue for the maintenance of device-specific code. Nevertheless, this issue has received little attention from researchers. Based on a manual study of device-specific code, we have identified a taxonomy of effects that an evolution in a driver support library can have on the interface with device-specific code and have characterized the collateral evolutions in device-specific code that these effects trigger. We have also developed automatic tools that assess the extent of the collateral evolution problem, based on analysis of Linux versions released over the last six years. As part of this analysis, we have identified the growing complexity of Linux driver support library interfaces, suggesting that collateral evolution will, if anything, be a greater problem in the future.

At present, our patch analyzer only detects micro collateral evolutions, while the collateral evolutions identified manually often involve a collection of such changes. We are planning to investigate heuristics for combining multiple micro collateral evolutions into a coherent macro collateral evolution. Such an automated detection of macro collateral evolutions would allow a finer assessment of the complexity of the collateral evolutions that occur across Linux versions and would improve the detection of protocols, as compared to our current analysis, which only considers individual added and dropped function calls.

The pervasiveness of the need for collateral evolutions in device-specific code, as identified in this paper, calls for the development of tools to aid the driver maintainer in this task. Our long term goal is to design a tool, Coccinelle,[1] that provides a formal notation for describing collateral evolutions and a transformation engine to assist developers in applying them. As a proof of concept, we plan to use this tool to return "back to the future" of Linux 2.4 and replay the evolution to Linux 2.6. The analysis of collateral evolutions conducted in this work thus represents a first step towards making collateral evolution easy and robust, in order to improve the reliability of device support in operating systems.

Acknowledgments

We would like to thank Damien Deville for his contributions to the initial study of collateral evolutions, and our shepherd Brian Bershad for his help in refining the paper. This work has been supported in part by the Agence Nationale de la Recherche (France) and the Danish Research Council for Technology and Production Sciences.

Availability

The various tools developed for this work, including the patch analyzer, are available at the following URL:

`http://www.emn.fr/x-info/coccinelle/`

7. REFERENCES

[1] J. Appavoo, M. Auslander, M. Burtico, D. D. Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42: an open-source Linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, Leuven, Belgium, Apr. 2006. To appear.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In SOSP'01 [21], pages 73–88.

[4] A. C. de Melo, D. Jones, and J. Garzik, 2001. http://umeet.uninet.edu/umeet2001/talk/15-12-2001/arnaldo-talk.html.

[5] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Diego, CA, Oct. 2000.

[6] D. R. Engler, D. Y. Chen, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In SOSP'01 [21], pages 57–72.

[7] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *2000 USENIX Annual Technical Conference*, pages 73–86, Monterey, CA, June 2002.

[8] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 38–51, Saint-Malo, France, Oct. 1997.

[9] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 38–51, Berlin, Germany, June 2002.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[11] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *International Conference on Software Maintenance (ICSM'00)*, pages 131–142, San Jose, CA, 2000. IEEE.

[12] A. E. Hassan. *Mining Software Repositories to Assist Developers and Support Managers.* PhD thesis, School of Computer Science, Faculty of Mathematics, University of Waterloo, Ontario, Canada, 2004.

[13] C. Hellwig, 2003. http://www.cs.helsinki.fi/linux/linux-kernel/2003-20/1120.html.

[14] P. Koellner, Feb. 2002. http://www.uwsg.iu.edu/hypermail/linux/kernel/0202.2/0106.html.

[15] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In OSDI'04 [19], pages 17–30.

[16] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In OSDI'04 [19], pages 289–302.

[17] LWN. API changes in the 2.6 kernel series, Oct. 2005. http://lwn.net/Articles/2.6-kernel-api/.

[18] D. S. Miller, Feb. 2002. http://www.ussg.iu.edu/hypermail/linux/kernel/0202.1/0855.html.

[19] *Proceedings of the Sixth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Fransisco, CA, Dec. 2004.

[20] A. Rubini and J. Corbet. *Linux Device Drivers, 2nd Edition.* O'Reilly, June 2001.

[21] *Proceedings of the 18th ACM Symposium on Operating System Principles*, Banff, Canada, Oct. 2001.

[22] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In OSDI'04 [19], pages 1–16.

[23] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, Feb. 2005.

[24] D. Wambolt, Dec. 2001. http://seclists.org/lists/linux-kernel/2001/Dec/2027.html.

[25] J. Weber, Feb. 2002. http://www.ussg.iu.edu/hypermail/linux/kernel/0202.1/0697.html.

[1]A coccinelle is a ladybug, which is an insect that eats smaller bugs.