Faults in Linux: Ten Years Later

Nicolas Palix *
DIKU, University of Copenhagen
npalix@diku.dk

Gaël Thomas
INRIA Regal/LIP6
Gael.Thomas@lip6.fr

Suman Saha
INRIA Regal/LIP6
Suman.Saha@lip6.fr

Christophe Calvès
INRIA Regal/LIP6
Christophe.Calves@lip6.fr

Julia Lawall
INRIA Regal/LIP6,
University of Copenhagen
julia@diku.dk

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

Abstract

In 2001, Chou *et al.* published a study of faults found by applying a static analyzer to Linux versions 1.0 through 2.4.1. A major result of their work was that the `drivers` directory contained up to 7 times more of certain kinds of faults than other directories. This result inspired a number of development and research efforts on improving the reliability of driver code. Today Linux is used in a much wider range of environments, provides a much wider range of services, and has adopted a new development and release model. What has been the impact of these changes on code quality? Are drivers still a major problem?

To answer these questions, we have transported the experiments of Chou *et al.* to Linux versions 2.6.0 to 2.6.33, released between late 2003 and early 2010. We find that Linux has more than doubled in size during this period, but that the number of faults per line of code has been decreasing. And, even though `drivers` still accounts for a large part of the kernel code and contains the most faults, its fault rate is now below that of other directories, such as `arch` (HAL) and `fs` (file systems). These results can guide further development and research efforts. To enable others to continually update these results as Linux evolves, we define our experimental protocol and make our checkers and results available in a public archive.

Categories and Subject Descriptors D.4 [*Operating Systems*]: Reliability

General Terms Reliability, Experimentation, Measurement

Keywords Linux, fault-finding tools

1. Introduction

The Linux operating system is widely used, on platforms ranging from embedded systems, to personal computers, to servers and supercomputers. As an operating system (OS) with a traditional monolithic kernel, Linux is responsible for the security and integrity of the interactions between software and the underlying hardware. Therefore, its correctness is essential. Linux also has a large developer base, as it is open source, and is rapidly evolving. Thus, it is critical to be able to continually assess and control the quality of its code.

Almost 10 years ago, in 2001, Chou *et al.* published a study of the distribution and lifetime of certain kinds of faults[1] in OS code, focusing mostly on the x86 code in the Linux kernel [5], in versions up to 2.4.1. The ability to collect fault information automatically from such a large code base was revolutionary at the time, and this work has been highly influential. Indeed, their study has been cited over 360 times, according to Google Scholar, and has been followed by the development of a whole series of strategies for automatically finding faults in systems code [1, 16, 33, 36, 38]. The statistics reported by Chou *et al.* have been used for a variety of purposes, including providing evidence that driver code is unreliable [12, 35], and evidence that certain OS subsystems are more reliable than others [10].

Linux, however, has changed substantially since 2001, and thus it is worth examining the continued relevance of Chou *et al.*'s results. In 2001, Linux was a relatively young OS, having first been released only 10 years earlier, and was primarily used by specialists. Today, well-supported Linux distributions are available, targeting servers, embedded systems, and the general public [11, 37]. Linux code is changing rapidly, and only 30% of the Linux 2.6.33 code is more than five years old [7, 8]. Linux now supports 23 architectures, up from 13 in Linux 2001, and the developer base has grown commensurately. The development model has also changed substantially. Until Linux 2.6.0, which was released at the end of 2003, Linux releases were split into stable versions, which were installed by users, and development versions, which accommodated new features. Since Linux 2.6.0 this distinction has disappeared; releases in the 2.6 series occur every three months, and new features are made available whenever they are ready. Finally, a number of fault finding tools have been developed that target Linux code. Patches are regularly submitted for faults found using checkpatch [4], Coccinelle [24], Coverity [9], smatch [36] and sparse [31].

In this paper, we transport the experiments of Chou *et al.* to the versions of Linux 2.6, in order to reevaluate their results in

* Now at Joseph Fourier University and INRIA Sardes, Grenoble, nicolas.palix@inria.fr

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

[1] Chou *et al.* used the terminology "errors." In the software dependability literature [13], however, this term is reserved for incorrect states that occur during execution, rather than faults in the source code, as were investigated by Chou *et al.* and are investigated here.

the context of the current state of Linux development. Because Chou *et al.*'s fault finding tool and checkers were not released, and their results were released on a local web site but are no longer available, it is impossible to exactly reproduce their results on recent versions of the Linux kernel.[2] To provide a baseline that can be more easily updated as new versions are released, we propose an experimental protocol based on the open source tools Coccinelle [24], for automatically finding faults in source code, and Herodotos [25], for tracking these faults across multiple versions of a software project. We validate this protocol by replicating Chou *et al.*'s experiments as closely as possible on Linux 2.4.1 and then apply our protocol to all versions of Linux 2.6. To ensure the perenity of our work, our tools and results are available in a public archival repository [27].

The contributions of our work are as follows:

- We provide a repeatable methodology for finding faults in Linux code, based on open source tools, and a publicly available archive containing our complete results.
- We show that the faults kinds considered 10 years ago by Chou *et al.* are still relevant, because such faults are still being introduced and fixed, in both new and existing files. These fault kinds vary in their impact, but we have seen many patches for all of these kinds of faults submitted to the Linux kernel mailing list [17] and have not seen any receive the response that the fault was too trivial to fix.
- We show that while the rate of introduction of such faults continues to rise, the rate of their elimination is rising slightly faster, resulting in a kernel that is becoming more reliable with respect to these kinds of faults. This is in contrast with previous results for earlier versions of Linux which found that the number of faults was rising with the code size.
- We show that the rate of the considered fault kinds is falling in the `drivers` directory, which suggests that the work of Chou *et al.* and others has succeeded in directing attention to driver code. The directories `arch` (HAL) and `fs` (file systems) now show a higher fault rate, and thus it may be worthwhile to direct research efforts to the problems of such code.
- We show that the lifespan of faults in Linux 2.6 is comparable to that observed for previous versions, at slightly under 2 years. Nevertheless, we find that fault kinds that are more likely to have a visible impact during execution have a much shorter average lifespan, of as little as one year.
- Although fault-finding tools are now being used regularly in Linux development, they seem to have only had a small impact on the kinds of faults we consider. Research is thus needed on how such tools can be better integrated into the development process. Our experimental protocol exploits previously collected information about false positives, reducing one of the burdens of tool use, but we propose that approaches are also needed to automate the fixing of faults, and not just the fault finding process.

The rest of this paper is organized as follows. Section 2 briefly presents our experimental protocol based on Coccinelle and Herodotos. Section 3 gives some background on the evolution of Linux. Section 4 establishes a baseline for our results, by comparing our results for Linux 2.4.1 with those of Chou *et al.* Section 5 presents a study of Linux 2.6, considering the kinds of code that contain faults, the distribution of faults across Linux code, the lifetime of faults, and effect of the use of fault-finding tools. Section 6 presents some limitations of our approach. Finally, Section 7 describes related work and Section 8 presents our conclusions.

[2] Chou *et al.*'s work did lead to the development of the commercial tool Coverity, but using it requires signing an agreement not to publish information about its results (`http://scan.coverity.com/policy.html#license`).

2. Experimental protocol

In laboratory sciences there is a notion of experimental protocol, giving all of the information required to reproduce an experiment. For a study of faults in operating systems code, such a protocol should include the definition of the fault finding tools and checkers, as well as the strategies for identifying false positives, as each of these elements substantially affects the results. In this section, we first present our checkers, both for finding faults and for assessing the fault rate, and then describe the tools that we have used in the fault finding and validation process. All of our results, as well as the scripts used and the source code of the Coccinelle and Herodotos tools are available on the open access archive HAL [28] and on the project website [29].

2.1 Fault finding checkers

Based on the descriptions of Chou *et al.*, we have implemented our interpretations of their **Block**, **Null**, **Var Inull**, **Range**, **Lock**, **Intr**, **LockIntr**, **Float**, and **Size** checkers. We omit the **Real** checker, related to the misuse of `realloc`, and the **Param** checker, related to dereferences of user-level pointers, as in both cases, we did not have enough information to define checkers that found any faults. In the description of each checker, the initial citation in italics is the description provided by Chou *et al.*

Block *"To avoid deadlock, do not call blocking functions with interrupts disabled or a spinlock held."* Implementing this checker requires knowing the set of functions that may block, the set of functions that disable interrupts, and the set of functions that take spinlocks. These functions vary across Linux versions. Identifying them precisely requires a full interprocedural analysis of the Linux kernel source code, including a precise alias analysis, as these operations may be performed via function pointers. To our knowledge, Chou *et al.*'s tool `xgcc` did not provide these features in 2001, and thus we assume that these functions were identified based on their examination of the source code and possibly heuristics for collecting functions with particular properties. We take the same approach, but add a simple interprocedural analysis, based on the iterative computation of a transitive closure though the call graph. This iterative analysis implies that our **Block** checker automatically takes into account the new blocking functions that are added in each version.

To identify blocking functions, we consider two kinds of functions as the starting point of our interprocedural analysis. First, we observe that basic memory allocation functions, such as the kernel function `kmalloc`, often take as argument the constant `GFP_KERNEL` when they are allowed to block until a page becomes available. Thus, we consider that a function that contains a call with `GFP_KERNEL` as an argument may block. Second, we observe that blocking is directly caused by calling the function `schedule`. Given this initial list of blocking functions, we then iteratively augment the list with the names of functions that call functions already in the list without first explicitly releasing locks or turning on interrupts, until reaching a fixed point.

