`patch (1)` Considered Harmful

Marc E. Fiuczynski
Princeton University

Robert Grimm
New York University

Yvonne Coady
University of Victoria

David Walker
Princeton University

Abstract

Linux is increasingly used to power everything from embedded devices to super computers. Developers of such systems often start with a mainline kernel from `kernel.org` and then apply patches for their application domain. Many of these patches represent *crosscutting concerns* in that they do not fit within a single program module and are scattered throughout the kernel sources—easily affecting over a hundred files. It requires nontrivial effort to maintain such a crosscutting patch, even across minor kernel upgrades due to the variability of the kernel proper. Moreover, it is a significant challenge to ensure the kernel's correctness when integrating multiple crosscutting concerns. To make matters worse, developers use simple code merging tools that are limited to textual substitution, with the result that patch maintenance is error prone and time consuming. In this paper, we propose a new *semantic patch* tool, called `c4`, designed to simplify the management of OS variations and thereby improving OS evolution.

1 Introduction

Over the past years open source operating systems, particularly Linux, have experienced tremendous growth. Industry and governments alike are relying upon such software to reduce the cost and time-to-market of developing WiFi routers, cell phones, and telecommunications equipment and of running services on specialized servers, clusters, and high-performance super computers. One important benefit of using Linux for these systems is that developers have access to all kernel sources and can easily create variants that are directly tailored for their application domains. As such, Linux also is an attractive platform for OS research, as it offers the potential for a speedy technology transfer.

Major variants to a mainline Linux kernel are typically represented in terms of higher-level extensions that are implemented through so-called patch sets or, simply, patches. For example, embedded systems require changes that reduce the kernel's memory footprint (e.g., Linux-Tiny [16] or uCLinux [26]), desktops require strong security mechanisms that reduce the impact of viruses and worms (e.g., LSM [15]), time-shared servers require resource management subsystems to isolate users from each other (e.g., VServer [18] or CKRM [4]), and super computers require special resource management modifications that scale the OS to a large number of components (e.g., CPUSETS [7]). Many of these kernel extensions do not fit within a single program module and are scattered throughout the kernel sources. As shown in the following table, each extension can easily cover a hundred existing kernel files, even though it represents a logical unit, expressing a single *crosscutting concern*:

Patch	New Files	Modified Kernel Files
Nooks [24]	68	108
CKRM [4]	48	53
LSM [15]	123	85
Kernel Probes [13]	13	20
LTT [17]	9	71
VServer [18]	40	211
Linux-Tiny [16]	7	142
CPUSETS [7]	1	3
ALSA [1]	200	540
LLA [19]	1	39

It requires non-trivial effort to maintain even a small crosscutting extension between minor kernel upgrades due to the variability of the kernel proper. Moreover, it is a tremendous challenge to ensure the kernel's correctness when integrating multiple crosscutting kernel extensions, as even for the small number of patch sets shown in the above table there is significant overlap in the files affected by the different extensions. To make matters worse, developers currently use simple code merging tools (e.g., `diff` and `patch`), which are limited to indicating conflicts based upon textual comparison. Experience with maintaining a variant of the Linux kernel for PlanetLab (which contains several major variants to the mainline code base) as well as anecdotal evidence from the Linux and OS research communities suggest that this approach is error prone and time consuming.

Developers wishing to merge their kernel extensions into the mainline code base must repeatedly go through this process, because any non-trivial change to Linux's architecture takes time to be reviewed and accepted.

Anecdotal evidence (e.g., LSM, LTT, ALSA) suggests that it may take anywhere from one to three years before a crosscutting kernel extension is fully integrated into the mainline kernel.

This leads to a natural selection of kernel extension developers: those that are persistent and those that are not. While this natural selection weeds out the weak, it also eliminates strong work done by members of the OS research community. For example, the Nooks [24] project for recoverable Linux device drivers has garnered best paper awards at both SOSP and OSDI. Yet the work remains relegated to Linux 2.4.18, which was the kernel version at the start of Nooks in Feb. 2002. The problem is not laziness! Rather, with today's tools, it is simply too tedious to keep up with the changes that occur between even minor releases of Linux, e.g., from 2.4.18 to 2.4.19.

