

# Depuración: Aplicando el Método Científico

TONI CASTILLO

toni.castillo@iberprensa.com

**T**oda pieza de software es susceptible de contener defectos en su código. Estos defectos pueden provocar infecciones, es decir, que ciertos estados del programa sean cualquier cosa menos la esperada por su programador. Y, por extensión, dichos estados infectados pueden conducir a un fallo. La acción sistemática de búsqueda de defectos en el software y su posterior corrección se conoce con el nombre de depuración. En este reportaje introduciremos al lector en esta apasionante área.

Que el software falla es una realidad incuestionable. Los errores de programación son variados, aunque por regla general, se hace siempre hincapié en los accesos erróneos a memoria. Es decir, cuando en un trozo de código se intenta, por ejemplo, direccionar una posición de un vector que, simplemente, no existe. Es uno de los errores clásicos ampliamente descritos en miles de libros dedicados al *Arte de la Depuración*.

Sorprendentemente, estos errores son atribuibles no solo a aquellos programadores noveles que empiezan a crear sus primeras rutinas, también a aquellos senior que llevan años en el desarrollo. Según el Profesor A. Zeller, la mejor depuración de todas es la no depuración en absoluto, es decir, cuando nuestro código es tan perfecto que sabemos que nunca nos hará falta depurarlo. Sin embargo, estamos muy lejos de un modelo ideal de código. Incluso el mejor de los programadores inserta defectos en el mismo. Algunos de estos defectos no son detectables jamás en la vida útil del software. Es perfectamente viable que un programa posea errores de diseño, o pequeños defectos que, simplemente, no sean perceptibles por el usuario final. Los programadores aplican tests sistemáticos sobre sus códigos, intentando comprobar su correcto funcionamiento en la mayor parte de las situaciones comunes (y, ge-

neralmente, en otras que no lo son tanto). Pero como bien apuntó Dijkstra, los tests jamás podrán demostrar la ausencia de defectos. De hecho, nada puede garantizarnos de una manera formal e inequívoca que un programa no posea defectos. Por eso, el trabajo de un programador debe extenderse un poco más allá de los tests y del desarrollo en sí mismo.

Conocer los principios de la depuración es esencial, como desarrollador o simplemente como Administrador de Sistemas, a fin y efecto de entender porque los programas fallan.

## INTRODUCCIÓN

Para introducirse en el mundo de la depuración, lo mejor es comprender qué es realmente y cómo podemos enfocarla. A lo largo de mi experiencia como Administrador de Sistemas, ofreciendo soporte técnico en entornos científicos con GNU/Linux, he descubierto que el conocimiento profundo de ciertas herramientas de depuración disponibles y ampliamente documentadas junto con una buena base teórica de sistemas operativos, me permite descubrir y solucionar una gran cantidad de problemas que, de otro modo, serían simplemente imposibles de abordar.

Este reportaje, pues, pretende ser una pequeña introducción a la depuración de software. Dividiremos el mismo en cuatro partes. La primera expondrá terminología ampliamente aceptada por autores como A. Zeller, Dijkstra, y otros. A lo largo de la segunda parte, discutiremos largo y tendido sobre el método científico aplicado a la depuración. La tercera parte mostrará brevemente las herramientas disponibles en GNU/Linux para poner todo lo anterior a prueba, intentando darle un último enfoque práctico a este reportaje después de tanta teoría formal en la cuarta y última parte del mismo.

## DEFECTOS Y MÁS DEFECTOS...

Es ampliamente aceptado que, cuando un programa falla, tiene un BUG. Pero como muchos autores recalcan, la definición de BUG está muy lejos de ser acertada en el caso particular del software. Cuando una aplicación falla, es porque tiene un defecto. Los defectos son insertados por el programador del código, porque el código no se crea a sí mismo: es pensado e implementado por uno o varios programadores. No tiene sentido, pues, hablar de BUG porque un BUG es un defecto que proviene del mundo exterior. En nuestro caso, un hipotético defecto no presente en el código. ¡Menuda barbaridad! Si un programa presenta un fallo, es incuestionable que posee un defecto, y dicho defecto debe buscarse en su código. El código de un programa no es más que la abstracción formal que lo define.

