APROXIMACIONES A LA FIABILIDAD DEL SOFTWARE LIBRE

ENRIQUE A. CHAPARRO
International Association of Cryptologic Research y
Fundación Vía Libre, Argentina
echaparro@vialibre.org.ar

RESUMEN. Este trabajo contribuye a la discusión sobre la seguridad del software libre, procurando determinar si sus características singulares de disponibilidad universal del código fuente y de desarrollo colaborativo influyen positiva o negativamente en su fiabilidad respecto del software privativo. Se demuestra que, con base en los supuestos estándar de la teoría de crecimiento de la fiabilidad, los programas libres pueden alcanzar un nivel de fiabilidad no inferior al de los privativos, y convergen mucho más rapidamente hacia un estado "cero bugs". Se establecen las restricciones de los modelos empleados, y se analiza la influencia que circunstancias de índole práctica tienen en la fiabilidad del software. Se concluye que, en términos generales, el software libre es potencialmente más seguro que el propietario bajo condiciones reales, y se formulan algunas cuestiones aún abiertas a la investigación.

1. INTRODUCCIÓN

La seguridad del software, es decir, la fiabilidad del código para garantizar atributos de confidencialidad, integridad y disponibilidad para la información que trata conforme a los propósitos para los que ha sido concebido, es una preocupación central en el diseño y desarrollo de sistemas de información. La seguridad de los sistemas de información es un proceso complejo, en el que la existencia de software sin defectos constituye una pieza indispensable: el empleo de programas libres de *bugs* no garantiza la seguridad de un sistema de información, pero esta es impensable sin aquellos.

La emergencia del software libre, y su rol cada vez más significativo en el tratamiento de aplicaciones críticas, ha despertado encendidas polémicas acerca de la seguridad relativa de los programas libres respecto de los que no lo son. Muchos argumentos se han cruzado entre los partidarios de unos y otros, a veces respaldados por evidencia empírica, pero la cuestión no parece haber recibido hasta ahora la intensidad de análisis teórico que merece. Este trabajo pretende ser un paso más en el camino de determinar cuál de los modelos es capaz de proporcionar software más seguro, en el caso de que los modos de producción y distribución tuviesen alguna influencia sobre la seguridad.

1.1. "All programs are buggy". Es un fenómeno bien conocido que todo programa no trivial, más allá de una cierta extensión, contiene defectos ("*bugs*" – *vid*., por ejemplo, [1]). De hecho, todo un promisorio campo teórico, el de la teoría de crecimiento de la fiabilidad (*reliability growth theory*) ha crecido alrededor de la necesidad de estudiar la dinámica de evolución de los defectos en el código, y determinar los métodos más eficientes para que los programas converjan a un estado "libre de bugs"[1]. En este estado libre de defectos, una pieza de software puede considerarse segura, aún cuando esta afirmación puede relativizarse en dos sentidos:

- Existen defectos que parecen no tener influencia en la seguridad de los sistemas involucrados, sea porque no afectan a ninguna de las propiedades que se procura

[1]Una explicación detallada sobre este campo teórico está fuera del alcance de este trabajo, pero el lector interesado podrá consultar el trabajo de BIROLI[2] y la bibliografía allí citada, y un análisis de modelos bayesianos puede verse en el trabajo de KOROLEV[3].

preservar, o porque no existe una forma conocida para que un adversario pueda explotar el defecto en su favor[2].

- La invulnerabilidad a todos los ataques conocidos en un momento dado no implica que no surjan posteriormente nuevas formas de ataque que explotan vulnerabilidades hasta entonces desconocidas. El caso de las "*format string vulnerabilities*", desconocidas hasta septiembre de 1999, es un ejemplo dramático de esta condición (*vid.* [4, 5, 6]).

Sin embargo, a los fines de este trabajo, aceptaremos la convención de que un programa es seguro cuando alcanza este estado ideal "libre de bugs". Ello, sin pérdida de la generalidad; es posible sostener que los programas se hallarán en algún punto de la función de convergencia hacia el estado "libre de bugs", sin que necesariamente alcancen el valor "cero bugs" en algún momento.

1.2. Caracterización del problema. Las objeciones centrales de quienes sostienen que el software libre es menos seguro que el privativo se centran en dos aspectos:

1. La disponibilidad universal del código fuente, de modo tal que cualquiera pueda inspeccionarlo y modificarlo, disminuye la seguridad de los programas
   a) porque los potenciales adversarios tienen la posibilidad de estudiarlo de cerca para determinar sus vulnerabilidades (por ejemplo, [8]); o bien
   b) porque depositar mayor confianza en que la realimentación producida por quienes estudien independientemente el código fuente no garantiza procesos metodológicamente correctos de testeo del software (por ejemplo, [9]).
2. El método de desarrollo "bazar" empleado usualmente por los proyectos de software libre implica la incapacidad de controlar el proceso de construcción, por lo que bajo los supuestos normales de la teoría de crecimiento de la confiabilidad convergerá más lentamente hacia un estado "libre de bugs" (por ejemplo, [10]).

