A Tale of Four Kernels

Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Patision 76, GR-104 34 Athens, Greece
dds@aueb.gr

ABSTRACT

The FreeBSD, GNU/Linux, Solaris, and Windows operating systems have kernels that provide comparable facilities. Interestingly, their code bases share almost no common parts, while their development processes vary dramatically. We analyze the source code of the four systems by collecting metrics in the areas of file organization, code structure, code style, the use of the C preprocessor, and data organization. The aggregate results indicate that across various areas and many different metrics, four systems developed using wildly different processes score comparably. This allows us to posit that the structure and internal quality attributes of a working, non-trivial software artifact will represent first and foremost the engineering requirements of its construction, with the influence of process being marginal, if any.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software process models*; D.2.8 [**Software Engineering**]: Metrics—*Product metrics*

General Terms

Measurement

1. INTRODUCTION

Arguments regarding the efficacy of open source products and development processes often employ external quality attributes [21], anecdotal evidence [17], or even plain hand waving [13]. Although considerable research has been performed on open source artifacts and processes [10, 36, 7, 9, 41, 3, 32], the direct comparison of open source products with corresponding proprietary systems has remained an elusive goal. The recent open-sourcing of the Solaris kernel and the distribution of large parts of the Windows kernel source code to research institutions has provided us with a window of opportunity to perform a comparative evaluation between the code of open source and proprietary systems.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

Here I report on code quality metrics I collected from four large industrial-scale operating systems: FreeBSD, Linux, OpenSolaris, and the Windows Research Kernel (WRK). The main contribution of this research is the finding that there are no significant across-the-board code quality differences between four large working systems, which have been developed using various open-source and proprietary processes. An additional contribution involves the proposal of numerous code quality metrics for objectively evaluating software written in C. Although these metrics have not been empirically validated, they are based on generally accepted coding guidelines, and therefore represent the rough consensus of developers concerning desirable code attributes.

2. MATERIALS AND TOOLS

The key properties of the systems I examine appear in Table 1(A), while Figure 1 shows their history and genealogy. All four systems started their independent life in 1991–1993. Two of the systems, FreeBSD and OpenSolaris, share common ancestry that goes back to the 1978 1BSD version of Unix. FreeBSD is based on BSD/Net2: a distribution of the Berkeley Unix source code that was purged from proprietary AT&T code. The code behind OpenSolaris goes further back, tracing the origins of its code back to the 1973 version of Unix, which was the first written in C [29, p. 54]. In 2005 Sun released most of the Solaris source code under an open-source license.

Linux was developed from scratch in an effort to build a more feature-rich version of Tanenbaum's teaching-oriented, POSIX-compatible Minix operating system [39]. Thus, although Linux borrowed ideas from both Minix and Unix, it did not derive from their code [40].

The intellectual roots of Windows NT go back to DEC's VMS through the common involvement of the lead engineer David Cutler in both projects. Windows NT was developed as Microsoft's answer to Unix, initially as an alternative of IBM's OS/2, and later as a replacement of the 16-bit Windows code base. The Windows Research Kernel (WRK) whose code I examine in this paper includes major portions of the 64-bit Windows kernel, which Microsoft distributes for research use [27]. The kernel is written in C with some small extensions. Excluded from the kernel code are the device drivers, and the plug-and-play, power management, and virtual DOS subsystems. The missing parts explain the large size difference between the WRK and the other three kernels.

Although all four systems I examine are available in source code form, their development methodologies are markedly different. OpenSolaris and WRK have been developed as pro-

