Tarantula: Killing Driver Bugs Before They Hatch

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

Gilles Muller and Richard Urunuela
OBASCO Group, École des Mines de Nantes-INRIA, LINA
44307 Nantes Cedex 3, France
gilles.muller@emn.fr, rurunuel@emn.fr

Abstract

The Linux operating system is undergoing continual evolution. Evolution in the kernel and generic driver modules often triggers the need for corresponding evolutions in specific device drivers. Such collateral evolutions are tedious, because of the large number of device drivers, and error-prone, because of the complexity of the code modifications involved. We propose an automatic tool, Tarantula, to aid in this process. In this paper, we examine some recent evolutions in Linux and the collateral evolutions they trigger, and assess the corresponding requirements on Tarantula.

1 Introduction

The Linux operating system (OS) is undergoing continual evolution to improve performance, meet new hardware requirements, and improve the software architecture. When evolution in one OS kernel module causes the interface of the module to change, the need for evolution percolates out into other OS services. This collateral evolution can become quite tedious when many modules depend on the interface and the modifications required are complex. It is also error-prone, because of the difficulty of understanding both the evolution and its impact on the dependent modules. As a result, some collateral evolutions happen very slowly and bugs are introduced. The problems are compounded for modules outside the kernel source tree, which are maintained by developers different from those performing the original evolution and who may not have access to complete information about evolution requirements.

Device drivers are particularly vulnerable to the need for collateral evolution. As illustrated in Figure 1, drivers depend on services provided by the kernel and by modules generic to various families of devices. Due to the rapid proliferation of new devices, there are many drivers. Indeed, an evolution in a generic function defined by the kernel can require modification of over a hundred driver files. Drivers are also a high priority for users, who, in an open system such as Linux, can submit patches to update the drivers for their machines, despite not having a complete understanding of the implications of the evolution.

A particularly striking example of the difficulty of driver evolution is the case of the function `check_region` used in driver initialization. In Linux 2.4.1, this function was called 322 times in 197 driver files. Starting in Linux 2.4.2 (Feb. 2001), the use of this function began to be eliminated, because changes in the driver initialization process implied that its use could cause race conditions. Eliminating `check_region` requires both replacing it with a call to `request_region` and introducing some cleanup code at any subsequent code point that indicates failure of the driver initialization process. Identifying the latter code points entails a non-trivial control-flow analysis possibly across multiple functions. Accordingly, bugs have appeared in the process of eliminating `check_region` and the evolution is not complete as of Linux 2.6.10 (Dec. 2004), even though the function has been deprecated since Linux 2.5.54 (Jan. 2003).

To reduce the difficulty of performing collateral evolution of device drivers, we propose to develop an automatic tool, Tarantula, to aid in the evolution process. Using Tarantula, a collateral evolution is described as a set of rewrite rules, referred to as a semantic patch, that specify the affected code patterns and associated changes. Given a driver and a semantic patch, Tarantula identifies driver code that matches the code patterns and interactively proposes the associated changes. If the user accepts a change, Tarantula transforms the code automatically. We envision that a developer who modifies the interface of a generic module also writes a corresponding semantic patch. This developer then applies the semantic

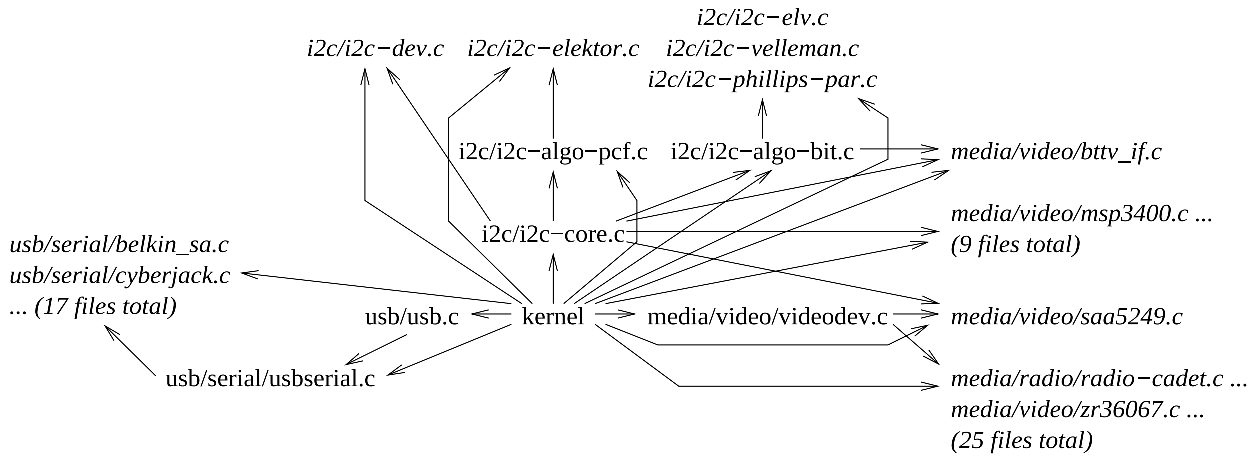

Figure 1: Some kernel dependencies in Linux 2.4.27 (device drivers are shown in italics)