To identify functions that turn off interrupts and take locks, we rely on our knowledge of a set of commonly used functions for these purposes, listed in the appendix. However, we observe that blocking with interrupts turned off is not necessarily a fault, and indeed core Linux scheduling functions, such as `interruptible_sleep_on`, call `schedule` with interrupts turned off. Thus, we only consider the case where a blocking function is called while holding a spinlock, which is always a fault. We refer to this checker as **BlockLock** to highlight the different design.

Null *"Check potentially `NULL` pointers returned from routines."* To collect a list of the functions that may return `NULL`, we follow the same iterative strategy as for the **Block** checker, with the starting point of the iteration being the set of functions that explicitly return `NULL`. Once the transitive closure is computed, we check the call sites of each collected function to determine whether the returned value is compared to `NULL` before it is used.

Var *"Do not allocate large stack variables (> 1K) on the fixed-size kernel stack."* Our checker looks for local variables that are declared as large arrays, *e.g.*, 1,024 or more elements for a `char` array. Many array declarations express the size of the array using macros, or even local variables, rather than explicit constants. Because Coccinelle does not expand macros, and indeed some macros may have multiple definitions across different architectures, we consider only array declarations where the size is expressed as an explicit constant.

Inull *"Do not make inconsistent assumptions about whether a pointer is `NULL`."* We distinguish two cases: **IsNull**, where a null test on a pointer is followed by a dereference of the pointer, and **NullRef**, where a dereference of a pointer is followed by a null test on the pointer. The former is always an error, while the latter may be an error or may simply indicate overly cautious code, if the pointer can never be `NULL`. Still, at least one **NullRef** fault has been shown to allow an attacker to obtain root access [34].

Range *"Always check bounds of array indices and loop bounds derived from user data."* We recognize the functions `memcpy_fromfs`, `copy_from_user` and `get_user` as giving access to user data. Possible faults are cases where a value obtained using one of these functions is used as an array index, with no prior test on its value, and where some value is checked to be less than a value obtained using one of these functions, as would occur in validating a loop index.

Lock and Intr *"Release acquired locks; do not double-acquire locks (**Lock**)." "Restore disabled interrupts (**Intr**)."* In early versions of Linux, locks and interrupts were managed separately: typically interrupts were disabled and reenabled using `cli` and `sti`, respectively, while locks were managed using operations on spinlocks or semaphores. In Linux 2.1.30, however, functions such as `spin_lock_irq` were introduced to combine locking and interrupt management. Our **Lock** checker is limited to operators that only affect locks (spinlocks and, from Linux 2.6.16, mutexes), our **Intr** checker is limited to operators that only disable interrupts, and for the combined operations, we introduce a third checker, **LockIntr**. The considered functions are listed in the appendix.

Free *"Do not use freed memory."* Like the **Null** checker, this checker first iteratively collects functions that always apply `kfree` or some collected function to some parameter, and then checks each call to `kfree` or a collected function for a use of the freed argument after the call.

Float *"Do not use floating point in the kernel."* Most uses of floating point in kernel code are in computations that are performed by the compiler and then converted to an integer or in code that is part of the kernel source tree, but is not actually compiled into the kernel. Our checker only reports a floating point constant that is not a subterm of an arithmetic operation involving another constant.

Size *"Allocate enough memory to hold the type for which you are allocating."* Because our checker works at the source code level without first invoking the C preprocessor, it is not aware of the sizes of the various data types on a particular architecture. We thus focus on the information apparent in the source code, considering the following two cases. In the first case, one of the basic memory allocation functions, `kmalloc` or `kzalloc`, is given a size argument involving a `sizeof` expression defined in terms of a type that is different from the type of the variable storing the result of the allocation. To reduce the number of false positives, the checker ignores cases where one of the types involved represents only one byte, such as `char`, as these are often used for allocations of unstructured data. We consider as a fault any case where there is no clear relationship between the types, whether the allocated region is too large or too small. In the second case, there is an assignment where the right hand side involves taking the size of the left hand side expression itself, rather than the result of dereferencing that expression. In this case, the allocated region has the size of a pointer, which is typically significantly smaller than the size intended.

	Find	Fix	Impact
Block	Hard	Hard	Low
Null	Hard	Hard	Low
Var	Easy	Easy	Low
IsNull	Easy	Easy	Low
NullRef	Easy	Hard	Low
Range	Easy	Easy	Low
Lock	Easy	Easy	High
Intr	Easy	Easy	High
LockIntr	Easy	Easy	High
Free	Hard	Easy	High
Size	Easy	Easy	High
Float	Easy	Hard	High

Table 1. Assessment of the difficulty of finding and fixing faults, and the potential of a fault to cause a crash or hang at runtime

These faults vary in how easy they are to find in the source code, how easy they are to fix once found, and the likelihood of a runtime impact. Table 1 summarizes these properties for the various fault kinds, based on our observations in studying the code. Faults involving code within a single function are often easy for both maintainers and tools to detect, and thus we designate these as "Easy." Finding "Hard" faults requires an interprocedural analysis to identify functions that have specific properties. Interprocedural analysis requires more effort or expertise from a maintainer, or more complexity in a tool. Fixing a fault may require only an easy local change, as in **Size**, where the fix may require only changing the argument of `sizeof` to the type of the allocated value. Cases that require creating new error handling code, such as **Null**, or choosing between several alternative fixes (*e.g.*, moving a dereference or dropping an unnecessary null test), such as **NullRef**, are more difficult. Instances of fault kinds that entail more difficult fixes may benefit less from the use of tools, as the tool user may not have enough expertise to choose the correct fix. Finally, we indicate a low impact when a crash or hang is only likely in an exceptional condition, and high when it is likely in normal execution.

2.2 Assessing the fault rate

The maximum number of faults that code can contain is the number of occurrences of code relevant to the fault, *i.e.* where a given kind of fault may appear. For example, the number of **Block** faults is limited by the number of calls to blocking functions. We follow Chou *et al.* and refer to these occurrences of relevant code as *notes*. Then,

$$\textit{fault rate} = \textit{faults}/\textit{notes}$$

We find the notes associated with each of our checkers as follows. For **Block**, **Null**, and **Free**, a note is a call to one of the functions collected as part of the transitive closure in the fault-finding process. For **Var**, a note is a local array declaration. For **Inull** (**IsNull** and **NullRef**), a note is a null test of a value that is derefenced elsewhere in the same function. For **Range** and for **Lock**, **Intr**, or **LockIntr**, a note is a call to one of the user-level access functions or locking

functions, respectively. For **Size**, a note is a use of `sizeof` as an argument to one of the basic memory allocation functions `kmalloc` or `kzalloc` when the argument is a type, or a use of `sizeof` where the argument is an expression. In the former case, as for the checker, we discard some cases that are commonly false positives such as when the argument to `sizeof` is a one-byte type such as `char`. Finally, we do not calculate the number of notes for **Float**, because we consider that every occurrence of a float in a context where it may be referenced in the compiled code is a fault, and thus the number of notes and faults is the same.

2.3 Tools

Our experimental protocol relies on two open-source tools: Coccinelle (v0.2.2), to automatically find potential faults and notes in the Linux kernels [24], and Herodotos (v0.6.0), to correlate the fault reports between versions [25]. We store the resulting data in a PostgreSQL database (v8.4), and analyze it using SQL queries.

Coccinelle performs control-flow based pattern searches in C code. It provides a language, the Semantic Patch Language (SmPL), for specifying searches and transformations and an engine for performing them. Coccinelle's strategy for traversing control-flow graphs is based on the temporal logic CTL [2], while that of the tool used by Chou *et al.* is based on automata. There are technical differences between these strategies, but we do not expect that they are relevant here.

A notable feature of Coccinelle is that it does not expand preprocessor directives. We have only found this feature to be a limitation in the **Var** case, as noted in Section 2.1. On the other hand, this feature has the benefit of making the fault-finding process independent of configuration information, and thus we can find faults across the entire Linux kernel source tree, rather than being limited to a single architecture.

To be able to understand the evolution of faults in Linux code, it is not sufficient to find potential faults in the code base; we must also understand the history of individual fault occurrences. To do so, we must be able to correlate potential fault occurrences across multiple Linux versions, even in the presence of code changes in the files, and manage the identification of these occurrences as real faults and false positives. For these operations, we use Herodotos. To correlate fault occurrences, Herodotos first uses `diff` to find the changes in each pair of successive versions of a file for which Coccinelle has produced fault reports. If a pair of reports in these files occur in the unchanged part of the code, at corresponding lines, they are automatically considered to represent the same fault, with no user intervention. Otherwise, if only one of a pair of reports occurs in the unchanged part of the code, then the reports are automatically considered to be unrelated. Finally, if both of a pair of reports occur in the changed part of the code, then their status is considered to be unknown, and the user must indicate, via an interface based on the emacs "org" mode, whether they represent the same fault or unrelated ones. Once the correlation process is complete, a similar interface is provided to allow the user to classify each group of correlated reports as representing either a fault or a false positive. Further details about the process of using Herodotos are provided elsewhere [25, 26].

Once the fault reports are correlated and assessed for false positives, we import their histories into the database, along with the associated notes. The database also contains information about Linux releases such as the release date and code size, and information about Linux files (size, number of modifications between releases) and functions (starting and ending line numbers). The complete database, including both the reports and the extra information, contains 1.5 GB of data. To analyze the collected data, we wrote over 2,000 lines of PL/pgSQL and SQL queries that extract and correlate information.