Our position is that a better method is needed—beyond `diff` and `patch`—that reduces the amount of work it takes to maintain and review a crosscutting kernel extension in Linux. The remainder of this paper describes our work towards such a solution: a *semantic patch system* called `c4` for CrossCutting C Code. A semantic patch basically amounts to a set of transformation rules that precisely specify the conditions under which changes need to be made and the means for rewriting the affected code. Its compact yet human readable form lets a community of developers easily understand and discuss a crosscutting kernel extension, thereby helping reduce the time and effort required to evolve the kernel.

2 Motivating Observations

We studied a number of patch files—LTT, Kernel Probes, LSM, and CKRM—that introduce new kernel extensions as well as patch files that update existing code in Linux. In general, the changes introduced by these patch files fall into three categories*:

- *Logic changes.* Modifications to the internal logic of a function (e.g., to eliminate a bug).
- *Behavior changes.* Modifications to the semantics of Linux's existing interfaces (e.g., changing all file operations to use ACLs rather than standard UNIX permissions).
- *Version changes.* Modifications to either a function's signature or the field makeup of a non-ADT data structure (e.g., adding, deleting, or changing the type of an argument or field).

Our preliminary analysis of patches that update existing kernel functionality shows that the bulk fall into the *logic changes* category with very few *version changes* and no *behavior changes*. In contrast, patches that introduce new kernel extensions fall primarily into both the *behavior* and *version* changes categories. The syntactic substitution approach embodied by `diff` and `patch` suffices for capturing and applying *logic changes*. After all, they are of syntactic nature! However, both *behavior* and *version* changes are of semantic nature, leading to significant problems when capturing and applying such changes with syntactic tools.

Behavior changes modify the semantics of system interfaces. They are usually implemented by introducing call-out hooks either at the *beginning* or *end* of the functions that implement these interfaces. Yet logically, the new functionality is executed either *before* or *after* the hooked functions. In other words, the change in implementation is of a syntactic nature (i.e., beginning or end of a function), whereas the behavioral change truly is of a semantic nature (i.e., before or after a function). The problem with patch sets is that any other textual modification at the the same locations will result in a conflict.

To illustrate this problem, consider the patch set for the Class-Based Kernel Resource Management (CKRM) project [4]. CKRM is a new kernel mechanism that provides differentiated services for shared system resources, including CPU time, tasks, memory, and disk I/O. The application of this patch set results in a new Linux kernel variant suitable for servers that require stronger resource usage guarantees than the egalitarian approach used by the unmodified mainline kernel.

The actual CKRM extension consists of a set of files that specify where "hunks" of code are applied by `patch`, identifying specific line numbers or relative offsets within specific files. For example, this portion of the CKRM patch:

```
--- a/kernel/sys.c Sat Sep 18 19:28:57 2004
+++ b/kernel/sys.c Tue Feb 01 22:03:15 2005
@@ -638,6 +642,9 @@
  else
     return -EPERM;
+
+ ckrm_cb_gid();
+
  return 0;
@@ -726,6 +733,8 @@
     current->suid = current->euid;
  current->fsuid = current->euid;
+
+ ckrm_cb_uid();
+
  return security_task_post_setuid(old_ruid,
```

instruments `kernel/sys.c` with calls to the ckrm_cb_gid()/uid() functions. Line numbers are represented as relative offsets indicated by `@@`*line-info*`@@`. Upon closer inspection of the patch we observe a pattern: all calls to ckrm_cb_gid()/uid() (6 in total) directly precede return statements.

The same pattern emerges for other kernel extensions, such as LSM and VServer, that hook themselves into specific Linux subsystems. Thus, composing several such kernel extensions or updating to a new release of the kernel may result in unnecessary patch conflicts. Such

*A fourth category comprises modifications to Makefiles etc., which we do not consider.

conflicts typically need to be resolved manually, which is clearly tedious. Section 3 presents our solution to this problem, which transforms *behavior changes* into *aspects* using aspect-oriented software development [2] (AOSD) techniques.

Version changes involve modifications to either a function's signature or the field makeup of a data structure. But changing a function's signature or deleting/changing a data structure's fields can have far-reaching consequences: it requires updating all modules that directly use them. Consequently, capturing such changes with `diff` and `patch` requires manually updating all dependent modules. This is prohibitive when the interface changes are in the kernel proper or in the generic device driver framework and trigger corresponding changes in specific device drivers—there might be hundreds.

