Maintainability of the kernels of open-source operating systems: A comparison of Linux to FreeBSD, NetBSD, and OpenBSD

Liguo Yu,[a,1] Stephen R. Schach, [a,*] Kai Chen, [a] Gillian Z. Heller, [b] Jeff Offutt [c]

[a] *Department of Electrical Engineering and Computer Science, Vanderbilt University, Station B Box 351679, Nashville, TN 37235–1679, USA. E-mail: liguo.yu@vanderbilt.edu, srs@vuse.vanderbilt.edu, kai.chen@vanderbilt.edu*

[b] *Department of Statistics, Macquarie University, Sydney, NSW 2109, Australia. E-mail: gheller@efs.mq.edu.au*

[c] *Department of Information and Software Engineering, George Mason University, Fairfax, VA 22030, USA. E-mail: ofut@ise.gmu.edu*

* **Corresponding Author**
E-mail: srs@vuse.vanderbilt.edu
Tel: +1.615.322.2924
Fax: +1.615.343.5459

[1] Present address: Computer Science Department, Tennessee Technological University, P.O. Box 5101, Cookeville, TN 38505, USA. E-mail: yul@csc.tntech.edu

Maintainability of the kernels of open-source operating systems: A comparison of Linux to FreeBSD, NetBSD, and OpenBSD

Abstract

We compared and contrasted the maintainability of four open-source operating systems: Linux, FreeBSD, NetBSD, and OpenBSD. We used our categorization of common coupling in kernel-based software to highlight future maintenance problems. An unsafe definition is a definition of a global variable that can affect a kernel module if that definition is changed. For each operating system we determined a number of measures, including the number of global variables, the number of instances of global variables in the kernel and overall, as well as the number of unsafe definitions in the kernel and overall. We also computed the value of each our measures per kernel KLOC and per KLOC overall. For every measure and every ratio, Linux compared unfavorably to FreeBSD, NetBSD, and OpenBSD. Accordingly, we are concerned about the future maintainability of Linux.

Keywords: Maintainability; Common coupling; Definition-use analysis; Open-source software; Linux.

1. Introduction

The *coupling* between two modules is a measure of the degree of interaction between those modules and, hence, of the dependency between the modules. Certain types of coupling, especially common (global) coupling, are considered to present risks for software development, especially for maintenance (Briand et al., 1988; Troy and Zweben, 1981). Two modules are defined to be *common coupled* if they both reference the same global variable. The open-source software development life-cycle model can best be described as continuous maintenance, as encapsulated in the dictum "release early and often" (Raymond, 2000). Accordingly, it is important that open-source software should have as little common coupling as possible.

In a longitudinal study of 400 successive versions of Linux (Offutt, 2002; Schach and Offutt, 2002; Schach et al., 2002), we showed that the number of lines of code in each kernel module increases *linearly* with version number, whereas the number of instances of common coupling between each kernel module and all the other Linux modules grows *exponentially*. Both results were significant at the 99.99% level. In view of the deleterious effect of common coupling, we concluded that the resulting dependencies between modules had the potential of rendering Linux hard to maintain in the future.

At conference presentations of our result (Offutt, 2002; Schach and Offutt, 2002), our conclusion was challenged on two grounds:

1. Not all instances of common coupling are equally bad. For example, if global variables can be changed in just a few places, Linux would be considerably more maintainable than if global variables can be changed in many places.

We responded to the first argument by performing a definition-use analysis of Linux. Our results were based on a new categorization of common coupling in which certain forms of common coupling are safer than others from the viewpoint of maintainability (Yu et al., 2004).

2. All operating systems have to use global variables to achieve efficiency, and there is no alternative to the widespread utilization of common coupling in Linux.

This paper is our response to the second argument. Specifically, we show that there is far more common coupling in Linux than in three other open-source operating systems, FreeBSD, NetBSD, and OpenBSD (in what follows, we refer collectively to FreeBSD, NetBSD, and OpenBSD as "the three BSDs"). Furthermore, Linux has a higher proportion of unsafe forms of common coupling than the three BSDs. This leads us to conclude that is possible to structure Linux (and other operating systems) in a way that would increase its maintainability without sacrificing efficiency.

2. Kernel-Based Software

In 1969, Per Brinch Hansen developed a multiprogramming operating system for the RC 4000 computer. The operating system consisted of a kernel ("nucleus") that handled program execution, whereas input–output was performed by hardware-specific nonkernel modules (Brinch Hansen, 1970). This concept was subsequently extended to the architecture of database management systems (Härden, 1986). Today, the kernel–nonkernel architecture is widely used in the design of operating systems, database management systems, and other systems software, including games systems (Xbox365, 2004). In this paper, we refer to software that is comprised of a kernel together with optional nonkernel modules as *kernel-based software*.

The operating systems we consider this paper, Linux and the three BSDs, are all kernel-based. That is, every installation of the operating systems consists of all the kernel modules, together with a subset of the nonkernel modules specific to that installation. A characteristic of all four open-source operating systems is that the kernel modules are under strict control, whereas users are encouraged to write nonkernel modules, for example, for specific architectures or hardware devices.

All four open-source operating systems are written in the programming language C. In this study, a *module* is defined to be a source code file (".c" file or ".h" file). The *size* of the product is measured in thousands of lines of code (KLOC), excluding comments.

Data regarding the number and total number of lines of code of kernel and nonkernel modules in the four operating systems are provided in Table 1. A key point is that the Linux kernel is far smaller than the kernels of the three BSDs, both with regard to the number of modules and the total number of lines of code.

Table 1.
The kernel and nonkernel structure of four open-source operating systems.

Version	Number of kernel modules	Number of nonkernel modules	Size of kernel (KLOC)	Total size (KLOC)
Linux 2.4.20	26	9,407	14.230	4,260.445
FreeBSD 5.1	131	3,353	108.475	1,793.294
NetBSD 1.6	85	11,527	64.554	3,329.809
OpenBSD 3.3	81	4,569	55.969	1,825.733

3. Module Dependencies

As stated in Section 1, the *coupling* between two units of a software system is a measure of the degree of interaction between those units and, hence, of the dependency between the units. Many different categorizations of coupling have been published (Offutt et al., 1993; Page-Jones, 1980; Schach, 2005; Stevens et al., 1974), but all agree that common coupling is undesirable.

In this paper, we consider the classical coupling category *common coupling*. It has been shown that, for a variety of coupling metrics, the stronger (more undesirable) the coupling, the greater the fault-proneness (Briand et al., 1988; Troy and Zweben, 1981). A major reason underlying this phenomenon is that dependencies within the code lead to regression faults.

