Finding Resource-Release Omission Faults in Linux

Suman Saha
LIP6-Regal
Suman.Saha@lip6.fr

Julia Lawall
DIKU, University of Copenhagen
julia@diku.dk

Gilles Muller
INRIA/LIP6-Regal
Gilles.Muller@lip6.fr

Abstract

The management of the releasing of allocated resources is a continual problem in ensuring the robustness of systems code. Missing resource-releasing operations lead to memory leaks and deadlocks. A number of approaches have been proposed to detect such problems, but they often have a high rate of false positives, or focus only on commonly used functions. In this paper we observe that resource-releasing operations are often found in error-handling code, and that the choice of resource-releasing operation may depend on the context in which it is to be used. We propose an approach to finding resource-release omission faults in C code that takes into account these issues. The approach is based on parsing C language. We use our approach to find over 100 faults in the `drivers` directory of Linux 2.6.34, with a false positive rate of only 16%, well below the 30% that has been found to be acceptable to developers.

1. Introduction

Omitting needed resource-releasing operations is a common problem in systems code, such as the Linux operating system, and can lead to memory leaks and deadlocks. A challenge in detecting resource-release omission faults is to identify the set of expected resource-releasing operations. One approach that has extensively been explored is to identify pairs of functions that first allocate and then release some resource, and then to scan the code base for occurrences of an allocation without a corresponding resource-releasing operation [2, 4, 6, 10]. For some kinds of resources, however, the choice of resource-releasing operation is context-sensitive, depending on the phase within a computation at which a release is needed or whether the current file has defined a specialized resource-releasing function. When the choice of resource-releasing operation depends on the context, fault-finding rules relating an allocation function to *e.g.* its most common resource-releasing operation will lead to false positives in cases where the context indicates that a different resource-releasing operation is required. On the other hand, ignoring cases where the set of possible resource-releasing functions appears to contain more than one element will lead to false negatives. We are not aware of existing approaches that have addressed these issues.

In this paper, we propose an approach to finding resource-releasing omission faults in the Linux operating system, building on the observation that many Linux functions need to deal with multiple possible failures, and thus appropriate error-handling code is often nearby. To exploit this nearby error-handling code, our approach first collects a list of calls to probable resource-releasing operations, from all error-handling code in a given function. It then searches for error-handling code that is missing calls to some of these operations. Finally, it uses some heuristics to determine whether these omissions are legitimate or they represent omission faults. By relying on a list of the resource-releasing operations actually used in the function's error-handling code, our approach naturally takes into account context-sensitive constraints on the choice of resource-releasing operations. Furthermore, unlike statistics-based approaches, our approach is independent of the usage frequency of a resource-releasing operation across the code base.

We have implemented our approach in OCaml, and have carried out preliminary experiments on the source code of the `drivers` directory of Linux 2.6.34. Our experiments find over 100 faults, with a low number of false positives. Our results furthermore show that content-sensitivity significantly reduces the number of false positives. Finally, to make it easier for the user to identify reports that represent probable faults, we propose a strategy for ranking the results, that takes into account context information. No false positives in our results receive a high rank according to this strategy.

The rest of this paper is organized as follows. Section 2 presents two examples that motivate our work. Section 3 then presents our fault-finding algorithm. Section 4 presents a preliminary evaluation of this algorithm. Finally, Section 5 presents related work and Section 6 concludes.

2. Motivating Examples

We first illustrate resource-release omission faults, using two examples from the Linux 2.6.34 kernel. These examples, as well as the other examples presented in the paper, have been found using the tool we have developed based on our approach.

The first example, shown in Figure 1, relates to the use of the functions `lock_kernel` and `unlock_kernel`. This example is typical of the use of locking functions. On line 3, the entire kernel is locked by calling `lock_kernel`. The error-handling code on lines 14-15 jumps to the label `bail`, which correctly calls `unlock_kernel` on line 23 to release the kernel lock. However, the preceding error-handling code, on lines 6 and 8, do not perform this operation. Both omissions will lead to a deadlock if an error is detected on line 5 or 7.

