RACCIS 5(1), pp. 49-57, 2015

Revista Antioqueña de las
Ciencias Computacionales y la Ingeniería de Software

ISSN: 2248-7441

www.fundacioniai.org/raccis
raccis@fundacioniai.org

Free and Licensed Operating Systems: Direct Comparison

Sistemas Operativos Libres y Propietarios: Comparación Directa

Samy Hagman[1], Keshava Indiresh[2]
Universidad de Harvard. *samyh(AT)harvard.edu*[1], *kindi(AT)harvard.edu*[2]

INFORMACIÓN DEL ARTÍCULO

Tipo
Investigación

Historia
Recibido: 10-03-2015
Correcciones:
Aceptado: 30-05-2015

Keywords
Open source, Software Engineering, software metrics, software quality.

Palabras clave
Calidad del software, código abierto, Ingeniería de Software, métricas del software.

ABSTRACT
When developers comparing open with proprietary code, which should be a civilized debate often degenerates into a flame war. This paper presents a report on code quality metrics that have been collected from four major operating systems on an industrial scale is as follows: FreeBSD, Linux, OpenSolaris and Windows Research Kernel (WRK). This article is not a mysterious crime and it can be concluded as the main finding no significant differences in the quality of the code of these systems.

RESUMEN
Cuando los desarrolladores comparan el código abierto con el propietario, lo que debería ser un debate civilizado a menudo degenera en una llama de guerra. En este trabajo se presenta un reporte sobre métricas de calidad del código, que se han recogido a partir de cuatro grandes sistemas operativos a escala industrial: FreeBSD, Linux, OpenSolaris y Windows Research Kernel (WRK). Este artículo no es un crimen misterioso y se puede concluir como principal hallazgo que no hay diferencias significativas en la calidad del código de estos sistemas.

1. Introducción

Cuando los desarrolladores comparan el código abierto con el software propietario, lo que debería ser un debate civilizado a menudo degenera en una llama de guerra. Esto no tiene que ser así, porque no hay mucho espacio para una comparación objetiva con cabeza fría. Los investigadores examinan la eficacia de los procesos de desarrollo de código abierto a través de enfoques complementarios:

- Un método consiste en examinar la calidad del código y sus atributos internos de calidad, tales como densidad de comentarios o uso de variables globales [1].
- Otro enfoque implica examinar los atributos de calidad externos del software que reflejan cómo se verá ante los usuarios finales [2].
- Otros, en lugar del producto, ven el proceso y examinan las medidas relacionadas con la construcción y el mantenimiento del código, tales como cuánto código se añade por semana o la rapidez con que se solucionan los errores [3].
- Otro enfoque consiste en discutir escenarios específicos. Por ejemplo, Hoepman y Jacobs [4] examinan la seguridad del software de código abierto mirando cómo se filtra el código fuente desde máquinas con Windows NT.
- Por último, una serie de argumentos se basan simplemente en revisar la escritura. Glass [5] identificó esta tendencia en la publicidad asociada con la aparición de Linux en la industria de TI.

Aunque en los años recientes muchos investigadores han examinado artefactos y procesos de código abierto [6-13], la comparación directa de sistemas de código abierto con los productos propietarios correspondientes sigue siendo un objetivo difícil de alcanzar. La razón es que es difícil encontrar un producto propietario comparable a una fuente abierta equivalente, y mucho más convencer al dueño de la patente para que proporcione el código fuente para una comparación objetiva. Sin embargo, el código abierto del *kernel* de Solaris (Oracle) y la distribución de grandes partes del código fuente del *kernel* de Windows a instituciones de investigación, ha proporcionado una oportunidad para llevar a cabo una evaluación comparativa entre el código fuente abierto y el código de los sistemas desarrollados como software propietario.

En este trabajo se presenta un reporte sobre métricas de calidad del código, que se han recogido a partir de cuatro grandes sistemas operativos a escala industrial: FreeBSD, Linux, OpenSolaris y Windows Research Kernel (WRK). Esta investigación no es un crimen misterioso y se puede concluir como principal hallazgo que no hay diferencias

significativas en la calidad del código de estos sistemas. Ahora que se sabe el final, se le sugiere al lector que siga leyendo, porque en las siguientes secciones encontrará no solamente cómo se llegó a esta conclusión, sino también numerosas métricas de calidad de código para evaluar objetivamente el software escrito en C, que también puede aplicar a su código. Aunque algunos de estos indicadores no se han validado empíricamente, se basan en directrices de codificación generalmente aceptadas, por lo que representan el consenso aproximado relativo que los desarrolladores atribuyen al código deseable [14].

2. Antecedentes

Por décadas los investigadores han estado estudiando los atributos de calidad del código de los sistemas operativos [15, 16]. Particularmente, hay estudios comparativos de sistemas operativos de código abierto [17, 18] y de sistemas de código abierto y cerrado [1, 3, 19]. Una comparación de atributos de mantenibilidad entre Linux y varios sistemas operativos Berkeley Software Distribution (BSD), encontró que Linux contenía más instancias de módulo de comunicación a través de variables globales (conocidas como acoplamiento común) que las variantes BSD. Los resultados que se presentan aquí corroboran este hallazgo para identificadores de archivos, pero no para identificadores globales. Además, una evaluación de la dinámica de crecimiento de los sistemas operativos FreeBSD y Linux encontró que ambos crecen a un ritmo lineal, y que son infundadas las pretensiones de los sistemas de código abierto de que crecen a un ritmo más rápido que los comerciales [18].

