

Proceedings of the 9th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '10)

hrsg. von
Bram Adams, Michael Haupt, Daniel Lohmann

Technische Berichte Nr. 33

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam



Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

**Proceedings of the 9th Workshop on
Aspects, Components, and Patterns
for Infrastructure Software
(ACP4IS '10)**

herausgegeben von

Bram Adams
Michael Haupt
Daniel Lohmann

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Universitätsverlag Potsdam 2010

<http://info.ub.uni-potsdam.de/verlag.htm>

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 4623 / Fax: 3474
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

Das Manuskript ist urheberrechtlich geschützt.

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam

URL <http://pub.ub.uni-potsdam.de/volltexte/2010/4122/>

URN [urn:nbn:de:kobv:517-opus-41221](http://nbn-resolving.org/urn:nbn:de:kobv:517-opus-41221)

<http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-41221>

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-043-4

9th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '10)

**co-located with the 9th International Conference on
Aspect-Oriented Software Development (AOSD)
March 16, 2010, Rennes, France**

Workshop Home Page: <http://aosd.net/workshops/acp4is/2010/>

Aspect oriented programming, component models, and design patterns are modern and actively evolving techniques for improving the modularization of complex software. In particular, these techniques hold great promise for the development of “systems infrastructure” software, e. g., application servers, middleware, virtual machines, compilers, operating systems, and other software that provides general services for higher-level applications. The developers of infrastructure software are faced with increasing demands from application programmers needing higher-level support for application development. Meeting these demands requires careful use of software modularization techniques, since infrastructural concerns are notoriously hard to modularize.

Aspects, components, and patterns provide very different means to deal with infrastructure software, but despite their differences, they have much in common. For instance, component models try to free the developer from the need to deal directly with services like security or transactions. These are primary examples of crosscutting concerns, and modularizing such concerns are the main target of aspect-oriented languages. Similarly, design patterns like Visitor and Interceptor facilitate the clean modularization of otherwise tangled concerns.

Building on the ACP4IS meetings at AOSD 2002–2009, ACP4IS '10 aims to provide a highly interactive forum for researchers and developers to discuss the application of and relationships between aspects, components, and patterns within modern infrastructure software. The goal is to put aspects, components, and patterns into a common reference frame and to build connections between the software engineering and systems communities.

Scope of the Workshop

The importance of “systems infrastructure” software—including application servers, virtual machines, middleware, compilers, and operating systems—is increasing as application programmers demand better and higher-level support for software development. Vendors that provide superior support for application development have a competitive advantage. The software industry as a whole benefits from an increased base level of abstraction, decreasing the need for application programmers to continually “reinvent the wheel”.

These trends, however, mean that the demands on infrastructure software are increasing. More and more features and requirements are being “pushed down” into the

infrastructure, and the developers of systems software need better tools and techniques for handling these increased demands. The design and implementation of systems-level software presents unique opportunities and challenges for AOSD techniques. These challenges include the need to address the inherent complexity of infrastructure software, the need for strong assurances of correct and predictable behavior, the need for maximum run-time performance, and the necessity of dealing with the large body of existing systems software components.

This workshop aims to provide a highly interactive forum for researchers and developers to discuss the application of and relationships between aspects, components, and patterns within modern infrastructure software. The goal is to put aspects, components, and patterns into a common reference frame and to build connections between the software engineering and systems communities.

This year's workshop puts special focus on the challenges in system's programming introduced by multi-core platforms. As hardware-supported parallelization becomes mainstream, there is an increasing pressure on systems infrastructure to exploit this new parallelism to its fullest. However, the non-modular nature of parallel execution, and the numerous levels at which parallelism can be achieved (application, systems infrastructure, hardware or even a combination thereof) make it hard to come up with an intuitive, yet efficient parallel architecture. We solicited novel ideas and experience reports on this emerging research area.

Other topics in the scope of the workshop include, but are not restricted to:

- Approaches that combine or relate component-, pattern-, and aspect-based techniques
- Dimensions of infrastructure software quality including comprehensibility, configurability (by implementers), customizability (by users), reliability, evolvability, scalability, and run-time characteristics such as performance and code size
- Merits and downsides of container-, ORB-, and system-based separation of concerns
- Architectural techniques for particular system concerns, e.g., security, static and dynamic optimization, and real-time behaviour
- Design patterns for systems software
- Component, pattern, and aspect "mining" within systems code
- Application- or domain-specific optimization of systems
- Reasoning and optimization across architectural layers
- Quantitative and qualitative evaluations

The workshop is structured to encourage fruitful discussions and build connections between workshop participants. To this end, approximately half of the workshop time

will be devoted to short presentations of accepted papers, with the remaining half devoted to semi-structured discussion groups. Participants will be expected to have read the accepted papers prior to the workshop, to help ensure focused discussions.