En consecuencia, será necesario probar: (a) si la apertura del código fuente favorece más a los defensores que a los atacantes, o viceversa; y (b) si el método de desarrollo "bazar" converge más rápido o más lentamente que el método "catedral" hacia un estado libre de bugs. Al análisis de estas cuestiones, tanto en el plano práctico como el teórico, está consagrado este trabajo.

1.3. Definiciones. Antes de entrar definitivamente en materia, resulta necesario definir algunos términos cuyo significado podría resultar ambiguo:

Software libre: Emplearemos la definición de "Open Source" de la Open Source Initiative, versión 1.9[3]. A los efectos del análisis, la definición también podría hacerse extensiva a software cuya licencia no permite la redistribución de obras derivadas (como el conocido programa *qmail*). Denominaremos *privativo*[4] al software cuya licencia no concede al usuario las libertadas definidas.

Modelo "bazar": El modelo de desarrollo característico de muchos proyectos de software libre (pero no todos ellos), que permite la colaboración amplia y se basa en el criterio de hacer público tempranamente el código, y liberar prontamente sucesivas versiones incrementales ("*release early, release often*"). El modelo ha sido descripto en [11].

[2]Pero esta condición puede cambiar en el tiempo, *vid.* [7] respecto de una condición considerada "no explotable" en el programa *wu-ftpd.*

[3]Ver http://www.opensource.org/docs/definition.php. Esta definición, a su vez, proviene de las Debian Free Software Guidelines.

[4]En la literatura suele llamársele también "propietario", por traducción deformada del inglés. El Diccionario de la Lengua Española, XXII Edición, define **"privativo, va**. (Del lat. privati-vus). 1. adj. Que causa privación o la significa. 2. adj. Propio y peculiar singularmente de alguien o algo, y no de otros." Ambas definiciones parecen singularmente apropiadas.

MTTF: "Mean time to failure", el tiempo promedio de ejecución del programa que transcurrirá hasta que una falla provocada por un defecto se manifieste.

2. DEFECTOS INDUCIDOS Y DEFECTOS ACCIDENTALES

Un programa puede contener defectos accidentales, provocados por errores en el diseño o la codificación, y defectos intencionales, introducidos ex-profeso para permitir su explotación por terceros por cualesquiera razones. Estos últimos son, lamentablemente, bastante comunes; en algunos casos, se trata de "puertas traseras" o mecanismos de acceso inseguro con escalamiento de privilegios, introducidos durante el desarrollo y la prueba del programa con el fin de acelerar los tests o la reparación de defectos, que debido a fallas en el proceso de estabilización del programa son incluidos en las versiones liberadas al uso público. En otros, el creador del programa los introduce intencionalmente para garantizar para sí o para terceros (por ejemplo, organizaciones criminales u organismos de inteligencia estatales) la posibilidad de acceder sin restricciones y sin ser detectado a la información del usuario del programa, alterarla o volverla inaccesible.

Si bien no resulta indispensable acceder al código fuente para encontrar defectos en un programa, pues desde el paper seminal de ED MOORE [12] es conocido que el análisis puramente externo de sistemas de cajas negras puede determinar frecuentemente detalles del estado interno, es también claro que la disponibilidad del código fuente acelera los procesos de descubrimiento de estos defectos. La posibilidad no descartable de que cualquier adversario, teniendo acceso al código fuente, pueda insertar código malicioso en un programa libre, se ve compensada con creces con la posibilidad de que cualquier usuario pueda inspeccionar los mecanismos internos del programa; adicionalmente, los repositorios públicos de software libre suelen emplear medidas de control de integridad de los programas que distribuyen.

El caso de los defectos inducidos intencionalmente no ha sido analizado en particular, pues puede quedar perfectamente incluido sin pérdida de generalidad en los casos que analizaremos en las secciones siguientes. Pero existe abundante evidencia empírica en favor de la seguridad del software libre. El sistema de gestión de base de datos Interbase de Borland es un excelente ejemplo en este sentido: en algún momento entre 1992 y 1994 Borland insertó una puerta trasera intencional en Interbase, que permitía a cualquier usuario local o remoto manipular cualquier objeto de la base de datos e instalar programas arbitrarios. La vulnerabilidad permaneció allí durante no menos de seis años. En julio de 2000 Borland convirtió el programa en libre, y publicó el código fuente. El proyecto "Firebird" comenzó a trabajar en el código fuente, y descubrió este serio problema de seguridad en diciembre de 2000[5].

3. VISIBILIDAD DEL CÓDIGO FUENTE

Analizaremos primero si la visibilidad del código favorece más a los atacantes o a los defensores. Para ello, será preciso determinar cómo se produce el crecimiento de la fiabilidad en sistemas libres y privativos. En los primeros, el código fuente es universalmente visible, es decir, está disponible tanto para atacantes cuanto para defensores. En los segundos, el código está oculto y es sólo visible por un pequeño número de personas (los desarrolladores), aunque como veremos más abajo la influencia relativa de estos desarrolladores en el crecimiento de la fiabilidad tiende a tornarse despreciable en sistemas grandes y complejos. El modelo que presente un mejor crecimiento de la fiabilidad será entonces el más favorable a los defensores, al menos en términos ideales.

[5] Lo que resulta realmente desalentador es que la puerta trasera podía ser hallada simplemente con un "dump" ASCII del programa. Es muy difícil determinar si esta vulnerabilidad fue abusada durante el largo tiempo en que el defecto no fue públicamente conocido.