El estado de un programa es un momento dado en su vida útil durante su ejecución, definido por una serie de variables y valores asociados a las mismas. Puede definirse formalmente un hipotético estado  $S_i$  como un grafo  $S_i(V, A)$ . Los nodos podrían ser las variables y sus valores, y las aristas las relaciones entre las mismas (ver Figura 1). Los estados de un programa pueden verse modificados por dichos defectos, o no. En el primer caso, hablaremos de un estado infectado, y se puede afirmar que el defecto ha infectado dicho estado. A veces, la infección puede propagarse entre diferentes estados. Esto es así porque ciertas rutinas dentro del código pueden recibir parámetros de estados infectados, generando pues una propagación. Aunque parezca un poco sorprendente, ciertos defectos puede que nunca lleguen a infectar estados. Del mismo modo, un estado infectado o un compendio de los mismos puede que nunca den lugar a un fallo. De hecho, es perfectamen-

**Cualquier depuración de código aplicando el método científico requiere de las fuentes originales del programa**

te demostrable que muchos programas parecen funcionar con absoluta normalidad a pesar de poseer defectos. Cuando se produce realmente un *fallo*, es porque el usuario final lo percibe. No todos los usuarios perciben todos los errores, simplemente porque o no usan ciertas funcionalidades del mismo, o porque simplemente no se dan cuenta. Algunas veces estos fallos son *reproducibles*, otras veces es más difícil lograr un estado infectado que conduzca a dicho fallo de manera sistemática. Por eso es imprescindible el estudio del código utilizando herramientas de depuración.

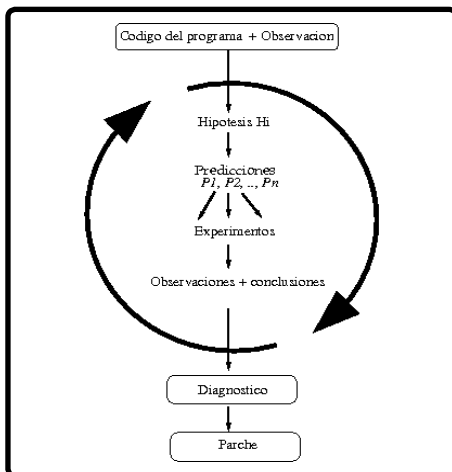


Figura 1. Un hipotético estado  $S_1$  con unas variables y las interrelaciones entre las mismas.

Un buen ejemplo de programa con al menos un defecto  $d_1$ , es el software de instalación de controladores para las tarjetas gráficas ATI. Concretamente, la versión 8.821. A pesar de poseer dicho defecto  $d_1$ , bajo ciertas condiciones es perfectamente viable no apreciar ningún fallo en absoluto. E incluso cuando dicho fallo es realmente detectado, el software completa su tarea correctamente (en este caso, la instalación de los controladores ATI). Realicé un exhaustivo estudio sobre dicho defecto accesible online. (Ver [Cuadro de referencias](#)).

Por tanto, no podemos afirmar que un programa que no presenta fallos no posea defectos. Este es, precisamente, el punto de vista de *Dijkstra*, *N. Matloff*, *A. Zeller*, *J. Salzman*, entre otros. Podemos llegar a otra afirmación por *deducción*: no podemos afirmar que un programa que posee un defecto  $d_1$ , no posea otros defectos adicionales  $d_2, d_3, \dots, d_n$ . Por ejemplo, el estudio del defecto  $d_1$  del controlador ATI no puede demostrarnos que no haya otros defectos presentes en el código. Así, el trabajo de un depurador

de software es, por generalización, aquel que comprende el estudio y análisis, utilizando las herramientas y técnicas que se introducirán a continuación, del *fallo* detectado a fin y efecto de localizar los *estados infectados*, que deberán conducirle hasta los *defectos* del código que convergen hacia los mismos. Parece una tarea bastante ingrata, y cuando uno desconoce las herramientas y las diferentes aproximaciones formales disponibles en el mundo de GNU/Linux, ciertamente lo puede llegar a ser. Pero realmente no es tan duro (aunque esto, claro, dependerá de la extensión del software a depurar).