A novelty at ACP4IS '10 is that we will invite workshop attendees to give “spontaneous” short presentations on their work if they see a relation to topics being presented and discussed at the workshop. These presentations will be limited to about ten minutes, and are intended to provide additional structured input to discussions. Spontaneous presentations will be asked for during the workshop; no paper needs to be submitted, and no publication is associated with them. There will be a session dedicated to them, just prior to discussion.

Organizers



Bram Adams is a post-doctoral fellow in the Software Analysis and Intelligence Lab at Queen's University (Canada), and is also affiliated with the SOFT lab at the Vrije Universiteit Brussel (Belgium). He obtained his PhD at the GH-SEL lab at Ghent University (Belgium). Bram has a wide range of research interests, ranging from software evolution in general, to the co-evolution of source code and the build system, and advanced separation of concerns. In his PhD, Bram developed a powerful aspect language for C (Aspicere), which he applied to large legacy C systems for reverse-engineering their behavior, re-engineering exception handling idioms and refactoring conditional compilation. Bram served in the program committees of WCRE, IWPSE and ACP4IS, and co-organized the first Practices of Linking Aspect Technology and Evolution workshop (associated with AOSD 2009). Bram is a member of the IEEE.



Michael Haupt is a post-doctoral researcher and lecturer in the Software Architecture Group at Hasso-Plattner-Institut in Potsdam. His research interests are in improving the modularity of complex software system architectures as well as in implementing programming languages, in which latter area his main focus is on faithfully regarding programming paradigms' core mechanisms as primary subjects of language implementation effort. Michael holds a doctoral degree from Technische Universität Darmstadt, where he has worked on the Steamloom virtual machine to provide run-time support for AOP languages. Michael has served as PC member for ECOOP 2008, as reviewer for TAOSD and IEEE TSE, and has been supporting reviewer for the AOSD, ECOOP, ICSE, FSE, MODELS, and VEE conference series. He has co-organized the Dynamic Aspects Workshop series in conjunction with the AOSD conferences, and the Virtual Machines and Intermediate Languages workshop series in conjunction with the AOSD and OOPSLA conferences. Michael is a member of the ACM.



Daniel Lohmann is an assistant professor at the Distributed Systems and Operating Systems group at Friedrich-Alexander University Erlangen-Nuremberg. He has been conducting research in the domain of (embedded) operating systems, software product lines, and aspect oriented programming since 2003. Daniel holds a doctoral degree from Friedrich-Alexander University; in his PhD he developed CiAO, the first purely aspect-oriented operating system. His current research activities are focused on applying AOP ideas for the fine-grained configuration of nonfunctional properties in system software and the new challenges of the many-core area. Daniel co-organized the MMB 2006 workshop on Nonfunctional Properties of Embedded Systems and the ACP4IS 2008 and 2009 workshops. Before joining the PhD programme at Friedrich-Alexander University he worked as a software developer, consultant and IT trainer. Daniel is a member of the ACM, GI, and EUROSYS.

Organization

Program Committee

Mehmet Aksit	University of Twente
Shigeru Chiba	Tokyo Institute of Technology
Eric Eide	University of Utah
Michael Engel	Technische Universität Dortmund
Franz Hauck	Ulm University
Julia Lawall	DIKU
Hidehiko Masuhara	University of Tokyo
Hridesh Rajan	Iowa State University
Doug Simon	Sun Microsystems Laboratories
Olaf Spinczyk	University of Dortmund
Eric Wohlstadter	University of British Columbia
Roel Wuyts	IMEC and K.U. Leuven

Steering Committee

Eric Eide	University of Utah
Olaf Spinczyk	University of Dortmund
Yvonne Coady	University of Victoria
David Lorenz	University of Virginia

Sponsor

The publication of this proceedings volume was sponsored by the AOSD-Europe Network of Excellence, <http://www.aosd-europe.net/>.



Table of Contents

Session 1: Adepts of Code and the Wizards of OS

Nicolas Palix, Julia L. Lawall, Gaël Thomas, Gilles Muller,
How Often Do Experts Make Mistakes?9

Maarten Bynens, Dimitri Van Landuyt, Eddy Truyen, Wouter Joosen,
Towards Reusable Aspects: the Callback Mismatch Problem17

Session 2: Scanners and Sensors for Components and Code

Abdelhakim Hannousse, Gilles Ardourel, Rémi Douence,
Views for Aspectualizing Component Models21

Fan Yang, Hidehiko Masuhara, Tomoyuki Aotani, Flemming Nielson,
Hanne Riis Nielson,
AspectKE: Security Aspects with Program Analysis for Distributed Systems*27

Session 3: Fantastic Frameworks and Infamous Infrastructures

Bholanathsingh Surajbali, Paul Grace, Geoff Coulson,
Preserving Dynamic Reconfiguration Consistency in Aspect Oriented Middleware ... 33

William Harrison,
Malleability, Obliviousness and Aspects for Broadcast Service Attachment41

How Often do Experts Make Mistakes?