patch to drivers in the kernel source tree, profiting from the interactivity of Tarantula to identify overlooked code patterns and code fragments that are matched inappropriately. When a semantic patch has been validated on the kernel sources, the developer makes it publicly available for use by the maintainers of drivers outside the kernel source tree.

In this paper, we present preliminary work in the development of Tarantula. Based on a study of evolution in driver code across versions of Linux 2.4 through 2.6, we present examples that illustrate the kinds of code modification that collateral evolution entails. In each case, we assess the corresponding requirements on the expressiveness of semantic patches and on the power of the underlying rewriting engine. In terms of expressiveness, we find the need for rewrite rules that describe control-flow paths, for which we propose to use temporal logic (CTL) [10]. To support such rules, we find the need for a rewriting engine that includes inter-procedural control-flow analysis, alias analysis, and constant propagation.

The rest of this paper is organized as follows. Section 2 presents some examples of evolution in Linux. Section 3 assesses these examples in terms of the requirements that they pose on Tarantula. Finally, Section 4 presents related work and Section 5 concludes.

2 Examples

In this section, we present some representative examples of evolution in Linux and the difficulties that have arisen in the collateral evolutions in driver code.

2.1 Elimination of `check_region`

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 [18]. In this case, a driver determines whether its device is attached to a given port as follows: (i) calling `check_region` to find out whether the memory region associated with the port is already allocated to another driver, (ii) if not, then performing some driver-specific tests to identify the device attached to the port, and (iii) if the desired device is found, then calling `request_region` to reserve the memory region for the current driver. In more recent versions of Linux, the kernel initializes device drivers concurrently [5]. 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. Starting with Linux 2.4.2, device drivers 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` is 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

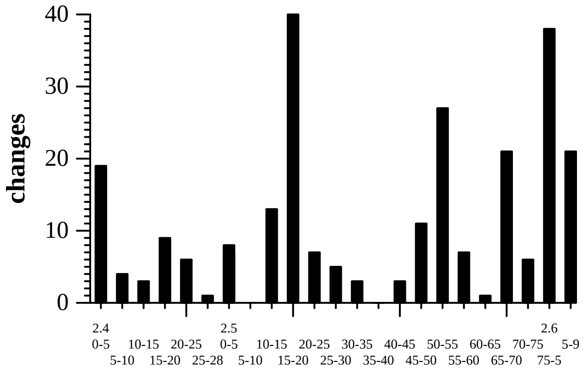

Figure 2: `check_region` elimination in Linux 2.4-2.6

to `check_region`. In Linux 2.4.1, the call to `request_region` is in the same function for only 56% of the calls to `check_region`.[1] In the remaining cases, an interprocedural analysis is needed. 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. Such points include returning an error value, as found in 75% of the functions calling `check_region`, and going around a loop that checks successive ports until finding one with the desired device, as found in 23% of these functions. At such code points, it may be the case that only a subset of the incoming paths contain a call to `check_region`, as occurs in 31% of the functions calling `check_region`. In this case, the call to `release_region` must be placed under a conditional.

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 2. The evolution is still not complete as of Linux 2.6.10.

2.2 An extra argument for `usb_submit_urb`

The function `usb_submit_urb`, defined until Linux 2.5.7 in the generic module `usb/urb.c`, until Linux 2.5.20 in the generic module `usb/core/usb.c`, and subsequently in the generic module `usb/core/urb.c`, implements the passing of a message, implemented as USB Request Block (urb), by a USB driver. 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 driver programmer 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 more generic modules. The only relevant documentation in the Linux code is the comments preceeding 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 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.

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 3 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

[1] This analysis and the other analyses reported for the elimination of `check_region` were carried out using CIL [17], which requires parsing each file. Due to problems obtaining appropriate compilation arguments and incompatibilities between the Linux 2.4.1 code and the gcc 3.3.3 compiler, we were only able to parse 78% of the driver files successfully. The percentages reported here are as compared to this set of parsable files.