Un estudio realizado por Paulson y sus colegas [3] compara patrones evolutivos entre tres proyectos de código abierto (Linux, GCC y Apache) y tres comerciales no-divulgados. Encontraron un ritmo más rápido de corrección de errores y adición de características en los proyectos de código abierto, algo que se esperaría para proyectos muy populares como los que se examinaron. En otro estudio centrado en la calidad del código [1], los autores aplicaron una herramienta comercial para evaluar 100 aplicaciones de código abierto, utilizando métricas similares a los reportados aquí pero midieron sobre una escala que va desde *aceptar* a *re-escribir*. Luego compararon los resultados con los puntos de referencia suministrados por el proveedor de la herramienta para proyectos comerciales. Encontraron que solamente la mitad de los módulos que examinaron serían considerados aceptables por las organizaciones que aplican normas de programación basados en métricas de software. Otro estudio del grupo [19] examinó la evolución de una medida llamada *índice de mantenibilidad* [20] entre una aplicación de código abierto y sus (semi) bifurcaciones propietarias. Llegaron a la conclusión que todos los proyectos sufrieron deterioro similar al índice de mantenibilidad en el tiempo.

Los cuatro sistemas comenzaron su vida independiente entre 1991 y 1993. En ese momento los computadores basados en microprocesadores que soportaban un espacio de direcciones y gestión de memoria de 32 bits condujeron a la explosión cámbrica de los sistemas operativos modernos. Dos de ellos, FreeBSD y OpenSolaris, comparten un ancestro común que se remonta a la versión 1BSD de Unix de 1978. FreeBSD está basado en BSD/Net2, una distribución del código fuente de Unix de Berkeley que fue purgado desde el código propietario de AT&T. Consecuentemente, mientras que tanto FreeBSD como OpenSolaris contienen código escrito en Berkeley, solamente OpenSolaris contiene código de AT&T. En concreto, el código detrás de OpenSolaris remonta sus orígenes a la versión de Unix de 1973, que fue el primero escrito en C [21]. En 2005 Sun lanzó la mayor parte del código de Solaris bajo una licencia de código abierto.

Linux fue desarrollado desde cero en un esfuerzo por construir una versión más rica en características que la orientada a la enseñanza de Tanenbaum, el sistema operativo POSIX compatible con Minix [22]. Por lo tanto, aunque Linux toma ideas prestadas de Minix y Unix, no se deriva de su código [23].

Las raíces intelectuales de Windows NT se remontan a DEC's y VMS a través de la participación común de David Cutler, ingeniero principal en ambos proyectos. Windows NT fue desarrollado como respuesta de Microsoft a Unix, inicialmente como una alternativa al OS/2 de IBM y más tarde como un reemplazo al código base del Windows de 16 bits. Windows Research Kernel (WRK) incluye grandes porciones del *kernel* del Windows de 64 bits, que Microsoft distribuye para su uso en investigación [24]. El *kernel* está escrito en C con algunas extensiones pequeñas. Se excluyen del código del *kernel* los controladores de dispositivo, el *plug-and-play*, la administración de energía y los subsistemas DOS virtuales. Las partes que faltan explican la gran diferencia de tamaño entre WRK y los otros tres *kernels*.

Aunque los cuatro sistemas están disponibles en forma de código fuente, sus metodologías de desarrollo son muy diferentes. Los ingenieros Microsoft y Sun construyeron Windows NT y Solaris como sistemas propietarios con un mínimo de participación de foráneos en el proceso. OpenSolaris tuvo una vida muy corta como un proyecto de código abierto, por lo que sólo un mínimo de código pudo haber sido aportado por desarrolladores externos a Sun. Además, Solaris fue desarrollado con énfasis en un proceso formal, mientras que Windows NT emplea métodos más ligeros [25]. FreeBSD y Linux fueron desarrollados utilizando métodos de desarrollo de código abierto [26], pero sus procesos también son diferentes. FreeBSD fue desarrollado principalmente por un grupo no-jerárquico de unos 220 participantes, que tenían acceso a un repositorio de software compartido basado inicialmente en CVS y actualmente en Subversion [27]. Por el contrario, los desarrolladores de Linux estaban organizados en una pirámide de cuatro niveles. En los dos niveles inferiores, miles de desarrolladores contribuyeron a parchar y mantener unos 560 subsistemas. En la parte superior de la pirámide, Linus Torvalds, asistido por un grupo de lugartenientes de confianza, era el único responsable de la adición de los parches al árbol Linux [28]. Hoy en día los desarrolladores de Linux coordinan los cambios de código a través de un sistema de control de versiones distribuido especialmente diseñado para ello.

3. Metodología

La mayoría de métricas presentadas aquí se calcularon mediante consultas SQL en una base de datos relacional, que contiene elementos de código comprendidos en cada sistema: módulos, identificadores, fichas, funciones, archivos, comentarios y sus relaciones. La base de datos para cada sistema se construyó ejecutando el navegador de refactorización CScout para código C [29, 30], sobre una serie de configuraciones de procesador específico de cada sistema operativo. Una configuración de procesador específico comprende algunas definiciones y archivos de macro únicos, y por lo tanto procesa el código de manera diferente.

Para procesar el código fuente de un sistema completo, CScout debe tener un archivo de configuración que especifique el entorno preciso utilizado para procesar cada unidad de compilación (archivo C). Para el FreeBSD y los *kernels* de Linux, este archivo de configuración se construyó mediante la instrumentación de servidores proxy para el compilador GNU C, el enlazador y algunos comandos *shell*. Estos registran sus argumentos (principalmente la ruta del archivo y las definiciones macro) en un formato que podría ser utilizado para construir un archivo de configuración CScout. Para OpenSolaris y WRK simplemente se realizó una construcción completa de las configuraciones investigadas, almacenando los comandos que se ejecutan en un archivo de registro, y luego se procesaron los comandos de compilación y vinculación que aparecen en el registro construido.