Coupling has not yet been explicitly shown to be related to maintainability. However, there is as yet no precise definition of maintainability, and therefore there are no generally accepted metrics for maintainability. Nevertheless, if a module is fault-prone then it will have to undergo repeated maintenance, and these frequent changes are likely to compromise its maintainability.

Furthermore, these frequent changes will not always be restricted to the fault-prone module itself; it is not uncommon to have to modify more than one module to fix a single fault. Thus, the fault-proneness of one module can adversely affect the maintainability of a number of other modules. In other words, it is easy to believe that strong coupling can have a deleterious effect on maintainability (Yu et al., 2004).

We consider common coupling in this paper for four reasons:

- In a case study on multiversion real-time software, we showed that the vast majority of the strong coupling introduced during the maintenance phase was common coupling (Wang et al., 2001).
- As previously stated, there are many categorizations of coupling. In addition, there is controversy as to what precisely constitutes weak or strong coupling. However, every categorization we have seen includes a form of coupling that corresponds to classical common coupling, and it is unanimously agreed that common coupling is undesirable.
- The number of instances of common coupling between a module P and the other modules can change dramatically, even if module P itself never changes, an effect that has been called clandestine common coupling (Schach et al., 2003). For example, if modules P and Q both reference global variable global_var, then there is one instance of common coupling between module P and the other modules. But if 100 new modules that reference global variable global_var are written, then the number of instances of common coupling between module P and the other modules increases to 101, even though module P itself is unchanged.
- The effect of clandestine common coupling can be especially severe in the case of kernel-based software. For every other form of coupling, the only way that coupling can be

introduced between an existing kernel module and a new nonkernel module is to explicitly change the kernel module. But where there is common coupling, a new nonkernel module can be coupled to an existing kernel module simply by including a reference to a global variable in that new nonkernel module. Consequently, it is possible for common coupling within a kernel-based module to increase without the knowledge of the developers responsible for the kernel, that is, in a clandestine way.

4. Definition-Use Analysis

Suppose that a variable yyy is declared in a program. Every occurrence of that variable can then be categorized as either

(a) A *definition* (or *def*) of that variable, that is, an assignment of a value to that variable (for example, read (yyy) or yyy = 3); or

(b)A *use* of that variable, that is, an access to the current value of that variable (for example, x = yyy + 3 or if (yyy > 7) return).

Suppose that modules M_1 and M_2 are common coupled because they both reference global variable global_var. There are three possible situations:

(1) Only M_1 can change the value of global_var. That is, global_var is defined in M_1 but only used in M_2.

(2) Only M_2 can change the value of global_var. That is, global_var is defined in M_2 but only used in M_1.

(3) Both M_1 and M_2 can change the value of global_var. That is, global_var is defined in both M_1 and M_2.

Situations (1) and (2) pose less risk for maintenance than (3) because there are fewer dependencies between the two modules when only one can change the value of global_var. The dependencies are localized, thus effects of changes can be easily determined. When only one

module can define global_var, changes to the other module cannot affect the defining module. Furthermore, within a given module that can change global variable global_var, fewer places that can change global_var is better.

In def-use analysis, each instance of a variable is labeled as either a definition or a use of that variable. The next section describes how def-use analysis can be utilized to characterize common coupling in kernel-based software.

5. Categorization of Common Coupling in Kernel-Based Software

This section provides an overview of our categorization of common coupling in kernel-based software and the associated graphical notation. There are five separate categories, which are fully described in a previous paper (Yu et al., 2004).

5.1. Category-1 Global Variables

Consider Figure 1. It depicts two modules, M_1 and M_2. The outer rectangle denotes the kernel, so M_1 is a kernel module and M_2 is a nonkernel module. The arrow from M_1 to M_2 denotes that M_1 defines gv_1 (at least once) and M_2 uses gv_1 (at least once).

A *category-1 global variable* is defined in one or more kernel modules, but is not used in any kernel modules. (It is used in one or more nonkernel modules, but that is not important here.)

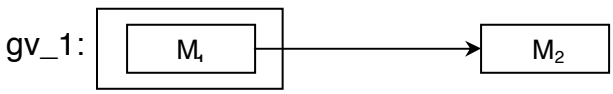

Figure 1: A category-1 global variable gv_1.

The key point is that a change to gv_1 inside the kernel cannot affect the rest of the kernel in any way. That is, a category-1 global variable is *kernel-to-kernel safe*, that is, a change to a kernel module cannot affect the kernel.

Now consider gv_1 from the viewpoint of a nonkernel module. Global variable gv_1 is not used in any kernel modules, so a change to gv_1 in a nonkernel module cannot affect a kernel module in any way. That is, a category-1 global variable is *nonkernel-to-kernel safe*, that is, a change to a nonkernel module cannot affect the kernel.

5.2. Category-2 Global Variables

Next, consider Figure 2, which depicts two kernel modules, M_1 and M_2, and global variable gv_2, which is defined in kernel module M_1 and used in kernel module M_2. A *category-2 global variable* is defined in one kernel module, and is used in one or more kernel modules. Category-2 global variables may be used in nonkernel modules, but that use is not important.

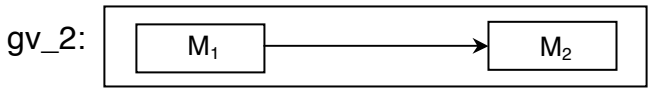

Figure 2: A category-2 global variable gv_2.

As with category 1, a modification to a category-2 global variable in a nonkernel module cannot affect a kernel module because there are no definitions of a category-2 global variable in a nonkernel module. That is, category-2 global variables are *nonkernel-to-kernel safe*. However, category-2 global variables are *kernel-to-kernel unsafe* because a change to the kernel module that defines the variable can affect the kernel module that uses it. By definition, however, a category-2 global variable is defined in only one kernel module, and thus it is *minimally kernel-to-kernel unsafe*.

5.3. Category-3 Global Variables

Now consider Figure 3, which depicts three kernel modules, M_1, M_2, and M_3, and global variable gv_3, which is defined in kernel modules M_1 and M_3 and used in kernel module M_2. A *category-3 global variable* is defined in more than one kernel module, and is also used in one or more kernel modules. Category-3 global variables may be used in nonkernel modules, but that use is not important.

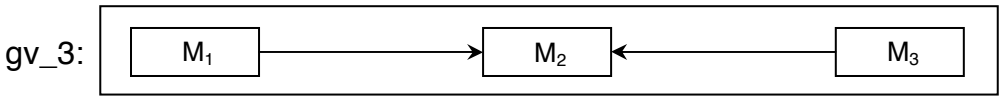

Figure 3: A category-3 global variable gv_3.

As with category 2, category-3 global variables are *nonkernel-to-kernel safe*. However, they are *kernel-to-kernel unsafe*. They are not *minimally kernel-to-kernel unsafe*, because a category-3 global variable is defined in more than one kernel module.