The second example, shown in Figure 2, relates to the use of a highly specialized memory allocation function, `wl1251_alloc_hw`. This example is typical of the use of allocation functions. On line 3, a resource is allocated by calling `wl1251_alloc_hw`, and the result is stored in the variable `hw`. The error-handling code on lines 8-9 and the error-handling code on lines 18-19 both jump to the label `out_free`, which calls `ieee80211_free_hw` on line 25 to release `hw`. However, the error-handling code in the middle of the function, on lines 13-14, does not have any operation that releases `hw`. This omission may lead to a memory leak. We note that the function `wl1251_alloc_hw` is only used twice in the Linux kernel, once

PLOS '11 October 23, 2011, Cascais, Portugal.
Copyright © 2011 ACM 978-1-4503-0979-0/11/10...$10.00

```
static int
hiddev_open(struct inode *inode, struct file *file) {
    lock_kernel();
    ...
    if (i >= HIDDEV_MINORS || i < 0 || !hiddev_table[i])
        return -ENODEV;
    if (!(list = kzalloc(sizeof(struct hiddev_list), GFP_KERNEL)))
        return -ENOMEM;
    ...
    if (list->hiddev->exist) {
        if (!list->hiddev->open++) {
            ...
            if (res < 0) {
                res = -EIO;
                goto bail;
            }
        }
    } ...
    unlock_kernel();
    return 0;
bail:
    ...
    unlock_kernel();
    return res;
}
```

Figure 1. Example of an omission fault that may lead to a deadlock (Linux-2.6.34/drivers/hid/usbhid/hiddev.c)

```
static int __devinit wl1251_spi_probe(struct spi_device *spi) {
  ...
  hw = wl1251_alloc_hw();
  if (IS_ERR(hw))
    return PTR_ERR(hw);
  ...
  if (ret < 0) {
    ...
    goto out_free;
  }
  ...
  if (!wl->set_power) {
    ...
    return -ENODEV;
  }
  ...
  if (ret < 0) {
    ...
    goto out_free;
  }
  ...
  return 0;
  ...
  out_free:
    ieee80211_free_hw(hw);
    return ret;
}
```

Figure 2. Example of an omission fault that may cause a memory leak (Linux-2.6.34/drivers/net/wireless/wl12xx/wl1251_spi.c)

with this resource-releasing operation and once without. Therefore, statistics-based approaches [2, 6] are not likely to detect this fault.

3. Detecting Resource-Release Omission Faults

We have designed an algorithm to detect resource-release omission faults based on the information available in the current function. This algorithm first globally analyzes a function's error handling code to identify its resource-releasing operations, and then finds omissions of these operations in the individual blocks of error-handling code. Concretely, the algorithm performs the following steps:

X	u(a)	x(d)		
Y	u(a)	v(b)	w()	x(d)
Z	u(a)	v(b)	w()	y()

Figure 3. A schematic representation of a function's error-handling code

1. Collect the complete set of resource-releasing operations used by a given function in its error-handling code.
2. Compare each block of error-handling code within the function with the set of collected resource-releasing operations to determine whether any resource-releasing operations are omitted.
3. Analyze the omitted resource-releasing operations using heuristics to determine whether they represent omission faults.

We now describe these steps in more detail.

3.1 Identify resource-releasing operations

The algorithm first analyzes the error-handling code in a given function to identify its probable resource-releasing operations. We consider error-handling code to be any conditional branch that ends with a return or goto, following Linux coding conventions. Within such code, we consider a probable resource-releasing operation to be a call to a function that has at most one pointer-typed argument and that does not have any explicit string arguments, as string arguments are typical of debugging code. The pointer-typed argument, if any, is considered to be the *released resource*. The identified set of operations is stored in the *function list*.

As a running example, consider a function containing three blocks of error-handling code X, Y, and Z, shown schematically in Figure 3. In this case, the function list is {*u(a)*, *v(b)*, *w()*, *x(d)*, *y()*}.

3.2 Finding omitted resource-releasing operations

For each block of error-handling code, the algorithm then calculates the set of omitted resource-releasing operations, which is the difference between the set of resource-releasing operations in the function list and the set of resource-releasing operations in the error-handling code. We refer to this difference as the *candidate set*.