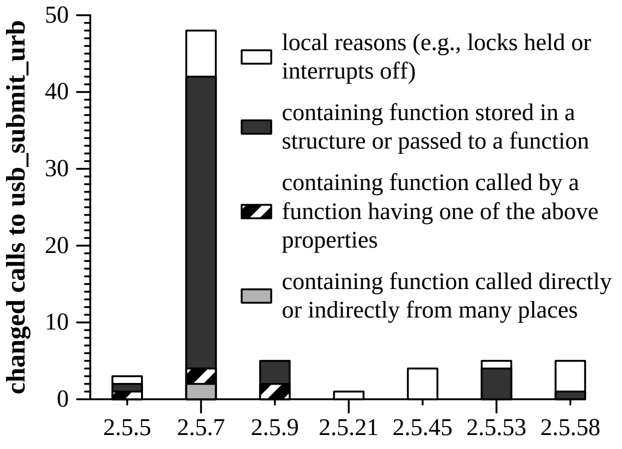

Figure 3: Linux 2.5 versions in which `GFP_ATOMIC` replaces `GFP_KERNEL` in a call to `usb_submit_urb`

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 2.6.10, in the file `usb/class/audio.c`, `GFP_KERNEL` is still used in one function where interrupts are turned off.

2.3 Introduction of `video_usercopy`

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, in the generic module `media/video/videodev.c`, a wrapper function was introduced to encapsulate this argument copying. This function was refined in Linux 2.5.8 and named `video_usercopy`. As of Linux 2.6.9, `video_usercopy` was used in 29 `media` files.

Introducing the use of `video_usercopy` requires primarily (i) identifying the ioctl function and (ii) rewriting its code to eliminate copying between user and kernel space. An ioctl function does not have a fixed name, but can be recognized as the value stored in the `ioctl` field of the structure implementing the driver interface. Copying between user and kernel space is typically implemented by using the functions `copy_from_user` and `copy_to_user` to copy information to and from a local structure specific to each ioctl command. `Video_usercopy` provides the ioctl code with a generic-typed kernel pointer to this information. The ioctl code must thus be modified to cast this pointer to the structure type used by each command and to replace 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 function `video_usercopy` is not specific to media drivers, and thus there has been interest in making the function more generally available [9]. 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.

3 Requirements

The semantic patches of Tarantula must (i) identify the code to modify, (ii) describe how to construct the new code, and (iii) describe the impact on the existing context. We review the above examples in terms of these issues, and identify the requirements they place on Tarantula. Required features are shown in italics.

In the `check_region` example, the code to modify is indicated by a use of the function name. The new code that replaces a call to `check_region` is determined by the call to `request_region` that would subsequently be executed at run time. To specify the connection between these calls, the rewrite rules must be able to describe a control-flow path. For this, we propose to use *temporal logic* [10], a logic that describes relationships between successive events, instantiated here as successive program constructs. So that the rewriting engine can identify such paths in the source program, it must include a *control-flow analysis.* Because the calls to `check_region` and `request_region` are not always in the same function, the control-flow analysis must be *inter-procedural.* Finally, replacement of `check_region` by `request_region` implies that calls to `release_region` must be inserted in the context. This again requires rewrite rules that describe paths, and temporal logic and control-flow analysis are useful here. Some of the paths requiring `release_region` are interprocedural error paths. *Constant propagation* of error return values is thus needed to restrict the analysis to meaningful control-flow paths.

In the `usb_submit_urb` example, the code to transform is again indicated by a use of the function name. The new argument is determined by properties of the enclosing calling context. Again, these properties are interprocedural and depend on control flow, and thus temporal logic and control-flow analysis are useful. In a few cases, functions containing calls to `usb_submit_urb` are stored in structures or variables local to the driver are subsequently invoked through these entities. These cases require *alias analysis.*

In the `video_usercopy` example, identifying the code to transform requires finding the ioctl function, which entails reasoning about *global structure declarations.* The introduction of `video_usercopy` has a significant effect on the context: calls to `copy_from_user` and `copy_to_user` disappear, and the types of the variables manipulated by these functions change. To express these modifications, the rewrite rules must be able to express properties of *local-variable declarations and uses.*

We have previously used rewrite rules including temporal logic to describe the modifications needed to reengineer the source code of a legacy OS to support the Bossa process scheduling framework [1, 16]. Those rules were implemented using the CIL infrastructure for C program analysis and transformation [17]. For Tarantula, we will generalize this work by extending the rewrite rule language to describe a more general set of transformations, and by improving the rewriting engine to include more complex variants of the analyses, such as inter-procedural analyses exploiting constant values. Of the required analyses, CIL already provides intra-procedural control-flow analysis, inter-procedural constant propagation, and inter-procedural alias analysis.

4 Related Work

Our work involves the description of code patterns requiring evolution and the transformation of code matching these patterns using rewrite rules. This work is related to pattern-based approaches to bug finding and to techniques that allow the description of code modifications such as Aspect-Oriented Programming (AOP).