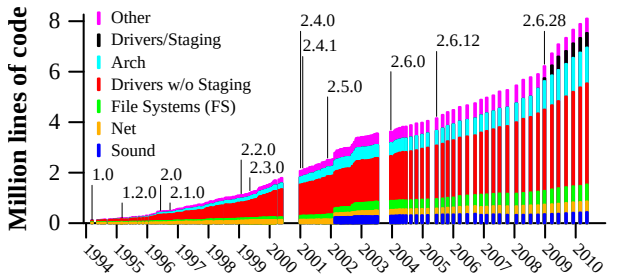


Figure 1. Linux directory sizes (in MLOC)

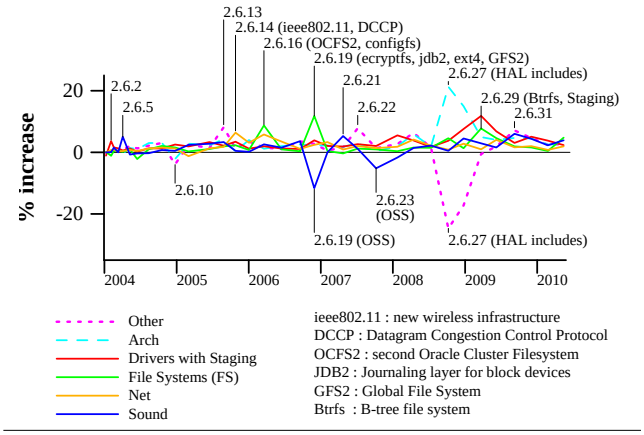


Figure 2. Linux directory size increase

Extending the results to new versions A benefit of our experimental protocol is that it makes it easy to extend the results to new versions of Linux. When a new version of Linux is released, it is only necessary to run the checkers on the new code, and then repeat the correlation process. As our collected data contains information not only about the faults that we have identified, but also about the false positives, Herodotos automatically annotates both faults and false positives left over from previous versions as such, leaving only the new reports to be considered by the user.

3. Evolution of Linux

To give an overview of the complete history of Linux, we first consider the evolution in code size of the Linux kernel between version 1.0, released in March 1994, and version 2.6.33, released in February 2010, as shown in Figure 1. This figure shows the size of the development versions, when available, as it is in these versions that new code is added, and this added code is then maintained in the subsequent stable versions. Code sizes are computed using David A. Wheeler's 'SLOCCount' (v2.26) [39] and include only the ANSI C code. The code sizes are broken down by directory, highlighting the largest directories: `drivers/staging`, `arch`, `drivers`, `fs` (file systems), `net`, and `sound`. `Drivers/staging` was added in Linux 2.6.28 as an incubator for new drivers that are not yet mature enough to be used by end users. Code in `drivers/staging` is not compiled as part of the default Linux configuration, and is thus not included in standard Linux distributions. `Sound` was added in Linux 2.5.5, and contains sound drivers that were previously in the `drivers` directory. The largest directory is `drivers`, which, including `drivers/staging`, has made up 57% of the source code since Linux 2.6.30.

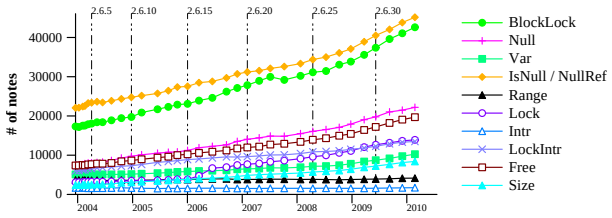


Figure 3. Notes through time per kind

For most directories, the code growth has been roughly linear since Linux 1.0. Some exceptions are highlighted in Figure 2, which shows the percentage code size increase in each directory from one version to the next. We have marked some of the larger increases and decreases. Many of the increases involve the introduction of new services, such as new file systems. In Linux 2.6.19 and 2.6.23, old OSS drivers already supported by ALSA were removed from `sound`, decreasing its size. In Linux 2.6.27, `arch` was reorganized, and received some large header files from `include`, adding around 180,000 lines of C code. Finally, `staging` grew substantially in 2.6.29. All in all, these changes have resulted in code growth from 2MLOC in 2001 to more than 8MLOC in 2010.

In our study, we are less interested in the absolute number of lines of code than the amount of code relevant to our fault kinds. As shown in Figure 3, the increase in code size has induced an almost linear increase in the number of notes, as defined in Section 2.2, in almost all cases. In fact, across all of Linux 2.6, the number of notes per line of code is essentially constant, between 0.027 and 0.030.

4. Linux 2.4.1

Linux 2.4.1 was the latest version of Linux considered by Chou *et al.* [5]. To validate our experimental protocol, we have used our checkers to find faults and notes in this version, and we compare our results to those provided in their paper. We focus on the results that are specific to Linux 2.4.1, rather than those that relate to the history of Linux up to that point, to avoid the need to study earlier versions that are of little relevance today.

4.1 What code is analyzed?

For the results of fault finding tools to be comparable, the tools must be applied to the same code base. Chou *et al.* focus only on x86 code, finding that 70% of the Linux 2.4.1 code is devoted to drivers. Nevertheless, we do not know which drivers, file systems, etc. were included. To calibrate our results, we use SLOCCount to obtain the number of lines of ANSI C code in the Linux kernel and in the `drivers` directory, considering three possibilities: all code in the Linux 2.4.1 kernel source tree ("All code"), the set of `.c` files compiled when using the default x86 configuration ("Min x86"),[3] and all 2.4.1 code except the `arch` and `include` subdirectories that are specific to non-x86 architectures ("Max x86"). Max x86 gives a result that is closest to that of Chou *et al.*, although the proportion of driver code is slightly higher than 70%. This is reasonable, because some driver code is associated with specific architectures and cannot be compiled for x86. Nevertheless, these results show that we do not know the precise set of files used in Chou *et al.*'s tests.

In our experiments, we consider the entire kernel source code, and not just the code for x86, as every line of code can be assumed to be relevant to some user.

[3] This configuration was automatically generated using `make menuconfig` without any modification of the proposed configuration. To collect the `.c` files, we compiled Linux 2.4.1 according to this configuration using a Debian 3.1 (Sarge) installation in a virtual machine, with `gcc` version 2.95.4 and `make` version 3.80.

	All code	Min x86	Max x86
Drivers LOC	1,248,930	71,938	1,248,930
Total LOC	2,090,638	174,912	1,685,265
Drivers %	59%	41%	74%

Table 2. Percentage of Linux code found in `drivers` calculated according to various strategies

4.2 How many faults are there?

For the entire Linux 2.4.1 kernel, using the checkers described in Section 2.1, we obtain 600 reports, of which we have determined that 467 represent faults and the remainder represent false positives. Chou *et al.*'s checkers find 1,025 faults in Linux 2.4.1. They have checked 602 of these reports; the remainder are derived from what they characterize as low false positive checkers. We have checked all of the reports included in our study.

Table 3 compares the number of faults found per checker. In most cases, we find fewer faults. This may be due to different definitions of the checkers, or different criteria used when identifying false positives. In the case of **Var**, we find fewer faults because we consider only cases where the size is explicitly expressed as a number. In a few cases, we find more faults. For example, Chou *et al.*'s **Inull** checker can be compared to our **IsNull** and **NullRef** checkers. We find fewer **IsNull** faults than their **Inull** faults, but far more **NullRef** faults. We also find slightly more **Free** faults. This may derive from considering a larger number of files, as we have found that only one of our **Free** faults occurs in a file that is compiled using the default x86 configuration. Results from Chou *et al.*'s checkers were available at a web site interface to a database, but Chou has informed us that this database is no longer available. Thus, it is not possible to determine the precise reasons for the observed differences.

Checker	Chou *et al.*		Our results
	checked	unchecked	
Block	206	87	N/A
BlockLock	N/A	N/A	43
Null	124	267	98
Var	33	69	13
Inull	69	0	N/A
IsNull	N/A	N/A	36
NullRef	N/A	N/A	221
Range	54	0	11
Lock	26	0	5
Intr	27	0	2
LockIntr	N/A	N/A	6
Free	17	0	21
Float	10	15	8
Size	3	0	3
Total	569	438	467

Table 3. Comparative fault count

4.3 Where are the faults?

Chou *et al.* find that the largest number of faults is in the `drivers` directory and that the largest number of these faults are in the categories **Block**, **Null**, and **Inull**, with around 180, 95, and 50 faults in `drivers`, respectively.[4] As shown in Figure 4(a), we also observe that the largest number of faults is in the `drivers` directory, with the largest number of these faults also being in **BlockLock**, **Null**, and **Inull** (**IsNull** and **NullRef**), although in different proportions.

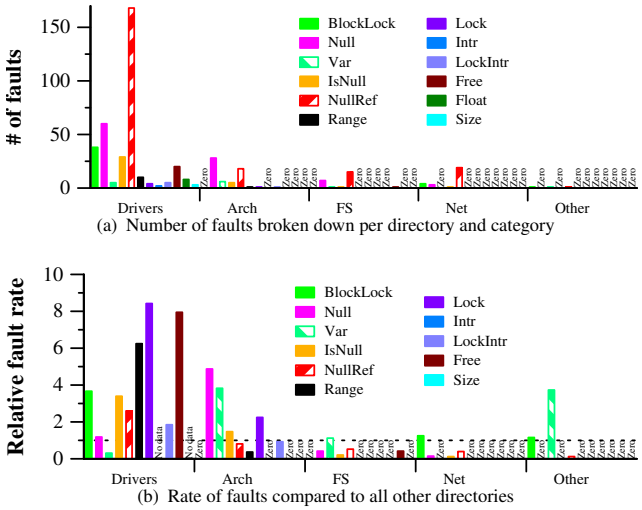
A widely cited result of Chou *et al.* is that the `drivers` directory contains almost 7 times as many of a certain kind of faults (**Lock**) as all other directories combined. They computed this ratio using

[4] These numbers are approximated from the provided graphs.

the following formula for each directory d, where d refers to the directory d itself and $\overline{d}$ refers to all of the code in all other directories:

$$fault\ rate_d / fault\ rate_{\overline{d}}$$

Figure 4(b) shows the values of the same formula, using our results. We obtain a similar ratio with a relative rate of over 8 for **Lock** in `drivers`, as compared to all other directories combined. We also find that the `drivers` directory has a rate of **Free** faults that is almost 8 times that of all other directories combined. Chou *et al.*, however, found a fault rate of only around 1.75 times that of all other directories combined in this case. With both approaches, however, the absolute number of **Free** faults is rather small. Like Chou *et al.*, we also observe a high fault rate in the `arch` directory for the **Null** checker, in both cases about 4.8 times that of all other directories combined. Finally, unlike Chou *et al.*, we observe a high rate of **Var** faults in both `arch` and *other*. In the `arch` case, all of the **Var** faults found are for architectures other than x86. Indeed, overall for `arch`, we find 60 faults, but only 3 (all **Null**) in the x86 directory.