Se eligieron y definieron las métricas con el fin de limitar el sesgo introducido en la selección de indicadores, y se recogieron antes de establecer los mecanismos de medición. Esto ayudó a evitar la selección sesgada de métricas basadas en resultados que se obtuvieran en el camino. Sin embargo, esta selección previa también ofreció muchos indicadores, tales como el número de caracteres por línea, que no suministra ninguna información interesante y no proporciona un claro ganador o perdedor. Por otro lado, la selección de métricas no estaba completamente blindada, porque en el momento que se diseñó el experimento el equipo ya estaba familiarizado con el código fuente del *kernel* de FreeBSD, y había visto el código fuente de Linux (9th Research Edition Unix) y otros controladores de dispositivos de Windows.

Otras limitaciones metodológicas de este estudio son: el pequeño número de sistemas estudiados, la especificidad del lenguaje de las métricas empleadas y la cobertura solamente de mantenibilidad y portabilidad del espacio de todos los atributos de calidad del software. Esta última limitación significa que el estudio no tiene en cuenta el gran e importante conjunto de atributos de calidad que se determinan normalmente en tiempo de ejecución: funcionalidad, fiabilidad, facilidad de uso y eficiencia. Sin embargo, los atributos que faltan a menudo dependen de factores que están fuera del control de los desarrolladores del sistema: configuración, puesta a punto y selección de la carga de trabajo. Estudiarlos sería introducir sesgos subjetivos adicionales, tales como configuraciones inadecuadas para algunas cargas de trabajo o entornos operativos. La controversia en torno a los estudios de comparación de los sistemas operativos, que compiten en áreas como la seguridad o el rendimiento, demuestra la dificultad de este tipo de enfoques.

La amplia diferencia de tamaño entre el código fuente de WRK y los otros sistemas no es tan problemático como inicialmente puede parecer. Un estudio anterior sobre la distribución del índice de mantenibilidad [20] a través de diversos módulos FreeBSD, mostró que se distribuye de manera uniforme, con pocos valores extremos [31]. Esto significa que se puede formar una opinión acerca de un sistema software grande mirando una pequeña muestra del mismo. Por lo tanto, el código WRK examinado puede ser tratado como un subconjunto representativo del *kernel* completo del sistema operativo Windows.

4. Resultados

Las propiedades clave que se examinaron en los sistemas aparecen en la Tabla 1. Las métricas de calidad recogidas se pueden clasificar a grandes rasgos en las áreas de: organización de archivos, estructura de código, estilo de código, pre-procesamiento y organización de datos. Cuando una métrica se puede representar solamente con un número, se presenta una lista de valores para cada uno de los cuatro sistemas, indicando si idealmente ese valor debe ser alto, bajo, o cerca de otro en particular.

Tabla 1. Métricas clave de los sistemas operativos evaluados

Métrica	FreeBSD	Linux	Solaris	WRK
Versión	HEAD 2006-09-18	2.6.18.8-0.5	2007-08-28	1.2
Configuración	i386 AMD64 SPARC64	AMD64	Sun4v Sun4u SPARC	i386 AMD64
Líneas	2.599	4.150	3.000	829
Comentarios	232	377	299	190
Declaraciones	948	1.772	1.042	192
Archivos fuente	4.479	8.372	3.851	653
Módulos vinculados	1.224	1.563	561	3
Funciones C	38.371	86.245	39.966	4.820
Definiciones macro	727.410	703.940	136.953	31.908

4.1 Organización de archivos

En el lenguaje C los archivos de código fuente juegan un papel importante en la estructuración de un sistema. Un archivo forma un alcance-límite, mientras que el directorio en el que se encuentra puede determinar la ruta de búsqueda para archivos de cabecera incluidos [32]. De este modo, la organización adecuada de definiciones y de declaraciones en archivos y de los archivos en directorios es una medida de la modularidad del sistema [33].

En el estudio realizado, la longitud de los archivos C y de cabecera en los sistemas operativos, tiene en su mayoría menos de 2.000 líneas. Los archivos demasiado largos (OpenSolaris y WRK) a menudo son problemáticos, porque pueden ser difíciles de manejar, crear muchas dependencias y pueden violar la modularidad. De hecho, el archivo de cabecera más largo (Winerror.h de WRK) con 27.000 líneas agrupa los mensajes de error de 30 áreas diferentes, la mayoría de los cuales no están relacionados con el *kernel* de Windows. Una medida relacionada examina el contenido de los archivos no en términos de líneas, sino en términos de entidades definidas. En los archivos de origen C se cuentan funciones globales. En los archivos de cabecera una entidad importante es una estructura, la abstracción más cercana a una clase disponible en C. Idealmente el número de funciones globales que se declaran en cada archivo C y el número de agregados (estructuras o uniones) que se definen en cada archivo de cabecera, debe ser pequeño, lo que indica una separación adecuada de este cometido. Los archivos C de OpenSolaris y WRK se valoran peor que los otros sistemas, mientras que un número significativo de archivos de cabecera de WRK se ve mal, porque define más de 10 estructuras para cada uno.

Los cuatro sistemas examinados tienen estructuras de directorios interesantes. Tres de los cuatro sistemas tienen un ancho de estructuras similar: FreeBSD, Linux y Solaris. El tamaño pequeño y la complejidad de WRK refleja el hecho de que Microsoft ha excluido de él muchas partes grandes del *kernel* de Windows. Los directorios en Linux están relativamente distribuidos de forma homogénea en todo el árbol de código fuente, mientras que en FreeBSD y OpenSolaris algunos directorios se juntan en grupos. Esto puede ser el resultado de un crecimiento orgánico en un período de tiempo más largo, debido a que ambos sistemas tienen más de 20 años de historia. La distribución más equitativa de los directorios de Linux también puede reflejar el carácter descentralizado de su desarrollo.

A un nivel más alto de granularidad se examinó el número de archivos que se encuentra en un directorio. Una vez más, poner muchos archivos en un directorio es como tener muchos elementos en un módulo. Un gran número de archivos puede confundir a los desarrolladores, que a menudo buscan a través de ellos con diversas herramientas, y esto puede dar lugar a colisiones accidentales del identificador a través de los archivos de cabecera compartidos. Los números encontrados se pueden ver en la Tabla 2.