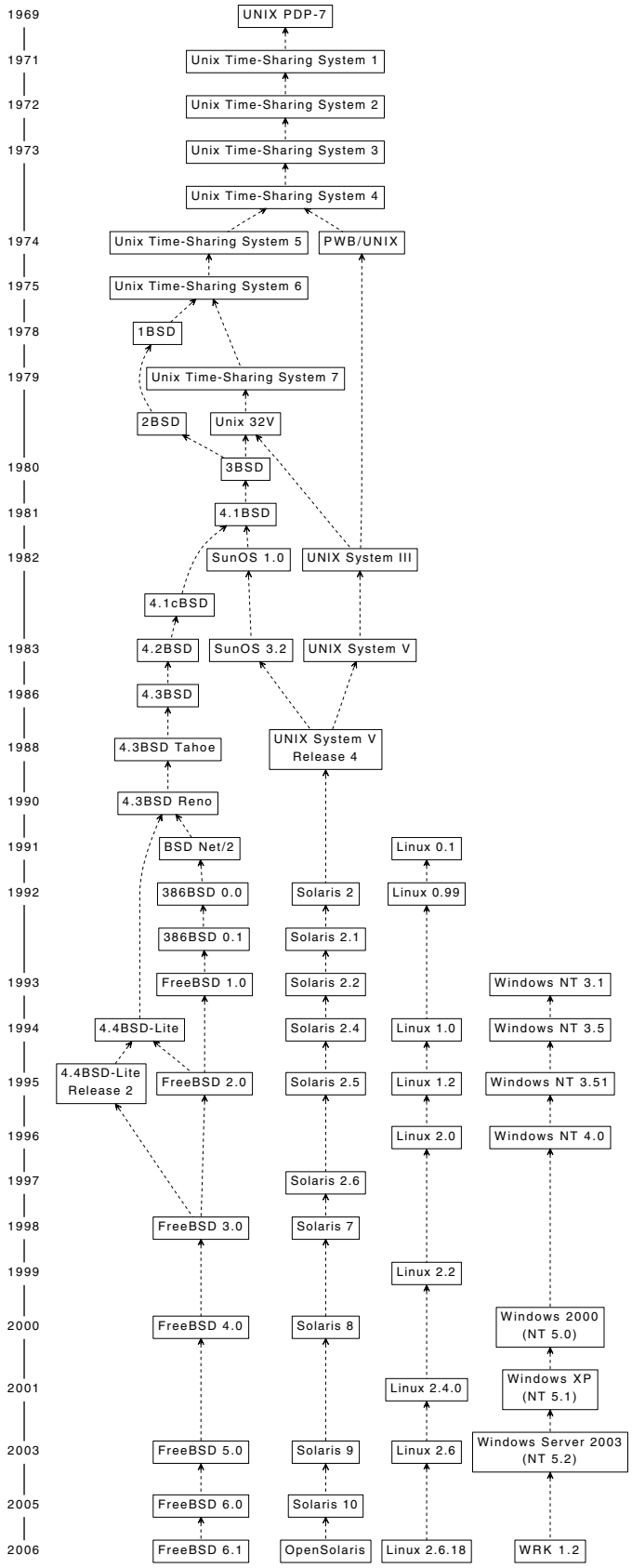

Figure 1: History of the examined systems.

prietary systems. (OpenSolaris has a very short life as an open-source project, and therefore only minimal code could have been contributed by developers outside Sun in the snapshot I examined.) Furthermore, Solaris has been developed with emphasis on a formal process [6], while the development of Windows NT employed more lightweight methods [5, pp. 223, 263, 273–274]. FreeBSD and Linux are both developed using open source development methods [8], but their development processes are also dissimilar. FreeBSD is mainly developed by a non-hierarchical group of about 220 committers who have access to a shared CVS-based software repository. In contrast, Linux's developers are organized in a four tier pyramid. At the bottom two levels thousands of developers contribute patches to about 560 subsystem maintainers. At the top of the pyramid Linus Torvalds, assisted by a group of trusted lieutenants, is the sole person responsible for adding the patches to the Linux tree [28].

Most of the metrics reported here were calculated by issuing SQL queries on a relational database containing the code elements comprising each system (identifiers, tokens, functions, files, comments, and their relationships).[1] The database for each system was constructed by running the *CScout* refactoring browser [33, 34] on the specified configurations of the corresponding operating system. (Each configuration comprises different macro definitions, and will therefore process code in a different way.) To process the source code of a complete system *CScout* must be given a configuration file that will specify the precise environment used for processing each compilation unit. For the FreeBSD and the Linux kernels I constructed this configuration file by instrumenting proxies for the GNU C compiler, the linker and some shell commands. These recorded their arguments in a format that could then be used to construct a *CScout* configuration file. For OpenSolaris and the WRK I simply performed a full build for the investigated configurations, and then processed the compilation and linking commands appearing in the build's log.

In order to limit bias introduced in the selection of metrics, I chose and defined the metrics I would collect before setting up the mechanisms to measure them. This helped me avoid the biased selection of metrics based on results I obtained along the way. However, this *ex ante* selection also resulted in many metrics—like the number of characters per line—that did not supply any interesting information, failing to provide a clear winner or loser. On the other hand my selection of metrics was not completely blind, because at the time I designed the experiment I was already familiar with the source code of the FreeBSD kernel and had seen source code from Linux, the 9th Research Edition Unix, and Windows device drivers.

Other methodological limitations of this study are the small number of (admittedly large and important) systems studied, the language specificity of the employed metrics, and the coverage of only maintainability and portability from the space of all software quality attributes. This last limitation means that the study fails to take into account the large and important set of quality attributes that are typically determined at runtime: functionality, reliability, usability, and efficiency. However, these missing attributes are affected by configuration, tuning, and workload selec-

[1]The databases (141 million records), their schema, and the corresponding queries are available online at `http://www.dmst.aueb.gr/dds/sw/4kernel/`.

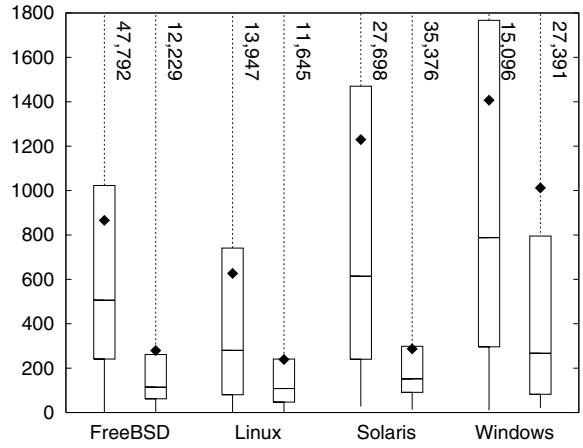

Figure 2: File length (in lines) of C files and headers.

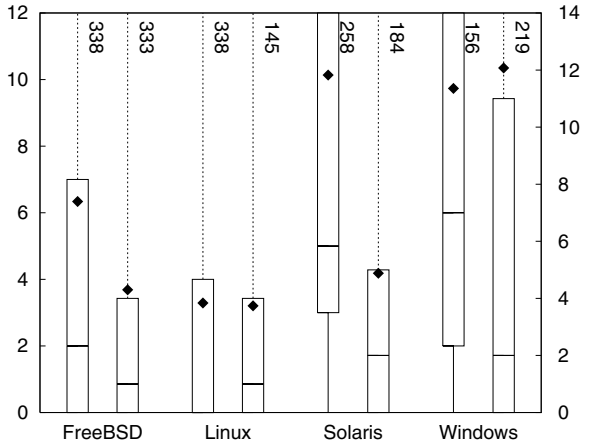

Figure 3: Defined global functions and structures.

tion. Studying them would introduce additional subjective criteria. The controversy surrounding studies comparing competing operating systems in areas like security or performance demonstrates the difficulty of such approaches.