Nicolas Palix
DIKU-APL
University of Copenhagen
Denmark
npalix@diku.dk

Julia L. Lawall
INRIA Regal/LIP6
University of Copenhagen
France/Denmark
julia@diku.dk

Gaël Thomas Gilles Muller
INRIA Regal/LIP6
France
{Gael.Thomas, Gilles.Muller}@lip6.fr

Abstract

Large open-source software projects involve developers with a wide variety of backgrounds and expertise. Such software projects furthermore include many internal APIs that developers must understand and use properly. According to the intended purpose of these APIs, they are more or less frequently used, and used by developers with more or less expertise. In this paper, we study the impact of usage patterns and developer expertise on the rate of defects occurring in the use of internal APIs. For this preliminary study, we focus on memory management APIs in the Linux kernel, as the use of these has been shown to be highly error prone in previous work. We study defect rates and developer expertise, to consider *e.g.*, whether widely used APIs are more defect prone because they are used by less experienced developers, or whether defects in widely used APIs are more likely to be fixed.

Categories and Subject Descriptors D.2.8 [Software Engineering]: Metrics—Process metrics, Product metrics; D.3.3 [Programming Languages]: Language Constructs and Features—Patterns

General Terms Measurement, Languages, Reliability

Keywords History of pattern occurrences, bug tracking, Herodotos, Coccinelle

1. Introduction

To ease development, large-scale software projects are often decomposed into multiple interdependent and coordinated modules. Software developers working on one module must then be aware of, and use properly, functions from the APIs of other modules. When a usage protocol is associated with these API functions, it must be carefully followed to ensure the reliability of the system. Large-scale software projects typically also impose coding conventions that should be followed throughout the software project and are not specific to any given API. These conventions ease code understanding, facilitate code review, and ease the maintenance process when many developers are involved in a particular piece of code and when new developers begin to work on the software project.

In this paper, we investigate the degree to which developers at different levels of expertise respect API usage protocols and coding conventions. We focus on the Linux operating system, which as an open source system makes its complete development history available. Furthermore, we focus on memory management APIs, as their use has been found to be highly error prone [3]. The Linux kernel indeed provides both a general-purpose memory management API and highly specialized variants. Thus, it is possible to compare defect rates in APIs that have a related functionality but that require different degrees of expertise to use correctly. We specifically assess the following hypotheses that may be considered to be generally relevant to open-source software:

1. Defects are introduced by less experienced developers.
2. Frequently used APIs are used by developers at all levels of experience, and thus have a high rate of defect occurrences. Nevertheless, these defects are likely to be fixed.
3. Rarely used APIs are used by only highly experienced developers, and thus have a low rate of defect occurrences. Nevertheless, these defects are less likely to be fixed.
4. Coding style conventions are well known to experienced developers.
5. The frequency of a defect varies inversely with its visible impact, *i.e.*, defects causing crashes or hangs occur less often, while defects that have a delayed or cumulative effect such as memory leaks occur more frequently.

To assess these hypotheses, several challenges must be addressed. First, we need to mine the Linux code base to find the occurrences of defective code across the history of the different versions of the software project. Next, we need to identify the developer who introduced each defect. Finally, we need a means to evaluate the level of expertise of the developer at the time the defect was introduced. To address these issues, we use the Coccinelle source code matching tool to detect defects in the uses of memory management functions, focusing specifically on code that violates the usage protocol of the memory management API, code that does not satisfy the global Linux kernel coding style, and code that uses memory management functions inefficiently. We then use the Herodotos tool [7] to correlate the defect occurrences across the different versions. Finally, we use the git [4] source code manager used by Linux kernel developers for version control in order to extract information about developer expertise.

2. Linux Memory Management APIs

In user-level code, the most common memory management functions are `malloc` and `free`. These functions are, however, not available at the kernel level. Instead, the kernel provides a variety of memory management APIs, some generic and others more special-purpose. We first describe the commonalities in these APIs and then present four Linux kernel memory management APIs in detail.