5.4. Category-4 Global Variables

Consider Figure 4, which depicts kernel module M_1 and nonkernel module M_2, and global variable gv_4. (Note that Figure 4 is the same as Figure 1, but with the direction of the arrow reversed). A *category-4 global variable* is defined in one or more nonkernel modules, and used in one or more kernel modules. As with category-2 and -3 global variables, uses in nonkernel modules are not important. Figure 4 depicts a category-4 global variable gv_4.

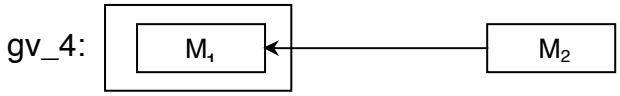

Figure 4: A category-4 global variable gv_4.

Category-4 global variables are highly undesirable. They are *kernel-to-kernel safe* but *nonkernel-to-kernel unsafe*. That is, a kernel module that uses a category-4 global variable is vulnerable to modifications to that global variable in a nonkernel module that defines the

variable. The principle of "separation of concerns" tells us that changes to nonkernel modules should not be able to affect kernel modules.

4.5. Category-5 Global Variables

Finally, consider Figure 5, which depicts kernel module M_1 and nonkernel module M_2, and global variable gv_5. A category-5 global variable is defined in one or more nonkernel modules, defined in one or more kernel modules, and used in one or more kernel modules. Figure 5 depicts a category-5 global variable.

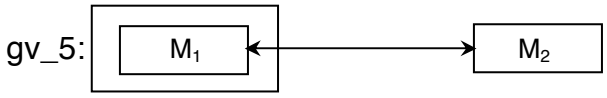

Figure 5: A category-5 global variable gv_5.

Category-5 global variables are both *kernel-to-kernel unsafe* and *nonkernel-to-kernel unsafe*. That is, a kernel module that contains a category-5 global variable is vulnerable to modifications to both a kernel module and a nonkernel module in which that global variable is defined. It is extremely difficult to minimize the impact of changes that involve category-5 global variables.

In summary, all global variables are unacceptable, but some are more unacceptable than others. Category-1 global variables are the least deleterious from the viewpoint of maintainability of the kernel, followed by categories 2 and 3, in that order. However, category-4 and category-5 global variables should be considered unacceptable. In what follows, we use the term *unsafe definition* to refer to a definition of a global variable that can affect a kernel module if that definition is changed. That is, an unsafe definition is a definition of a category-2, -3, or –5 global variable in a kernel module, or a definition of a category-4 or –5 global variable in a nonkernel module.

6. Common Coupling In Open-Source Operating Systems

We analyzed Linux and the three BSDs using our categorization of common coupling in kernel-based software. The Linux cross-referencing tool lxr was used to identify the global variables. For each global variable, lxr was then used to determine in which modules the global variable appears, and extract the corresponding lines of code. For each instance of a global variable, we manually checked whether it is a definition or use, a straightforward determination. To be sure that we had counted correctly, two researchers (Yu and Chen) performed the study independently. There were only a few discrepancies, all of which were clerical errors and therefore easy to reconcile.

An overview of our results is shown in Table 2. As shown in the table, Linux has 99 distinct global variables. Altogether, there are 1,022 instances of global variables in kernel modules. However, if multiple instances of a given global variable in a module are ignored, there are 193 unique instances of a global variable in kernel modules. The other entries are similar.

The rightmost two columns of Table 2 reveal that Linux has a disproportionately large number of instances of global variables in both kernel and nonkernel modules. The large number of instances of global variables in kernel modules, 1022, is surprising in view of the relatively small size of the Linux kernel, as shown in Table 1. However, from the viewpoint of maintainability, what must be considered is not the total number of instances of global variables but rather the breakdown of those instances by definitions and uses in each of the five categories described in Section 5. The data are given in Tables 3, 4, 5, and 6.

Table 2.
Global variables in open-source operating systems.

Operating system	Total number of global variables	Number of unique instances of a global variable in kernel modules	Number of unique instances of a global variable in nonkernel modules	Total number of instances of global variables in kernel modules	Total number of instances of global variables in nonkernel modules
Linux	99	193	2,808	1,022	14,088
FreeBSD	75	166	338	483	770
NetBSD	66	112	411	378	1,222
OpenBSD	75	122	268	343	521

Table 3.
Definitions and uses of global variables in Linux 2.4.20.

Category number	Number of global variables	Kernel modules			Nonkernel modules		
		Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses	Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses
1	23	25	35	–	220	0	389
2	28	76	36	208	1,041	–	4,437
3	4	10	25	15	91	–	302
4	24	27	–	65	66	40	171
5	20	55	180	458	1,390	1,627	7,122
Overall	99	193	276	746	2,808	1,667	12,421

Table 4.
Definitions and uses of global variables in FreeBSD 5.1.

Category number	Number of global variables	Kernel modules			Nonkernel modules		
		Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses	Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses
1	22	23	53	–	73	0	87
2	35	104	74	251	172	–	504
3	8	18	23	28	36	–	70
4	4	8	–	25	24	22	21
5	6	13	6	23	33	24	42
Overall	75	166	156	327	338	46	724

Table 5.
Definitions and uses of global variables in NetBSD 1.6.

Category number	Number of global variables	Kernel modules			Nonkernel modules		
		Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses	Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses
1	19	22	34	–	159	0	336
2	28	57	49	162	129	–	547
3	0	0	0	0	0	–	0
4	4	6	–	18	33	31	76
5	15	27	38	77	90	50	182
Overall	66	112	121	257	411	81	1,141

Table 6.
Definitions and uses of global variables in OpenBSD 3.3.

Category number	Number of global variables	Kernel modules			Nonkernel modules		
		Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses	Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses
1	22	22	31	–	63	0	107
2	27	61	49	148	71	–	131
3	6	14	18	20	10	–	18
4	10	11	–	25	75	29	111
5	10	14	14	38	49	55	70
Overall	75	122	112	231	268	84	437

7. Results and Statistical Analyses

Critical aspects of Tables 1 through 6 are summarized in Table 7 and depicted in Figures 6 and 7. From the rightmost two columns of the table, we see that Linux has far more unsafe definitions of global variables than any of the BSDs. Specifically, there are 241 unsafe definitions in the Linux kernel, which is only 14.230 KLOC in size, as opposed to 103, 87, and 81 definitions in FreeBSD, NetBSD, and OpenBSD, despite the fact that the kernels of the three BSDs are 7.6, 4.5, and 3.9 times larger than the Linux kernel.

Turning to nonkernel modules, the Linux nonkernel modules contain 1,667 unsafe definitions, as opposed to just 46, 81, and 84 for the three BSDs. The results of this and the previous paragraph are summarized in Table 8 and depicted in Figure 8.