Recent years have seen a surge of interest in automatic approaches to detecting bugs in large pieces of software, including the Linux operating system [6, 7, 8, 14]. These approaches rely on identifying required code patterns and then detect code fragments that are inconsistent with these patterns. In the context of Linux, most of the bugs found using these approaches are in device driver code. We believe that the patterns used by these approaches derive largely from the interface provided by the kernel and generic modules. In the context of evolution of this interface, existing approaches detect bugs after they appear, while our approach prevents bugs by providing assistance in the evolution process. Our work can also be viewed as introducing a new source of code patterns into consideration. While previous work has focused on patterns identified within a single version of Linux, we consider patterns derived from evolution.

AOP is a programming paradigm that isolates the implementation of a modular crosscutting concern in a single unit, known as an aspect [12]. An aspect includes both code implementing the concern and directives indicating how to integrate this code with an existing base program. Coady *et al.* have investigated the use of aspects in OS code to improve modularity, and have considered the impact of OS evolution on these aspects [2, 3, 4]. Semantic patches can be viewed as a form of aspects, as they specify code and a means of determining where this code should be introduced. Nevertheless, the goals of our approach, and hence the mechanisms employed, are different. AOP is directed towards the complete implementation of a functionality that is somewhat orthogonal to the base program. Thus, for example, the widely-used aspect system, AspectJ [11], does not permit fine-grained modification of the base program, such as changing the type of a local variable. Our approach is directed towards specifying modifications to a portion of an integral functionality, specifically the interaction with the interface of a more generic module. Accordingly, our approach allows describing much more invasive, finer-grained transformations and requires more complex supporting analyses.

The Splice aspect system allows an aspect to use program analysis to specify where a base program should be transformed [15]. The specification is described in terms of logic programming rules combined with operators expressing temporal properties. Based on our previous experience in describing temporal properties in the reengineering of Bossa, we plan to use temporal logic directly, rather than via logic programming. The precision of the analyses used by Splice has been restricted to ensure scalability to large programs. Because we have observed that device drivers typically have shallow call graphs, we plan to favor analysis precision over efficiency. Finally, Splice has only been used to implement lock insertion and a loop transformation, whereas we target a much wider range of transformations.

Our use of temporal logic was originally inspired by that of Lacey *et al.* on using temporal logic to specify program transformations [13].

5 Conclusion

Keeping drivers up to date is known to be difficult, due to the large number of drivers and the varying levels of programmer expertise. In this paper, we have proposed Tarantula to provide automatic assistance in evolving a driver to match changes in the interface of more generic parts of the OS. Tarantula is based on semantic patches, which provide (i) precise description of the contexts in which evolution is required, (ii) encapsulation of relevant information about external functions and data structures, and (iii) help with the tedious process of analyzing the driver file to determine where the evolution applies. So far, besides the examples cited here, we have found around 30 evolutions in driver directories such as `cdrom`, `ide`, `pcmcia`, and `usb` where Tarantula would be useful. We plan to continuing studying driver code to find a more complete set of examples. Our next step will be to refine the language of semantic patches and develop the supporting program analysis infrastructure.

Acknowledgments We would like to thank Anne-Françoise Le Meur and Laurent Réveillère for their comments on earlier versions of this paper.

References

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 196–204, Montreal, Canada, Oct. 2003. IEEE.

[2] S. Bray, M. Yuen, Y. Coady, and M. E. Fiuczynski. Managing variability in systems: Oh what a tangled OS we weave. In *Workshop on Managing Variabilities Consistently in Design and Code*, Vancouver, Canada, Oct. 2004.

[3] Y. Coady. *Improving Evolvability of Operating Systems with AspectC.* PhD thesis, The University of British Columbia, July 2003.

[4] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 50–59, Boston, MA, Mar. 2003.

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

[6] 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.

[7] 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 *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 57–72, Banff, Canada, Oct. 2001.

[8] 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.

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

[10] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 2000.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 – Object-Oriented Programming, 15th European Conference*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 – Object-Oriented Programming, 11th European Conference*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 1997.

[13] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Compiler Construction, 10th International Conference, CC 2001*, number 2027 in Lecture Notes in Computer Science, pages 52–68, Genova, Italy, 2001.

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

[15] S. McDirmid and W. C. Hsieh. Splice: Aspects that analyze programs. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*, number 3286 in Lecture Notes in Computer Science, pages 19–38, Vancouver, Canada, Oct. 2004.

[16] G. Muller, J. Lawall, J.-M. Menaud, and M. Südholt. Constructing component-based extension interfaces in legacy systems code. In *ACM SIGOPS European Workshop 2004 (EW2004)*, pages 80–85, Leuven, Belgium, Sept. 2004.

[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002*, number 2304 in Lecture Notes in Computer Science, pages 213–228, Grenoble, France, Apr. 2002.

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