3.1. Correlación entre MTTF y tiempo de prueba. En principio, debemos considerar un hecho conocido desde hace tiempo en la comunidad de la ingeniería de software de sistemas críticos: para un sistema grande y complejo, serán necesarias t horas de prueba para hallar un defecto con un MTTF igual a t [13]. Este hecho fue observado originalmente en un estudio sobre la historia de los bugs en los sistemas operativos de mainframes IBM [14], y avalado posteriormente por extensivas investigaciones empíricas. El primer modelo teórico explicativo fue publicado en 1996 [15], probando que bajo supuestos estándar este sería el comportamiento en el peor caso. Este resultado fue luego ajustado, demostrándose que hasta un factor constante este es también el comportamiento esperado [16].

El modelo BRADY-ANDERSON-BALL citado más arriba muestra que, después de un largo período inicial de pruebas y remoción de defectos, el efecto neto de los bugs remanentes convergerá a una distribución polinómica en lugar de una exponencial, bajo ciertos supuestos. Así, si la probabilidad de que el i-ésimo bug quede indetectado después de t pruebas al azar es $e^{-E_i t}$, entonces la probabilidad E de una falla de seguridad en el instante t cuando n bugs ya han sido removidos es

$$E = \sum_{i=n+1}^{\infty} e^{-E_i t} \approx K/t \tag{1}$$

para un amplio rango de valores de t. En consecuencia, el tiempo de falla observado por quien prueba depende sólo de la calidad inicial del código (la constante de integración K) y el tiempo hasta entonces invertido en pruebas.

3.2. Tornando la tarea más compleja. ¿Qué sucedería al complicar más la tarea de quien prueba? Supongamos que después de las pruebas alfa iniciales del programa, todas las pruebas que siguen son realizadas por personas sin acceso al código fuente, que sólo pueden intentar diversas combinaciones de input para intentar causar fallas. Si la tarea de quien prueba se hace λ veces más difícil, la probabilidad de que el i-ésimo bug permanezca oculto se volverá $e^{-E_i t/\lambda}$, y la probabilidad de que el sistema falle en la próxima prueba será

$$E = \sum_{i=n+1}^{\infty} e^{-E_i t/\lambda} \approx K/\lambda t \tag{2}$$

Dicho de otro modo, la tasa de falla del sistema ha caído por un factor λ; pero si todas las pruebas hasta el presente se han llevado a cabo bajo este régimen más difícil, sólo $1/\lambda$ parte de las pruebas efectivas se habrán llevado a cabo, los factores λ se cancelan mutuamente, y la probabilidad de falla E no sufrirá cambios. Habremos removido menos bugs, pero la tasa a la que los descubrimos continuará siendo la misma. FENTON y NEIL [17] sugieren que λ está entre 3 y 5 para sistemas maduros.

Consideremos ahora que el software privativo es probado primero por empleados del proveedor con acceso al código fuente, y luego por usuarios y "beta-testers" sin dicho acceso. En un producto de gran volumen, docenas de probadores pueden trabajar durante meses en el código, después de lo cual el producto entrará en la fase de pruebas beta con probadores con acceso al código objeto solamente. Puede haber decenas de miles de "beta-testers"[6], asi que aunque λ sea mucho mayor que lo que Fenton y Neil predicen, el efecto del testeo alfa inicial será rápidamente superado por el mucho mayor esfuerzo total de los "beta testers". El esfuerzo de pruebas beta se convierte rápidamente en la fuerza dominante en el crecimiento de la fi abilidad.

[6]Considerando los niveles de falla en las etapas iniciales de muchos programas privativos ampliamente usados, se podría alegar que en realidad las pruebas beta son realizados por los usuarios finales de programas supuestamente estables, que en lugar de ser remunerados por su trabajo pagan por realizar el testeo.

3.3. Conclusiones del modelo. De acuerdo con el modelo, entonces, podemos esperar que el software libre y el privativo exhibirán un crecimiento de fiabilidad similar, bajo los siguientes supuestos estándar:

- Que hay suficientes bugs como para realizar estadísticas;
- Que los bugs son independientes y están idénticamente distribuidos;
- Que son descubiertos al azar; y
- Que son reparados apenas se los encuentra.

Hemos establecido, pues, una equivalencia teórica en el crecimiento de la fiabilidad entre software libre y privativo. Este teorema de equivalencia fue probado por ROSS ANDERSON [18]. Pero ello no significa, en modo alguno, que esta equivalencia se sostenga en situaciones específicas dadas. Esto se debe, por un lado, a que los supuestos estándar no siempre se cumplen en la realidad; y por otro, que existen numerosos factores adicionales a considerar. Trataremos estas cuestiones más adelante.

4. VELOCIDAD DE CONVERGENCIA HACIA EL ESTADO "CERO BUGS"

Muy recientemente[7] DAMIEN CHALLET y YANN LE DU, del Departamento de Física Teórica de la Universidad de Oxford, desarrollaron un nuevo modelo de dinámica de bugs de software[19]. Este modelo microscópico se diferencia de aproximaciones anteriores[20] y por su interés y novedad lo reproduciremos aquí.