Continuing with the example of Figure 3, suppose that we are analyzing the block of error-handling code labelled X, indicated in grey. In this case, the difference between the function list, {*u(a)*, *v(b)*, *w()*, *x(d)*, *y()*} and the resource-releasing operations in X, {*u(a)*, *x(d)*}, is {*v(b)*,*w()*, *y()*}, which is the candidate set for X. The remainder of the algorithm then analyzes this candidate set to determine which of its elements represent omission faults.

3.3 Finding resource-releasing omission faults

The third step analyzes each element of a candidate set to determine whether omitting the operation in the current error-handling code is legitimate or represents a probable omission fault, taking into account the context in which the omission occurs. The algorithm considers the following heuristics to identify cases in which a given resource-releasing operation is *not* actually needed:

1. The variables used to describe the released resource are undefined or have a different definition at the point of the error-handling code than at the point of the occurrence in the code of the element of the candidate set.
2. The released resource is returned by the error-handling code.
3. The resource is released in an alternate way.

We now describe these cases in more detail.

Definitions of variables The algorithm uses a dataflow analysis to identify the reaching definitions for the variables used to describe the released resource, both at the point of the omission and at the point of the occurrences of the associated element of the candidate set in the source code. If at the point of the omission some of these variables are not yet defined or contain different values than at the occurrence of the element of the candidate set, then there is no evidence that the given block of error-handling code needs the omitted resource-releasing operation. This dataflow analysis takes into account not only explicit definitions, but also information that can be inferred from test expressions. For example, in Figure 2, at line 3, a resource is allocated by calling `wl1251_alloc_hw`, and the result is stored in the variable `hw`. From the analysis of the complete function, we find that the resource-releasing function for `hw` is `ieee80211_free_hw`, which is not found in the error-handling code on line 5. However, the associated conditional tests whether `hw` is an error value, and thus in the error-handling code we consider that the definition of `hw` is at the point of the `if` test, not the call to `wl1251_alloc_hw`. Therefore, there is no need to release `hw` in the error-handling code.

Finally, the algorithm treats specially the case of a function that has no arguments, and thus has no explicit released resource. In Figure 1, the error-handling code on lines 6 and 8 is missing the resource-releasing operation `unlock_kernel`. Since this operation does not have any arguments, the algorithm assumes that the missing operation is a probable omission fault and further analyzes this operation.

Return of the resource The algorithm checks the return statement of the error-handling code to determine whether it uses the released resource. If the resource is returned, then it should not be released. For example, in Figure 4, a resource is allocated and stored into the variable `bh` on line 1. The code on line 7 returns `bh` and thus it is not appropriate to release this value.

```
 1   bh = udf_tread(sb, block);
 2   if (...) {
 3       ...
 4       goto error_out;
 5   }
 6   ...
 7   if (...) return bh;
 8   ...
 9   error_out:
10       brelse(bh);
11       return NULL;
```

Figure 4. Example of returning a resource (Linux-2.6.34/drivers/input/xen-kbdfront.c)

Alternate release of the resource Rather than releasing a resource in the error-handling code using the identified element of the candidate set, the resource can be released earlier or using a different function, depending on the context. Figure 5 illustrates four ways in which this can occur. First, the missing resource-releasing operations may appear in the execution path prior to the error-handling code. For example, in scenario 1, line 9 calls `kfree(attr)` and thus this call is not needed in the subsequent error-handling code on lines 12-13. Second, a resource can be released by some operation other than the omitted resource-releasing operation either before or inside the error-handling code. In scenario 2, the error-handling code on lines 2-4 uses `kfree(fw)` to release `fw`. However, the subsequent error-handling code, on lines 7-8, uses `free_fw` to release `fw`. This operation is also used to release `fw` in line 11 prior to the error-handling code on line 14, and thus no call to `kfree(fw)` is needed in either case. Third, a resource can be referenced via another pointer. Any resource-releasing operation on that pointer may release the resource as well. In scenario 3, in line 6, the variable `pr` is stored in the structure `device`. The subsequent error-handling code on lines 9-10 releases the resource using `device` instead of `pr` on line 14. In the final case, an intervening function takes the resource as an argument to perform a specific task but is not able to perform this task successfully and releases the resource. No subsequent resource-releasing operation is needed. This case is illustrated by scenario 4, where the resource `command` is passed to the function `usb_bulk_msg` on line 4. On failure, `usb_bulk_msg` releases `command`. Therefore, the subsequent error-handling code on lines 7-8 does not need to release this resource.