Tabla 2. Métricas de organización de archivos

Métrica	FreeBSD	Linux	Solaris	WRK
Archivos por directorio	6.8	20.4	8.9	15.9
Archivos cabecera por archivos fuente C	1.05	1.96	1.09	1.92
Promedio de la complejidad de la estructura en archivos	$2.2x10^{14}$	$1.3x10^{13}$	$5.4x10^{12}$	$2.6x10^{13}$

Una pauta de estilo común para el código C consiste en colocar la interfaz de cada módulo en un archivo de cabecera separado y su aplicación en otro archivo C correspondiente. Por lo tanto, un radio de cabecera para archivos C de alrededor de 1 es el óptimo; los números significativamente divergentes a partir de 1 pueden indicar una distinción poco clara entre la interfaz y la implementación. Esto puede ser aceptable para un sistema pequeño (la relación en la aplicación del lenguaje de programación awk es 3/11), pero sería un problema en un sistema que consiste de miles de archivos. Todos los sistemas analizados tienen puntuación aceptable en esta métrica.

Finalmente, la última línea de la Tabla 2 proporciona una métrica relacionada con la complejidad de las relaciones de archivo. Esto se estudia mirando los archivos como nodos en un grafo dirigido. En este estudio se define el número de referencias salientes que hace un archivo a los elementos declarados o definidos en otros archivos. Por ejemplo, un archivo C que utiliza el símbolo FILE, *putc* y *malloc* (definidos en los archivos de cabecera *stdio.h* y *stdlib.h*) tiene un abanico de salida de 3. En consecuencia, el abanico de entrada de un archivo es el número de referencias entrantes hechas por otros archivos. Así, en el ejemplo anterior, el de *stdio.h* sería 2. Para mirar las relaciones correspondientes entre archivos se utilizó el flujo de información de la métrica de Henry y Kafura [15]. Los estados selectos internos derivan, para todas las definiciones externas o referencias asociadas a cada archivo, un conjunto de identificadores únicos y de archivos cuando éstos se definen o referencian. Un valor grande para esta métrica se asocia con una alta incidencia de cambios y defectos estructurales.

4.2 Estructura del código

La estructura del código de los cuatro sistemas ilustra cómo los problemas similares pueden ser abordados a través de diversas estructuras de control y separación de intereses. También permite hacer lo mismo en el diseño de cada sistema. En este estudio la distribución se encontró a través de la métrica de funciones de complejidad ciclomática extendida [34]. Esta es una medida del número de caminos independientes contenidos en cada función. El número resultante tiene en cuenta los operadores de evaluación condicionales de Boole (porque introducen rutas adicionales) pero no la sentencias *switch* de múltiples vías, porque podría afectar de manera

desproporcionada los resultados en código que es típicamente de molde similar. La métrica fue diseñada para medir la capacidad de prueba, la comprensibilidad y la mantenibilidad de un programa [35]. En este sentido, las puntuaciones de Linux son mejores que las de los otros sistemas (341), mientras que la de WRK es la peor (364) y la de FreeBSD es de 344 y la de Solaris de 352. También se analizó el número de declaraciones C por función. Idealmente debería ser un número pequeño (alrededor de 20), porque permite código completo de función que puede encajar en la pantalla del programador. De nuevo Linux obtiene mejores puntuaciones que los otros sistemas (1.084).

También se calculó la distribución del volumen de complejidad de Halstead [36]. Esta teoría afirma que el valor debe ser bajo, lo que refleja un código que no requiere mucho esfuerzo mental para comprenderlo. Sin embargo, esta métrica a menudo ha sido criticada. Como en el caso con la complejidad ciclomática, la puntuación de Linux es mejor y la WRK peor que los otros sistemas. Las interacciones entre funciones representan acoplamiento común, mostrando los identificadores únicos que aparecen en el cuerpo de una función, y que provienen ya sea desde el ámbito de la unidad de compilación o del alcance del proyecto. Ambas formas de acoplamiento son indeseables para los identificadores globales, pero consideradas peores para los archivos de ámbito. Los puntajes de Linux son mejores que los de los otros sistemas en el caso del acoplamiento común en el ámbito global, pero, y probablemente debido a esto, sus resultados son peores cuando se analiza en el ámbito de archivo. Todos los demás sistemas están más equilibrados uniformemente.

Otros indicadores relacionados con la estructura de código aparecen en la Tabla 3. El porcentaje de funciones globales indica funciones visibles en todo el sistema. El número en WRK, casi el 100%, es sorprendentemente alto. Sin embargo, puede reflejar el uso de Microsoft de diferentes técnicas, tales como enlazadores en bibliotecas compartidas con símbolos explícitamente exportados para evitar choques de identificadores.

Tabla 3. Métricas de la estructura del código

Métrica	FreeBSD	Linux	Solaris	WRK
% de funciones globales	36.7	21.2	45.9	99.8
% de funciones estrictamente estructuradas	27.1	68.4	65.8	72.1
% de declaraciones etiquetadas	0.64	0.93	0.44	0.28
Promedio de parámetros para funciones	2.08	1.97	2.20	2.13
Profundidad media de anidación máxima	0.86	0.88	1.06	1.16
Cadenas por declaración	9.14	9.07	9.19	8.44
% de cadenas en código replicado	4.68	4.60	3.00	3.81
Promedio de la complejidad de las estructuras en las funciones	7.1×10^4	1.3×10^8	3.0×10^6	6.6×10^5

Las funciones estructuradas estrictamente son aquellas que siguen las reglas de la programación estructurada: un único punto de salida y sin sentencias *goto*. La simplicidad de estas funciones hace que sea más fácil razonar acerca de ellas. Su porcentaje se calcula observando el número de palabras clave dentro de cada línea. En la misma línea el porcentaje de declaraciones etiquetadas indica orígenes *goto*: una violación grave de los principios de la programación estructurada. Aquí se midieron declaraciones etiquetadas en lugar de sentencias *goto*, porque muchos de los orígenes sucursales son más confusos que muchas fuentes sucursales. A menudo se utilizan múltiples sentencias *goto* a una sola etiqueta para salir de la función mientras se realiza una pequeña limpieza, equivalente de una cláusula de excepción *finally*.