4.1. El modelo Challet-Le Du.

1. El programa está dividido en L partes $i = 1, \cdots, L$; cada parte puede verse como una funcionalidad básica (tal como cargar un archivo). Cada parte tiene M subpartes. La subparte j de la parte $i (j = 1, \cdots, M)$ o bien está libre de bugs, lo que denotamos por $s_{i,j} = 0$, o tiene bugs ($s_{i,j} = 1$). Al momento t, la subparte i tiene $b_i t = \sum_j s_{i,j}(t)$ bugs, y el número total de errores es $B(t) = \sum_i b_i(t)$. Finalmente, el número de partes que contienen al menos un defecto es $D(t) = \sum_i \Theta(b_i(t))$ donde $\Theta(x) = 1$ si $x > 0$ y 0 en caso contrario es la función de Heavyside.
2. Hay N_u usuarios. Por hipótesis, cada usuario usa una parte del programa en cada intervalo de tiempo, y reporta el comportamiento defectuoso con una probabilidad

$$P_u = \delta b_i / M \tag{3}$$

   es decir, la fracción de defectos en la parte i multiplicada por un factor que toma en cuenta el número medio de subpartes usadas en cada intervalo temporal y su propensión a reportar bugs. P_u debe ser proporcional a b_i.
3. Cada reporte de bugs consiste sólo del número de la parte defectuosa (porque, por ejemplo, si la carga del archivo falla el usuario no puede describir con más detalles las fallas del programa). En consecuencia, la lista de bugs es una tabla que indica qué partes contienen bugs. El número de partes defectuosas reportadas es $R(t)$.
4. Hay N_p programadores. Además de encontrar bugs conforme a 3, cada uno de ellos intenta corregir un bug extraído al azar de la lista y revisa todas las subpartes de una parte dada. Se supone que este proceso permite reparar una subparte defectuosa con probabilidad ϕ e introducir defectos en una subparte correcta con probabilidad β. En términos matemáticos,

$$P_p(s_{i,j} = 1 \to s'_{i,j} = 0) = \phi \tag{4}$$

$$P_p(s_{i,j} = 0 \to s'_{i,j} = 1) = \beta \tag{5}$$

   Una vez que el programador ha modificado el código, envía un patch al mantenedor.

[7] La última versión corregida fue prepublicada el 25 de julio de 2003.

5. El rol del mantenedor (que puede ser al mismo tiempo un programador) es determinar si un parche mejora o no el código. Supondremos que el mantenedor mide el número de defectos en el código actual $d(s_i)$ y en la nueva versión propuesta $d(s_i')$. La medición se realiza de este modo: si la subparte j de la característica i tiene bugs, el mantenedor detecta la circunstancia con probabilidad υ y puede clasificar correctamente una parte que opera correctamente con probabilidad ω. Luego, el mantenedor acepta el parche si cree que contiene menos bugs que la versión actual ($d(s', i) < d(s, i)$), y actualiza la lista de bugs.
6. Conforme al modelo de desarrollo:
   a) Código libre: todos los usuarios usan el código modificado en el momento $t+1$ y reportan bugs exclusivamente sobre el código actualizado. Esto supone que todos los usuarios actualizan su software con cada release.
   b) Código cerrado: el programa modificado se pone a disposición de los usuarios cada $T > 1$ pasos temporales. Entre dos releases, los usuarios continúan reportando bugs sobre el último release, y los programadores trabajan sobre código aún no liberado.
7. Los pasos 2 a 6 se repiten hasta que no quedan mas bugs en el código.

Este modelo se diferencia de trabajos previos en el campo por dos características singulares: 1) el programa se divide en partes que contienen subpartes, lo cual causa el lento descenso del número de bugs $R(t)$ cuando M es muy grande, y 2) la retroalimentación de la fracción de bugs a la tasa de descubrimiento de bugs (ecuación 3) es uno de los supuestos centrales del modelo, responsable de la posibilidad de convergencia hacia el estado "libre de bugs" aún con programadores y mantenedor imperfectos.

En un contexto de software privativo, los programadores enfrentan un dilema cuando un usuario informa que una parte es defectuosa cuando ya ha sido reparada desde el último release. En efecto, los usuarios reportan bugs del último release, mientras los programadores trabajan sobre el siguiente, sin sincronía. Los programadores podrían ignorar la información, o modificar nuevamente el código actual; en este último caso la modificación podría ser sistemática, o posterior a verificar de que la parte en cuestión en el código actual también es defectuosa (conforme a la ecuación 3). Sin verificación, $D(t)$ no decrecerá monótonamente, pues una parte libre de bugs podría ser afectada por este proceso.

4.2. Resultados. La figura 1 muestra el comportamiento dinámico de proyectos exitosos con estado inicial $s_{i,j} = 1 \; \forall \; i, j$ en términos del número de partes defectuosas $D(t) \; \alpha \exp(-\lambda t)$ para t grande. A nivel más microscópico es posible distinguir dos fases: en la temprana, los usuarios hallan y reportan muchos bugs, manteniendo $R(t) \gg N_p$. La gran mayoría de los bugs se reparan durante esta fase, en la que $B(t)$ decrece linealmente con el tiempo. Cuando $R(t) \sim N_p$, aparece un régimen más lento en que $B(t) \sim \exp(-t/\tau)$; en este régimen, el número de bugs por parte defectuosa $B(t)/D(t)$ es pequeño y fluctúa alrededor de un valor que depende de los parámetros escogidos.