Table 7.
Key aspects of Tables 1 through 6.

Operating system	Total number of global variables	Total number of instances of global variables		Number of instances of category-4 and -5 global variables		Percentage of instances of category-4 and -5 global variables	
		Kernel modules	Nonkernel modules	Kernel modules	Nonkernel modules	Kernel modules	Nonkernel modules
Linux	99	1,022	14,088	703	8960	69%	64%
FreeBSD	75	483	770	54	109	11%	14%
NetBSD	66	378	1,222	133	339	35%	28%
OpenBSD	75	343	521	77	265	22%	51%

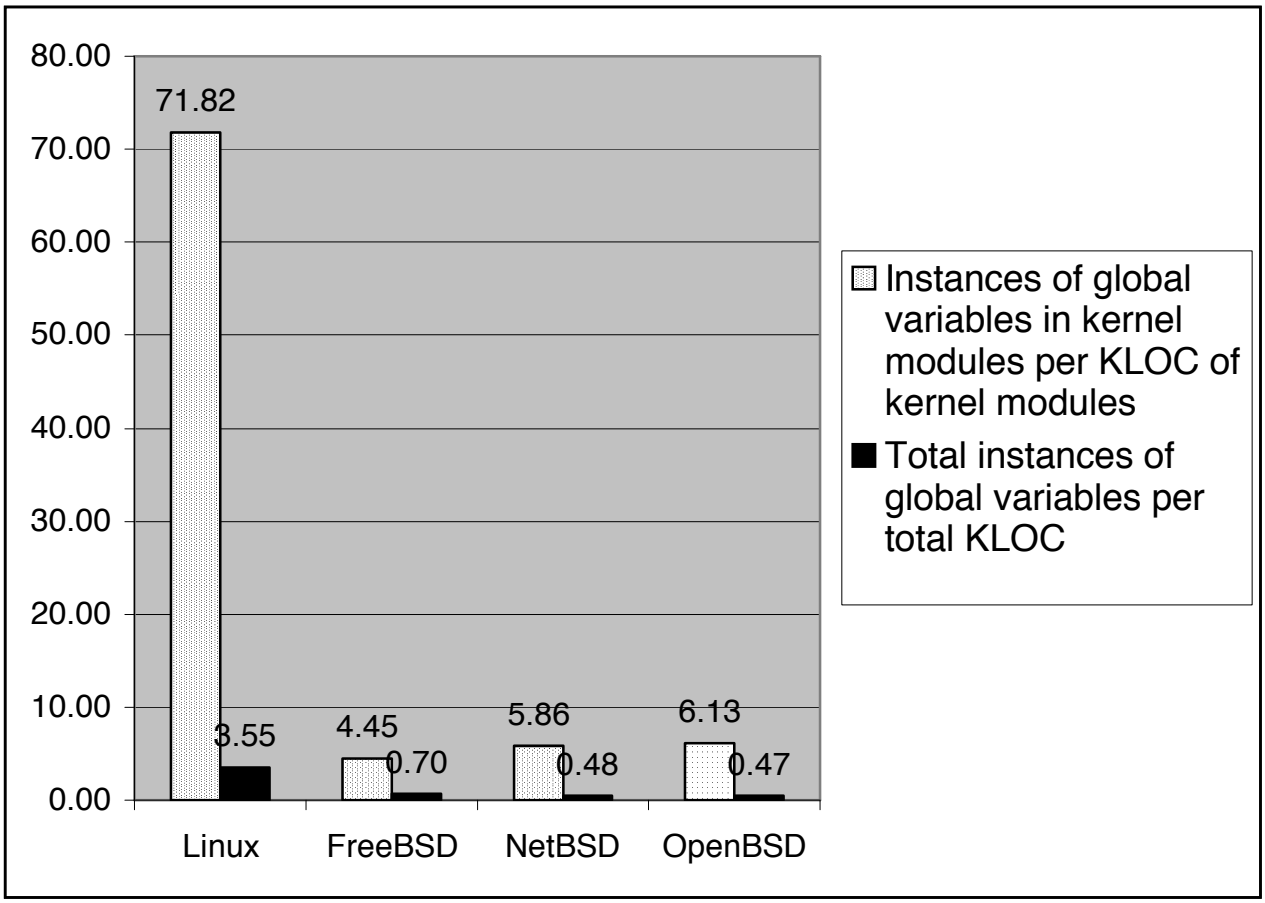

Figure 6: Instances of global variables per KLOC.