Muller et. al [22] are exploring how to percolate such interface changes throughout the Linux code base. Their approach, similar to ours, builds on a kind of semantic patch, which relies on code rewriting rules to automate the task of updating dependent modules. While this appears promising, we observe that *version changes* might be better handled by: (1) a systematic conversion of non-ADTs used across Linux subsystems to ADTs, and (2) using well established interface versioning techniques such as Microsoft's Component Object Model (COM).

3 The `c4` Semantic Patch Compiler

Recognizing that *behavioral changes* are common to new kernel extensions for Linux, our approach is to make them part of the kernel's architecture by leveraging AOSD techniques. More specifically, our approach is to express *behavioral changes* as semantic patches using aspects, which provide a language-supported methodology for integrating crosscutting concerns with a program. The benefits of aspects are twofold. First, they provide a well-defined specification of domain-specific features that is separate from baseline functionality, yet can be automatically integrated with the kernel. Second, they enable automatic reasoning about the implications of composing several crosscutting concerns and the identification of semantic conflicts.

The main research questions raised by our approach are (1) how to extend C with aspects without impacting compatibility, readability, or performance and (2) how to automate the identification and resolution of conflicts between aspects. However, fully exploring these research questions requires building the corresponding tools. To reduce the required engineering effort, we are not implementing a self-contained C compiler for our AOSD-enhanced C language, called `c4` for CrossCutting C Code. Rather, we leverage existing platform support for C and rely on a pipeline that first invokes the C preprocessor, which resolves all # directives, then the `c4` compiler to translate aspect-enhanced code to plain C, and finally gcc, which performs traditional optimizations and code generation. To further reduce the engineering effort required for building `c4`, we are implementing `c4` on top of the `xtc` compiler framework [11], which provides a toolkit for building extensible source-to-source transformers. In the rest of this section, we present the proposed aspect-oriented language enhancements to C by example and then discuss our approach to non-interference analysis for aspects.

3.1 The `c4` Language

In `c4`, which is based on AspectC [6], aspects structure and modularize concerns that crosscut functions. Due to space constraints, we do not define the `c4` language in detail. Rather, we illustrate the gist of its features on the example of instrumenting the kernel with calls to ckrm_cb_gid()/uid() *after* the *execution* of sys_setregid()/setreuid(), sys_setgid()/setuid(), and sys_setresgid()/setresuid(), respectively:

```
aspect {
     pointcut setuid() :
       execution(long sys_setreuid(..))  &&
       execution(long sys_setresuid(..)) &&
       execution(long sys_setuid(..));

     after setuid() { ckrm_cb_uid(); }

     pointcut setgid() :
       execution(long sys_setregid(..))  &&
       execution(long sys_setresgid(..)) &&
       execution(long sys_setgid(..));

     after setgid() { ckrm_cb_gid(); }
}
```

The *execution* keyword refers to principled points in the execution of a program called *join points* (e.g., `sys_setreuid`). A *pointcut* statement groups one or more join points, which can then be referenced by *advice* to define actions for these join points. In our example, we only use *after* advice, which indicates that the actions (the explicit C code) should be performed after the *execution* of the join points.

The aspect code thus structures the modifications to the mainline code, which are automatically merged, or *weaved*, with the appropriate C code by the `c4` compiler. In contrast to the syntactic patch shown in Section 2, the interaction with the kernel becomes explicit at the level of functions and parameters involved; hence, code becomes more amenable to semantic analysis and developers can reason about any interactions at a higher level. Previous work has shown that this reduces the complexity of crosscutting concerns [6, 23].

Note that the `c4` language is richer than suggested by this example. In particular, it supports not only *after*, but also *before* and *around*, with the latter replacing an existing mainline function. Coady describes this in further detail for AspectC [6], upon which `c4` is based. Further note that we aim to reduce developers' exposure

to `c4` as much as possible. In particular, we are exploring how to support simple annotations of the form `aspect(Name){...}`, which can be added inline at the *beginning* or *end* of system functions and are then automatically extracted and converted into fully-featured aspects by the `c4` compiler.

