(19) OFICINA ESPAÑOLA DE PATENTES Y MARCAS

ESPAÑA

(11) Número de publicación: **2 999 381**

(51) Int. Cl.:

G06F 11/36 (2006.01)

(12) TRADUCCIÓN DE PATENTE EUROPEA T3

(86) Fecha de presentación y número de la solicitud internacional: **27.11.2017 PCT/EP2017/080481**

(87) Fecha y número de publicación internacional: **31.05.2019 WO19101341**

(96) Fecha de presentación y número de la solicitud europea: **27.11.2017 E 17807816 (8)**

(97) Fecha y número de publicación de la concesión europea: **23.10.2024 EP 3718012**

(54) Título: **Autodepuración**

(45) Fecha de publicación y mención en BOPI de la traducción de la patente:
25.02.2025

(73) Titular/es:

NAGRAVISION SÀRL (100.00%)
Route de Genève 22-24
1033 Cheseaux-sur-Lausanne, CH

(72) Inventor/es:

DORE, LAURENT;
ORAKZAI, ASFANDYAR;
WYSEUR, BRECHT y
XU, YIHUI

(74) Agente/Representante:

VALLEJO LÓPEZ, Juan Pedro

DESCRIPCIÓN

Autodepuración

Campo

La presente divulgación se refiere a la seguridad de software, en particular a la protección de software tal como aplicaciones o bibliotecas contra los ataques que usan técnicas de depuración.

Antecedentes

La depuración es el proceso mediante el que se pueden identificar errores en el código. Una herramienta para esto es el depurador, un tipo de utilidad que muchos sistemas operativos permiten emparejar con el código a depurar. Cuando se produce una excepción u otro error, esto se informa al depurador que, a continuación, puede inspeccionar el código e identificar el origen de este problema.

La capacidad de emparejar un depurador con código ha sido utilizada por partes malintencionadas para comprometer la seguridad de ese código. En particular, ya que un depurador puede identificar la operación del código, puede ser una fuente de vulnerabilidad.

Se han desarrollado técnicas para intentar proteger el código contra tales ataques. Estas técnicas incluyen intentos de permitir que el código identifique cuando un depurador activo ha sido ilícitamente acoplado al código. Otro enfoque es diseñar el código para que, cuando se ejecute, inicie por sí mismo un depurador (este depurador puede denominarse "autodepurador"). La mayoría de los sistemas operativos únicamente permiten emparejar un único depurador con un proceso dado, lo que significa que el autodepurador ocupa el espacio que un depurador malicioso podría desear usar.

El documento "Tightly-Coupled Self-Debugging Software Protection" de Abrath et al. divulga una técnica de autodepuración estrechamente acoplada, donde se migran fragmentos de código completo de la aplicación al depurador, dificultando al atacante la ingeniería inversa del programa y su deconstrucción en el programa original desprotegido para acoplar un depurador o recopilar trazas.

Sumario

Aunque la invención se define en las reivindicaciones independientes, se exponen aspectos adicionales de la invención en las reivindicaciones dependientes, los dibujos y la siguiente descripción.

Breve descripción de los dibujos

La Figura 1 es una ilustración esquemática de las principales características de un proceso de código de la técnica anterior, y del proceso de código y el proceso de depuración acoplados de una primera realización;

La Figura 2 es un diagrama de flujo que muestra etapas de tiempo de ejecución de acuerdo con la primera realización;

La Figura 3 muestra aspectos primarios de la generación de binarios de acuerdo con la primera realización;

La Figura 4 es una ilustración esquemática del proceso de código acoplado y del proceso de depurador de una segunda realización;

La Figura 5 es un diagrama de flujo que muestra las etapas de tiempo de ejecución de acuerdo con la segunda realización; y

La Figura 6 muestra una infraestructura de hardware para implementar una realización preferida.

Descripción detallada de los dibujos

En resumen, se proporcionan métodos para asegurar la operación de código. De acuerdo con la divulgación, un método puede comprender el lanzamiento de un proceso de código y la inicialización de un proceso de depurador adjunto al proceso de código. Durante la ejecución del proceso de código, se pueden realizar operaciones de críticamente relevantes para la funcionalidad del proceso de código dentro del proceso de depurador. Como resultado, el proceso de depurador no puede ser sustituido o subvertido sin afectar a la funcionalidad del proceso de código. Por tanto, el proceso del código puede protegerse de la inspección mediante técnicas de depuración modificadas o malintencionadas.