2.1 Common behavior and potential defects

All of the memory management APIs defined in the Linux kernel impose essentially the same usage protocol, as shown in Figure 1 and illustrated by the following code:

```
x = alloc(size, flag);
if (x == NULL) { ... return -ENOMEM; }
...
free(x);
```

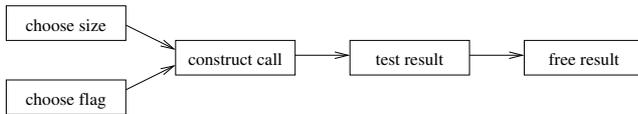


Figure 1. Usage protocol for Linux kernel memory management functions

Name	Description	Potential Impact
sizeof	Size argument expressed as the size of a type rather than the result of dereferencing the destination location.	coding style
noderef	Size argument expressed as the size of a pointer, rather than the pointed type.	buffer overflow
flag	Flag that allows locking when a lock is already held.	hang
cast	Cast on the result of an allocation function.	coding style
null test	Missing NULL test on the result of an allocation function (inverted when NULL test is not required).	crash
free	Missing deallocation of a pointer that is only accessible locally.	memory leak
memset	Explicit zeroing of the allocated memory rather than allocating using a zeroing allocation function.	inefficient
array alloc	Allocation of an array without using a dedicated array-allocating function.	buffer overflow

Table 1. Defect kinds studied

In this usage protocol, the allocation function takes two arguments: a size indicating the number of bytes to allocate and a flag indicating how the allocation may be carried out. The allocation function then returns either a pointer to the allocated region of memory or NULL, indicating that some failure has occurred. This result must thus be tested for NULL before using the allocated memory. Finally, the allocated memory should be freed when it is not useful any more, using the corresponding deallocation function.

Each step in this usage protocol introduces possibilities for defects. These defects may be violations of the Linux kernel coding style, that at best only have an impact on the maintainability of the code, or they may induce runtime errors, such as buffer overflows, hangs, crashes, or memory leaks. These defects are summarized in Table 1 and are described in detail below, for each step of the usage protocol:

Choose size The size argument to a memory allocation function is typically determined by the type of the location that stores the result of the call. One possibility is to express the size explicitly in terms of the type of this location (defect “sizeof”):

```
x = kmalloc(sizeof(struct foo),...);
```

The Linux kernel **coding style**, however, suggests to express the size as the size of the result of dereferencing the location itself:

```
x = kmalloc(sizeof(*x),...);
```

This strategy makes the size computation robust to changes in the type of **x**.

The approach preferred by the Linux kernel coding style, however, introduces the possibility of another kind of defect, in which the size is computed in terms of the pointer itself, instead of what it references, *e.g.*:

```
x = kmalloc(sizeof(x),...);
```

In this case, only a few bytes are allocated, leading to a likely subsequent **buffer overflow** (defect “noderef”).

Pattern	Memory Management API			
	Standard	Node	Cache	Bootmem
basic	4 240	52	264	105
array	363	N/A	N/A	N/A
zeroing	5 125	25	96	N/A
TOTAL	9 728	77	360	105

Table 2. Number of occurrences of the memory allocation functions in Linux 2.6.32 (released December 2009)

Choose flag The flag argument indicates some constraints on the memory allocation process. The most common values are **GFP_KERNEL**, indicating that the memory allocation process may sleep if adequate memory is not immediately available, and **GFP_ATOMIC**, indicating that such sleeping is not allowed, typically because the function is called in a context in which interrupts are turned off. Using **GFP_ATOMIC** where **GFP_KERNEL** could be used can cause the memory allocation to fail unnecessarily, while using **GFP_KERNEL** where **GFP_ATOMIC** is required can **hang** the kernel (defect “flag”).

Construct call The Linux kernel memory allocation functions have return type **void ***, while the location that stores the result typically has some other type, such as that of a pointer to some structure. Some programmers thus cast the result of the memory allocation to the destination type. Such a cast is, however, not required by the C standard and is against the Linux kernel **coding style** (defect “cast”).

Test result If the pointer resulting from a call to an allocation function is not immediately tested for being NULL, then the first dereference of a NULL result will normally **crash** the kernel (defect “null test”). This dereference may be far from the allocation site, making the problem difficult to diagnose.

Free result In Linux kernel code, a common pattern is for one function to allocate multiple resources. Each of these allocations may fail, in which case all of the previously allocated resources must be freed. Neglecting to free allocated memory in the case of such a failure causes a **memory leak** (defect “free”).

2.2 The specific APIs

The Linux kernel provides a number of different memory management APIs for different purposes. These differ in when they can be invoked and the features they provide. The APIs we consider are described below. Table 2 summarizes the usage frequency of their allocation functions.

Standard **kmalloc** is the standard memory allocation function in the Linux kernel, comparable to **malloc** at the user level. Two variants have recently been introduced. **kcalloc** was introduced in Linux 2.6.9 (October 2004) for allocating arrays. This function takes the number of elements and the size of each element as separate arguments, and protects against the case where their product overflows the size of an integer. The elements of the array are also initialized to 0. **kzalloc** was introduced in Linux 2.6.14 (October 2005) for allocating a region of memory in which all elements are initialized to 0 but that is not an array. Memory allocated with all of these functions is freed using **kfree**.