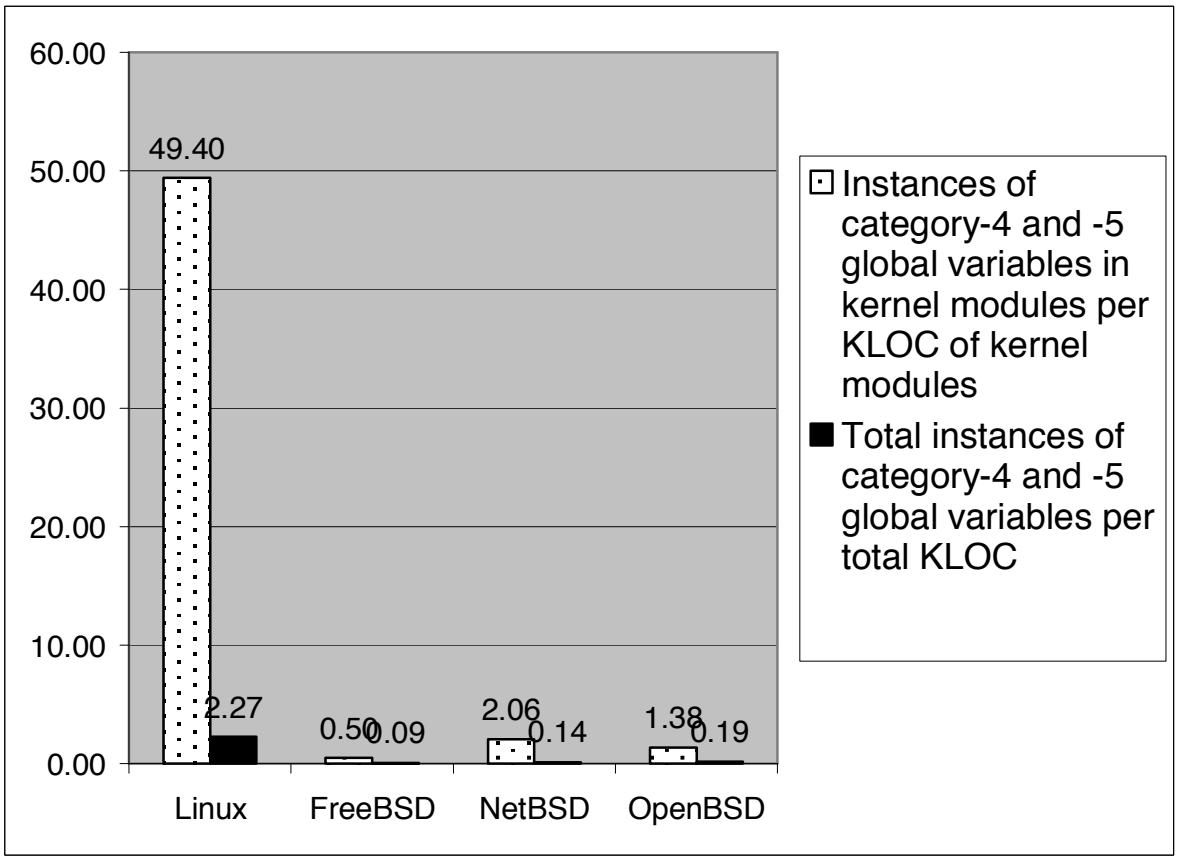

Figure 7: Instances of category-4 and –5 global variables per KLOC.

Table 8.
Unsafe definitions of global variables.

Operating system	Total number of global variables	Size of kernel (KLOC)	Total size (KLOC)	Number of unsafe definitions		Number of unsafe definitions per KLOC	
				Kernel modules	Nonkernel modules	Kernel modules	Nonkernel modules
Linux	99	14.230	4,260.445	241	1,667	16.936	0.393
FreeBSD	75	108.475	1,793.294	103	46	0.950	0.027
NetBSD	66	64.554	3,329.809	87	81	1.348	0.025
OpenBSD	75	55.969	1,825.733	81	84	1.447	0.047

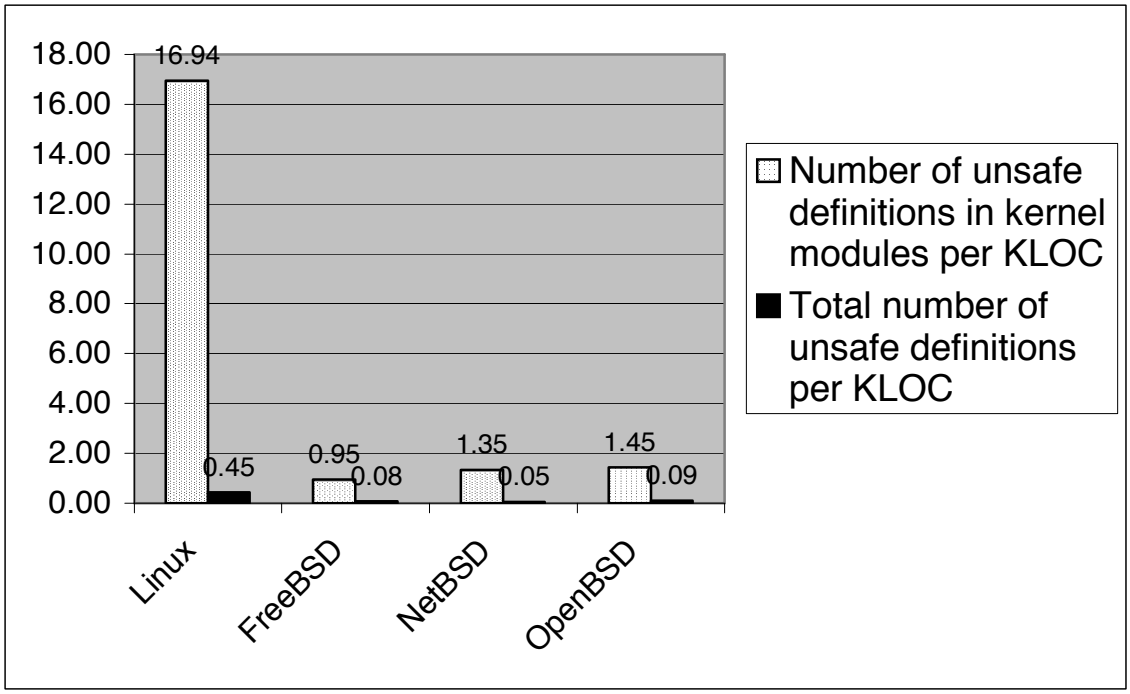

Figure 8. Number of unsafe definitions per KLOC.

Our results show that Linux has many more instances of global variables and far more instances of unsafe definitions than the three BSDs. In order to quantify these differences, we performed two sets of statistical tests. First, we tested differences between the three BSDs, in order to be able to pool the results. On finding that the BSDs were statistically different, we had to compare Linux with each of the BSDs separately; these constituted our second set of statistical tests.

In more detail, first we tested the following two null hypotheses:

- *H_{01}: There is no significant difference between the distribution of the total number of instances of global variables in the three BSD-based open-source operating systems and the distribution of the size (in KLOC) of those operating systems.*

- *H_{02}: There is no significant difference between the distribution of the total number of unsafe definitions in the three BSD-based open-source operating systems and the distribution of the size (in KLOC) of those operating systems.*

We constructed the relevant contingency tables and performed the chi-square test of independence. In both cases, the corresponding *P*-value was less than 0.0001, so we rejected the corresponding null hypotheses at the 99.99% level of significance. Accordingly, we could not pool the BSD data, but had to compare Linux separately with each BSD.

In our second set of statistical tests, we compared Linux pairwise with each BSD. First we considered FreeBSD. We tested the following four null hypotheses:

- *H_{03}: There is no significant difference between the distribution of the total number of instances of global variables in Linux and FreeBSD, and the distribution of the size (in KLOC) of those two operating systems.*

- *H_{04}: There is no significant difference between the distribution of the number of instances of global variables in the Linux and FreeBSD kernels, and the distribution of the size (in KLOC) of the two operating system kernels.*

- *H_{05}: There is no significant difference between the distribution of the total number of unsafe definitions of global variables in Linux and FreeBSD, and the distribution of the size (in KLOC) of those two operating systems.*

- *H_{06}: There is no significant difference between the distribution of the number of unsafe definitions of global variables in the Linux and FreeBSD kernels, and the distribution of the size (in KLOC) of the two operating system kernels.*

We constructed the relevant contingency tables and performed the chi-square tests. In each case, the *P*-value was less than 0.0001, so we rejected all four null hypotheses at the 99.99% level of significance. We concluded that there are significant differences between Linux and FreeBSD with respect to:

- The total number of instances of global variables per KLOC;
- The number of instances of global variables in the kernel per KLOC;
- The total number of unsafe definitions of global variables per KLOC; and
- The number of unsafe definitions of global variables in the kernel per KLOC.

We remark that we could not directly test any of the above four ratios, because the chi-square test can be applied to only counts (numbers) like number of global variables and number of lines of code, and not to ratios like the number of global variables per KLOC.