El número de argumentos a una función es un indicador de la calidad de la interfaz: cuando se debe pasar muchos argumentos empaquetados en una sola estructura se reduce el desorden y abre oportunidades para la optimización en estilo y rendimiento. Dos métricas de seguimiento a la comprensibilidad del código son la profundidad media de anidación máxima y el número de cadenas por declaración. Estas métricas se basan en la teoría de que tanto las estructuras profundamente anidadas como las declaraciones largas son difíciles de comprender [37]. El código replicado se asocia con las fallas [38] y los problemas de mantenibilidad [31]. La métrica correspondiente (% de cadenas en código replicado) muestra el porcentaje que participa en por lo menos un conjunto clon. Para obtener esta métrica se utilizó la herramienta CCFinderX para localizar las líneas de código duplicado y un *script* de Perl para medir la relación de dichas líneas. Por último, para calcular la complejidad media de las estructuras en las funciones se utilizó nuevamente la métrica de Henry y Kafura [15], para mirar las relaciones entre las funciones. Lo ideal sería que este número fuera bajo, lo que indica una separación adecuada entre proveedores y consumidores de funcionalidad.

4.3 Estilo del código

Algunas características de sangría, espaciado, nombres de identificadores, representaciones de constantes y convenciones de nombres pueden afectar la presentación de código sin alterar su funcionalidad [39-42]. En otros casos, la consistencia es más importante que la convención del estilo de código específico elegida. Para este estudio se midió la consistencia de estilo de cada sistema mediante la aplicación del programa de formato *indent* sobre el código fuente completo de cada uno, y contando las líneas que lo modifican. Los resultados se muestran en la Tabla 4. El comportamiento de *indent* se puede modificar utilizando varias opciones con el fin de que coincida con las directrices de un formato de estilo.

Al analizar la distribución de la longitud de dos clases de identificadores importantes de C: los objetos visibles a

nivel global (variables y funciones) y las variables utilizadas, se identifican los agregados (estructuras y uniones). Debido a que cada clase suele utilizar un único espacio de nombres, es importante elegir nombres distintos y reconocibles [43]. Para estas clases de identificadores es preferible nombres más largos y WRK sobresale en ambos casos, como cualquiera programador que haya utilizando la API de Windows podría verificar fácilmente. Otros indicadores relacionados con el estilo del código aparecen en la Tabla 4. La consistencia se determinó a través de la inspección a la convención de código utilizada para denominar *typedefs* y *aggregate tags*, y luego se contaron los identificadores de las clases que no responden a la convención.

Tabla 4. Métricas de estilo del código

Métrica	FreeBSD	Linux	Solaris	WRK
% de líneas conformes al estilo	77.27	77.96	84.32	33.30
% de identificadores conformes al estilo	57.1	59.2	86.9	100.0
% de etiquetas conformes al estilo	0.0	0.0	20.7	98.2
Caracteres por línea	30.8	29.4	27.2	28.6
% de constantes numéricas en operadores	10.6	13.3	7.7	7.7
% de uso de macros	3.99	4.44	9.79	4.04
% palabras mal escritas con comentarios	33.0	31.5	46.4	10.1
% palabras mal escritas sin comentarios	6.33	6.16	5.76	3.23

Las siguientes son métricas orientadas a la identificación de prácticas de programación que típicamente desalientan las directrices de estilo:

- Líneas de código demasiado largas (métrica de caracteres por línea).
- El uso directo de números mágicos en el código (% de constantes numéricas).
- La definición de macros como funciones que pueden comportarse mal cuando se colocan después de una sentencia *if* (% de uso de macros).

Otro elemento importante del estilo son los comentarios en el código, aunque es difícil juzgar objetivamente la calidad de los mismos. Los comentarios pueden ser superfluos o incluso equivocados. No es posible juzgar la calidad de la programación en ese nivel pero se puede medir fácilmente la densidad de comentarios. En los archivos de cabecera se midió la relación de elementos definidos que normalmente requieren un comentario explicativo (enumeraciones, agregados y sus miembros, declaraciones de variables y macros con funciones similares). En este sentido y con notable consistencia, las puntuaciones WRK son mejores que las de los otros sistemas, y las de Linux las peores. Curiosamente el valor medio de la densidad de comentario es mucho más alto que la media, lo que indica que algunos archivos requieren sustancialmente más comentarios que otros. También se midió el número de errores de ortografía en los comentarios como un *proxy* para su calidad. Para esto se corrió el texto de los comentarios a través del corrector ortográfico *aspell* y con un diccionario personalizado que consta de todos los identificadores del sistema y los nombres de los archivos. El bajo número de errores en WRK refleja la corrección ortográfica explícita que de acuerdo con la documentación adjunta se lleva a cabo antes que se liberé el código.

Aunque no se midió objetivamente la portabilidad, el trabajo involucrado en el procesamiento del código fuente con CScout permitió obtener una sensación de la portabilidad del código fuente de cada sistema para distintos compiladores. El código de Linux y WRK parece ser el más fuertemente unido a un compilador específico. Linux utiliza numerosas extensiones de lenguaje previstas por el compilador GNU C, a veces incluyendo código ensamblador apenas disimulado. WRK utiliza considerablemente menos extensiones de lenguaje, pero depende en gran medida de la extensión *try catch* de C, que soporta el compilador Microsoft. El *kernel* de FreeBSD solamente utiliza unas pocas extensiones *gcc*, que a menudo se aíslan en el interior de macros de envoltura. El *kernel* de OpenSolaris representó una agradable sorpresa: fue el único cuerpo de código fuente que no requirió ninguna extensión a CScout para compilar.