Node **kmalloc_node** targets NUMA architectures, where memory may be local to a processor or shared between a subset of the processors, and access to non-local memory is very expensive. This function thus takes an extra argument that specifies the node that should be associated with the allocated memory. **kzalloc_node** is **kmalloc_node**’s zeroing counterpart. Memory allocated with both of these functions is freed using **kfree**. Some other variants

of these functions exist that aid in debugging, but these are rarely used and we do not consider them further.

Cache `kmem_cache_alloc` allocates memory from a previously allocated memory cache. `kmem_cache_zalloc` is its zeroing counterpart. Memory allocated with both of these functions is freed using `kmem_cache_free`.

Boot These functions must be used to allocate memory during the booting process. They are analogous to `kzalloc` in that the memory is already zeroed. They furthermore always return a valid pointer, never NULL; in the case of an allocation error, the kernel panics. These functions do not take a flag argument. We consider only the allocation functions `alloc_bootmem`, `alloc_bootmem_low`, `alloc_bootmem_pages`, and `alloc_bootmem_low_pages`. Memory allocated with all of these functions is freed using `free_bootmem`.

In addition to the defect types outlined in Section 2.1, the different features of the memory management functions within each API introduce the possibility of using one of these functions in the wrong situation. In terms of defects, we consider cases where the zeroing and array allocating variants, if available, are not used and the corresponding code is inlined into the call site (defects “memset” and “array alloc”, respectively). For the `alloc_bootmem` functions, which do zero the memory and do not return NULL, we consider code that performs unnecessary zeroing and NULL test operations. These mistakes essentially only impact the efficiency of the code, but may also impact readability, and thus subsequent code maintenance.

3. Tools

To carry out our study, we use the following tools: 1) Coccinelle to find occurrences of defects in recent versions of the Linux source tree, 2) Herodotos to correlate these occurrences across multiple versions, and 3) git to identify the developer responsible for introducing each defect occurrence and to obtain information about the other patches submitted by this developer. Coccinelle is applicable to any software implemented in C. Herodotos is applicable to any software at all, as it is language-independent. Git can also be used to access developer information for any software, as long as it or some compatible tool has been used as the version control manager during the software’s development.

3.1 Coccinelle

Coccinelle is a tool for performing control-flow based pattern searches and transformations in C code [2, 6]. It provides a language, SmPL, for specifying searches and transformations and an engine for performing them. In this work, we use SmPL to create patterns representing defects and then use Coccinelle to search for these patterns across different versions of the Linux source tree. Patterns are expressed using a notation close to source code, but may contain *metavariables* to represent arbitrary sub-terms. A metavariable may occur multiple times, in which case all occurrences must match the same term. SmPL furthermore provides the operator “...”, which connects separate patterns that should be matched within a single control-flow path. This feature allows, for example, matching an execution path in which there is first a call to a memory allocation function and then a return with no intervening save or free of the allocated data, amounting to a memory leak. More details about Coccinelle, including numerous examples, are found in previous work [2, 5, 6].

3.2 Herodotos

To understand how defects have been introduced in the Linux kernel, we have to *correlate* the defect occurrences found by Coc-

Pattern	Memory Management API			
	Standard	Node	Cache	Bootmem
sizeof	30.64%	28.57%	N/A	16.19%
noderef	0	0	N/A	0.95%
flag	0.01%	0	N/A	N/A
cast	0.79%	2.60%	6.39%	29.52%
null test	1.04%	6.49%	3.06%	8.57%
free	0.10%	0	0.56%	0
memset	2.92%	1.92%	3.03%	1.90%
array alloc	3.32%	2.60%	0.28%	0.95%

Table 3. Comparison for Linux 2.6.32

cinelle across multiple versions. Indeed, the position of a defect may change across versions due to the addition or removal of other code in the same file. To correlate defect occurrences, we use the Herodotos tool [7]. Herodotos uses Unix diff to identify the differences in each affected file from one version to the next and thereby predicts the change in position of a defect. If a defect of the same type is reported in the predicted position in the next version, they are considered to be the same defect. Otherwise, the defect is considered to have been corrected. Herodotos also can be configured to produce a wide variety of graphs and statistics representing the defect history.

3.3 Git

Since version 2.6.12 (June 2005) Linux has used the `git` version control system [4]. Git maintains a graph representing the project history, including commits, forks and merges. Each of these operations is referenced by a SHA1 hash code. This hash code gives access to the changes in the repository and some related meta-information. For instance, Git registers the name and the email of the author and the committer of a change, short and long descriptions of the change, and the date on which the change was committed.