The large size difference between the WRK source code and the other systems, is not as problematic as it may initially appear. An earlier study on the distribution of the maintainability index [4] across various FreeBSD modules [35, Figure 7.3] showed that their maintainability was evenly distributed, with few outliers at each end. This makes me believe that the WRK code examined can be treated as a representative subset of the complete Windows operating system kernel.

3. METHODS AND RESULTS

The metrics I collected can be roughly categorized into the areas of file organization, code structure, code style, preprocessing, and data organization.

3.1 File Organization

In the C programming language source code files play a significant role in structuring a system. A file forms a scope boundary, while the directory it is located may determine the search path for included header files [15, p. 45]. Thus, the appropriate organization of definitions and declarations into files, and files into directories is a measure of the system's modularity [25].

Figure 2 shows the length of C and header files.[2] Most files are less than 2000 lines long. Overly long files are often problematic, because they can be difficult to manage, they may create many dependencies, and they may violate modularity. Indeed the longest header file (WRK's winerror.h) at 27,000 lines lumps together error messages from 30 different areas; most of which are not related to the Windows kernel.

A related measure examines the contents of files, not in terms of lines, but in terms of defined entities. In a C source file the main entity is a global function, while for header files, an important entity is a structure; the closest abstraction to a class that is available in C. Figure 3 shows the number of global functions that are declared in each C file and the number of aggregates (structures or unions) that are defined in each header file. Ideally, both numbers should be small, indicating an appropriate separation of concerns.

At a higher level of granularity, I examine the number of files located in a single directory. Again here, putting many files in a directory is like having many elements in a module. A large number of files can confuse developers, who often search through these files as a group with tools like *grep*, and lead to accidental identifier collisions through shared header files. The numbers I found in the examined systems can be seen in Table 1(B).

The next line in the table describes the correspondence between header files and C (proper) files. A common style guideline for C code involves putting each module's interface in a separate header file, and its implementation in a corresponding C file. Thus a ratio of header to C files around 1 is the optimum; numbers significantly diverging from this figure may indicate an unclear distinction between interface and implementation. This can be acceptable for a small system (the ratio in the implementation of the *awk* programming language is 3/11), but will be a problem in a system consisting of thousands of files.

Finally, the last line in Table 1(B) provides a metric related to the relationships between files when these are regarded as first-class entities. I define a file's *fan-out* as the number of efferent references it makes to elements declared or defined in other files. For instance, a C file including the headers *stdio.h* and *stdlib.h* that uses the symbols *FILE*, *putc*, and *malloc* will have a fan-out of 3. Correspondingly, I define as a file's *fan-in* the number of afferent references coming in from other files. Thus, in the previous example, the fan-in of *stdio.h* would be 2. I used Henry and Kafura's information flow metric [16] to look at the corresponding relationships between files. The value I report is

$$(fanIn \times fanOut)^2$$

A large value of this metric has been associated with the occurrence of changes and structural flaws.

[2]Each candlestick in the figures depicts the minimum, lower (25%) quartile, median, upper (75%) quartile, and maximum values. The diamond indicates the mean. When two candlesticks are shown for each system, the caption's first element (C files in this case) is shown on the left, and the second (header files here) on the right. Minima and maxima lying outside the graph's range are indicated with a dashed line.

Metric		FreeBSD	Linux	Solaris	WRK
A. Overview					
Version		HEAD 2006-09-18	2.6.18.8-0.5	2007-08-28	1.2
Configuration		i386 AMD64 SPARC64	AMD64	Sun4v Sun4u SPARC	i386 AMD64
Lines (thousands)		2,599	4,150	3,000	829
Comments (thousands)		232	377	299	190
Statements (thousands)		948	1,772	1,042	192
Source files		4,479	8,372	3,851	653
Linked modules		1,224	1,563	561	3
C functions		38,371	86,245	39,966	4,820
Macro definitions		727,410	703,940	136,953	31,908
B. File Organization					
Files per directory	↘	6.8	20.4	8.9	15.9
Header files per C source file	≈ 1	1.05	1.96	1.09	1.92
Average structure complexity in files	↘	2.2×10^{14}	1.3×10^{13}	5.4×10^{12}	2.6×10^{13}
C. Code Structure					
% global functions	↘	36.7	21.2	45.9	99.8
% strictly structured functions	↗	27.1	68.4	65.8	72.1
% labeled statements	↘	0.64	0.93	0.44	0.28
Average number of parameters to functions	↘	2.08	1.97	2.20	2.13
Average depth of maximum nesting	↘	0.86	0.88	1.06	1.16
Tokens per statement	↘	9.14	9.07	9.19	8.44
% of tokens in replicated code	↘	4.68	4.60	3.00	3.81
Average structure complexity in functions	↘	7.1×10^{4}	1.3×10^{8}	3.0×10^{6}	6.6×10^{5}
D. Code Style					
% style conforming lines	↗	77.27	77.96	84.32	33.30
% style conforming typedef identifiers	↗	57.1	59.2	86.9	100.0
% style conforming aggregate tags	↗	0.0	0.0	20.7	98.2
Characters per line	↘	30.8	29.4	27.2	28.6
% of numeric constants in operands	↘	10.6	13.3	7.7	7.7
% unsafe function-like macros	↘	3.99	4.44	9.79	4.04
% misspelled comment words	↘	33.0	31.5	46.4	10.1
% unique misspelled comment words	↘	6.33	6.16	5.76	3.23
E. Preprocessing					
% of preprocessor directives in header files	↘	22.4	21.9	21.6	10.8
% of non-#include directives in C files	↘	2.2	1.9	1.2	1.7
% of preprocessor directives in functions	↘	1.56	0.85	0.75	1.07
% of preprocessor conditionals in functions	↘	0.68	0.38	0.34	0.48
% of function-like macros in defined functions	↘	26	20	25	64
% of macros in unique identifiers	↘	66	50	24	25
% of macros in identifiers	↘	32.5	26.7	22.0	27.1
F. Data Organization					
% of variable declarations with global scope	↘	0.36	0.19	1.02	1.86
% of variable operands with global scope	↘	3.3	0.5	1.3	2.3
% of identifiers with wrongly global scope	↘	0.28	0.17	1.51	3.53
% of variable declarations with file scope	↘	2.4	4.0	4.5	6.4
% of variable operands with file scope	↘	10.0	6.1	12.7	16.7
Variables per typedef or aggregate	↘	15.13	25.90	15.49	7.70
Data elements per aggregate or enumeration	↘	8.5	10.0	8.6	7.3

Metric interpretation: ↘ means lower is better; ↗ means higher is better.