4.4 Pre-procesamiento

La relación entre el propio lenguaje C y su pre-procesador se puede describir mejor como incómoda. Aunque los programas de C y de la vida real se basan en gran medida en el pre-procesador, sus características crean a menudo problemas de portabilidad, mantenibilidad y fiabilidad. El pre-procesador, como instrumento poderoso pero contundente, causa estragos en los ámbitos del identificador, la capacidad de analizar y refactorizar código sin procesar y las formas en que el código se compila en diferentes plataformas. Por lo tanto, la mayoría de las directrices de programación C recomiendan moderación en el uso de las construcciones del pre-procesador. También por esta razón los lenguajes modernos basados en C han tratado de reemplazar las características proporcionadas por el pre-procesador C con alternativas más disciplinadas. Por ejemplo, C++ proporciona constantes y plantillas potentes como alternativas a las macros de C, y C# proporciona la funcionalidad del pre-procesador solamente para ayudar a la compilación condicional y a los generadores de código.

El uso de las características del pre-procesador se puede medir por la cantidad de expansión o contracción que se produce cuando se corre sobre el código. En este trabajo se aplicaron dos medidas: una para el cuerpo de funciones (que representan la expansión de código) y una para los elementos por fuera del cuerpo de funciones (que

representan las definiciones y declaraciones de datos). Las dos mediciones se realizaron mediante el cálculo de la relación de *tokens* que llegan al pre-procesador, y para aquellos que salen de él. Tanto la expansión como la contracción son preocupantes. Expansión significa la ocurrencia de macros complejas y contracción es un signo de compilación condicional, que también se considera perjudicial [44]. Por lo tanto, los valores de estas métricas deben rondar alrededor de 1. En el caso de las funciones, las puntuaciones de OpenSolaris son mejores que las de los otros sistemas y las de FreeBSD peores, mientras que en el caso de archivos las puntuaciones de WRK son sustancialmente peores que todos los demás sistemas. Otras cuatro métricas que figuran en la Tabla 5 miden usos cada vez más inseguros del pre-procesador:

- Directivas en archivos cabecera (a menudo requeridas).
- Directivas *no-#include* en archivos C (rara vez necesarias).
- Directivas del pre-procesador en funciones (de dudoso valor).
- Condicionales del pre-procesador en funciones (un riesgo de portabilidad).

Tabla 5. Métricas de pre-procesamiento

Métrica	FreeBSD	Linux	Solaris	WRK
% de directivas del preprocesador en archivos cabecera	22.4	21.9	21.6	10.8
% de directivas *no-#include* en archivos C	2.2	1.9	1.2	1.7
% de directivas del preprocesador en funciones	1.56	0.85	0.75	1.07
% de condicionales del preprocesador en funciones	0.68	0.38	0.34	0.48
% de macros como funciones en funciones definidas	26	20	25	64
% de macros en identificadores únicos	66	50	24	25
% de macros en identificadores	32.5	26.7	22.0	27.1

Las macros del pre-procesador normalmente se utilizan en lugar de variables (macros *objectlike*) y funciones (macros como funciones). En el C moderno las macros como objeto a menudo se pueden sustituir a través de miembros de enumeración, y las macros como funciones a través de funciones en línea. Ambas alternativas se adhieren a las reglas de ámbito de bloques C y son, por lo tanto, considerablemente más seguras que las macros cuyo alcance típicamente se extiende a una amplia unidad de compilación. Las últimas tres métricas de uso del pre-procesador en la Tabla 5 miden la ocurrencia de macros como funciones y como objeto. Dada la disponibilidad de alternativas viables y los peligros asociados con las macros, idealmente todos deberían tener valores bajos.

4.5 Organización de datos

El conjunto final de medidas se refiere a la organización de los datos de cada *kernel* (en memoria). Una medida de la calidad de esta organización en código C puede ser determinada por la definición del alcance de los identificadores y el uso de estructuras. A diferencia de muchos lenguajes modernos, C proporciona algunos mecanismos para controlar la contaminación del espacio de nombres. Las funciones pueden ser definidas solamente con dos posibles alcances (archivo y global), las macros son visibles a través de toda la unidad de compilación en la que se definen, y las etiquetas agregadas suelen vivir juntas en el espacio de nombres globales. Por razones de mantenibilidad esto es importante para los sistemas a gran escala, como los cuatro examinados en esta investigación, para usar juiciosamente los pocos mecanismos disponibles para controlar el gran número de identificadores que pueden entrar en conflicto.

En este trabajo se buscó el nivel de contaminación del espacio de nombres en archivos C y los resultados promedian el número de identificadores y macros que son visibles en el inicio de cada función. Con un promedio de cerca de 10.000 identificadores visibles en un momento dado a través de los sistemas examinados, es obvio que la contaminación del espacio de nombres es un problema en el código. Sin embargo, las cifras de FreeBSD son mejores que la de los otros sistemas y las de WRK las peores. Las tres primeras medidas en la Tabla 6 examinar cómo se ocupa cada sistema de su escaso recurso de nombres y de los identificadores de variables globales.

Tabla 6. Métricas de organización de datos

Métrica	FreeBSD	Linux	Solaris	WRK
% de variables declaradas con alcance global	0.36	0.19	1.02	1.86
% de operandos declarados con alcance global	3.3	0.5	1.3	2.3
% de identificadores con alcance global erróneo	0.28	0.17	1.51	3.53
% de variables declaradas con alcance archivo	2.4	4.0	4.5	6.4
% de operandos declarados con alcance archivo	10.0	6.1	12.7	16.7
Variables por *definición de tipos o agregados*	15.13	25.90	15.49	7.70
Elementos de datos por agregados o enumeración	8.5	10.0	8.6	7.3

Con el fin de reducir al mínimo la contaminación del espacio de nombres sería conveniente minimizar el número de declaraciones de variables que tienen lugar en el alcance global. Por otra parte, al reducir al mínimo el porcentaje de operandos que se refieren a las variables globales se reduce el acoplamiento y se disminuye la carga cognitiva en el lector de código (los identificadores globales se pueden declarar en cualquier parte de los millones de líneas que componen el sistema). La última métrica relativa a objetos globales cuenta identificadores

que se declaran como globales, pero que podrían haber sido declarados con un alcance estático ya que se acceden solamente dentro de un archivo. Las siguientes dos métricas miran las declaraciones de variables y operandos con alcance de archivo. Estas son más benignas que las variables globales, pero peores que las variables con alcance a nivel de bloque.