The above-mentioned alternate ways of releasing resources can be identified by analyzing the execution path from the allocation of the resource to the return statement of the error-handling code, that omits the resource-releasing operation. The use of alternate operations to release a resource in scenarios 2 and 3 can be identified by interprocedural analysis and alias analysis, respectively. In the case of scenario 4, the approach assumes that the intervening function releases the resource for the given error-handling code if the subsequent error-handling code does contain the omitted resource-releasing operations.

4. Evaluation

We have implemented a tool based on our omission-fault finding algorithm. This tool consists of around 1200 lines of OCaml code, not including the code for the C code parser and C abstract syntax,

```
 1   attr = kmalloc(...);
 2   ...
 3   if (ret) {
 4     ...
 5     kfree(attr);
 6     return ERR_PTR(ret);
 7   }
 8   ...
 9   kfree(attr);
10   pool = kmalloc(...);
11   if (!pool) {
12     ...
13     return ERR_PTR(-ENOMEM);
14   }
```
a) Scenario 1

```
 1   if (...) {
 2     ...
 3     kfree(fw);
 4     return -ENOMEM;
 5   }
 6   if (...) {
 7     free_fw(fw);
 8     return -EFAULT;
 9   }
10   ...
11   free_fw(fw);
12   ...
13     if (...)
14       return err;
```
b) Scenario 2

```
 1   if (...) {
 2     kfree(pr);
 3     return ;
 4   }
 5   ...
 6   device->driver_data = pr;
 7   ...
 8   if (...) {
 9     ...
10     goto err_remove_fs;
11   }
12   ...
13   err_remove_fs:
14     acpi_processor_remove_fs
15                       (device);
```
c) Scenario 3

```
 1   if (!result)
 2     goto no_result_buffer;
 3   ...
 4   ret = usb_bulk_msg(...,
 5                 command, ...)
 6   if (ret) {
 7     ...
 8     goto no_firmware;
 9   }
10   ...
11   no_firmware:
12     ...
13     return -ENODEV;
14   ...
15   no_result_buffer:
16     kfree(command);
```
d) Scenario 4

Figure 5. Ways of releasing a resource (Linux-2.6.34).
a. ib_create_fmr_pool in drivers/infiniband/core/fmr_pool.c
b. vx_hwdep_dsp_load in sound/drivers/vx/vx_hwdep.c
c. acpi_processor_add in drivers/acpi/processor_driver.c
d. whiteheat_firmware_attach in drivers/usb/serial/whiteheat.c

which we have borrowed from the implementation of Coccinelle [8]. We have evaluated our tool on the Linux 2.6.34 kernel, released in May 2010, focusing in these preliminary experiments only on the `drivers` directory. All of our experiments were carried out on one core of an 8-core 3GHz machine with 16GB memory.

We first present our overall results on the `drivers` directory, assess these results in light of data-mining strategies, and quantify the importance of context-sensitivity to our results. Finally, we propose and evaluate a ranking strategy, that highlights the reports that most probably represent actual faults.

Results As shown in Table 1, the tool generates a total of 126 reports for the `drivers` directory, relating to code within 78 functions. We manually investigated all of these reports and found that 103 of the reports represent actual faults, which come from 65 different functions. Our assessment of the faults is mainly based on our own understanding of the code. We have also submitted patches based on some of the reports to the maintainers of the affected code, and these patches have been accepted.[1]

The impact of these faults depends on the kind of code in which they occur. 63% of the faults are in driver initialization functions, and thus can only occur at most once in normal usage of the device. All of these faults are memory leaks. The remainder of the faults occur in driver open, ioctl, or read/write functions. Most of these faults are again memory leaks, but some can lead to deadlocks. A few faults are only related to debugging code. Open and ioctl functions typically are invoked only a small number of times by each application that uses the device, but could be invoked repeatedly in the case of a denial of service attack. Read/write functions are typically invoked many times in the normal usage of the device. Faults in this kind of code can thus have a high impact.

We also found 20 false positives within 10 functions, *i.e.*, 16% of the total number of reports. A false positive rate of under 30% has been found to be acceptable in practice [1], and indeed the absolute number of false positives is not high. We are still investigating 3 reports, due to insufficient expertise in the associated APIs.