Git includes various options for browsing the commit history. In this work, we use git to trace the contributions of each developer. Starting from the earliest version in which Coccinelle finds a given defect, we use the *blame* option to find the name of the developer who has most recently edited the defective line in a prior commit. To evaluate the level of expertise of this developer, we then count the number of patches from this developer that were accepted prior to the one introducing the defect and the number of days between the developer’s first accepted patch and the defective one. We consider the level of expertise of the developer to be the product of these two quantities.

4. Assessment

We now assess the hypotheses presented in Section 1 for the memory management APIs. To support our assessment, we have collected various statistics. Table 3 presents the percentage of defect occurrences as compared to all occurrences of each kind of memory allocation function for Linux 2.6.32, which is the most recent version. Figure 2 presents the same information, but for all versions since Linux 2.6.12. The defect Flag is omitted, because its frequency is very close to 0. Figure 3 presents the average lifespan of these defects. Finally, Figure 4 presents the number of developers introducing each kind of defect (on the X axis) and their average level of expertise, calculated as described in Section 3.3.

Our assessment of each of the five hypotheses is as follows:

Defects are introduced by less experienced developers Figure 4 shows that in most cases, the expertise of the developers who introduce defects is indeed low, *i.e.*, they have participated in kernel development for only a short time and have submitted only a few patches. But for two defect types for the Node API and for one

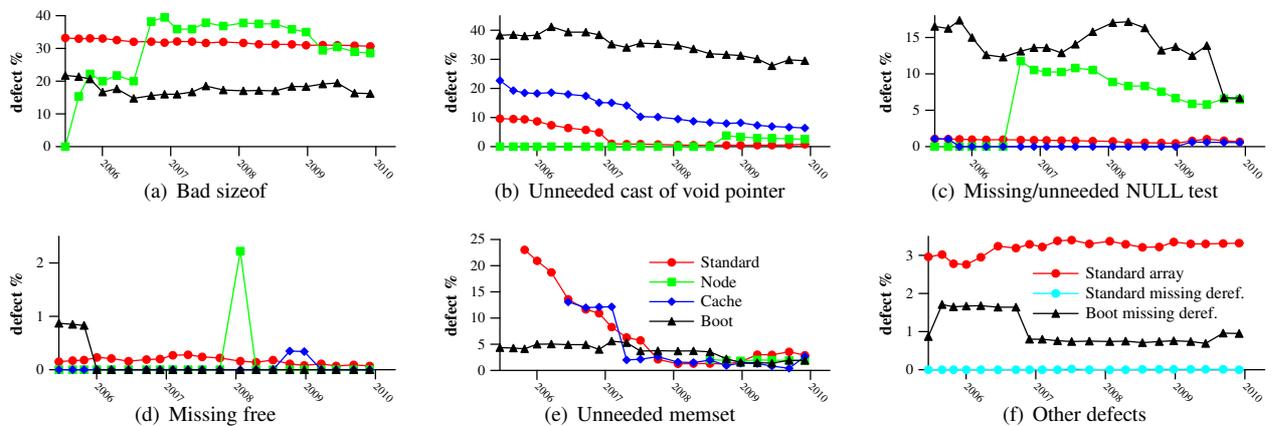


Figure 2. Defect ratio per uses for each defect kind

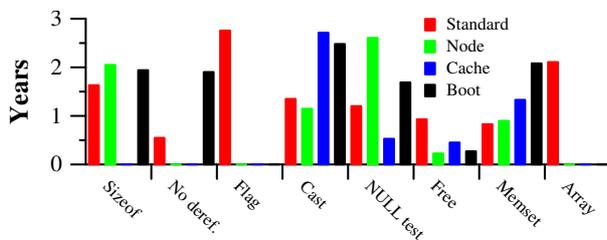


Figure 3. Average defect lifespan

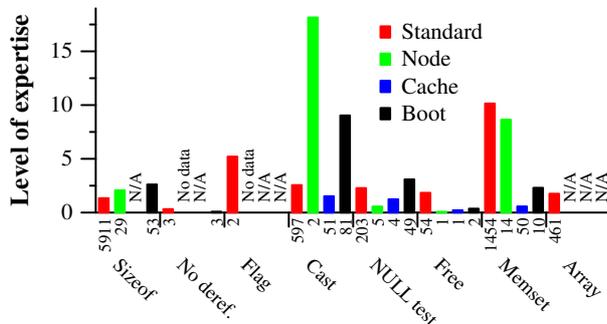


Figure 4. Developer expertise per allocator kind. No data means no defects of the given type. N/A means the defect type is irrelevant to the given API.

defect type for the Boot API, the average level of expertise is relatively high. We conjecture that these APIs are mostly used by experienced developers, and thus only experienced developers introduce the defects. For Node, at least, the number of developers in these cases is also very small.