Las últimas dos métricas relativas a la organización de datos proporcionan una medida aproximada de los mecanismos de abstracción utilizados en el código. El tipo y las definiciones agregados son los dos principales mecanismos de abstracción de datos a disposición de los programas en C. Por lo tanto, el número de declaraciones de variables que corresponden a cada tipo o definición agregada proporciona una indicación de cuánto se han empleado estos mecanismos de abstracción. Estas declaraciones miden el número de elementos de datos por agregado o enumeración en relación con los elementos de datos como conjunto. Esto es similar a la métrica de Chidamber y Kemerer [45], quienes aplican métodos orientados a objeto ponderados por clase para codificar. Un valor alto podría indicar que una estructura intenta almacenar demasiados elementos dispares.

5. Conclusiones

Hay que aclarar que los pesos de las métricas en esta investigación no están calibrados de acuerdo con su importancia; además, están lejos de ser claras porque son funcionalmente independientes, por lo que pueden no proporcionar una imagen completa o representativa de la calidad del código C. Finalmente, la colocación de las marcas es subjetiva, tratando de identificar los casos claros de diferenciación en las métricas particulares. Sin embargo, al observar la distribución y la agrupación de las marcas se puede llegar a algunas conclusiones importantes y plausibles.

El hallazgo más interesante que llamó la atención de los investigadores es la similitud de los valores entre los sistemas. A través de diversas áreas y muchas métricas diferentes, los cuatro sistemas analizados usan procesos ampliamente diferentes pero con puntuación comparable. Los resultados indican por lo menos que la estructura y los atributos de calidad interna de un amplio y complejo artefacto de software representan ante todo los formidables requisitos de ingeniería de su construcción, con la influencia de algunos procesos marginales. Al construir un sistema operativo del mundo real, tales como las unidades de control electrónico de un auto, un sistema de control del tráfico aéreo, o el software para el aterrizaje de una sonda en Marte, no importa si el equipo hace la gestión de desarrollo mediante software propietario o de código abierto, lo importante es no escatimar en calidad. Esto no significa que el proceso sea irrelevante, sino que los procesos compatibles con los requisitos del artefacto conducen a resultados más o menos similares. En el campo de la arquitectura de este tipo de fenómenos se ha popularizado el lema: *la forma sigue a la función* [46].

También se puede llegar a conclusiones interesantes al agrupar métricas particulares. Linux se destaca en varias métricas de estructura de código, pero se queda corto en el estilo del código. Esto podría atribuirse al trabajo de los programadores, porque aunque brillantes y motivados no logran, sin embargo, efectivamente prestar atención a los detalles de estilo. Por el contrario, las altas calificaciones de WRK en el estilo de código y sus marcas bajas en la estructura de código podrían atribuirse a un efecto contrario: los programadores están efectivamente micro-gestionados para preocuparse por los detalles de estilo, pero no se les da suficiente libertad creativa para desarrollar técnicas, patrones de diseño y herramientas que les permitan conquistar la complejidad a gran escala.

Las altas calificaciones de OpenSolaris en pre-procesamiento también podrían atribuirse a la disciplina de programación. Los problemas derivados del uso del pre-procesador son bien conocidos, pero su atractivo es seductor. A menudo es tentador usarlo para crear elaboradas construcciones de programación específicas de dominio. También es fácil a menudo solucionar un problema de portabilidad mediante directivas de compilación condicional. Sin embargo, ambos enfoques pueden ser problemáticos a largo plazo y se puede plantear la hipótesis de que en una organización como Sun los programadores son animados a confiar en el pre-procesador.

Un interesante *cluster* definitivo aparece en las marcas bajas de uso de pre-procesador en el *kernel* de FreeBSD. Esto podría atribuirse a la edad del código base en conjunto con una actitud de programación *gung-ho*, que asume que el código será leído por desarrolladores, al menos, tan inteligentes como el que lo escribió. Sin embargo, el nivel particularmente bajo de contaminación del espacio de nombres a través del código fuente de FreeBSD podría ser el resultado de usar el pre-procesador para configurar y acceder de forma conservadora el alcance de las estructuras de datos.

A partir de los resultados de esta investigación se puede observar, a pesar de varias pretensiones en cuanto a la eficacia de determinados métodos de desarrollo de fuente abierta o cerrada, que no existe un claro ganador (o perdedor). Un sistema con un pedigrí comercial (OpenSolaris) tiene el mayor equilibrio entre las métricas positivas y negativas. Por otro lado, WRK tiene el mayor número de métricas negativas y OpenSolaris tiene el segundo número más bajo de métricas positivas. En cuanto a los sistemas de código abierto, aunque FreeBSD tiene el mayor número de métricas negativas y el menor de positivas, Linux tiene el segundo mayor número de métricas positivas. Por lo tanto, lo más que se puede leer en el balance global de métricas es que los enfoques de desarrollo de código abierto no producen software con una marcadamente mayor calidad que los de desarrollo de software propietario.

Referencias

[1] Stamelos, I. et al. (2002). Code quality analysis in open source software development. Information Systems Journal 12(1), pp. 43-60.