El modelo muestra también que el tiempo requerido para obtener un proyecto "libre de bugs" crece en forma aproximadamente lineal con T, y por lo tanto los proyectos de software privativo son siempre más lentos en alcanzar un estado perfecto. La razón de ello es simple: después de cada release, el número de reportes de bugs relevantes provenientes de los usuarios cae rápidamente a cero, alternándose los regímenes rápido y lento. Otro hallazgo importante, a partir de la aplicación al modelo de parámetros del desarrollo de un proyecto libre grande, complejo y muy conocido (el kernel Linux™) muestra que es posible sostener un crecimiento supralineal de software sin desmedro de la calidad si se dispone de un número suficiente de programadores calificados, contradiciendo las "leyes" de Lehmann sobre crecimiento de software[21].

4.3. Conclusiones del modelo. El modelo permite entonces concluir que los proyectos libres desarrollados con metodología "bazar" convergen siempre más rápido hacia el estado

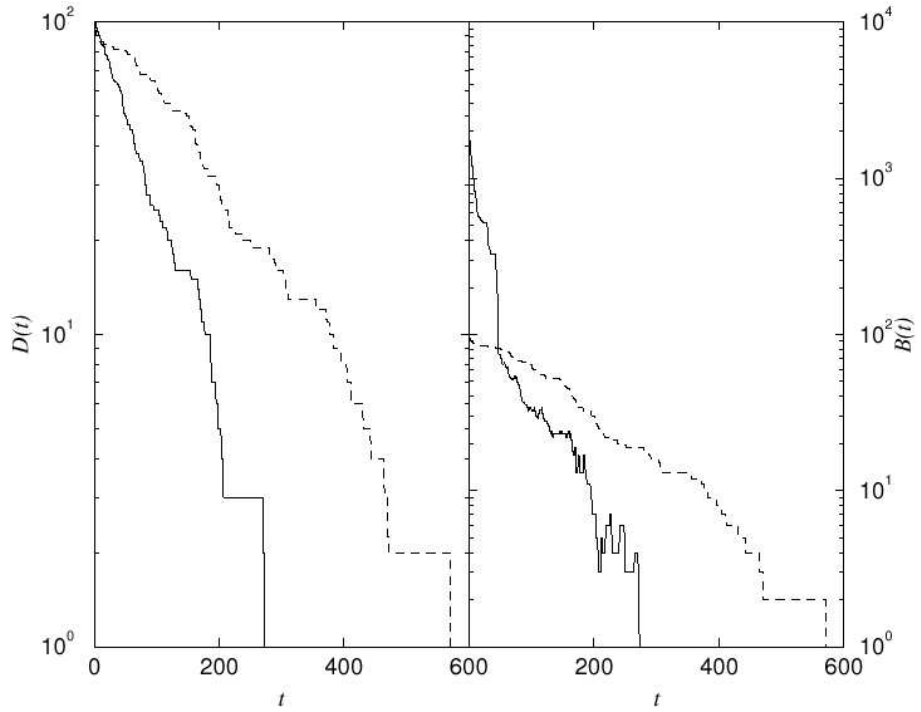

FIGURA 1. Número de partes defectuosas (izquierda) y número de bugs (derecha) en un proyecto libre (líneas continuas) y en un poyecto privativo sin re-presentación de reportes de bugs permitida entre releases (línea punteada) con $L = 100$, $M = 20$, $N_u = 100$, $N_p = 10$, $\delta = 1$, $\phi = 0{,}9$, $\beta = 0{,}1$, $\omega = 0{,}9$, $\upsilon = 0{,}9$, $T = 1$ para código libre y $T = 50$ para código privativo. Extraído de [19].

"libre de bugs" que los proyectos de software privativo, a igualdad de los demás parámetros. Esto confirma el argumento central de [11]. No obstante ello, esto no significa que los proyectos libres "bazar" sean siempre más rápidos: proyectos privativos con un mejor conjunto de parámetros (más y/o mejores programadores, más usuarios que reporten defectos) podrían obtener mejor resultado que los libres aún con largo tiempo entre releases.

5. CONFRONTANDO LA REALIDAD

Como conclusión de las secciones anteriores, hallamos que en términos ideales la visibilidad universal del código fuente no afecta adversamente la seguridad de los programas libres, y que estos – cuando se desarrollan bajo el modelo "bazar" – convergen más rápidamente a un estado "libre de bugs" que los privativos, a igualdad de parámetros. Ahora bien, estas conclusiones resultan útiles para acotar la cuestión en el marco teórico, pero las circunstancias del mundo real son usualmente lo suficientemente complejas como para poder ser capturadas en los modelos simples que hemos revisado. Intentaremos, pues, incorporar al análisis algunas de estas variables exógenas.