Frequently used APIs have a high rate of defects, but these defects are quickly fixed The Standard API is used much more frequently than the others, and in some cases, notably sizeof and array, it has a higher defect frequency as well. But it also has a lower defect frequency for the remaining defect kinds. Some defect kinds show a slight or substantial decrease in their frequency for the Standard API, notably the use of memset rather than the zeroing function kcalloc. Such defects have an average lifespan of around one year, while the lifespan of the comparable defect for the Boot API is two years. Other defect kinds essentially remain steady, notably the non-

use of the array allocation function kcalloc, which has an average lifespan of two years. The frequency of this defect, however, is consistently very low.

Rarely used APIs have a low rate of defects, but these defects are rarely fixed The data in Figure 2 does not support the hypothesis of a low rate of defects, as it is often the rarely used APIs Node, Cache, and Boot that have the highest frequency of defects. For example, Boot has the highest rate of improper NULL tests. In this case, the API has the special property that a NULL test is not needed, and thus the results show that developers are not fully familiar with the features of this API. The defect rates are relatively stable, without the substantial drops over time seen in the case of the much more frequently used Standard. Defects also have a long lifespan, notably of typically two years or more for Boot.

Coding style conventions are well known to experienced developers This hypothesis is also not supported by Figure 2. The coding style defects Sizeof and Cast indeed have the highest frequency of all of the considered defects, and the frequencies are highest for Node and Boot, respectively. These are highly specialized APIs and thus are only likely to be used by experienced developers. It may be that such developers have a more specialized focus, and are thus not aware of these conventions. Or these APIs may be considered less often when doing coding style cleanups.

The frequency of a defect varies inversely with its impact Noderef is likely to lead to a kernel crash, as much less memory is allocated than intended. Flag can cause a kernel hang. A missing free can cause a memory leak. All of these defect kinds do indeed occur very infrequently, as shown by Table 3. Missing NULL tests, however, are relatively frequent, found in up to 10% of all occurrences for the Node API. We conjecture that the memory allocation functions do not return NULL very often, and thus the impact of the defect is not seen very often in practice.

5. Related work

The closest work to this one is our paper at AOSD 2010 [7], which presents Herodotos. The experiments in that paper have a larger scope, as they consider four open source projects and a wider range of defects. In this paper, on the other hand, we consider in more depth the defects in the use of a single type of API, in one software project. We have also added the assessment of developer expertise to the collected statistics. In future work, we will apply the analyses presented here to the wider set of examples considered in the AOSD paper.

Zhou and Davis assess the appropriateness of statistical models for describing the patterns of bug reports for eight open source projects [9]. They do not, however, distinguish between different defect types, nor do they study the level expertise of the developers who introduce the bugs. Chou *et al.* [3] do consider specific bug types in earlier versions of Linux, but do not study developer expertise.

Anvik and Murphy [1], and Schuler and Zimmermann [8] propose approaches to determine implementation expertise based on mining bug reports and code repositories. However, they determine who has expertise on a particular piece of code while we want to investigate the expertise of a developer who commits code containing a particular defect.

6. Conclusion

In this paper, we have studied the history of a set of defect types affecting a range of memory management APIs in the Linux kernel code, considering both the percentage of defects as compared to the total number of occurrences of each considered function and the expertise of developers that introduced these defects. Based on this information, we have assessed a collection of hypotheses, differentiating between widely used APIs and more specialized ones. The hypotheses that the developers who introduce defects have less experience, that defects in the use of widely used APIs are fixed quickly, and that defects in the use of rarely used APIs tend to linger are largely substantiated. On the other hand, the percentage of defect occurrences does not appear to be correlated to the frequency of use of the API, expert developers do not seem to be more aware of coding conventions than less expert ones, and the frequency of a defect is not always inversely related to its potential impact. More work will be required to assess these hypotheses in the context of Linux and the memory management APIs, as well as other software projects and APIs.

Availability

The data used in this paper are available at <http://www.diku.dk/~npalix/acp4is10/>.

References

- [1] J. Anvik and G. C. Murphy. Determining implementation expertise from bug reports. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, Minneapolis, USA, May 2007. IEEE Computer Society.
- [2] J. Brunel, D. Doligez, R. R. Hansen, J. Lawall, and G. Muller. A foundation for flow-based program matching using temporal logic and model checking. In *The 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 114–126, Savannah, GA, USA, Jan. 2009.
- [3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 73–88, Banff, Canada, Oct. 2001.
- [4] Git: The fast version control system. <http://git-scm.com/>.
- [5] J. L. Lawall, J. Brunel, R. R. Hansen, H. Stuart, G. Muller, and N. Palix. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. In *The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, (DSN 2009)*, pages 43–52, Estoril, Portugal, June 2009.
- [6] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, Mar. 2008.
- [7] N. Palix, J. Lawall, and G. Muller. Tracking code patterns over multiple software versions with Herodotos. In *Proc. of the ACM International Conference on Aspect-Oriented Software Development, AOSD'10*, Rennes and Saint Malo, France, Mar. 2010. To appear.
- [8] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 121–124, Leipzig, Germany, May 2008.
- [9] Y. Zhou and J. Davis. Open source software reliability model: an empirical approach. In *5-WOSSE: Proceedings of the fifth workshop on Open source software engineering*, pages 1–6, St. Louis, MO, USA, 2005. ACM.