(a) Number of faults broken down per directory and category

(b) Rate of faults compared to all other directories

Figure 4. Faults in Linux 2.4.1

4.4 How are faults distributed?

Chou *et al.* plot numbers of faults against the percentage of files containing each number of faults and find that for all of the checkers except **Block**, the resulting curve fits a log series distribution, with a θ value of 0.567 and a degree of confidence (p-value) as measured by the χ^2 test of 0.79 (79%). We observe a θ value of 0.562 and a p-value of 0.234 without **Block**. We conjecture that this low p-value is due to the fact that two files have 5 and 6 faults while three files have 7 faults each; such an increase for larger values is not compatible with a log series distribution. Nevertheless, the values involved are very small, and, as shown in Figure 5, the curve obtained from our experimental results fits well with the curve obtained using the corresponding calculated θ value. The curve obtained from Chou *et al.*'s value of θ is somewhat lower, because they found a smaller number of faulty files, probably due to having considered only the x86 architecture.

Chou *et al.* also find that younger files and larger functions have a higher fault rate, of up to 3% for the **Null** checker. We also find fault rates of around 3% for the **Null** checker, for files of all ages and for larger functions. Overall, we find that younger files have a fault rate of 0.8% while middle aged files, *i.e.*, those with an average age of over 6 years, have a significantly higher fault rate, of 0.4%. We also find a definite increase in fault rate as function size

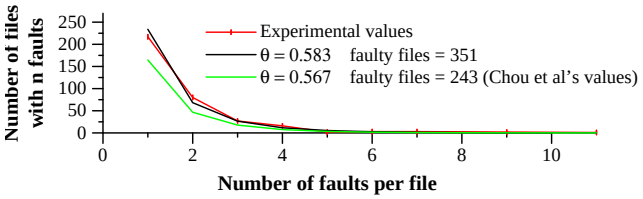


Figure 5. Comparison of the error distributions

increases: from 0.2% for functions containing up to 16 lines, to 0.6% for functions containing between 17 and 36 lines, to 0.8%-1.0% for larger functions.

4.5 Assessment

In this section, we have seen that our checkers do not find the same number of faults in Linux 2.4.1 code as those of Chou *et al.* We recall that Chou *et al.*'s checkers are not very precisely described, and thus we expect that most of the differences are in how the checkers are defined. Furthermore, we do not consider exactly the same set of files. Nevertheless, the distribution of these faults among the various directories is roughly comparable, and their distribution among the files is also comparable. We thus conclude that our checkers are sufficient to provide a basis for comparison between Linux 2.6 and previous versions studied by Chou *et al.*

5. Linux 2.6 kernels

We now assess the extent to which the trends observed for Linux 2.4.1 and previous versions continue to apply in Linux 2.6, and study the points of difficulty in kernel development today. We consider what has been the impact of the increasing code size and the addition of new features on code quality, and whether drivers are still a major problem.

Concretely, we study a period of over 6 years, beginning with the release of Linux 2.6.0 at the end of 2003 and ending in early 2010 with the release of Linux 2.6.33. For the entire Linux 2.6 kernel, using the checkers described in Section 2.1, we obtain 3,915 different reports (after correlation), of which we have determined that 2,370 represent faults and the rest represent false positives. We also consider some new checkers related to the use of the recently added RCU locking API. Finally, because one of our goals is to provide a means of repeating our work, we consider how our experimental protocol eases the extension of the results to new Linux versions.

5.1 How many faults are there?

We first analyze the relation between the code growth and the total number of faults in Linux 2.6. As shown in Figure 6(a), the number of the faults considered has held roughly steady over this period, with an overall increase of only 7%, and indeed a decrease of 11% from 2.6.0 to 2.6.28. This is quite remarkable given that the code size has more than doubled since Linux 2.6.0 (Figure 1). Indeed, the rate of faults per line of code has significantly decreased, by 50%, as shown in Figure 6(b). These observations are quite different from those for versions up through Linux 2.4.1: there was a code size increase of over 17 times between Linux 1.0 and Linux 2.4.1 and a corresponding increase in the number of the faults considered of over 33 times [5]. Figure 6(c) shows that faults are still introduced, indeed at a growing rate. But in most versions even more faults are eliminated.

Figure 7 shows the number of each kind of fault found in Linux 2.6, separated for readability into those that have increased in number between the beginning and the end (Figure 7(a)) and those

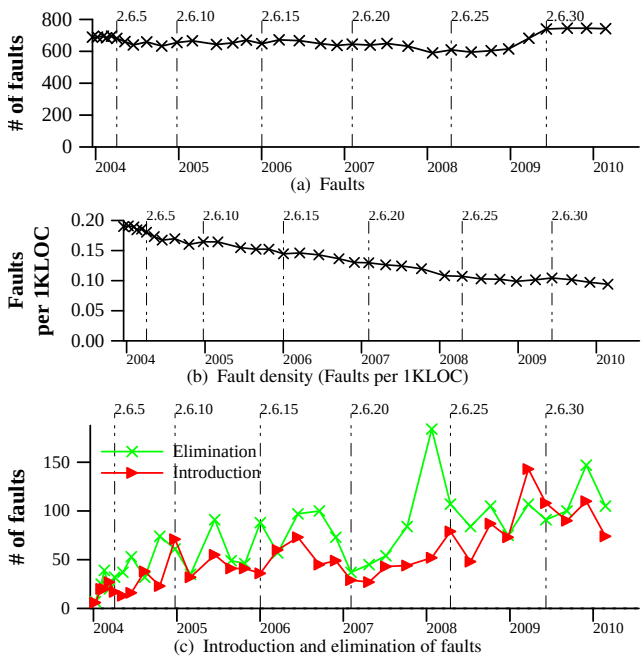


Figure 6. Faults in Linux 2.6.0 to 2.6.33

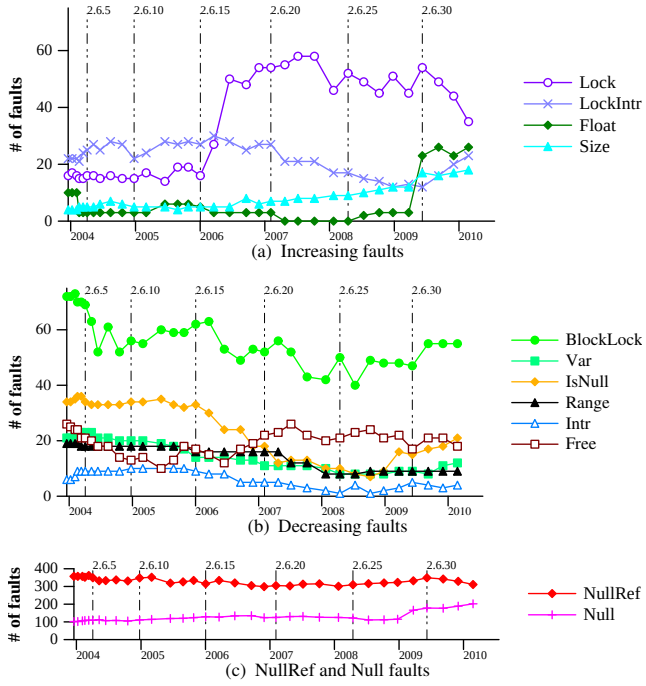


Figure 7. Faults through time

that have decreased in number in the same versions (Figure 7(b)). **NullRef** and **Null** are further separated from the others. For many fault kinds, the number of faults is essentially constant over the considered period.

Three notable exceptions to the stability in the number of Linux 2.6 faults are **Lock**, **Null**, and **Float**, in Linux 2.6.16 and 2.6.17,

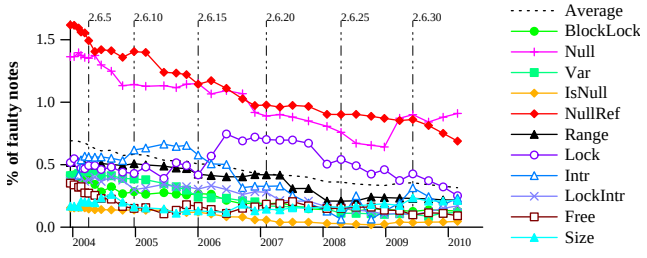


Figure 8. Fault rate per fault kind

Linux 2.6.29, and Linux 2.6.30, respectively (Figures 7(a) and 7(c)). In Linux 2.6.16, the functions `mutex_lock` and `mutex_unlock` were introduced to replace mutex-like occurrences of the semaphore functions `down` and `up`. 9 of the 11 **Lock** faults introduced in Linux 2.6.16 and 23 of the 25 **Lock** faults introduced in Linux 2.6.17 were in the use of `mutex_lock`. In Linux 2.6.29, the `btrfs` file system was introduced, as seen in Figure 2. 33 **Null** faults were added with this code. 7 more **Null** faults were added in `drivers/staging`, which more than tripled in size at this time. 29 other **Null** faults were also added in this version. Finally, in Linux 2.6.30 there was a substantial increase in the number of Comedi drivers [6] in `drivers/staging`. All of the 21 **Float** faults introduced in this version were in two Comedi files. These faults are still present in Linux 2.6.33. Recall, however, that `staging` drivers are not included in Linux distributions.