5.1. Limitaciones de los modelos. Es posible observar que la principal limitación imbuida en los modelos de las secciones anteriores es la hipótesis de independencia de los bugs. Desafortunadamente, esta independencia no existe: las partes de un programa están vinculadas a través de una red libre de escala[22] altamente asimétrica[23] y en consecuencia los bugs pueden propagarse en ese grafo y afectar otras partes. Esta y otras distribuciones no uniformes, como la posibilidad de que la disminución de bugs no resulte

exponencial sino una ley de potencia[8], o la asignación de prioridades relativas de reparación de bugs, pueden cambiar la dinámica de los modelos. Adicionalmente, el supuesto de que todo bug descubierto es reparado presente en ambos modelos es muchas veces erróneo en la realidad; el estudio ya citado de ADAMS [14] indica que IBM reparaba los bugs de sistemas operativos de mainframe a la octava vez que eran reportados, y la dinámica de emisión de paquetes de reparaciones ("service packs", "program temporary fixes", "fix packages") por parte de los proveedores de software privativo muestra que estos no suelen seguir un mecanismo de inmediatez a menos que se trate de bugs de alta criticidad. Por otro lado, algunos proyectos libres como *OpenBSD*[9] exhiben una preocupación extrema por auditar permanentemente el código y reparar inmediatamente cualquier falla.

5.2. Los actores. Otro problema de la simplicidad de los modelos es que los programas son hechos y usados por personas. Cada persona es singular: sus habilidades, su propensión al riesgo, su disposición para colaborar, sus motivaciones, pueden ser distintas; y la combinación de todos esos factores seguramente hará de cada individuo un caso único. Entonces, los resultados en la práctica variarán significativamente en función de las cualidades individuales de desarrolladores y usuarios. Como señalamos antes, de la aplicación del modelo de Challet y Le Du es posible inferir que los programadores del kernel Linux™ son especialmente hábiles, pues esta habilidad es la que permite sostener el crecimiento supralineal de un sistema que es conocido como razonablemente estable[10]; pero ¿es este fenómeno generalizable a todos los proyectos de software libre? Seguramente no, si observamos la evidencia del mundo real.

Pero, por otra parte, de la aplicación de la teoría económica al campo de la fiabilidad de software es posible extraer algunas guías sobre la variación en la provisión de un bien público (fiabilidad del software, en este caso) en términos de agregación social, con las tecnologías analizadas por Hirshleifer [24]: esfuerzo total, eslabón más débil ("weakest link") y mejor opción ("best shot")[11]. Varian [25] aporta interesantes conclusiones, entre las cuales: los sistemas se tornarán crecientemente fiables con el incremento del número de agentes en el caso de esfuerzo total, pero decrecientemente fiables con el aumento del número de agentes en el caso del eslabón más débil. Si extendemos legítimamente la hipótesis de Varian para incluir a los usuarios entre los agentes, hallaremos que la mayor propensión a cooperar en las comunidades formadas alrededor de los proyectos de software libre resulta en un mayor empleo de la tecnología de esfuerzo total. Ello explica en parte el éxito de muchos proyectos libres en alcanzar rápidamente altas tasas de fiabilidad.

Adicionalmente, factores motivacionales inciden sobre la calidad del código. Por un lado, el hecho de que el código sea universalmente visible motivará a los programadores a escribir más elegante y cuidadosamente que en el caso en que el código fuente sea sólo accesible a un número muy limitado de personas. Por otra, como sostiene Rijmen, porque el modelo colaborativo del software libre obliga a la gente a escribir código más claro y adherir a estándares. Finalmente, mientras que mucho del software libre es construido mediante esfuerzo voluntario atrayendo a programadores con habilidad y motivación, o en proyectos académicos que incorporan estudiantes graduados del cuartil superior de carreras informáticas e ingeniería, los plazos comerciales pueden imponer presiones extraordinarias a los

[8] De hecho, es observable que el número de modificaciones por programador en el kernel Linux entre las versiones 2.5.5 y 2.5.69 es una ley de potencia truncada, así como lo es el número de bugs asignados y corregidos por programador en el proyecto Mozilla. En el primer caso, para un número de modificaciones m por programador y su rango r, una función de aproximación es $r(m) \sim r^{-0,55} \exp(-C_m x)$. En el segundo, para b bugs asignados por programador y su rango r, una función de aproximación es $r \sim b^{-0,38} \exp(-C_b b)$.

[9] http://www.openbsd.org/

[10] Al menos, tanto o más estable que sistemas operativos privativos.

[11] En la tecnología de esfuerzo total, la fiabilidad dependerá de la suma de esfuerzos ejercidos por los individuos; en el caso del eslabón más debil, la fiabilidad dependerá del mínimo esfuerzo; y en el de la mejor opción, dependerá del esfuerzo máximo. En la práctica, la mayoría de los sistemas exhiben una mezcla de los tres casos.

desarrolladores de software privativo, causando que hasta los buenos programadores trabajen menos cuidadosamente. En este plano, entonces, el modelo de desarrollo de software libre cuenta con una ventaja en el factor *K* del modelo Brady-Anderson-Ball.