## EL MÉTODO CIENTÍFICO

Un depurador debe ser formal. Es imprescindible poder aplicar un método ordenado al aparente caos que puede generar un programa defectuoso. Ciertamente, algunas veces, no es necesario sobrecargar la sesión de depuración con métodos formales. Por ejemplo, algunas veces es preferible utilizar la *intuición* para llegar a encontrar un defecto. Esto, generalmente, funciona con código propio o bien como aplicación de la *inducción*. Ya sabemos, por la introducción de este reportaje, que la mayor parte de los errores (¡incluso hoy en día!), son aquellos que tienen como base un mal direccionamiento de memoria. Es todo un clásico intentar acceder al  $i$ -ésimo elemento de un vector cuando este solo posee  $i-1$  elementos. Bien, en casos así es muy probable que un simple vistazo del código nos revele dicho *defecto*. ¡Y sin necesidad de leerse este artículo! Por desgracia, no siempre es así. Ciertos autores consideran que, si somos incapaces de localizar un defecto en el código utilizando la depuración en sucio (mezclando técnicas inductivas y depuradores) en menos de diez minutos, es probable que ya seamos incapaces de encontrarlo. Es en estos casos cuando debemos utilizar el *Método Científico*.

## ■ APLICÁNDOLO A LA DEPURACIÓN

Vamos a tomar prestado dicho método para poderlo aplicar a la búsqueda de defectos en el software. En realidad, el método científico es simple y brutalmente coherente. No pretendemos explicar aquí teoría ampliamente documentada en Internet, echad un vistazo al [Cuadro de referencias](#) si estáis interesados en ampliar información sobre el mismo. Lo que interesa ahora es adaptarlo a nuestro caso particular. Queremos un modelo formal de análisis que nos permita localizar el *defecto* y finalmente aplicar una solución. Es

evidente que en el mundo de la programación, la solución pasará por alterar algunas líneas de código, lo que se conoce como un *parche*.

Estos son los diferentes pasos que deberemos aplicar sistemáticamente:

### ▶ PASO 1

Observar el *fallo*  $F$ .

### ▶ PASO 2

Crear una hipótesis,  $H_i$ . Esta hipótesis debe intentar explicar por qué ocurre  $F$ , y debe ser coherente con la observación de  $F$ .

### ▶ PASO 3

Utilizando  $H_i$  definiremos predicciones:  $P_1, P_2, \dots, P_n$ .

### ▶ PASO 4

Probaremos  $H_i$  a base de *experimentos* y futuras *observaciones*. Así, si los experimentos demuestran  $P_1, P_2, \dots, P_n$ , podremos mejorar  $H_i$  (volveremos al punto 3). Si tan sólo tenemos una predicción  $P_i$  que no se cumple, es decir si  $\exists P_i \in \{P_1, P_2, \dots, P_n\}$  que no prueba  $H_i$ , deberemos crear una nueva hipótesis (iremos al punto 2).

### ▶ PASO 5

Repetiremos los puntos 3 y 4 hasta que  $H_i$  ya no pueda perfeccionarse más.

¡Realmente parece muy formal! Algunas veces deberemos definir muchas hipótesis. Los experimentos que deberemos llevar a cabo para probar todas las  $P_n$  asociadas a la hipótesis  $H_i$  serán, por regla general, realizados utilizando un programa de depuración. La tercera parte de este artículo trata este tema ampliamente. Por ejemplo, podría tratarse simplemente de leer el valor de una variable y comprobar que es el que esperamos, o bien alterarlo. Esto es perfectamente viable sin necesidad de recompilar el código porque los depuradores cumplen este cometido a la perfección.

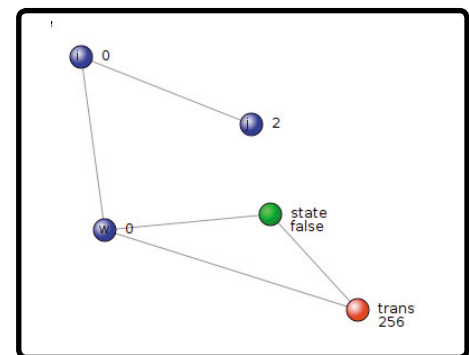


Figura 2. El método científico aplicado a la depuración.

## HERRAMIENTAS PARA LA BÚSQUDA DE DEFECTOS

Pero no solo de terminología y método científico vive el depurador.