Comparison with data-mining strategies Data-mining based approaches to identifying pairs of related allocation and resource-releasing operations and other similar protocols typically use thresholds defined in terms of *support* (the number of occurrences of the protocol) and *confidence* (the number of occurrences of some relevant information that match an expected pattern vs. the number that do not) to reduce the number of false positives. The data-mining-based protocol-finding tool PR-Miner [6], for example, only reports on sets of functions that occur together at least 15 times, with a confidence of at least 90%.[2] We have evaluated our identified faults with respect to these parameters, as shown in Figure 6. The $\times$s and circles represent the 30 pairs of allocation and resource-releasing operations associated with our 103 identified faults, and the y-axis indicates support, while the x-axis indicates confidence. The figure shows that only two pairs, marked as $\times$, have support greater than 15 and confidence greater than 90%. These two pairs are associated with only 10 of the 103 faults found by our approach. Thus, most of the faults we have found would be missed by approaches using these thresholds. On the other hand, reducing the support or confidence thresholds used by data-mining-based approaches could drastically increase their number of false positives.

Context sensitivity We have previously observed that for some kinds of resources, the choice of resource-releasing operation may depend on the context in which releasing the resource is needed. Searching for one type of resource-releasing operation when the context indicates that another one should be used results in false positives. In order to reduce the number of false positives, the tool must be aware of the context in which error-handling code appears.

Scenarios 2 to 4, illustrated in Figure 5 to motivate the third step of our algorithm, involve issues of context sensitivity. We analyze how often the strategies derived from these scenarios discard reports, avoiding false positives. Our tool found 331 resource allocations for which at least one resource-releasing operation seems to be omitted. We refer to these as *candidate resources*. In scenario 2, a resource-releasing operation seems to be omitted, but another one is used instead. Table 2 shows the number of the candidate resources that are associated with only one kind of resource-releasing operation in a given function, and the number that are associated with more than one in the given function. 22.4% of the candidate resources are released within a single function by two different operations while 4.2% of the candidate resources are released by three different operations. In scenario 3, a resource is accessible via another pointer, and is released via this pointer. Table 3 shows that 5.2% of the candidate resources are released in this manner. Finally, in scenario 4, a resource is released by an intervening function when this function fails on some task. Table 4 shows that 15.7% of the candidate resources are released in this manner. Moreover, 14.8% are released by a call to some other function that is defined in the same file (Table 5). Our tool takes these issues into account and does not generate reports in these cases.

[1] Examples, from linux-next [7]: 7febe2be36035e5c75128e8cc3baeb1f30fa2bc4, b722dbf176b67c75fe0f5a6b1b31f5ea8aa6117d

[2] Concretely, we compute the confidence as the number of times the allocation function occurs with the given resource-releasing operation as compared to the total number of times the allocation function occurs.

	Total reports	Faults	FP	TODO
Fault count	126	103	20	3
Function count	78	65	10	3

Table 1. Total number of Faults, False Positives (FP), and TODO, and the number of different functions that contain them

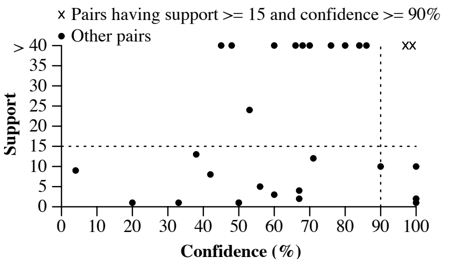

Figure 6. Support and confidence associated with the functions found in the faults reported by our algorithm. The dotted lines mark the thresholds of support 15 and confidence 90%.

resource released by one operation	resource released by two operations	resource released by three operations	Total
243(73.4%)	74(22.4%)	14(4.2%)	331

Table 2. Number of different operations used to release a resource

released via another pointer	not released via another pointer	Total
17 (5.2%)	314 (94.8%)	331

Table 3. Number of resources that are released via other pointers and that are only released directly

released via other operations	not released via other operations	Total
52 (15.7%)	279 (84.3%)	331

Table 4. Number of resources that are released via other operations and that are only released directly

released via a local function	not released via a local function	Total
49(14.8%)	282(85.2%)	331

Table 5. Number of resources that are released by calling another function defined in the same file, and number of resources that are only released by non local function calls

Ranking the fault reports As our algorithm does report some false positives, we have experimented with a ranking strategy to draw the user's attention to the most probable fault instances.