[2] Kuan, J. (2003). Open source software as lead user's make or buy decision: A study of open and closed source quality. Proceedings second conference on the Economics of the Software and Internet Industries (pp. 1-39). January 17-18. Toulouse, France.

[3] Paulson, J., Succi, G. & Eberlein, A. (2004). An empirical study of open-source and closed-source software products. IEEE Transactions on Software Engineering 30(4), pp. 246-256.

[4] Hoepman, J. & Jacobs, B. (2007). Increased security through open source. Communications of the ACM 50(1), pp. 79-83.

[5] Glass, R. (1999). Of open source, Linux...and hype. IEEE Software 16(1), pp. 126-128.

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

[7] Spinellis, D. & Szyperski, C. (2004). How is open source affecting software development? IEEE Software 21(1), pp. 28-33.

[8] Feller, J. (2005). 5-WOSSE: Proceedings of the fifth Workshop on Open Source Software Engineering. USA: ACM Press.

[9] Feller, J. et al. (2005). Perspectives on free and Open Source Software. Cambridge: MIT Press.

[10] von Krogh, G. & von Hippel, E. (2006). The promise of research on open source software. Management Science 52(7), pp. 975-983.

[11] Capiluppi, A. & Robles, G. (2007). Proceedings of the first international workshop on emerging trends in floss research and development. USA: IEEE Computer Society.

[12] Sowe, S., Stamelos, I. & Samoladas, I. (2007). Emerging free and open source software practices. Hershey: IGI Publishing.

[13] Stol, K. et al. (2009). The use of empirical methods in open source software research: Facts, trends and future directions. Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (pp. 19-24). May 18. Vancouver, Canada.

[14] Spinellis, D. (2008). A tale of four kernels. Proceedings 30th International Conference on Software Engineering (pp. 381–390). May 10-18. Leipzig, Germany.

[15] Henry, S. & Kafura, D. (1981). Software structure metrics based on information flow. IEEE Transactions on Software Engineering 7(5), pp. 510–518.

[16] Yu, L. et al. (2004). Categorization of common coupling and its application to the maintainability of the Linux kernel. IEEE Transactions on Software Engineering 30(10), pp. 694-706.

[17] Yu, L. et al. (2006). 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), pp. 807-815.

[18] Izurieta, C. & Bieman, J. (2006). The evolution of FreeBSD and Linux. Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (pp. 204-211). September 21-22. Rio de Janeiro, Brazil.

[19] Samoladas, I. et al. (2004). Open source software development should strive for even greater code maintainability. Communications of the ACM 47(10), pp. 83-87.

[20] Coleman, D. et al. (1994). Using metrics to evaluate software system maintainability. Computer 27(8), pp. 44-49.

[21] Salus, P. (1994). A quarter century of UNIX. Boston: Addison-Wesley.

[22] Tanenbaum, A. (1987). Operating Systems: Design and Implementation. Englewood Cliffs: Prentice Hall.

[23] Torvalds, L. & Diamond, D. (2001). Just for Fun: The story of an accidental revolutionary. New York: Harper Information.

[24] Polze, A. & Probert, D. (2006). Teaching operating systems: The Windows case. Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (pp. 298-302). March 1- 5. Houston, USA.

[25] Cusumano, M. (1998). Microsoft secrets: How the world's most powerful software company creates technology, shapes markets and manages people. New York: The Free Press.

[26] Feller, J. & Fitzgerald, B. (2001). Understanding Open Source software development. Reading: Addison-Wesley.

[27] Jørgensen, N. (2001). Putting it all in the trunk: Incremental software development in the FreeBSD open source project. Information Systems Journal 11(4), pp. 321-336.

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

[29] Spinellis, D. (2003). Global analysis and transformations in preprocessed languages. IEEE Transactions on Software Engineering 29(11), pp. 1019-1030.

[30] Spinellis, D. (2010). CScout: A refactoring browser for C. Science of Computer Programming 75(4), pp. 216-231.

[31] Spinellis, D. (2006). Code Quality: The Open Source perspective. Boston: Addison-Wesley.

[32] Harbison, S. & Steele, G. (2002). C: A reference manual. Englewood Cliffs: Prentice Hall.

[33] Parnas, D. (1972). On the criteria to be used for decomposing systems into modules. Communications of the ACM 15(12), pp. 1053-1058.

[34] McCabe, T. (1976). A complexity measure. IEEE Transactions on Software Engineering 2(4), pp. 308-320.

[35] Gill, G. & Kemerer, C. (1991). Cyclomatic complexity density and software maintenance productivity. IEEE Transactions on Software Engineering 17(12), pp. 1284-1288.

[36] Halstead, M. (1977). Elements of Software Science. New York: Elsevier New Holland.

[37] Cant, S., Jeffery, D. & Henderson, B. (1995). A conceptual model of cognitive complexity of elements of the programming process. Information and Software Technology 37(7), pp. 351-362.

[38] Li, Z. et al. (2006). CP-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on Software Engineering 32(3), pp. 176-192.

[39] Kernighan, B. & Plauger, P. (1978). The elements of programming style. New York: McGraw-Hill.

[40] FreeBSD Project. (1995). Style-Kernel source file style guide. FreeBSD Kernel Developer's Manual: style (9).

[41] Cannon, L. et al. (1991). Recommended C style and coding standards. Online [Feb. 2015].

[42] Stallman, R. et al. (2010). GNU coding standards. Online [Jan. 2015].

[43] McConnell, S. (2004). Code complete: A practical handbook of software construction. Redmond: Microsoft Press.

[44] Spencer, H. & Collyer, G. (1992). #ifdef considered harmful or portability experience with C news. Proceedings of the summer 1992 USENIX Conference (pp. 185-198). June 8-12. San Antonio, USA.

[45] Chidamber, S. & Kemerer, C. (1994). A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20(6), pp. 476-493.

[46] Greenough, H. & Small, H. (1947). Form and function: Remarks on art, design, and architecture. Berkeley: University of California Press.