Evidentemente, existen herramientas en el mundo del código libre que nos permitirán aplicar todos estos nuevos conocimientos de un modo práctico y relativamente sencillo. La herramienta central de toda buena aplicación del método científico en el mundo de la depuración es el depurador. Linux posee uno de los mejores: **gdb**. Su uso puede resultar un poco crudo para los neófitos en la materia, porque se ejecuta íntegramente en modo texto. Así, existen *frontends* como **ddd**, que para muchos puede llegar a resultar bastante más cómodo de utilizar. A efectos de nuestra discusión, vamos a considerar **gdb**. El uso del depurador será, por regla general, imprescindible para poder ejecutar ciertos experimentos que comprueben nuestras predicciones. Pero no será la única herramienta a utilizar. La depuración implica la ejecución de *scripts*, o incluso de software que permita trazar el comportamiento del programa en tiempo de ejecución, como **strace** o **ltrace**. Algunas veces, la combinación de todas estas herramientas, siempre bajo el paraguas del método científico, nos ayudará a localizar el defecto rápidamente.

Un profundo conocimiento del código a depurar será imprescindible. Por supuesto, disponer de las fuentes del código es una parte muy importante en toda sesión de depuración. Esto es así en un 99% de los casos, porque estamos tratando la depuración de código en entornos GNU/Linux. Todos sabemos que disponemos de todas las fuentes de la mayor parte de los programas que ejecutamos en nuestro ordenador, porque es código libre. Sin embargo, algunas veces deberemos lidiar con la lacra de código privativo para el que, lógicamente, no tendremos las fuentes. Sería muy atrevido afirmar que, aun y así, podremos localizar el defecto. Esto no siempre será factible. En los casos en los que realmente seamos capaces de aislar el mismo, tal vez nos resulte muy complicado aplicar una solución, es decir, un *patch*, que lo arregle definitivamente. Esto es así porque, si no disponemos del código fuente del programa, deberemos analizar su código desensamblado. El análisis del defecto en el programa de instalación de los controladores gráficos de ATI (ver **Cuadro de referencias**), utiliza una herramienta integrada en la *glibc*, **MALLOC\_CHECK\_**, para determinar que el software tiene, realmente, un de-

## Herramientas de depuración y otros ▶

El *debugger* por antonomasia en entornos GNU/Linux: **gdb**.

Traza de las llamadas al sistema ejecutadas: **strace**.

Traza de las llamadas a librerías: **ltrace**.

Desensamblador de código: **objdump**.

Información sobre binarios (segmentos, etc): **readelf**.

Frontend para el depurador **gdb**: **ddd**.

Comprobador de corrupción del *HEAP* de la *glibc*: **MALLOC\_CHECK\_**.

Detección de errores de memoria dinámica: **Electric Fence**, **Valgrind**.

Navegador de fuentes en C: **cscope**.

Analizador de flujo para fuentes en C: **cflow**.

Analizador de flujo para fuentes en Fortran: **fflow**.

fecto. A partir de dicho conocimiento, únicamente utilizamos **gdb** para demostrar que el defecto es coherente con lo que el código realiza internamente. Sin embargo, es un caso ligeramente diferente porque ya sabemos *a priori* cuál es el defecto. Esto, por regla general, no será así. Es decir, el defecto no estará claro, porque será el objeto de nuestra búsqueda.

Tratar con código *assembler* es complejo. Requiere de un profundo conocimiento del sistema donde se está ejecutando el código defectuoso. Por ejemplo, en lenguaje de programación Fortran, que es un lenguaje de alto nivel, podemos hacer esta asignación a una variable:

```
y1=agetab(j1,i1)
```

pero en un equipo con procesador Intel Core2Duo, ejecutando una distribución Debian de 64 bits, compilado con Intel Fortran 11.0, el código *assembler* equivalente será algo más o menos similar a esto:

```
movsd
0x7be848(%rsi,%r8,1),%xmm8
```

Claramente, entre la primera línea de código y la segunda hay enormes diferencias. Por definición, cualquier depuración de código aplicando el método científico y las diferentes herramientas disponibles, requiere de las fuentes originales del programa. En caso de no cumplirse este requisito, la búsqueda del defecto puede ser imposible de llevar a cabo.

Cuando no disponemos de las fuentes originales del programa, no podemos alterar su código; por extensión no podemos recompilarlo. En estos casos, si realmente sabemos dónde está el defecto y cómo solucionarlo, deberemos alterar el código a bajo nivel directamente sobre el binario. Este es el principio subyacente en los ya tristemente famosos *cracks* para software privativo, usualmente asociados a sistemas operativos Microsoft.

Por fortuna, poseeremos las fuentes del código la mayor de las veces. Así, podremos centrarnos en aplicar todo lo descrito hasta ahora sin preocuparnos por lidiar con código a muy bajo nivel.