As shown in Figure 8, the fault rate, *i.e.*, the ratio of observed faults to the number of notes, for the considered fault kinds confirms the increase in reliability (Float is omitted, as described in Section 2.2). As the number of notes increases roughly with the size of the Linux kernel while the number of faults is relatively stable, the fault rate tends to decline. The main increases, in **Lock** and **Null**, are due to the introduction of `mutex_lock` and the `btrfs` file system, respectively, as mentioned previously.

5.2 Where are the faults?

The presence of a high rate of faults in a certain kind of code may indicate that this kind of code overall needs more attention. Indeed, Chou *et al.*'s work motivated studies of many kinds of driver faults, going beyond the fault kinds they considered. Many properties of the Linux kernel have, however, changed since 2001, and so we reinvestigate what kind of code has the highest rate of faults, to determine whether attention should now be placed elsewhere.

As shown in Figure 9, the largest number of faults is still in `drivers`, which indeed makes up over half of the Linux kernel source code. The second-largest number of faults is in `arch`, accompanied by `fs` and `drivers/staging` in recent versions. In contrast to the case of Linux 2.4.1, however, as shown in Figure 10, `drivers` no longer has the largest fault rate, and indeed since Linux 2.6.19 its fault rate has been right at the average. There was not a large increase in the number of `drivers` notes at that time, so this decrease is indicative of the amount of attention drivers receive in the peer reviewing process. `Arch` on the other hand has many faults and relatively little code, and so it has the highest fault rate throughout most of Linux 2.6. Around 30% of the `arch` faults are **Null** faults, although there appears to be no pattern to their introduction. Over 90% of the `arch` faults are outside of the `x86/i386` directories, with many of these faults being in the `ppc` and `powerpc` code. The largest numbers of faults in `fs` are in `cifs`, with over 40 faults in Linux 2.6.0 but a decreasing number after, in `ocfs2`, with 10-15 faults per version starting in Linux 2.6.17, and in `btrfs`, with 36 in Linux 2.6.29 and 38 in Linux 2.6.30 and gradually fewer after that. All of these are recently introduced file systems: `cifs` was introduced in

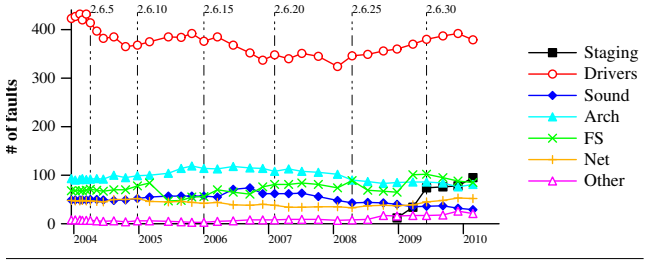


Figure 9. Faults per directory

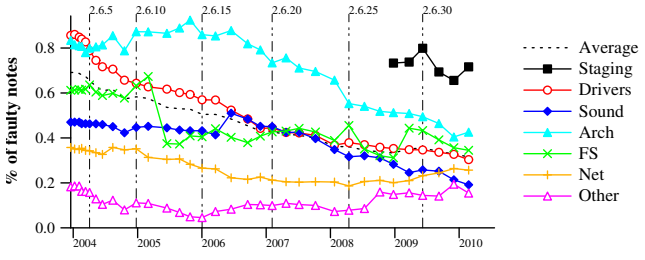


Figure 10. Fault rate per directory

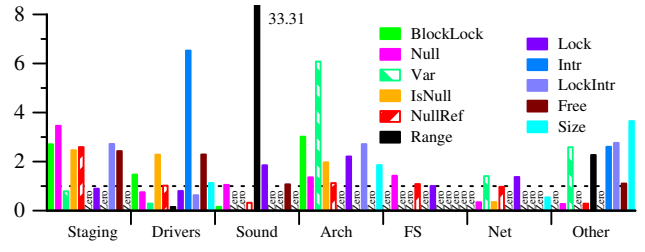


Figure 11. Fault rates compared to other directories (Linux 2.6.33)

Linux 2.5.42, `ocfs2` in Linux 2.6.16, and `btrfs` in Linux 2.6.29. `Drivers/staging`, introduced in Linux 2.6.28, also has a high fault rate, exceeding that of `arch`. This directory is thus receiving drivers that are not yet mature, as intended. The introduction of `drivers/staging`, however, has no impact on the fault rate of `drivers`, as `drivers/staging` accommodates drivers that would not otherwise be accepted into the Linux kernel source tree. Such drivers benefit from the expertise of the Linux maintainers, and are updated according to API changes with the rest of the kernel.

For Linux 2.4.1, we observed that `drivers` had a much higher fault rate for certain kinds of faults than other directories. Figure 11 shows that `drivers` has a high rate of **Intr** faults in Linux 2.6.33 as compared to other directories, but there are very few **Intr** fault in this version. `Sound`, which was part of `drivers` in 2.4.1, has a high rate of **Range** faults, as compared to the other directories, but again the actual number of faults is relatively small. Overall, while `drivers` has a high rate as compared to other directories for some fault kinds, it is more common that `drivers/staging`, `arch`, or *other* has the highest fault rate, indicating again that the drivers that are intended for use in the Linux kernel are no longer the main source of faults.

Finally, in Figure 12, we consider the number of faults per file that contains at least one fault. The highest average number of faults per faulty file is for `fs` in the versions prior to 2.6.12. In this case, there was a single file with many **NullRef** faults, as many as 45 in Linux 2.6.11. In later versions, the highest average is for `drivers/staging`, for which the average was over 2 in Linux

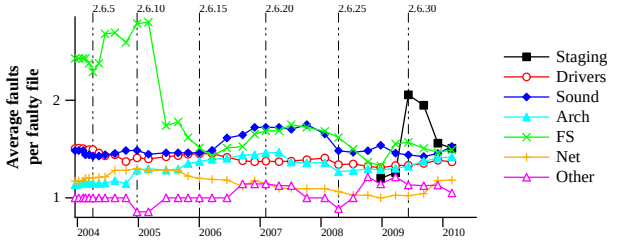


Figure 12. Faults per faulty file per directory

2.6.30. At that point, a large number of drivers had recently been introduced in this directory. Many of these faults have been corrected and the rate of entry of new drivers has slowed, and thus the average has dropped to around 1.5, close to that of other directories. `Sound` had a relatively high number of faults per faulty file starting in Linux 2.6.16 with the introduction of `mutex_lock`; faulty functions often contain more than one `mutex_lock`, and thus a single omitted `mutex_unlock` may result in multiple **Lock** reports.

5.3 How long do faults live?

Eliminating a fault in Linux code is a three step process. First, the fault must be detected, either manually or using a tool. Then it must be corrected, and a patch submitted to the appropriate maintainers. Then, the patch must be accepted by a hierarchy of maintainers, ending with Linus Torvalds. Finally, there is a delay of up to 3 months until the next release. The lifespan of a fault, modulo this three month delay, is an indication of the efficiency of this process.

Fault lifespans Figure 13 presents the average lifespan of faults across Linux 2.6, by directory and by fault kind. We omit `drivers/staging` because it was only introduced recently. Some faults were present before Linux 2.6.0 and some faults were still present in Linux 2.6.33. For the average lifespan calculation, in the former case, we assume that the fault was introduced in Linux 2.6.0 and in the latter case, we assume that the fault was eliminated in Linux 2.6.34, thus potentially underestimating the lifespan of such faults.

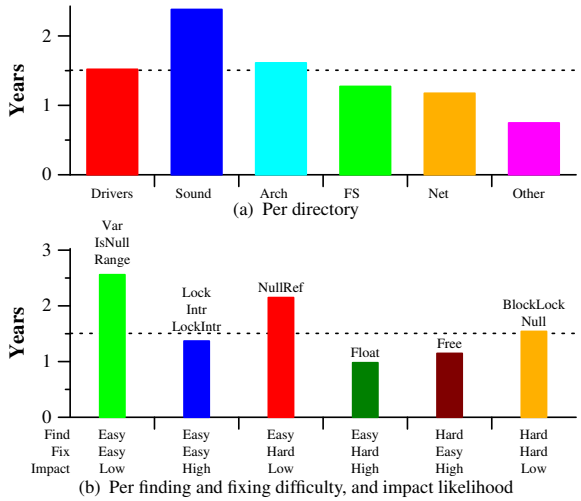


Figure 13. Average fault lifespans (without staging)

The average fault lifespan across all files of Linux 2.6 is 1.5 years, as indicated by the horizontal dotted line in Figure 13 The lifespans vary somewhat by directory. As shown in Figure 13(a), the average lifespan of faults in the `drivers` directory is the same as

the average lifespan of all faults, and indeed is less than the average lifespan of faults in the `sound` and `arch` directories. `Sound` faults now have the longest average lifespan. `Sound` used to be part of `drivers`; it may be that the sound drivers are no longer benefiting from the attention that other drivers receive.

For the fault kinds, Figure 13(b) shows that the average lifespans correspond roughly to our assessment of the difficulty of finding and fixing the faults and their likelihood of impact (Table 1). In particular, all of the fault kinds we have designated as having high impact, meaning that the fault is likely to have an observable effect if the containing function is executed, are fixed relatively quickly. On the other hand, the ease of finding and fixing the faults has little impact on their lifespan, showing that developers are willing to invest in tracking down any faults that cause obvious problems.