3.2 Program Analysis

Our initial research goal is to support the *syntactic* separation of crosscutting concerns through aspects. On their own, syntactic separation and automatic weaving of crosscutting concerns free developers from many low-level, time-consuming, and error-prone details of maintaining and applying kernel patches. However, in addition to supporting syntactic separation of crosscutting concerns, we are also targeting *semantic* separation through the detection of interference between concerns. When two concerns are semantically separate, the execution of one concern is guaranteed not to change the execution behavior of the other. For instance, semantically separate concerns do not mutate shared data structures either directly or indirectly through a series of function calls. Semantically separate concerns are of critical importance in large systems such as Linux, in which multiple developers work independently on their own system extensions. When concerns are semantically separate, these independent developers need not coordinate their work, analyze the code of the other developers, or even be aware that other projects are being developed. By definition, the work of one developer does not interfere with the other.

In addition to separating multiple "after-the-fact" concerns, it is useful to determine the degree to which a particular concern is separate from, or, conversely, interferes with, the mainline code. If a developer can prove, via an automated program analysis, that their concern does not interfere with the mainline code, then owners of the mainline are much more likely to integrate it into their system. Even if the owners themselves will not integrate the new concern, users will be less hesitant to download and apply the non-interfering patch. We believe that analysis of noninterference properties of aspects can greatly speed technology transfer and integration of new ideas into Linux (and other open source software).

We have begun to investigate how to design a static program analysis that will detect whether a new concern interferes with the mainline computation [8] or with another, existing concern [3]. This analysis makes use of previous work developed by programming language and security researchers on detecting and enforcing data integrity properties via information flow analysis. Our analysis is designed as a form of type-and-effect system that separates state into different logical protection domains, with one protection domain for each concern and one domain for the mainline computation. The analysis is designed to detect situations, in which code from one domain mutates state in another, either directly or indirectly through a series of function calls. We are in the process of formally proving a powerful non-interference result for our analysis.

While an important step, there still are considerable challenges to using this analysis in the context of C and the Linux kernel. A first step for this research will be to refactor crosscutting concerns in Linux and to analyze the degree to which various concerns really are separate from one another and the mainline kernel. This experience will be crucial in refining the theoretical analysis and in understanding the specific noninterference properties that will be useful (and possible) to specify and enforce. The next step will be to extend `c4` with a system of annotations that let developers specify their noninterference and semantic separation requirements. Even without an analysis, the annotation system will be useful as a systematic form of documentation of developer intentions and requirements. The last step is to implement the analysis itself and test it on kernel extensions in Linux. As we refine and develop our analysis, we intend not only to implement it, but also to formalize it and prove its correctness. Developing the formal techniques to do this is an important research challenge that will have a significant impact on the programming languages community.

4 Related Work

Both IBM and Microsoft recently announced their intentions on using AOSD to improve the evolvability of complex software systems [14, 25]. Our work differs from these industry initiatives in three important aspects. First, we are investigating aspect-oriented programming within C as opposed to existing efforts on C++, C#, or Java. Consequently, our work directly applies to a large, existing code base. Second, we are specifically targeting the software architecture of a major open source operating system, which provides us with an opportunity to address a real-world problem faced by many organizations. Finally, we plan to develop formal semantics for reasoning about aspect-oriented technology in this domain, and use this formalization to develop program analysis tools to further aid systems programmers in general.

AOSD techniques have been previously applied to operating systems. Both Coady et al. [6] and Spinczyk et al. [20] demonstrate that concerns that crosscut traditional layers in OS structure can be cleanly defined and applied using aspects.

Over the last several years, a number of researchers, including Walker et al. [27], have begun to build semantic foundations for aspect-oriented programming [28, 9, 12, 21, 5]. This foundational, theoretical work provides a starting point for analyzing the properties of aspect-

oriented programs, developing principled new programming features, and deriving useful program analyses. We plan to exploit our knowledge of and experience with these semantic foundations and type-based analyses as we develop the `c4` language.