Podemos consultar el **Cuadro** adjunto sobre Herramientas de depuración para ver una lista de las herramientas que son de uso generalizado entre la comunidad de depuradores de software. Todas ellas son código libre, y generalmente su instalación en sistemas Debian o derivados será tan simple como usar el gestor de paquetes apt. ¡Lo difícil será aprender a utilizarlas correctamente!



## MANOS A LA OBRA: UN CASO REAL.

A continuación, vamos a intentar aplicar el método científico descrito previamente en este tutorial, utilizando la terminología introducida: *defecto*, *estado*, *infección*, *propagación*, *fallo*, *parche*, para localizar un defecto y aplicar un *parche* a un programa que presenta un *fallo*. Vamos a utilizar **gdb** para dicha finalidad.

Cuando un código presenta un fallo, se puede afirmar categóricamente que la primera hipótesis,  $H_1$ , siempre será rechazada. Esto es así porque la primera hipótesis en un programa que posee un fallo será: "el programa funciona." Y eso no es cierto.

### ■ OBSERVACIÓN

Recientemente, una compañera me hizo llegar un código escrito íntegramente en

C que presentaba un *fallo*. El fallo en cuestión era una **violación de segmento** al cabo de ejecutar  $n$  iteraciones de un hipotético bucle, aunque dicho fallo no siempre sucedía en la misma  $i$ -ésima iteración. Esto es lo que podía observarse cuando se ejecutaba el código:

```
(...)ssssssssssssssssssss
Segmentation fault
```

Bien, sin lugar a dudas la observación del fallo, punto 1 de nuestro método científico, está muy clara. Así, tal y como hemos demostrado previamente, podemos rechazar  $H_1$ , y procederemos a definir  $H_2$ . Algunos autores definen  $H_1$  como una fase previa de *preparación*, asociándola con la *observación*. Creemos que ese enfoque no mantiene la coherencia con la descripción del método científico.

Así pues, podemos volver a observar el fallo y obtener más información si ejecutamos el programa anterior dentro del depurador gdb. Para ello, es totalmente necesario que compilemos el programa añadiendo la directiva `-g` de `gcc`, que añade **la tabla de símbolos** a la imagen de nuestro programa:

```
~ gcc -g -lm MD.c -o MD.e
~ gdb ./MD.e
```

Ahora podemos observar el fallo desde dentro del depurador. Esto nos mostrará mucha más información que el simple *"segmentation fault"*. Es siempre un requisito fundamental para saber por dónde podemos comenzar el análisis, es decir, cómo podemos crear la hipótesis  $H_2$ . Una vez dentro del depurador, podremos introducirle comandos mediante el teclado. Para ejecutar dichos comandos, pulsaremos <Enter>. Así, como queremos ejecutar el código defectuoso dentro del depurador, escribiremos <r> y pulsaremos <Enter>:

```
(gdb) r
ssssssssssssssssssss(...)
Program received signal SIGSEGV,
Segmentation fault.
0x0000000000402019 in Calc_vacf
() at MD.c:633
633     vacf[j-i]=0;
```

El programa se ejecutará hasta llegar al *fallo*. Esta vez, sin embargo, el depurador nos ofrece algo más de información. Definiremos nuestra segunda hipótesis a partir de esta información adicional

## Principales comandos gdb utilizados en el reportaje ▶

Para facilitar la correcta comprensión de este artículo, recogemos aquí los comandos ejecutados y una pequeña descripción de los mismos. El lector puede dirigirse en todo momento a la página de manual del depurador gdb (*man gdb*) o a la abundante documentación sobre el mismo disponible en Internet.

Comando	Descripción
<code>print var</code>	Muestra el valor de una variable del programa.
<code>list</code>	Lista el código fuente dentro de la ventana del depurador. La línea actualmente ejecutada queda centrada en la misma.
<code>b</code>	Inserta un punto de interrupción en el flujo del programa. Cuando se llega a dicho punto de interrupción, gdb detiene su ejecución para que podamos analizar el estado del programa.
<code>set variable var = value</code>	Altera el valor de una variable en tiempo de ejecución. Ideal para ejecutar experimentos que demuestren nuestras hipótesis.
<code>c</code>	Cotinúa (continue) con la ejecución del programa después de alcanzar un punto de interrupción.