5.3. Factores que alteran el equilibrio. Existen numerosos factores exógenos al software que tienen incidencia clave en la fiabilidad última de los sistemas:

Incentivos políticos al proveedor. Las organizaciones gubernamentales del país del proveedor de un programa privativo podrían preferir (como en el caso de los Estados Unidos) que las vulnerabilidades en algunos productos sean reportadas primero a las autoridades, de modo que puedan ser explotadas temporalmente por agencias de inteligencia o seguridad, y que los parches sean liberados sólo cuando otros atacantes comienzan a explotar el defecto. En este plano, el software libre se halla en posición ventajosa.

Incentivos de comercialización. La economía de la industria del software (altos costos fijos, costos variables bajos, efectos de red, "lock-in") lleva a mercados con corporaciones dominantes, con fuertes incentivos para despachar rápidamente productos mientras establecen una posición de control. En consecuencia, las corporaciones tenderán a liberar un producto tan pronto como sea "suficientemente bueno"; simultáneamente, dado que reparar bugs lleva tiempo y costo, podría suceder que reparasen sólo los suficientes como para mantenerse a tono con la competencia percibida. Los plazos de puesta en el mercado dirigidos por estas variables económicas hacen que bugs cada vez más severos permanezcan en el código a medida que la fecha de comercialización anunciada se aproxima, si repararlos no es tarea trivial. Estos estímulos son mucho menos intensos en el software libre, por lo que éste toma ventajas también en este plano.

Crecimiento de la complejidad. La teoria de crecimiento de la fiabilidad nos permite esperar que la fiabilidad total estará dominada por las adiciones más recientes de código, así que es altamente probable que nunca obtengamos sistemas realmente en equilibrio, o en estado de "cero bugs". Al mismo tiempo, la velocidad de adición de nuevo código afectará la fiabilidad. En consecuencia, la práctica de las empresas de software privativo de agregar arbitrariamente nuevas funcionalidades a los programas (un fenómeno que algunos denominan "featuritis") por razones económicas, es decir, para maximizar el lock-in y los efectos de red, conspira contra la calidad del código. Los desarrolladores de programas libres intentarán más bien resolver un problema específico que atraer (o cautivar) más clientes[12].

Otros factores. A los citados, cabría agregar otras variables como por ejemplo el efecto de los estados de transición (entre el descubrimiento de un defecto, el desarrollo del parche correspondiente, y su despliegue en los sistemas existentes), la efectividad de las pruebas (las realizadas por probadores hostiles tienden a ser más efectivas [28]) y los incentivos de relaciones públicas sobre el proveedor. En la mayoría de estos casos, la evidencia empírica muestra que los proyectos de software libre mantienen un camino más consistente hacia la provisión de programas seguros.

6. CONCLUSIONES

Las conclusiones a las que arribamos no permiten inferir que un cierto programa libre es más seguro que uno privativo. De hecho, existen programas libres que son una pesadilla en términos de seguridad, programas privativos que muestran una notable fiabilidad, y viceversa. Sin embargo, sí es posible determinar cuál modo de producción y distribución posee mayor potencialidad para crear sistemas más seguros.

[12] Programas grandes, complejos y extensamente utilizados como LaTeX han sido estables, sin agregado de nuevas funcionalidades, por más de diez años.

Por un lado, contamos con los resultados de los modelos de Brady-Anderson-Ball y de Challet-Le Du: la libre disponibilidad del código fuente no hace que el software libre sea menos seguro que el privativo, y si se desarrolla conforme al modelo "bazar" converge más rápidamente hacia el estado "libre de bugs". Por otro, cuando aplicamos a estas conclusiones teóricas un análisis de las condiciones del mundo real, hallamos que bajo la mayoría de los supuestos el software libre es potencialmente más seguro. Sin embargo, convendrá ser prudente en este aspecto, y continuar las investigaciones; mientras tanto, convendrá analizar las cuestiones de crecimiento de la fiabilidad caso por caso, cuando existan desarrollos libres y privativos comparables y datos suficientes para obtener estadísticas.

Parece necesario realizar estudios más detallados de los factores no contemplados en la teoría de crecimiento de la fiabilidad, como los indicados en 5.3. Al mismo tiempo, se requiere estudiar el modelo Challet-Le Du en redes libres de escala, y hallar las leyes de decrecimiento del número de bugs en ese contexto.

Agradecimientos. La idea de este paper comenzó a desarrollarse durante una charla de almuerzo (y sobremesa) con Eric Raymond y Ross Anderson en el simposio sobre seguridad y privacidad de IEEE en Oakland, California, en mayo de 2000. Tiempo después, Ross Anderson tuvo la gentileza y la paciencia de explicarme la prueba del teorema de [18]. Rodolfo Pilas, del UYLUG, fue el primero en advertirme acerca del trabajo de Challet y Le Du. Muchos valiosos comentarios del público en varias conferencias en que presenté versiones preliminares ayudaron a llevar este trabajo a su actual estado. A todos ellos, mi más sincero agradecimiento.

REFERENCIAS

[1] Cheswick, W. R. y S. Bellovin, *Firewalls and Internet Security, Repelling the Wily Hacker*. Reading, MA, Addison-Wesley Publishing Co., 1994, p. 7.

[2] Biroli A., *Reliability Engineering. Theory and Practice*, Berlin, Springer, 1999.