We then compared Linux with NetBSD and OpenBSD, and obtained identical results. That is, we rejected the corresponding sets of null hypotheses at the 99.99% level of significance. These differences are due to the fact that Linux has many more instances of unsafe definitions of global variables than FreeBSD, OpenBSD, or NetBSD both in kernel and nonkernel modules, as reflected in Table 8.

There are some clear threats to the validity of our results. First, as with any study of four software systems, there is an external threat in that these results cannot be guaranteed to apply to other software systems. However, our goal (as expressed at the end of Section 1) was to decide

whether Linux is less maintainable than the three BSDs. We can think of no reason why equally credible results will not be obtained when our method is applied to other kernel-based software systems.

There are also two internal threats to validity. Our results rely on counting various data definitions and uses, and then categorizing them. We used an automated tool to reduce the threat from inaccurate counting. The categorization is somewhat more problematic because it was done by hand using judgment that is inherently subjective. To ameliorate this internal threat, the categorization was performed by two different individuals and the results were then reconciled, as described in Section 6.

8. Conclusions

This paper is an application of a new classification of common coupling. The classification is based on the definition-use characteristics of global variables in kernel and nonkernel modules. Our results show there is considerably more common coupling in Linux than in FreeBSD, NetBSD, and OpenBSD. In particular, Linux contains far more unsafe definitions of global variables, especially definitions of category-4 and -5 global variables. Consequently, from the viewpoint of common coupling, we are concerned that Linux will be more difficult to maintain in the future than the three BSDs.

Linux compares unfavorably to the three BSDs with respect to every measure we considered, including:

- Total number of global variables
- Total number of instances of global variables in the kernel and overall
- Total number of instances of global variables per KLOC in the kernel and overall

- Number of unsafe definitions of global variables in the kernel and overall
- Number of unsafe definitions of global variables per KLOC in the kernel and overall
- Number of instances of category-4 and -5 global variables in kernel and nonkernel modules
- Number of instances of category-4 and -5 global variables per KLOC in the kernel and overall
- Percentage of instances of category-4 and -5 global variables in kernel and nonkernel modules

We are also concerned that maintainability is not being sufficiently considered by the Linux development team. The size of Linux is continuously growing (version 2.4.40 comprises over 4 million lines of code), yet there has not yet been a large-scale restructuring.

As we noted in Section 1, this paper is a response to the claim that the widespread usage of global variables in Linux is *necessary* for efficiency. Our results show that the three open-source BSDs have far fewer instances of global variables than Linux, and that it therefore is possible to design an efficient operating system without a plethora of global variables. As we have pointed out, common coupling in general and category-4 and category-5 global variables in particular, are potential threats to the maintainability of the kernel. We believe that Linux developers need to consider controlling the use of global variables in order to balance maintainability and system efficiency.

Acknowledgment

This work was sponsored in part by the National Science Foundation under grant number CCR–0097056.

References

Briand, L.C., Daly, J., Porter, V., Wüst, J., 1998. A comprehensive empirical validation of design measures for object-oriented systems. In: Proceedings of the 5th International Software Metrics Symposium, Bethesda, Maryland, pp. 246–257.

Brinch Hansen, P., 1970. The nucleus of a multiprogramming system. Communications of the ACM 4 (4), 238–241.

Härden, T., 1986. New Approaches to Object Processing in Engineering Databases, Abstract in: Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, CA, p. 217, September 1986. Full text available at: https://portal.acm.org/poplogin.-cfm?dl=ACM&coll=GUIDE&comp_id=318875&want_href=delivery%2Ecfm%3Fid%3D318875%26type%3Dpdf&CFID=19366447&CFTOKEN=6322313.

Offutt, J., 2002. Open-source software: more or less secure and reliable? Panel at the International Symposium on Software Reliability Engineering (ISSRE '02), Annapolis, MD.

Offutt, J., Harrold, M. J., Kolte, P., 1993. A software metric system for module coupling. Journal of Systems and Software 20 (3), 295–308.

Page-Jones, M., 1980. The Practical Guide to Structured Systems Design. Yourdon Press, New York.

Raymond E. S., 2000. The cathedral and the bazaar. www.catb.org/~esr/writings/cathedral-bazaar/

Schach, S. R., 2005. Object-Oriented and Classical Software Engineering, Sixth Edition. McGraw-Hill, Boston, MA.

Schach, S. R., Jin, B., Wright, D. R., Heller, G. Z., Offutt, A. J., 2002. Maintainability of the Linux kernel. IEE Proceedings—Software 149 (2), 18–23.

Schach, S. R., Jin, B., Wright, D. R., Heller, G. Z., Offutt, J., 2003. Quality impacts of clandestine common coupling. Software Quality Journal 11 (3), 211–218.

Schach, S. R., Offutt, J., 2002. On the nonmaintainability of open-source software. In: Proceedings of the 2nd Workshop on Open-Source Software Engineering, Orlando, FL, pp. 47–49.

Schach, S.R., Jin, B., Wright, D.R., Heller, G.Z., Offutt, A.J., 2003. Quality impacts of clandestine common coupling. Software Quality Journal 11 (7), 211–218.

Stevens, W.P., Myers, G.J., Constantine, L.L., 1974. Structured design. IBM Systems Journal 13 (2), 38–54.

Troy, D.A., Zweben, S.H., 1981. Measuring the quality of structured design. Journal of Systems and Software 2 (2), 113–120.

Wang, S., Schach, S. R., Heller, G. Z., 2001. A case study in repeated maintenance. J. Software Maintenance and Evolution: Research and Practice 13 (2), 127–141.

Xbox365, 2004. Xbox system software overview. www.xbox365.com/stories/xdkcomplete.-shtml

Yu, L., Schach, S. R., Chen, K., Offutt, J., 2004. Categorization of common coupling and its application to the maintainability of the Linux kernel. IEEE Transactions on Software Engineering 30 (10), 694–706.

Figure Captions

Figure 1: A category-1 global variable `gv_1`.

Figure 2: A category-2 global variable `gv_2`.

Figure 3: A category-3 global variable `gv_3`.

Figure 4: A category-4 global variable `gv_4`.

Figure 5: A category-5 global variable `gv_5`.

Figure 6: Instances of global variables per KLOC.

Figure 7: Instances of category-4 and –5 global variables per KLOC.

Figure 8. Number of unsafe definitions per KLOC.

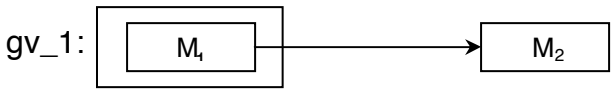

Figure 1: A category-1 global variable gv_1.