Figure 14 examines the fault lifetimes in more detail, by showing the number of faults that have been fixed in less than each amount of time. Again we include the faults already present in Linux 2.6.0 and the faults still remaining in Linux 2.6.33. While half of the faults we found where fixed within just under one year, it took about 5.5 years (almost half of the period studied) to fix 80% of the faults. For the `drivers` and `fs` directories, half of the faults were fixed within just under one year, while 2.5 years were required for `sound`. On the other hand, half of the `staging` and *other* faults were fixed within 6 months.

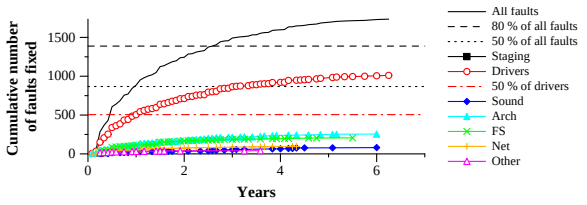


Figure 14. Cumulative number of fixed faults per amount of elapsed time

Origin of faults Figure 15 shows the lifetime of all of the Linux 2.6 faults in our study, with Linux 2.6.1 at the bottom of the graph and Linux 2.6.33 at the top. The 689 faults in Linux 2.6.0 are omitted to save space; their lifetime within Linux 2.6 can be seen later as the leftmost curve in Figure 16. They may, however, have been introduced earlier. Of the faults introduced and eliminated within the period considered, 36% of the faults introduced in or after Linux 2.6.0 were introduced with the file and 12% of the faults eliminated before Linux 2.6.33 were eliminated with the file.[5]

While we have seen that the total number of faults is essentially constant across the versions, Figure 15 shows that since Linux 2.6.27 a significantly larger number of faults have been introduced. **Null** and **NullRef** faults predominate, with for example 51% of the added faults in Linux 2.6.27 being **NullRef** faults, most of which were introduced in various drivers. In Linux 2.6.30 and Linux 2.6.32, 43% and 26% of the introduced faults were in `drivers/staging`. In each case, about half of the introduced faults were fixed within a few versions.

Figure 16 shows the number of faults in each version that are still present in each of the previous and successive versions. Except for the increase at Linux 2.6.29 and 2.6.30, as previously noted, the height and angle of all of the lines is fairly similar, indicating that the rate of introductions and eliminations of faults across the versions is relatively stable. These facts indicate a maturity in the Linux code and its development model.

[5] The degree to which this is visible depends on the image quality. It may be useful to print this page rather than view it on a screen.

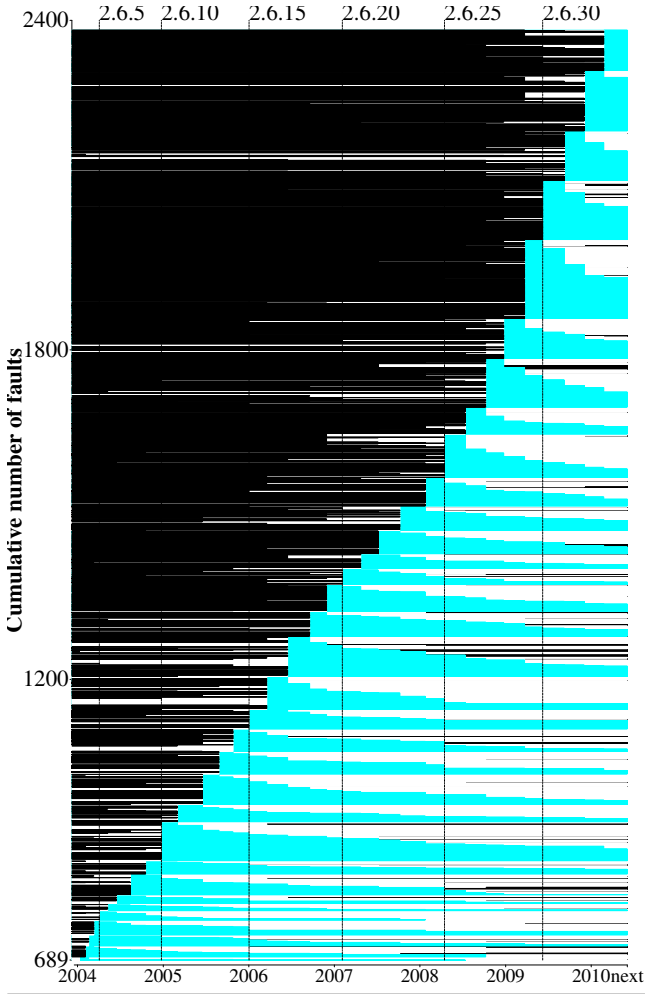


Figure 15. Lifetime of faults. Each row represents a separate fault. Blue (grey) lines indicate the periods where the fault is present. Black lines indicate the period where the file containing the fault does not exist and white lines indicate the period where the file does exist but the fault does not. The 689 faults in Linux 2.6.0 are omitted.

Developer and maintainer activity In the Linux development model, anyone (an *author*) can submit a patch, and the patch is then picked up by a maintainer, who *commits* it into his git repository that is then propagated to Linus Torvalds. The number of patch authors is thus an indicator of the number of participants in the Linux development process, and the number of committers is an indicator of the amount of manpower that is available to begin the integration of patches into a release. Figure 17 shows the number of authors and committers associated with the patches included in each version, both in total and broken down by directory.

For `drivers`, the numbers of authors and committers are rising at the same rate, roughly at the rate of the increase in the code size. For `arch` and `fs`, however, where we have previously noted a higher fault rate, the number of authors is rising significantly more slowly than the number of committers. The lower number of authors may suggest that potential authors are not able to develop adequate expertise to keep up with the number of new architectures and file systems (Figure 2). Finally, the small number of `sound` authors may explain the previously observed long life of `sound` faults (Figure 13(a)).

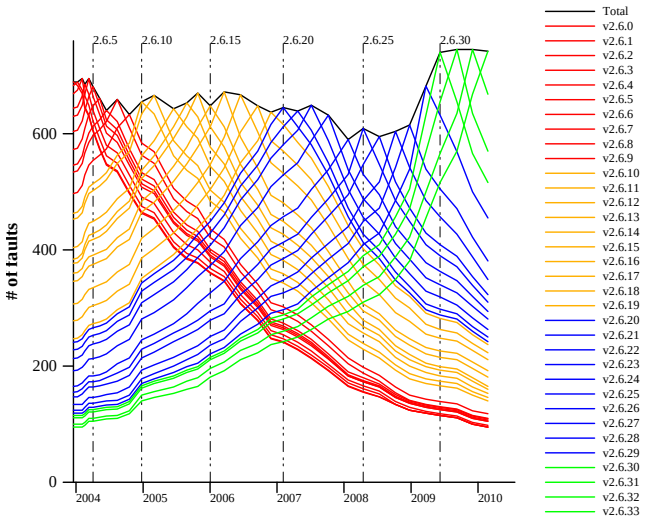


Figure 16. Lifetime of faults across versions

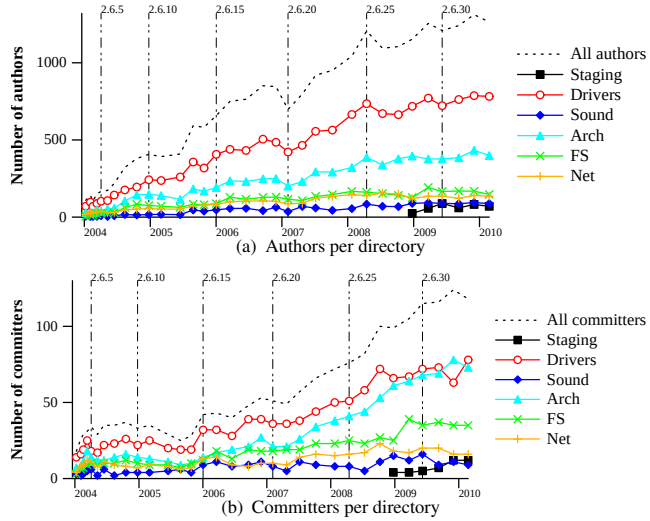


Figure 17. Authors and committers per version

Use of tools In principle, since the work of Chou *et al.*, it has been possible to find all of the considered faults using tools. Figure 18 shows the number of patches in each Linux version that mention one of the fault-finding tools Coccinelle [24] (used in this paper), Coverity [9] (the commercial version of Chou *et al.*'s `xgcc` tool), sparse [31, 33], and smatch [36]. As developers are not obliged to mention the tools they use, these results may be an underestimation.

The use of the various tools is somewhat variable across the different versions. In the case of Coverity, the main use occurred between 2006 and 2009, when its application to open-source software, including Linux, was funded by the US Department of Homeland Security [3]. Sparse usage shows several peaks, at 2.6.25 and at 2.6.29-30. No particular pattern emerges for 2.6.25. For versions 2.6.29 and 2.6.30, a single developer was responsible for 140 of the 253 sparse-related patches. He fixed faults across the entire kernel. The tools have also been used to find a wider range of faults than those considered in this paper. For example, in Linux 2.6.24, only about half of the patches that mention Coverity relate to the kinds of faults we consider, particularly **Null**, **IsNull**, **NullRef**, and **Free**. Overall, despite the variability in usage, the results show a willingness on the part of the Linux developers to use fault-finding tools and to pay attention to the kinds of faults that they find.

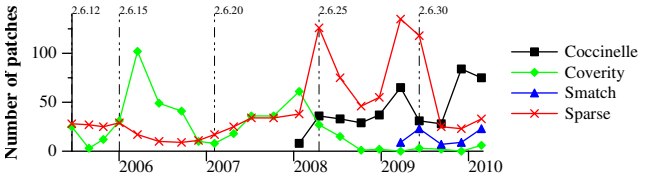


Figure 18. Tool usage since the introduction of git

5.4 Is code quality predictable?

In the software engineering community, substantial work has been done on identifying metrics that predict code quality. We consider three possible metrics: code churn, file age, and function size. We also evaluate the quality of Linux code in light of the conjecture that open source code is more reliable because it can be examined by many people.