We rank the reports as *high* and *low*. A report is ranked *high* if the omitted function appears in both a preceding block of error-handling code and a following block of error-handling code. In this case, we have high confidence that the resource has both been allocated (based on the preceding error-handling code) and has not yet been released (based on the following error-handling code). Other faults are ranked *low*. For example, in Figure 2, the tool found an omission of `ieee80211_free_hw` in the error-handling code on lines 13-14. The omitted resource-releasing function appears in both the preceding error-handling code on lines 8-9 and the following error-handling on lines 18-19, and thus the fault receives rank *high*. On the other hand, in Figure 1, the tool found omissions of `unlock_kernel` in the blocks of error-handling on lines 6 and 8. There is no preceding error-handling code that contains `unlock_kernel`, and thus these faults receive rank *low*.

Table 6 shows the total number of *high* and *low* ranked reports. 22 reports are ranked *high* and 104 are ranked *low*. No false positives have *high* rank. Still, many actual faults are given a *low* rank. We envisage that the user may want to study the high ranked reports first, to get an overall understanding of the problem of resource-release omissions, and then consider the low ranked reports, taking into account the acquired intuitions.

	Total reports	Faults	FP	TODO
High	22	21	0	1
Low	104	82	20	2

Table 6. Ranking reports

5. Related Work

A number of approaches have been proposed to detect the omission of certain operations in systems code. One well known technique is to use some form of data mining to extract implicit programming rules from the software source code and then to use static analysis to detect faults based on those programing rules. Engler et al. [2] and Li et al. [6] both propose variations of this approach. Kremenek et al. [3] present a framework based on factor graphs for automatically inferring specifications directly from programs. Ramanathan et al. [9] integrate mining within a path-sensitive dataflow framework to define potential preconditions of a procedure. Le Goues and Weimer [5] integrate extra information about nonfunctional code characteristics, such as churn and author expertise. Our approach is completely different, in that it relies entirely on local information rather than a global analysis of the software. As compared to other approaches, our approach may result in false negatives, when error-handling code is omitted and there is no relevant code nearby. But, as we have shown, it can also find faults in the use of protocols that are likely to be overlooked or given a low rank by other approaches.

In previous work, we have considered how to restructure Linux error-handling code to introduce the use of `goto`s, which jump to a cascade of resource-releasing operations at the end of the current function. This structure for error-handling code is suggested by the Linux coding style guidelines. A number of the faults found in this work stem from cases where direct returns are mixed in with error-handling code using `goto`s, and the direct returns omit some required resource-releasing operations.

6. Conclusion

In this paper, we have provided an approach to finding resource-release omission faults that takes context information into account. Our proposed approach finds a number of probable faults in Linux kernel code with only a small number of false positives. We have shown that taking context information into account significantly reduces the number of false positives. Moreover, the approach ranks the generated reports to draw the user's attention to the more probable fault instances.

Our approach only detects the omission of resource-releasing operations, but does not fix these faults. In future work, we will extend the algorithm to consider this issue. The proposed approach keeps track of NULL values, but does not fully use this information, *e.g.*, to find probable NULL pointer dereferences. We will consider the benefit of such checks in future work. Finally, we will consider the applicability of our approach to other kinds of systems software.

References

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, Feb. 2010.

[2] 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.

[3] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *OSDI*, pages 161–176, Nov. 2006.

[4] 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 *DSN*, pages 43–52, Estoril, Portugal, June 2009.

[5] C. Le Goues and W. Weimer. Specification mining with few false positives. In *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 292–306, York, UK, Mar. 2009.

[6] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*, pages 306–315, Lisbon, Portugal, Sept. 2005.

[7] Linux. Linux-next gitweb, 2011. `http://git.kernel.org/?p=linux/kernel/git/next/linux-next.git;a=summary`.

[8] 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.

[9] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *ICSE*, pages 240–250, Minneapolis, MN, USA, May 2007.

[10] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 461–476, Edinburgh, UK, Apr. 2005.