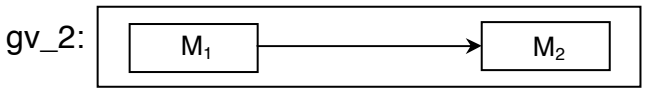

Figure 2: A category-2 global variable gv_2.

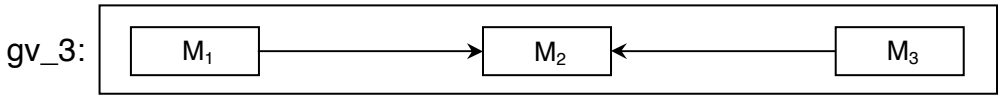

Figure 3: A category-3 variable gv_3.

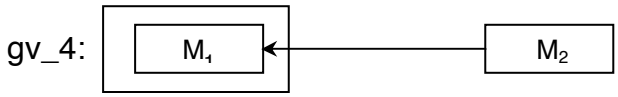

Figure 4: A category-4 variable gv_4.

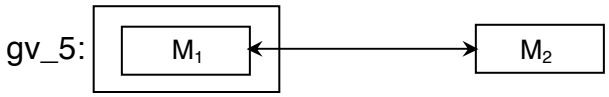

Figure 5: A category-5 global variable gv_5.

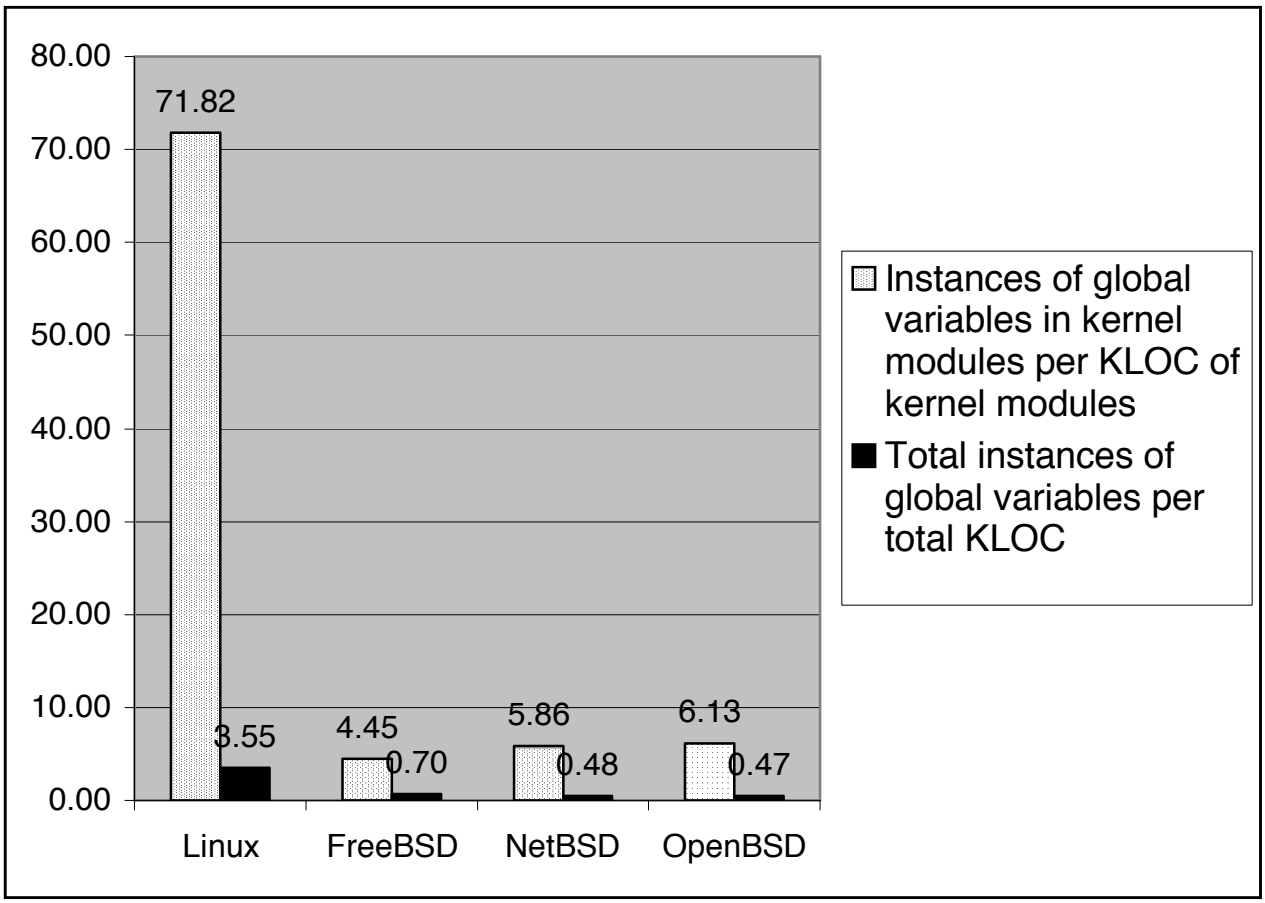

Figure 6: Instances of global variables per KLOC.