A. SmPL files

A.1 alloc_size

```
virtual org

@ r depends on org disable sizeof_type_expr @
type T,Tl;
T *x;
expression n;
position p;
@@

(
  x@p = (Tl)\(kmalloc \| kzalloc \)(<+... sizeof(T) ...+>, ...)
|
  x@p = (Tl)kcalloc(n, <+... sizeof(T) ...+>, ...)
)

@script:python@
p << r.p;
x << r.x;
xtype << r.T;
@@

msg = "var: %s type: %s " % (x,xtype)
cocci.lib.org.print_safe_todo(p[0],msg)
cocci.include_match(False)
```

A.2 alloc_noderef

```
virtual org

@ r depends on org @
expression *x;
expression n;
position p;
@@

(
  \| (kmalloc \| kzalloc \)(<+... sizeof@p(x) ...+>, ...)
|
  kcalloc(n, <+... sizeof@p(x) ...+>, ...)
)

@bad_deref@
position r,p;
expression e;
constant c;
@@

\| ( sizeof@p(*e) \| sizeof@p(c) \| )

@script:python depends on !bad_deref@
p << r.p;
x << r.x;
@@

msg = "var: %s" % (x)
cocci.lib.org.print_safe_todo(p[0],msg)
cocci.include_match(False)
```

A.3 gfp_kernel

```
virtual org

@r depends on org@
expression E1;
position p;
@@

(
read_lock_irq
|
write_lock_irq
|
read_lock_irqsave
|
write_lock_irqsave
|
local_irq_save
|
spin_lock_irq
|
spin_lock_irqsave
)(E1,...);
... when != E1
    when any
    \((kmalloc\|kcalloc\|kzalloc)\)(...,<+... GFP_KERNEL@p ...+>)

@script:python@
p << r,p;
@@

msg="%s::%s" % (p[0].file, p[0].line)
cocclib.org.print_todo(p[0], msg)
cocci.include_match(False)
```

A.4 cast_alloc

```
virtual org
virtual patch

@ depends on patch && !org disable drop_cast @
type T;
@@

- (T *)
\((kcalloc \| kmalloc \| kzalloc \)(...)

@r depends on !patch && org disable drop_cast @
type T;
position p;
@@

(T@p)\((kmalloc\|kzalloc\|kcalloc)\)(...)

@script:python@
p << r,p;
t << r,T;
@@

msg = "%s" % (t)
cocclib.org.print_todo(p[0], msg)
cocci.include_match(False)
```

A.5 alloc_nulltest

```
virtual org

@r depends on org@
type T;
expression *x;
identifier f;
constant char *C;
position p1,p2;
@@
```

```
x@p1 = (T)\((kmalloc\|kcalloc\|kzalloc)\)(...);
... when != x == NULL
    when != x != NULL
    when != (x || ...)

(
kfree(x)
|
f(...,C,...,x,...)
|
f@p2(...,x,...)
|
x->f@p2
)

@script:python@
x << r,x;
p1 << r,p1;
p2 << r,p2;
@@

msg = "%s" % (x)
cocclib.org.print_todo(p1[0],msg)
cocclib.org.print_link(p2[0])
cocci.include_match(False)
```

A.6 kmalloc

```
virtual org

@r exists@
type T;
local idexpression x;
statement S;
expression E;
identifier f,l;
position p1,p2,p3;
expression *ptr != NULL;
@@

(
if ((x@p1 = (T)\((kmalloc\|kzalloc\|kcalloc)\)(...)) == NULL) S
|
x@p1 = (T)\((kmalloc\|kzalloc\|kcalloc)\)(...);
...
if (x == NULL) S
)
<... when != x
    when != if (...) { <+...x...+> }

(
goto@p3 l;
|
x->f = E
)
...>
(
return \((0\|<+...x...+>\|ptr\));
|
return@p2 ...;
)

@script:python@
x << r,x;
p1 << r,p1;
p2 << r,p2;
@@

cocclib.org.print_todo(p1[0], x)
for p in p2:
    cocclib.org.print_link(p)
cocci.include_match(False)
```