Table 1: Key scalar metrics

3.2 Code Structure

The code structure of the four systems illustrates how similar problems can be tackled through different control structures and separation of concerns. It also allows us to peer into the design of each system.

Figure 4 shows the distribution across functions of the extended cyclomatic complexity metric [23]. This is a measure of the number of independent paths that are contained in each function. The number shown takes into account the Boolean and conditional evaluation operators (because these introduce additional paths), but not the multi-way switch statements, because these would disproportionably affect the result for code that is typically cookie-cutter similar. The metric was designed to measure a program's testability,

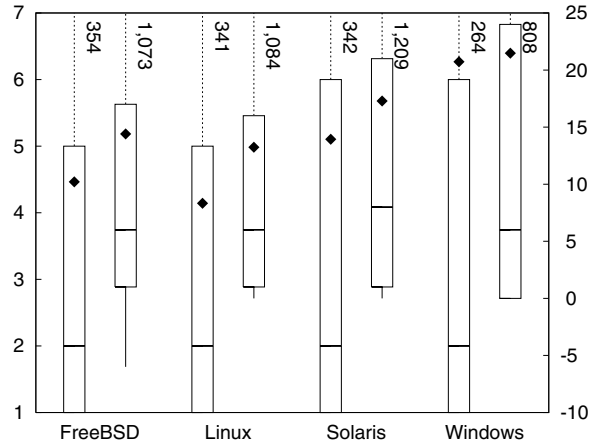

Figure 4: Extended cyclomatic complexity and number of statements per function.

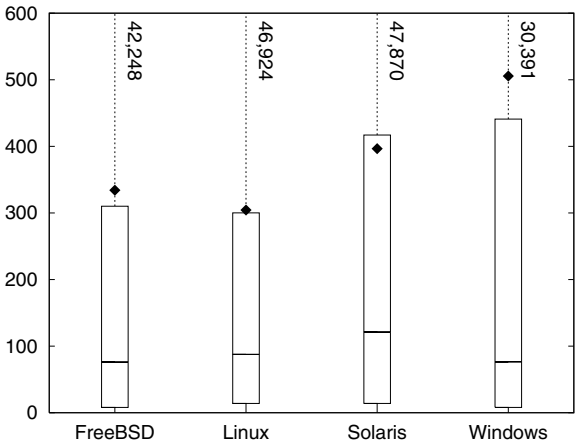

Figure 5: Halstead complexity.

understandability, and maintainability [12]. The same figure also shows the number of statements per function. Ideally, this should be a small number (e.g. around 20) allowing the complete body a function to fit on the developer's screen.

In Figure 5 we can see the distribution of the (often criticized) Halstead volume complexity [14]. Ideally, this should be low, reflecting code that doesn't require a lot of mental effort to comprehend.

Taking a step back to look at interactions between functions, Figure 6 depicts common coupling in functions by showing the percentage of the unique identifiers appearing in a function's body that come either from the scope of the compilation unit (static) or from the project scope (global). Both forms of coupling are undesirable, with the coupling through global identifiers being worse than that occurring through file-scoped ones.

Other metrics associated with code structure appear in Table 1(C). The percentage of global functions indicates the functions visible throughout the system. The number of such functions in the WRK (nearly 100%; also verified by hand) is shockingly high. It may however reflect Microsoft's use of different techniques—such as linking into shared libraries with explicitly exported symbols—for avoiding identifier clashes.

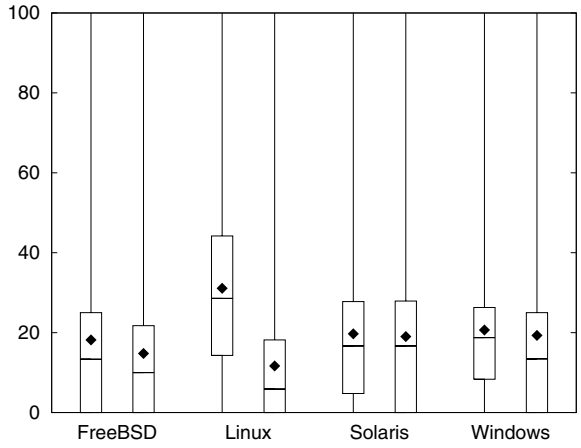

Figure 6: Common coupling at file and global scope.

Strictly structured functions are those following the rules of structured programming: a single point of exit and no *goto* statements. Such functions may be easier to reason about. Along the same lines, the percentage of labeled statements indicates *goto* targets: an severe violation of structured programming principles. I measured labeled statements rather than *goto* statements, because many branch targets are a lot more confusing than many branch sources. Often multiple *goto* statements to a single label are used to exit from a function while performing some cleanup—the equivalent of an exception's *finally* clause.

The number of arguments to a function is an indicator of the interface's quality: when many arguments must be passed, packaging them into a single structure reduces clutter and opens up style and performance optimization opportunities.

Two metrics tracking the code's understandability are the average depth of maximum nesting and the number of tokens per statement. Both deeply nested structures and long statements are difficult to comprehend [2].