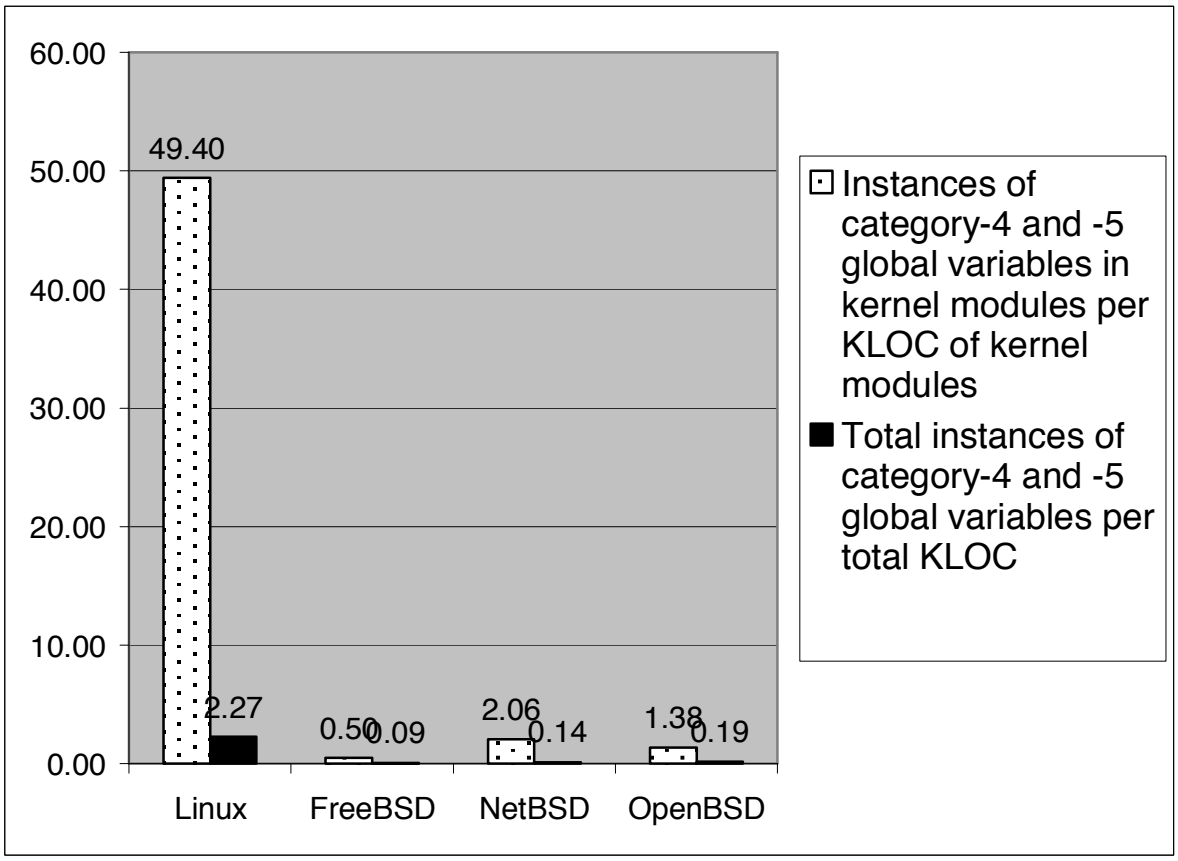

Figure 7: Instances of category-4 and –5 global variables per KLOC.

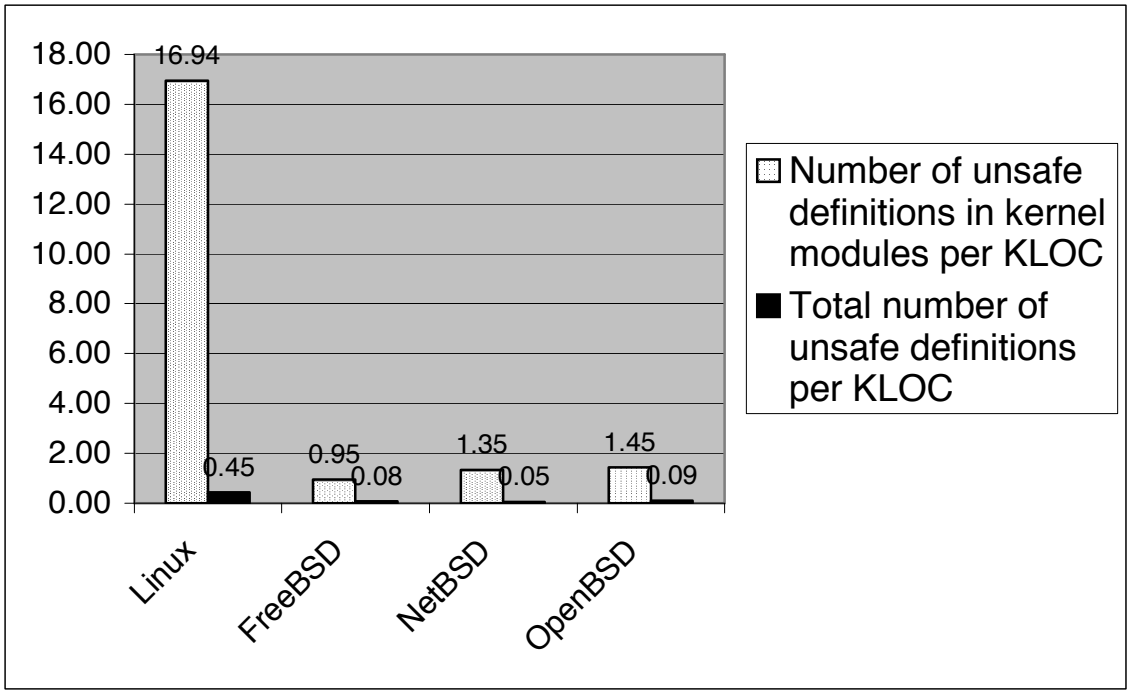

Figure 8. Number of unsafe definitions per KLOC.

Brief biographies

Liguo Yu obtained the PhD degree in computer science from Vanderbilt University. He is an assistant professor of computer science at Tennessee Technological University. His research concentrates on the maintainability of the Linux kernel and open-source software development. Before working in software engineering, his research focused on modeling and system identification, and fault detection and isolation of hybrid systems. E-mail: liguo.yu@vanderbilt.edu

Stephen R. Schach is an Associate Professor in the Department of Electrical Engineering and Computer Science at Vanderbilt University, Nashville, Tennessee. Steve is the author of over 115 refereed research papers. He has written ten software engineering textbooks, including *Object-Oriented and Classical Software Engineering*, Sixth Edition (McGraw-Hill, 2005). He consults internationally on software engineering topics. Steve's research interests are in software maintenance and open-source software engineering. He obtained his PhD from the University of Cape Town. E-mail: srs@vuse.vanderbilt.edu.

Kai Chen is a PhD student in the Department of Electrical Engineering and Computer Science at Vanderbilt University. His current research interests include development and maintenance of open-source software, embedded software design, domain-specific modeling language design, formal methods, model verification, and model-integrated computing. E-mail: kai.chen@vanderbilt.edu

Gillian Z. Heller is an Associate Professor in the Department of Statistics at Macquarie University, Sydney, Australia, where she has been for the last 12 years. Her B.Sc. (Hons) and Ph.D. degrees are in Mathematical Statistics, from the University of Cape Town, South Africa, and her M.Sc. in Operations Research is from the University of South Africa. Gillian's research interests are in discrete distribution theory, with applications in biostatistics. E-mail: gheller@efs.mq.edu.au.

Jeff Offutt is an Associate Professor of Information and Software Engineering at George Mason University. His current research interests include software testing, analysis and testing of web applications, object-oriented program analysis, module and integration testing, formal methods, and software maintenance. He has published over eighty research papers in refereed computer science journals and conferences. Offutt was program chair for ICECCS 2001 and is on the editorial boards for the *IEEE Transactions on Software Engineering*, the *Journal of Software Testing, Verification and Reliability*, the *Journal of Software and Systems Modeling*, and the *Software Quality Journal*. He received the Best Teacher Award from the School of Information Technology and Engineering in 2003. Offutt received a PhD degree in Computer Science from the Georgia Institute of Technology, and is a member of the ACM and IEEE Computer Society. E-mail: ofut@ise.gmu.edu.