[3] Korolev V. Yu., "Mathematical Theory of Software Reliability Growth", trabajo presentado en la 4th International Conference on Applied Informatics, Eger-Noszvaj, Hungría, septiembre 1999.

[4] Newsham T., *Format String Attacks*, 2000, http://www.guardent.com/docs/FormatString.PDF

[5] Scut/Team Teso, *Exploiting Format String Vulnerabilities 1.2*, 2000, http://www.team-teso.net/articles/formatstring/

[6] Thuemmel A., *Analysis of Format String Bugs v1.0*, 2001, http://downloads.securityfocus.com/library/format-bug-analysis.pdf

[7] Ghosh, A.K., T. O'Connor y G. McGraw, "An Automated Approach for Identifying Potential Vulnerabilities in Software" en *Proceedings – 1998 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, IEEE Computer Society, 1998, pp. 104-114

[8] Brown, K., *Opening the Open Source Debate*, West Lebanon, NH, Alexis de Tocqueville Institution, 2002.

[9] Lipner, S.B., "Security and Source Code Access" en *Proceedings – 2000 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, IEEE Computer Society, 2000, pp. 124-125.

[10] Schneider, F.B., "Open Source and Security: Visiting the Bizarre"en *Proceedings – 2000 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, IEEE Computer Society, 2000, pp. 126-127.

[11] Raymond, E.S., *The Cathedral & the Bazaar. Musings on Linux and Open Source by an Accidental Revolutionary*, Sebastopol, CA, O'Reilly & Associates, 1999.

[12] Moore, E.F., "Gedanken Experiments on Sequential Machines", en C.E. Shannon y J. McCarthy, eds., *Automata Studies*, Annals of Mathematical Studies, 34, Princeton, NJ, Princeton University Press, 1956, pp.129-153.

[13] Butler, R.W. y G.B. Finelli, "The Infeasibility of Experimental Quantification of Life-Critical Software Reliability", en *ACM Symposium on Software for Crytical Systems*, New Orleans, 1991, pp. 66-76.

[14] Adams, E.N., "Optimising Preventive Maintenance of Software Products", en *IBM Journal of Research & Development*, Vol. 28 No. 1, 1984, pp. 2-14.

[15] Bishop, P. y R. Bloomfield, "A Conservative Theory for Long-Term Reliability Growth Prediction", en *IEEE Transactions on Reliability*, Vol. 45 No. 4, 1996, pp. 550-560.

[16] Brady, R.M., R.J. Anderson y R.C. Ball, *Murphy's law, the fitness of evolving species,and the limits of software reliability*, Cambridge University Computer Laboratory Technical Report No. 471, septiembre 1999.

[17] Fenton, N.E. y M. Neil, "A Critique of Software Defect Prediction Models", en *IEEE Transactions on Software Engineering*, Vol. 25 No. 5, sept./oct. 1999, pp. 675-689.

[18] Anderson, R.J., “Security in Open Versus Closed Systems – The Dance of Boltzmann, Coase and Moore”, trabajo presentado en Open Source Software Economics 2002, Institut d’Economie Industrielle, Université Toulouse 1, 2002.
[19] Challet, D. e Y. Le Du, “Closed source versus open source in a model of software bug dynamics”, Theoretical Physics, Oxford University, 2003. Presentado a *Elsevier Science*; preimpreso electrónico disponible en http://arxiv.org/pdf/cond-mat/0306511.
[20] Botting, R., “Some Stochastic Models of Software Evolution”, trabajo presentado en Systemics, Cybernetics and Informatics 2002, Orlando, FL, julio 2002.
[21] Lehman, M.M., “Programs, life cycles and laws of software evolution”, en *IEEE* Vol. 68 (9), 1980, pp. 1060-1076.
[22] Valverde, S., R. Ferrer i Cancho y R.V. Sole, “Scale free networks from optimal design” en *Europhyisics Letters* 60, 2002, pp. 512-517.
[23] Challet, D. y A. Lombardoni, “Asymmetry of software structures”, presentado a *Physical Review 3*, 2003; prepublicación electrónica disponible en http://arxiv.org/pdf/cond-mat/0306509.
[24] Hirshleifer, J., “From weakest-link to best-shot: the voluntary provision of public goods”, en *Public Choice* ,41, 1983, pp. 371-86.
[25] Varian, H., “System Reliability and Free Riding”, trabajo presentado en Workshop on Economics and Information Security, University of California, Berkeley, 16 y 17 de mayo de 2002; disponible en http://www.sims.berkeley.edu/resources/affiliates/workshops/econsecurity/econws/49.pdf .
[26] Rijmen, V., "LinuxSecurity.com Speaks With AES Winner", reportaje en *LinuxSecurity.com*, disponible en http://www.linuxsecurity.com/feature_stories/interview-aes-3.html.
[27] Anderson, R.J., “Why Information Security is Hard – An Economic Perspective”, en *Proceedings of the Seventeenth Computer Security Applications Conference*, Los Alamitos, CA, IEEE Computer Society, 2001, pp.358-365.
[28] Anderson, R.J. y S.J. Beduidenhout, “On the Reliability of Electronic Payment Systems”, en *IEEE Transactions on Software Engineering*, Vol. 22 No. 5, mayo 1996, pp. 294-301.