Replicated code has been associated with bugs [22] and maintainability problems [35, pp. 413–416]. The corresponding metric (% of tokens in replicated code) shows the percentage of the code's tokens that participate in at least one clone set, as identified by the tool *CCFinderX*.[3]

Finally, the average structure complexity in functions uses again Henry and Kafura's information flow metric [16] to look at the relationships between functions. Ideally we would want this number to be low, indicating an appropriate separation between suppliers and consumers of functionality.

3.3 Code Style

The same code can be written using various choices of indentation, spacing, identifier names, representations for constants, and naming conventions [20, 11, 1, 37]. In most cases consistency is more important than the actual choice.

I measured each system's consistency of style by applying the formatting program *indent*[4] on the complete source code of each system, and counting the lines that *indent* modified. The result appears on the first line of Table 1(D). The behavior of *indent* can be modified using various options in or-

[3] `http://www.ccfinder.net/`

[4] `http://www.gnu.org/software/indent/`

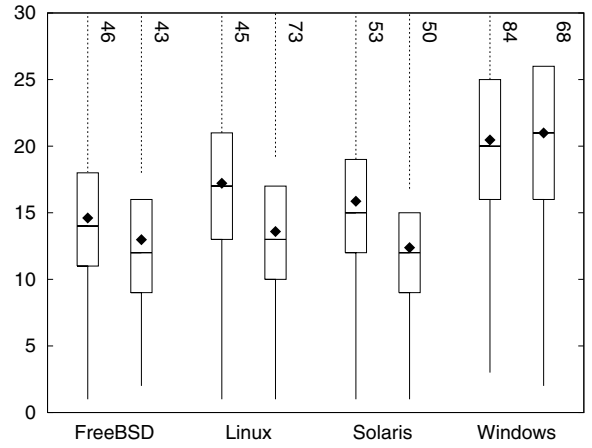

Figure 7: Length of global and aggregate identifiers.

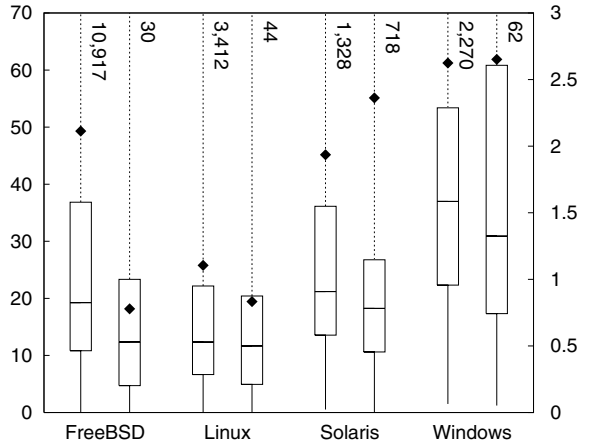

Figure 8: Comment density in C and header files.

der to match corresponding formatting styles. For instance, one can specify the amount of indentation and the placement of braces. In order to determine each system's formatting style and use the appropriate formatting options, I first run *indent* on each system with various values of the 15 numerical flags, and turning on or off each one of the 55 Boolean flags. I then chose the set of flags that produced the largest number of conforming lines. For example, on the OpenSolaris source code *indent* with its default flags would reformat 74% of the lines. This number shrank to 16% once the appropriate flags were determined (-i8 -bli0 -cbi0 -ci4 -ip0 -bad -bbb -br -brs -ce -nbbo -ncs -nlp -npcs).

Figure 7 depicts the length distribution of two important classes of C identifiers: those of globally visible objects (variables and functions) and the tags used for identifying aggregates (structures and unions). With a single name space typically used for each class throughout the system, it is important to choose names that distinct and recognizable (see chapter 31 of reference [24]). For these classes of identifiers longer names are preferable.

Further metrics related to code style appear in Table 1(D). Two related consistency measurements I performed involved manually determining the convention used for naming *typedef*s and aggregate tags, and then counting the identifiers of those classes that did not match the convention.

Three other metrics aimed at identifying programming practices that style guidelines typically discourage: overly long lines of code (characters per line metric), the direct use of "magic" numbers in the code (% of numeric constants in operands), and the definition of function-like macros that can misbehave when placed after an *if* statement (% unsafe function-like macros).[5]

Another important element of style involves commenting. It is difficult to judge objectively the quality of code comments. Comments can be superfluous or even wrong. Yet, we can easily measure the comment density. In Figure 8 we see the comment density in C files as the ratio of comment characters to statements. In header files I measured it as the ratio of defined elements that typically require an explanatory comment (enumerations, aggregates and their members, variable declarations, and function-like macros) to the number of comments.

I also measured the number of spelling errors in the comments as a proxy for their quality. For this I applied on the text of the comments the *aspell* spelling checker with a custom dictionary consisting of all the system's identifier and file names. The low number of errors in the WRK reflects the fact that, according to accompanying documentation, these were explicitly spell-checked before the code was released.

Although I did not measure portability objectively, the work involved in processing the source code with *CScout* allowed me to get a feeling of the portability of each system's source code between different compilers. The code of Linux and WRK appears to be the one most tightly bound to a specific compiler. Linux uses numerous language extensions provided by the GNU C compiler; in some places having assembly code thinly disguised in what *gcc* passes as C syntax. The WRK uses considerably fewer language extensions, but relies significantly on the *try catch* extension to C that the Microsoft compiler supports. The FreeBSD kernel uses only a few *gcc* extensions, and these are often isolated inside wrapping macros. The OpenSolaris kernel was a welcomed surprise: it was the only body of source code that did not require any extensions to *CScout* in order to compile.