Recently, programming language researchers have begun to try to understand and analyze interactions between separate concerns. For instance, Bauer et al. [3] introduces a theoretical language that includes several different ways for combining concerns and a type system for detecting when concerns apply to the same program points. In similar work, Douence et al. [10] analyze aspects defined by recursion together with parallel and sequencing combinators. They develop a number of formal laws for reasoning about their combinators and an algorithm that is able to detect aspect independence. These proposals present interesting techniques for detecting interference, but it appears that additional reasoning facilities will be required for analyzing crosscutting concerns in the Linux kernel, as many of the "separate" concerns actually reference the same program points. We believe that more recent work by Dantas et al. [8], which analyzes aspect code to determine its memory effects, will help solve this problem. However, all of these proposals are quite theoretical in nature, and it will take substantial research to understand how to modify and apply these ideas in the context of `c4` and the Linux kernel.

5 Summary

Our position is that current techniques for evolving operating systems are ineffective, since they solely operate at the syntactic level. Our work introduces a semantic patch system based on *aspects* that offers the ability to more rapidly and seamlessly move from idea to design to implementation for new OS features. Aspects' inherent separation of code from an operating system's mainline eases the maintenance of crosscutting concerns, thus speeding up the technology transfer of a new kernel extension from early prototype, through multiple design iterations, to a mainlined feature of an operating system that continues to evolve.

References Cited

[1] Advanced Linux Sound Architecture. `www.alsa-project.org`.

[2] Aspect Oriented Software Development. `www.aosd.net`.

[3] L. Bauer, J. Ligatti, and D. Walker. Types and effects for non-interfering program monitors. In *International Symposium on Software Security*, Tokyo, Japan, Nov. 2002.

[4] Class-Based Kernel Resource Management. `ckrm.sf.net`.

[5] C. Clifton and G. T. Leavens. Assistants and observers: A proposal for modular aspect-oriented reasoning. In *Foundations of Aspect Languages*, Apr. 2002.

[6] Y. Coady. Bibliography on aspect-oriented software development research. `www.cs.uvic.ca/~ycoady/aspectresearch.html`.

[7] CPUSETS for Linux. `www.bullopensource.org/cpuset`.

[8] D. S. Dantas and D. Walker. Harmless advice. In *Workshop on Foundations of Object-oriented Languages*, Jan. 2005. To appear.

[9] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Third International Conference on Metalevel architectures and separation of crosscutting concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Berlin, Sept. 2001. Springer Verlag.

[10] R. Douence, O. Motelet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Conference on Aspect-Oriented Software Development*, pages 141–150, Mar. 2004.

[11] R. Grimm. Practical packrat parsing. Technical Report TR2004-854, New York University, Mar. 2004.

[12] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[13] Kernel Probes. `www-124.ibm.com/linux/projects/kprobes`.

[14] G. Kiczales. The more the merrier. Software Development Magazine `www.sdmagazine.com/documents/s=8993/sdm0410g`, 2004.

[15] Linux Security Modules. `lsm.immunix.org`.

[16] Linux-Tiny. `www.selenic.com/tiny-about`.

[17] Linux Trace Toolkit. `www.opersys.com/LTT`.

[18] Linux VServer Project. `linux-vserver.org`.

[19] Low Latency Audio. `www.linuxdj.com/audio/lad/resourceslatency.php`.

[20] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schröder-Preikschat. An aspect-oriented implementation of interrupt synchronization in the Pure operating system family. In *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems*, Malaga, Spain, June 2002.

[21] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In G. T. Leavens and R. Cytron, editors, *Foundations of Aspect-Oriented Languages Workshop*, pages 17–25, Apr. 2002.

[22] G. Muller. Private communication.

[23] G. C. Murphy, R. J. Walker, and E. L. A. Baniassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Transactions on Software Engineering*, 25(4), 1999.

[24] M. Swift. Nooks: Improving the reliability of commodity operating systems. `nooks.cs.washington.edu`.

[25] TheServerSide at AOSD 2004: Part Two. `www.theserverside.com/articles/article.tss?l=AOSD2004-2`, 2004.

[26] uCLinux. `www.uclinux.org`.

[27] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ACM SIGPLAN International Conference on Functional Programming Languages*, Uppsala, Sweden, Aug. 2003.

[28] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *Foundations of Aspect-Oriented Languages Workshop*, pages 17–25, Apr. 2002. Iowa State University University technical report 02-06.