Churn Elbaum and Munson observed that code churn, *i.e.*, the number of times a file is modified, is a good predictor of fault rate [21]. Nagappan and Ball reached a similar conclusion in a study that used metrics relating to the development of Windows Server 2003 to predict SP1's fault rate [22]. Figure 19 shows the relationship between the average churn per day preceding the release of each version and the number of the considered faults added in that version. The relationship between churn and fault rate is similar. There is an overall tendency of high-churn versions to contain more new faults, even if some high-churn versions have a smaller number of faults than lower churn versions. Recall, however, that in most versions, more faults were eliminated than added; the churn includes the elimination of faults as well.

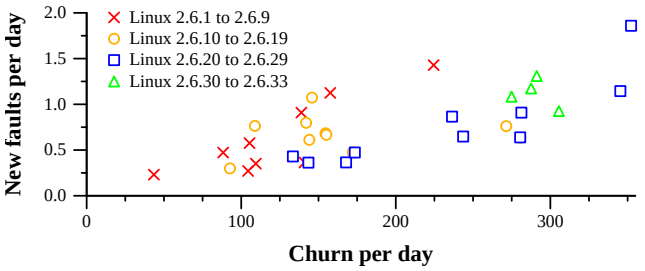


Figure 19. Churn vs. new faults

Kernel configuration In *"The Cathedral and the Bazaar"* [30], Eric S. Raymond formalized Linus' law: "given enough eyeballs, all bugs are shallow." But, in practice, code that is frequently executed, or at least frequently compiled, is more likely to be reviewed than the rest. When code is frequently executed, many users are likely to encounter any faults, and some may fix the faults themselves or submit a request that the faults be fixed by a kernel maintainer. When code is frequently compiled, even if it is not frequently executed, it can easily be submitted to fault-finding tools that are integrated with the kernel compilation process. "Eyeballs" may also focus on fixing faults in code that they are able to compile, as even standard compilers such as `gcc` perform some sanity checks that provide some confidence that a fix has not *e.g.*, introduced a typographical error, even if the code cannot be tested.

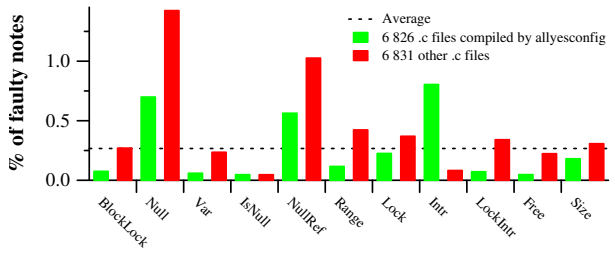


Figure 20. Fault rate compared between configurations (Linux 2.6.33)

Figure 20 compares the number of faults found in the `.c` files that are compiled using the configuration generated on an x86 architecture by the Linux Makefile argument `allyesconfig` to the number of faults found in the `.c` files found in the rest of the Linux kernel.[6] The Makefile argument `allyesconfig` creates a configuration file for the given architecture that includes as many options as possible without causing a conflict and without including `drivers/staging`. Thus, it can be assumed to trigger the compilation of a set of well-tested files and a superset of what is normally included with a Linux distribution, and thus what is executed by ordinary users. In most cases, we do find that the `allyesconfig` files have a lower fault rate than the other files. The only exception is for **Intr** faults, but there are only 4 faults in this case.

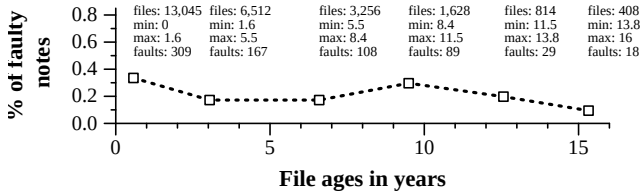
File age and function size One may expect that as a file ages the number of faults would decrease, and that large functions would tend to harbor more faults. Indeed, Chou *et al.* found these trends in Linux 2.4.1, to a varying degree for the different fault kinds. Figure 21 considers the relationship between file age or function size and fault rate in Linux 2.6.33. Files and functions are organized by increasing age or size, respectively, then collected into buckets containing an exponentially decreasing number of elements, from the smallest age or size to the largest. This strategy permits a fine degree of granularity for the files with smaller ages or sizes, respectively. Each graph then shows the average age or size of the files or functions in each bucket and their average fault rate.

Figure 21 shows that in Linux 2.6.33, the youngest half of the files, represented by the leftmost point, has a fault rate about twice that of the next youngest quarter of the files. For older files, however, the relation between age and fault rate is less clear, as the rate first increases and then decreases as the file age increases. On the other hand, the average fault rate clearly increases as the function size increases. Indeed, the bucket with the smallest functions (up to 14 lines) has a significantly lower fault rate than the next bucket. There is also an increase at 54 lines. This suggests that larger functions, although there are relatively fewer of them, may need more attention. As shown by our exponential bucketing strategy, there are relatively few large functions, meaning that it should be feasible to check them more carefully, either manually or using tools.

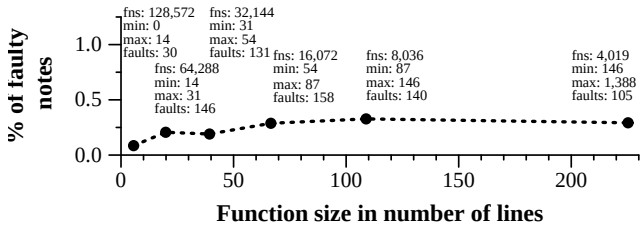
5.5 How does the fault rate in new APIs compare

The fault kinds that we have considered until now involve code structures that are either common to all C code or that have been present in Linux for a long time. To give an alternate perspective on the robustness of Linux 2.6 code, we consider the use of a more recent and specialized locking API: the one that implements the Read-Copy-Update (RCU) mechanism [20]. RCU is a lightweight synchronization mechanism that protects readers against writers.

[6] Compilation was done on an Ubuntu 10.04 (Lucid Lynx) installation with `gcc` 4.4.3 and `make` 3.81.



(a) Fault rate by file age in 2.6.33



(b) Fault rate by function size in 2.6.33

Figure 21. Impact of file age and function size. Each point represents twice as many files or functions as the next point. At each point, "files" and "fns" refers respectively to the number of files and functions considered to compute the point, "min" and "max" refer to the minimal or maximal age or size represented by the point, and faults indicates the number of faults that occur in the code in that range

Reads are wait-free with very low overhead while writes are more expensive. Therefore, RCU particularly favors workloads that mostly read shared data rather than updating it. RCU has been increasingly used in Linux 2.6, but is still used less often than spinlocks and mutexes, as shown in Figure 22.

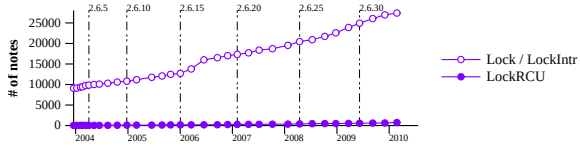


Figure 22. The use of spinlock and mutex locking functions as compared to the use of RCU locking functions

The main functions that we consider in the RCU API are the locking functions: `rcu_read_lock`, `srcu_read_lock`, `rcu_read_lock_bh`, `rcu_read_lock_sched`, and `rcu_read_lock_sched_notrace`. We identify as faults any cases where a blocking function is called while a RCU lock is held (**BlockRCU**) and where a RCU lock is taken but not released (**LockRCU**). These are analogous to the previously considered **BlockLock** and **Lock** fault kinds, respectively. Double-acquiring an RCU lock is allowed, and thus this is not considered to be a fault. Finally, we identify as a fault any case where shared data is accessed using `rcu_dereference` when an RCU lock is not held (**DerefRCU**).

The largest number of uses of the RCU lock functions is in `net`, followed by `drivers` in earlier versions and *other* in later versions, with `kernel`, `include`, `mm`, and `security` being the main *other* directories where it is used. Correspondingly, as shown in Figure 23, most faults are found in `net`, and the few remaining faults are found in *other*. The average lifespan of the `net` faults is around a year, while that of the *other* faults is a few months longer. As shown in Figure 24, most of the faults are **BlockRCU** faults, and the largest number of faults per version is only 10, in Linux 2.6.26. Thus, overall, the fault rates are substantially below the fault rates

observed for any of the previously considered fault kinds, and in particular far below the rates observed for the various locking faults (**Lock**, **Intr**, and **LockIntr**). Nevertheless, the developers who have added calls to the various RCU locking functions are relatively experienced, having at the median 123 patches accepted between Linux 2.6.12 and Linux 2.6.33. Still, these results suggest that Linux developers can be successful at adopting a new API and using it correctly.

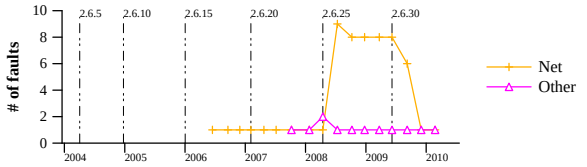


Figure 23. RCU faults per directory

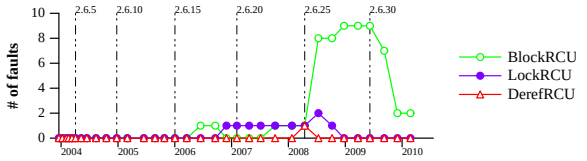


Figure 24. RCU faults through time

5.6 Processing New Versions

As presented in Section 2, the automatic report correlations provided by Herodotos make it easy to extend an existing set of results to the next version of Linux. For example, for the **BlockLock** checker, suppose one starts with a set of already annotated reports for at least Linux 2.6.32. For Linux 2.6.33, as shown in Figure 25, the checker produces 70 reports of potential faults of which 51 are faults inherited from Linux 2.6.32, and 14 are false positives also present in that version. Herodotos would annotate these reports automatically, leaving only five reports to be annotated by the user. In this case, four are faults. Overall, for Linux 2.6.33 there are 232 new reports to consider. Half of these are for **Float**, but of these most follow the same pattern, and can be dealt with in a few minutes.