3.4 Preprocessing

The relationship between the C language proper and its (integral) preprocessor can at best be described as uneasy. Although C and real-life programs rely significantly on the preprocessor, its features often create portability, maintainability, and reliability problems. The preprocessor, as a powerful but blunt instrument, wrecks havoc with identifier scopes, the ability to parse and refactor unpreprocessed code, and the way code is compiled on different platforms. For this reason, modern languages based on C have tried to replace features provided by the C preprocessor with more disciplined alternatives, while programming guidelines recommend moderation in the use of preprocessor constructs.

A global measure on the use of preprocessor features is the amount of expansion that occurs when processing code. Figure 9 contains two such measures: one for the body of functions (representing expansion of code), and one for elements outside the body of functions (representing data definitions and declarations). Both measurements were made by

[5]Function-like macros containing more than one statement should have their body enclosed in a dummy *do ... while(0)* block in order to make them behave like a call to a real function.

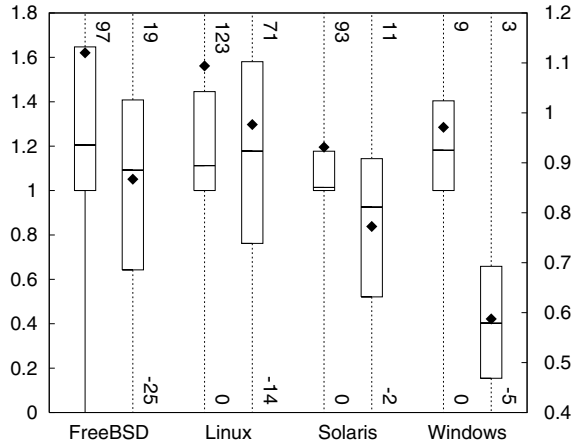

Figure 9: Preprocessing expansion in functions and files.

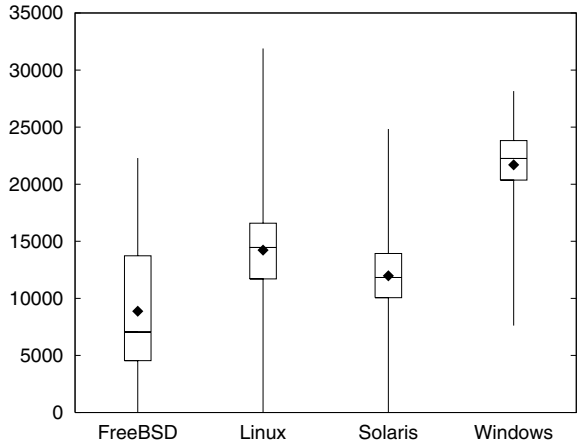

Figure 10: Average level of namespace pollution in C files.

calculating the ratio of tokens arriving into the preprocessor to those coming out of it.

Four further metrics listed in Table 1(E) measure increasingly unsafe uses of the preprocessor: directives in header files (often required), non-#include directives in C files (rarely needed), preprocessor directives in functions (of dubious value), and preprocessor conditionals in functions (a portability risk).

Preprocessor macros are typically used instead of variables (object-like macros) and functions (function-like macros). In modern C object-like macros can often be replaced through enumeration members and function-like macros through inline functions. Both alternatives adhere to the scoping rules of C blocks and are therefore considerably safer than macros whose scope typically spans a whole compilation unit. The last three metrics of preprocessor use in Table 1(E) measure the occurrence of function and object-like macros in the code. Given the availability of viable alternatives and the dangers associated with macros, ideally all should have low values.

3.5 Data Organization

The final set of measurements concerns the organization of each kernel's (in-memory) data. A measure of the quality of this organization in C code can be determined by the scoping of identifiers and the use of structures.

In contrast to many modern languages there is a paucity of mechanisms in C for controlling namespace pollution. Functions can only be defined in only two possible scopes (file and global), macros are visible throughout the compilation unit in which they are defined, and aggregate tags live all in the same (block scoped) namespace. Judiciously using the few mechanisms available to control the number of possibly interfering identifiers is an important requirement for the maintainability of large-scale systems, like the ones I examine. Figure 10 shows the level of namespace pollution in C files by averaging the number of identifiers and macros that are visible at the start of each function. With roughly 10,000 identifiers visible on average at any given point across the systems I examine, it is obvious that namespace pollution is a problem in C code, and that developers should try to keep this number low.

The first three measures in Table 1(F) examine how each system deals with the most scarce resource, that of global variable identifiers. One would like to minimize the number of variable declarations that take place at the global scope in order to minimize namespace pollution. Furthermore, minimizing the percentage of operands that refer to global variables reduces coupling and lessens the cognitive load on the reader of the code (global identifiers can be declared anywhere in the millions of lines comprising the system). The last metric concerning global objects counts identifiers that are declared as global, but could have been declared with a static scope, because they are only accessed from a single file. The next two metrics look at variable declarations and operands with file scope. These are more benign than global variables, but still worse than variables scoped at a block level.

The last two metrics concerning the organization of data provide a crude measure of the abstraction mechanisms used in the code. Type and aggregate definitions are the two main data abstraction mechanisms available to C programs. Thus, counting the number of variable declarations corresponding to each type or aggregate definition provides an indication of the extent that these abstraction mechanisms have been employed in the code. The number of data elements per aggregate or enumeration is to data elements what Chidamber and Kemerer's WMC object-oriented weighted methods per class (WMC) metric is to code. A high value could indicate that a structure tries to store too many disparate elements.

4. RELATED WORK

Considerable work has been performed in the area of open source software evaluation; see the pointers listed in the Introduction and the references therein. Studies of operating system code quality attributes have been conducted for more than two decades [16, 43]. Particularly close to our work are comparative studies of open source operating systems [42, 19], and studies comparing open and closed source systems [26, 38, 30].