### ■ HIPÓTESIS $H_2$ .

Sabemos que el fallo se presenta en forma de violación de segmento. Por tanto, estamos ante un caso de fallo relacionado con accesos ilegales a memoria. La línea de código donde aparece el fallo nos muestra que se está intentando escribir el valor 0 sobre un vector en la posición  $j-i$ . Empezaremos, pues, por definir  $H_2$  tal que:

- ▶ **Hipótesis:** El acceso al vector `vacf[]` es erróneo, por tanto  $i, j$  o las dos tienen valores erróneos.
- ▶ **Predicción:** Se generará un **segmentation fault** porque  $j-i$  pertenece a una posición que está fuera del ámbito del vector `vacf[]`. El valor de  $j-i$  deberá rebasar el valor máximo de elementos de `vacf[]`, es decir,  $j-i > \text{len}(\text{vacf}[\ ])$
- ▶ **Experimentación:** Utilizando gdb, vamos a proceder a comprobarlo. Justo cuando el programa muestra el fallo, dentro del depurador, podemos fácilmente comprobar los valores de  $i, j$ :

```
(gdb) print i
$1 = 0
(gdb) print j
$2 = 703832156
```

El valor de  $i$  podría ser correcto. El valor de  $j$  parece ser excesivamente elevado. El experimento no serviría de nada sin las fuentes del código, porque desconoceríamos el valor máximo de elementos del vector `vacf[]`.

Observando el código C, vemos que `vacf[]` es un vector acotado con:

```
# define tau_max 1000
...
double vacf[tau_max];
```

Sabemos, así, que si operamos  $j-i$ , obtendremos un elemento erróneo para el vector `vacf[]`. Esto es así porque la única manera de acceder al vector correctamente sería si  $0 \leq (j-i) < \text{tau}_{\text{max}}$ .

- ▶ **Observación:** Ejecutando el comando anterior, observamos que  $j$  está *infectada*. No podemos afirmar nada sobre  $i$  (que valga 0 no quiere decir que NO esté infectada, pero el valor de  $j$  sabemos que SI lo está, por tanto debemos focalizar nuestra atención en  $j$ ).
- ▶ **Conclusión:**  $H_2$  está confirmada.

### ■ HIPÓTESIS $H_3$ .

Debemos encontrar el origen de la infección de  $j$ . O bien esta infección se produce porque  $j$  es un parámetro pasado a la función en la que nuestro código falla, `Calc_vacf()`, cuyo valor ya está infectado (*propagación* de la infección), o bien se infecta dentro de la misma. Por tanto, de nuevo debemos conocer el código fuente para saber dónde está definida  $j$ . Resulta que es una **variable local** definida dentro de `Calc_vacf()` tal y como vemos en el **Listado 1**.

El comando `list` de gdb nos permite leer el código fuente del programa que está siendo depurado sin salir de la sesión. ¡Es realmente útil!

Así, vemos que  $j$  es una variable definida en la línea 629 de la función `Calc_vacf()`. Vamos a seguir razonando del mismo modo, aunque a simple vista ya se ve claramente el motivo de la infección de  $j$ :



## Listado 1. Variable local

```
(gdb) list
628     printf("estic en calcul de vacf      ");
629     int i,j,k;
630     double sum,norma;
631
632     for(i=0;i<tau_max;i++)
633     vacf[j-i]=0;
```

- ▶ **Hipótesis:** Inicializar el vector `vacf[]` en la función `Calc_vacf[]` genera una violación de segmento porque el valor de `j` está infectado. Si el valor de `j` no estuviera infectado, el vector se inicializaría correctamente.
- ▶ **Predicción:** Si alteramos el valor de `j` tal que `j = 0`, el programa no generará un **segmentation fault** porque la dirección de memoria a la que se intentará escribir el valor 0 formará parte como mínimo del segmento de datos de nuestro programa. Por supuesto, el valor computado será igualmente erróneo para indexar el vector, puesto que se cumplirá que:  $j - i < 0 \text{ para } i \geq 1$ .
- ▶ **Experimentación:** Usando `gdb`, vamos a alterar el valor de `j` antes de entrar en el código de inicialización del vector, esto es, justo antes de la línea de código `for(i=0;i<tau_max;i++)`:

```
(gdb) b MD.c:632
Breakpoint 1 at 0x402000: file
MD.c, line 632.
```

Con el comando anterior `<b MD.c:632>`, le hemos dicho al depurador que nos pare la ejecución del código (se entiende la próxima vez que lo ejecutemos), justo al llegar a la línea 632 del archivo de código C `MD.c`.