6. Limitations

The main limitations of our work are in the choice of faults considered and the definition of the checkers. We have focused on the same kinds of faults as Chou *et al.*, to be able to assess the changes in Linux since their work. Current fault finding tools, including Coccinelle, are able to find other kinds of faults, such as memory leaks. The considered set of faults also does not include concurrency faults, which are becoming increasingly important with the prevalence of multicore architectures. A recent study of concurrency faults in infrastructure software, however, has shown that over 20% of deadlocks are caused by a thread reacquiring a resource it already holds [18], amounting to a double lock, as detected by our **Lock** checker.

Our checkers could also be improved to reduce the number of false positives. In particular, as we have seen for **Float** in Section 5.6, some kinds of false positives are due to recurring patterns specific to certain Linux files. Taking these patterns into account in the checkers would avoid generating large numbers of trivial false positive reports. Finally, we have tried to be conservative in our identification of real faults, and this may have lead to an underestimation of their number. By making our results available in a public archive, we hope to benefit from feedback from the Linux community to improve our classification strategies.

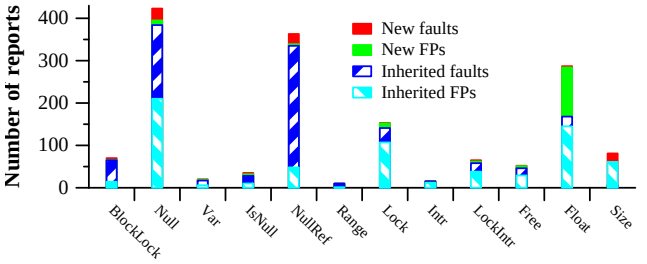


Figure 25. Reports generated for Linux 2.6.33

7. Related Work

Chou *et al.* briefly considered OpenBSD, as well as Linux [5]. Because of its wider use and more active development, we have focused only on Linux instead, comparing the properties of old and new versions.

In previous work, Palix *et al.* have used Coccinelle to conduct a study of faults in versions of Linux and several other open source projects released between 2005 and 2009 [25]. They proposed Herodotos, in order to correlate the faults between releases. They did not consider fault kinds requiring an interprocedural analysis, nor did they consider lock-related faults. As shown in Figure 7, these are among the most prevalent and also have high impact. Lawall *et al.* proposed a methodology allowing to find interprocedural faults in Linux code, but consider only a single Linux version [15].

Israeli and Feitelson have studied 810 versions of the Linux kernel, from Linux 1.0 to Linux 2.6.25 [14]. They considered traditional source-code metrics to measure complexity [19] and maintainability [23], rather than actual numbers of faults. They found that the complexity per Linux function is decreasing, and the maintainability is increasing. Nevertheless, they did find that `arch` and `drivers` do contain some very high complexity functions, typically interrupt handlers or `ioctl` functions, and that `arch` and `drivers` code is somewhat less maintainable than the code in other parts of the kernel. Their work is complementary to ours, and reaches some of the same conclusions.

Song *et al.* have studied the reasons for software hangs in open source infrastructure software, such as MySQL and Apache [32]. They focused on bug reports rather than analysis of the source code. They found that most types of concurrency faults are fixed on average within 100 days. Nevertheless, they did not take into account the time elapsed between the introduction of the fault and the time when it was first detected, and so the actual fault lifetime may be more in line with what we have observed. Lu *et al.* also considered concurrency faults in infrastructure software, primarily focusing on the kinds of tools that would be helpful to address them [18].

8. Conclusion

During the last 10 years, much of the research in operating system reliability has been predicated on the assumption that drivers are the main problem. The first major result of our study is that while `drivers` still has the largest number of faults in absolute terms, it no longer has the highest fault rate in Linux kernel code, having been supplanted by the HAL. The second major result of our study is that even though faults are continually being introduced, the overall code quality is improving. Our work thus shows the importance of being able to periodically repeat the study of faults in source code in order to revise research priorities as the fault patterns change in response to research efforts. Because the priorities of individuals and individual institutions change over time, the need to repeat

such studies implies that the tools and other data required must be available in an public archival repository.

Our study also shows that tools, while used, are under-exploited. Tools are indeed available to find all of the fault kinds considered in this paper. The fact that these kinds of faults remain and have a relatively long lifespan suggests that research is needed on how to design tools that are better integrated into the Linux development process. Another potential problem is the reactiveness of maintainers. Indeed, some services have no maintainer, but remain in the kernel source tree. This may somewhat artificially increase the number of faults. Still, any such faults can impact anyone who uses the code.

Our study has identified 736 faults in Linux 2.6.33, including RCU faults, some of which have not yet been corrected in the current developer snapshot, `linux-next`. We have submitted a number of patches based on our results.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Frans Kaashoek, for their feedback. We also thank Emmanuel Cecchet and Willy Zwaenepoel for comments on an earlier version of this paper. This work was partially supported by the Agence Nationale de la Recherche (France) under the contract ANR-09-BLAN-0158-01 and the Danish Research Council for Technology and Production Sciences.

References

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07*, pages 43–48, San Diego, CA, June 2007.

[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] How is the department of homeland security involved?, 2009. `http://scan.coverity.com/faq.html`.

[4] Checkpatch. `http://www.codemonkey.org.uk/projects/checkpatch/`.

[5] 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.

[6] Comedi: Linux Control and Mesurement Device Interface. `http://www.comedi.org/`.

[7] J. Corbet. The age of kernel code in various subsystems, Feb. 2010. `http://lwn.net/Articles/374622/`.

[8] J. Corbet. How old is our kernel?, Feb. 2010. `http://lwn.net/Articles/374574/`.

[9] Static source code analysis, static analysis, software quality tools by Coverity Inc. `http://www.coverity.com/`, 2008.

[10] A. Depoutovitch and M. Stumm. Otherworld – giving applications a chance to survive OS kernel crashes. In *ACM EuroSys*, pages 181–194, Paris, France, Apr. 2010.

[11] Fedora project, 2010. `http://fedoraproject.org/`.

[12] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Fault isolation for device drivers. In *2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 33–42, Estoril, Portugal, June 2009.

[13] IEEE std 982.2-1988 IEEE guide for the use of IEEE standard dictionary of measures to produce reliable software, 1988.

[14] A. Israeli and D. G. Feitelson. The Linux kernel as a case study in software evolution. *Journal of Systems and Software*, 83(3):485–501, 2010.

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

[16] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–315, Lisbon, Portugal, Sept. 2005.

[17] Lkml: The Linux kernel mailing list. `http://www.tux.org/lkml/`.

[18] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–339, Seattle, WA, USA, Mar. 2008.

[19] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, July 1976.

[20] P. E. McKenney and J. Walpole. Introducing technology into the Linux kernel: a case study. *ACM SIGOPS Operating Systems Review*, 42(5):4–17, 2008.

[21] J. C. Munson and S. G. Elbaum. Code churn: A measure for estimating the impact of code change. In *International Conference Software Maintenance (ICSM)*, pages 24–31, 1998.

[22] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *27th International Conference on Software Engineering (ICSE)*, pages 284–292, St. Louis, Missouri, USA, May 2005.

[23] P. Oman and J. Hagemeister. Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, 24(3):251–266, 1994.

[24] 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.

[25] 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*, pages 169–180, Rennes and Saint Malo, France, Mar. 2010.

[26] N. Palix, J. L. Lawall, and G. Muller. Herodotos: A tool to expose bugs' lives. Research report RR-6984, INRIA, July 2009.

[27] N. Palix, S. Saha, G. Thomas, C. Calvès, J. Lawall, and G. Muller. Database of Faults in Linux: Ten Years Later, Aug. 2010. `http://hal.inria.fr/docs/00/50/92/56/ANNEX/10years.sql.pg_dump`.

[28] N. Palix, S. Saha, G. Thomas, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. Research report RR-7357, INRIA, Aug. 2010.

[29] N. Palix, S. Saha, G. Thomas, C. Calvès, J. Lawall, and G. Muller. Website of Faults in Linux: Ten Years Later, Dec. 2010. `http://faultlinux.lip6.fr/`.

[30] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., 2001.

[31] D. Searls. Sparse, Linus & the Lunatics, Nov. 2004. Available at `http://www.linuxjournal.com/article/7272`.

[32] X. Song, H. Chen, and B. Zang. Why software hangs and what can be done with it. In *International Conference on Dependable Systems and Networks (DSN 2010)*, Chicago, IL, USA, June 2010.

[33] Sparse. `https://sparse.wiki.kernel.org/index.php/Main_Page`.

[34] B. Spencer. Local kernel exploit in /dev/net/tun. `http://grsecurity.net/~spender/cheddar_bay.tgz`.

[35] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4):333–360, 2006.

[36] The Kernel Janitors. Smatch, the source matcher, 2010. Available at `http://smatch.sourceforge.net`.

[37] Ubuntu, 2010. `http://www.ubuntu.com/`.

[38] D. Wheeler. Flawfinder home page. Web page: `http://www.dwheeler.com/flawfinder/`, Oct. 2006.

[39] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.

Appendix

Locking functions: {mutex,spin,read,write}_lock, {mutex,spin,read,write}_trylock

Interrupt disabling functions: cli, local_irq_disable

Functions combining both: {read,write,spin}_lock_irq, {read,write,spin}_lock_irqsave, local_irq_save, save_and_cli