A comparison of maintainability attributes between the Linux and various BSD operating systems found that Linux contained more instances of common coupling than the BSD variants. Our results corroborate this finding for file-scoped identifiers, but not for global identifiers (see Figure 6). An evaluation of growth dynamics of the FreeBSD and Linux

Metric	FreeBSD	Linux	Solaris	WRK
File Organization				
Length of C files			–	–
Length of header files		+		–
Defined global functions in C files			–	–
Defined structures in header files				–
Files per directory		–		
Header files per C source file				
Average structure complexity in files	–		+	
Code Structure				
Extended cyclomatic complexity		+		–
Statements per function		+		
Halstead complexity		+		–
Common coupling at file scope		–		
Common coupling at global scope		+		
% global functions		+		–
% strictly structured functions	–			+
% labeled statements		–		+
Average number of parameters to functions				
Average depth of maximum nesting			–	–
Tokens per statement				
% of tokens in replicated code	–	–	+	
Average structure complexity in functions	+	–		
Code Style				
Length of global identifiers				+
Length of aggregate identifiers				+
% style conforming lines			+	–
% style conforming typedef identifiers	–	–		+
% style conforming aggregate tags	–	–	–	+
Characters per line				
% of numeric constants in operands		–	+	+
% unsafe function-like macros			–	
Comment density in C files		–		+
Comment density in header files		–		+
% misspelled comment words				+
% unique misspelled comment words				+
Preprocessing				
Preprocessing expansion in functions	–		+	
Preprocessing expansion in files				+
% of preprocessor directives in header files		–	–	+
% of non-#include directives in C files	–		+	
% of preprocessor directives in functions	–		+	
% of preprocessor conditionals in functions	–	+	+	
% of function-like macros in defined functions		+		–
% of macros in unique identifiers	–		+	+
% of macros in identifiers	–		+	
Data Organization				
Average level of namespace pollution in C files	+			–
% of variable declarations with global scope		+		–
% of variable operands with global scope	–	+		
% of identifiers with wrongly global scope		+		–
% of variable declarations with file scope	+			–
% of variable operands with file scope		+		–
Variables per typedef or aggregate		–		+
Data elements per aggregate or enumeration		–		+

Table 2: Result summary

operating systems found that both grow at a linear rate, and that claims of open source systems growing at a faster rate than commercial systems are unfounded [19].

The study by Paulson and his colleagues [26] compares evolutionary patterns between three open source projects (Linux, GCC, and Apache) and three non-disclosed commercial ones, finding a faster rate of bug fixing and feature addition in the open source projects. In another study focusing on internal quality attributes [38] the authors used a commercial tool to evaluate 100 open source applications using metrics similar to those reported here, but measured on a scale ranging from *accept* to *rewrite*. They then compared the results against benchmarks supplied by the tool's vendor for commercial projects. Their results were inconclusive, with the modules roughly split in half between *accept* and *rewrite*. A related study by the same group [30] examined the evolution of the maintainability index [4] between an open source application and its (semi)proprietary forks. They concluded that all projects suffered from a similar deterioration of the maintainability index over time.

5. SUMMARY AND DISCUSSION

The study has a number of limitations A summary of the results I obtained appears in Table 2. In the table I have marked cells where an operating system excels with a + and corresponding laggards with a −. For a number of reasons it would be a mistake to read too much from this table. First of all, the weights of the table's metrics are not calibrated according to their importance. In addition, it is far from clear that the metrics I used are functionally independent, and that they provide a complete or even representative picture of the quality of C code. Finally, I entered the +/− markings subjectively, trying to identify clear cases of differentiation in particular metrics.

Nevertheless, by looking at the distribution and clustering of markings we can arrive at some important plausible conclusions. The most interesting result from both the detailed results listed in the previous sections and the summary in Table 2 is the similarity of the values among the systems. Across various areas and many different metrics, four systems developed using wildly different processes score comparably. At the very least, the results indicate that the structure and internal quality attributes of a working, nontrivial software artifact will represent first and foremost the engineering requirements of its construction, with the influence of process being marginal, if any. This does not mean that process is irrelevant, but that processes compatible with an artifact's requirements lead to roughly similar results. In the field of architecture this phenomenon has been popularized under the motto "form follows function" [31].

One can also draw interesting conclusions from the clustering of marks in particular areas. Linux excels in various code structure metrics, but lags in code style. This could be attributed to the work of brilliant motivated programmers who aren't however efficiently managed to pay attention to the details of style. In contrast, the high marks of WRK in code style and low marks in code structure could be attributed to the opposite effect: programmers who are efficiently micro-managed to care about the details of style, but are not given sufficient creative freedom to structure their code in an appropriate manner.

The high marks of Solaris and WRK in preprocessing could also be attributed to programming discipline. The problems from the use of the preprocessor are well-known, but its allure is seductive. It is often tempting to use the preprocessor in order to create elaborate domain-specific programming constructs. It is also often easy to fix a portability problem by means of conditional compilation directives. However, both approaches can be problematic in the long run, and we can hypothesize that in an organization like Sun or Microsoft programmers are discouraged from relying on the preprocessor.

A final interesting cluster appears in the low marks for preprocessor use in the FreeBSD kernel. This could be attributed to the age of the code base in conjunction with a gung-ho programming attitude. However, a particularly low level of namespace pollution across the FreeBSD source code could be a result of using the preprocessor to setup and access conservatively scoped data structures.

Despite various claims regarding the efficacy of particular open or close-source development methods, we can see from the table that there is no clear winner (or loser). The two systems with a commercial pedigree (Solaris and WRK) have slightly more positive than negative marks. However, WRK also has the largest number of negative marks, while Solaris has the second lowest number of positive marks. Therefore, the most we can read from the overall balance of marks is that open source development approaches do not produce software of markedly higher quality than proprietary software development.