Ejecutamos de nuevo la aplicación y `gdb` detiene la ejecución justo antes de la línea 632:

```
(gdb) r
The program being debugged has
been started already.
Start it from the beginning? (y
or n) y
ssssssssssssssssssssssssssssssss
sssssss(...)
Breakpoint 1, Calc_vacf () at
MD.c:632
632     for(i=0;i<tau_max;i++)

Alteramos el valor de j:

(gdb) set variable j=0
```

Continuamos ejecutando el programa. A pesar de lo que está escrito en el código, ahora el valor de `j` ha sido alterado en tiempo de ejecución. Pasemos a la observación de nuestra hipótesis:

```
(gdb) c
Continuing.
ssssssssssssssssssssssssssssss
ss(...)
Program exited normally.
```

- ▶ **Observación:** Tal y como hemos predicho, el programa finaliza correctamente.
- ▶ **Conclusión:**  $H_3$  se confirma.

Así, hemos determinado que el valor de `j` está infectado en la función `Calc_vacf[]`, y que dicha infección se produce directamente en el propio cuerpo de la misma. Esto es así porque `j` no está inicializada, adquiriendo el valor tan elevado que hemos podido observar durante la predicción de la hipótesis  $H_2$ .

El desarrollo del *parche* es trivial, basta modificar la línea 634 del archivo `MD.c` para que se inicialice el vector de la siguiente manera:

```
vacf[i] = 0
```

## CONCLUSIONES FINALES

El uso de `gdb` en este caso nos ha ayudado a ejecutar los diferentes experimentos para probar nuestras predicciones. Algunas veces, será necesario recurrir a herramientas adicionales. El ejemplo pro-

puesto era trivial, pero esto no va a ser siempre así. A pesar de ello, aplicando el método formal anteriormente descrito nos ayudará a ordenar los experimentos y a seguir un camino lógico hasta acotar al máximo los puntos donde el defecto (o defectos) puedan estar localizados. De este modo, podremos generalizar.

Aquellos programadores que insertan mensajes en sus aplicaciones para determinar cuando su código llega a cierto punto y que valor tiene cierta variable, como método de depuración, erran. Esto siempre es una mala práctica. Entre otros motivos, la depuración de defectos como el descrito ampliamente en la web que aparece en el **Cuadro de referencias**, *Fine-Tuning the previous Fortran analysis*, demuestra claramente que uno no puede fiarse de la ejecución de código como `printf()`, `write()`, o derivativos.

Entender completamente el uso de `gdb` se escapa del ámbito de este artículo introductorio. El lector puede encontrar abundante documentación sobre el uso de esta herramienta en Internet o en su propia página de manual (*man gdb*). Una gran e imprescindible obra sobre la depuración usando `gdb` y `ddd` que todo programador debería tener en casa es *The Art Of Debugging*, de N. Matloff y P. J. Salzman, de la cual también indicamos dónde conseguir más abajo.

Como nota final, desearía animar a todos aquellos Administradores de Sistemas o informáticos que dan soporte en entornos GNU/Linux a que utilicen estos métodos y herramientas en su trabajo diario. Muchos de los errores que afectan a los sistemas son debidos al software. Cuando el software tiene un *fallo*, siempre es debido a uno o más defectos en el código. Si no existe el *parche* que lo soluciona, entonces solo podemos hacer dos cosas: o le decimos al usuario que todavía no hay solución -¿para qué nos pagan entonces?-, o bien nos sentamos frente al ordenador, ponemos un poco de música y abrimos `gdb`. ¡Y eso si es ser informático! ■

## Referencias ▶

- ▶ **T. Castillo, Dealing with ATI Drivers and heap corruption:**  
<http://disbauxes.upc.es/?p=2964>
- ▶ **A. Zeller, Why Programs Fail (Chapter 6, Scientific Debugging):**  
<http://www.whypogramsfail.com/>
- ▶ **T. Castillo, Fine-tuning the previous Fortran analysis:**  
<http://disbauxes.upc.es/?p=2921>
- ▶ **N. Matloff & P.J. Salzman, The Art Of Debugging:**  
<http://my.safaribooksonline.com/book/software-engineering-and-development/ide/9781593271749/copyright/vi>