Acknowledgments and Disclosure of Interest

I wish to thank Microsoft, Sun, and the members of the FreeBSD and Linux communities for making their source code available in a form that allows analysis and experimentation. I also thank Fotis Draganidis, Robert L. Glass, Markos Gogoulos, Georgios Gousios, Panos Louridas, and Konstantinos Stroggylos for their help, comments, and advice on earlier drafts of this paper. This work was partially funded by the European Community's Sixth Framework Programme under the contract IST-2005-033331 "Software Quality Observatory for Open Source Software (SQO-OSS)".

The author has been a source code committer in the FreeBSD project since 2003, and has participated as an invited guest in three Microsoft-sponsored academic initiatives.

6. REFERENCES

[1] L. W. Cannon et al. Recommended C style and coding standards. Available online (February 2008) `http://sunland.gsfc.nasa.gov/info/cstyle.html`.

[2] S. N. Cant, D. R. Jeffery, and B. L. Henderson-Sellers. A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, 37(7):351–362, June 1995.

[3] A. Capiluppi and G. Robles, editors. *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*. IEEE Computer Society, May 2007.

[4] D. Coleman, D. Ash, B. Lowther, and P. W. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[5] M. A. Cusumano and R. W. Selby. *Microsoft Secrets*. The Free Press, New York, 1995.

[6] K. Dickinson. Software process framework at Sun. *StandardView*, 4(3):161–165, 1996.

[7] J. Feller, editor. *5-WOSSE: Proceedings of the Fifth Workshop on Open Source Software Engineering*. ACM Press, 2005.

[8] J. Feller and B. Fitzgerald. *Understanding Open Source Software Development*. Addison-Wesley, Reading, MA, 2001.

[9] J. Feller, B. Fitzgerald, S. Hissam, and K. Lakhani, editors. *Perspectives on Free and Open Source Software*. MIT Press, Boston, 2005.

[10] B. Fitzgerald and J. Feller. A further investigation of open source software: Community, co-ordination, code quality and security issues. *Information Systems Journal*, 12(1):3–5, 2002.

[11] The FreeBSD Project. *Style—Kernel Source File Style Guide*, Dec. 1995. FreeBSD Kernel Developer's Manual: style(9). Available online `http://www.freebsd.org/docs.html` (January 2006).

[12] G. K. Gill and C. F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12):1284–1288, 1991.

[13] R. L. Glass. Of open source, Linux ... and hype. *IEEE Software*, 16(1):126–128, January/February 1999.

[14] M. H. Halstead. *Elements of Software Science*. Elsevier New Holland, New York, 1977.

[15] S. P. Harbison and G. L. Steele Jr. *C: A Reference Manual*. Prentice Hall, Englewood Cliffs, NJ, third edition, 1991.

[16] S. M. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, 1981.

[17] J.-H. Hoepman and B. Jacobs. Increased security through open source. *Communications of the ACM*, 50(1):79–83, 2007.

[18] D. M. Hoffman and D. M. Weiss, editors. *Software Fundamentals: Collected Papers by David L. Parnas*. Addison-Wesley, Boston, MA, 2001.

[19] C. Izurieta and J. Bieman. The evolution of FreeBSD and Linux. In *ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 204–211. ACM Press, 2006.

[20] B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, New York, second edition, 1978.

[21] J. Kuan. Open source software as lead user's make or buy decision: A study of open and closed source quality. In *Second Conference on The Economics of the Software and Internet Industries*, Jan. 2003.

[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

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

[24] S. C. McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Redmond, WA, second edition, 2004.

[25] D. L. Parnas. On the criteria to be used for decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972. Also in [18] pp. 145–155.

[26] J. W. Paulson, G. Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering*, 30(4):246–256, Apr. 2004.

[27] A. Polze and D. Probert. Teaching operating systems: The Windows case. In *SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pages 298–302. ACM Press, 2006.

[28] P. C. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. Technical Report DCS-305-IR, University of Victoria, January 2006.

[29] P. H. Salus. *A Quarter Century of UNIX*. Addison-Wesley, Boston, MA, 1994.

[30] I. Samoladas, I. Stamelos, L. Angelis, and A. Oikonomou. Open source software development should strive for even greater code maintainability. *Communications of the ACM*, 47(10):83–87, 2004.

[31] H. A. Small, editor. *Form and Function: Remarks on Art by Horatio Greenough*. University of California Press, Berkeley and Los Angeles, 1947.

[32] S. K. Sowe, I. G. Stamelos, and I. Samoladas, editors. *Emerging Free and Open Source Software Practices*. IGI Publishing, Hershey, PA, 2007.

[33] D. Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11):1019–1030, Nov. 2003.

[34] D. Spinellis. The CScout refactoring browser. Technical report, Athens University of Economics and Business, Athens, Greece, 2004. Available online.

[35] D. Spinellis. *Code Quality: The Open Source Perspective*. Addison-Wesley, Boston, MA, 2006.

[36] D. Spinellis and C. Szyperski. How is open source affecting software development? *IEEE Software*, 21(1):28–33, January/February 2004.

[37] R. Stallman et al. GNU coding standards. Available online `http://www.gnu.org/prep/standards/` (January 2006), Dec. 2005.

[38] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, 2002.

[39] A. S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice Hall, Englewood Cliffs, NJ, 1987.

[40] L. Torvalds and D. Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. HarperInformation, New York, 2001.

[41] G. von Krogh and E. von Hippel. The promise of research on open source software. *Management Science*, 52(7):975–983, July 2006.

[42] L. Yu, S. R. Schach, K. Chen, G. Z. Heller, and J. Offutt. Maintainability of the kernels of open source operating systems: A comparison of Linux with FreeBSD, NetBSD and OpenBSD. *Journal of Systems and Software*, 79(6):807–815, 2006.

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