En este contexto, "críticamente" puede entenderse en el sentido de que la salida producida por esas operaciones

llevadas a cabo en el proceso de depurador sirve como entrada para la parte restante del proceso de código, y que esa entrada es necesaria para permitir que el proceso de código genere su salida correcta dada la otra entrada de ese proceso de código.

En algunos aspectos de la divulgación se proporciona un método para asegurar software. El método puede comprender el lanzamiento de un proceso de software y adjuntar un proceso de depurador al proceso de software. A continuación, puede ejecutarse el proceso de código de tal manera que el proceso de depurador sea invocado al menos una vez. Tras la invocación, una o más funciones pueden realizarse dentro del proceso de depurador, estas funciones tienen una salida dependiente de los datos asociados con el proceso de software. Dado que la salida puede variar dependiendo de los datos asociados con el proceso de software (es decir, no está predeterminada), la funcionalidad global únicamente se consigue cuando tanto el proceso de software como el proceso de depurador operan correctamente. Esto no deja espacio para interferir con el proceso de depurador para analizar el código.

El proceso de software de este aspecto puede considerarse un proceso "a depurar", ya que ocupa el lugar del proceso que se está depurando por el depurador. El proceso de depurador puede inicializarse cuando se lanza el proceso de software o en un momento posterior. Por ejemplo, un proceso de depurador puede inicializarse cuando se carga cierta funcionalidad (por ejemplo, una biblioteca) en el proceso de software). En algunos ejemplos, el proceso de software se bifurca para inicializar el proceso de depurador. En otros ejemplos, el proceso de depuración puede inicializarse en primer lugar, y, a continuación, bifurcarse para generar el proceso de software.

En algunos ejemplos, la salida comprende una salida de datos para su uso por el proceso de software. Por lo tanto, el resultado de las funciones dentro del proceso de depurador puede influir directamente en la operación posterior del proceso de software, de esta manera, los dos procesos están estrechamente acoplados de un modo difícil de romper. La salida de la función en el proceso de depurador comprende una entrada de datos para el proceso de software, siendo dicha entrada de datos crítica para la ejecución del proceso de software.

El proceso de software puede generar una estructura de datos que comprende parámetros requeridos para la realización de una o más funciones dentro del proceso de depurador antes de la invocación del proceso de depurador. Por lo tanto, el proceso de software puede hacer preparativos para permitir un fácil acceso a los datos desde su memoria al proceso de depurador. Esto es particularmente aplicable cuando se ha empleado la reescritura del código fuente o del código de bits para generar el programa asociado con el proceso de depurador. La reescritura a este nivel puede permitir la implementación de técnicas para facilitar la generación de una estructura de datos apropiada para las funciones realizadas por el proceso de depurador. La estructura de datos puede ser una estructura de estado.

En algunos ejemplos, el proceso de software actúa para depurar el proceso de depurador. Como tal, se proporciona una disposición de depuración "circular", en la que ambos procesos actúan para depurar al otro. Esto puede impedir que un depurador externo adjunte procesos a cualquier proceso.

El método puede lanzar un proceso adicional para depurar el proceso de depurador. Se pueden proporcionar además procesos adicionales para continuar la cascada, depurándose cada proceso por otro. De nuevo, puede proporcionarse una disposición circular. Por ejemplo, el proceso de software puede depurar el proceso adicional para que ningún proceso esté disponible para un depurador externo.

En algunos ejemplos, la salida de una función dada puede indicar múltiples puntos de retorno dentro del proceso de software para continuar la ejecución. Como tal, el punto de retorno de al menos una función es variable (en lugar de fijo). El flujo de control es, por lo tanto, variable de acuerdo con el comportamiento del proceso de depurador y no puede inferirse ni recrearse fácilmente.

En algunos ejemplos, el proceso de depurador proporciona capacidades de soporte de memoria para permitir que la una o más funciones recuperen datos desde la memoria dentro del espacio de direcciones del proceso de software. Como tal, las funciones relevantes para el programa pueden tener la capacidad de procesar datos como si se llevaran a cabo dentro del proceso de software.

El proceso de depurador puede invocarse cuando se alcanza un punto de interrupción dentro del proceso de código. El proceso de depurador puede des-adjuntarse del proceso de software cuando finaliza el proceso de software. El proceso de software puede finalizar porque se ha completado, o, de otra manera, (tal como cuando se aborta). Como alternativa, el proceso de depuración puede des-adjuntarse del proceso de software cuando finaliza la funcionalidad dentro del proceso de software, en lugar de esperar a que finalice el proceso en su conjunto.

En algunos ejemplos, el proceso de software implementa un ejecutable, tal como una aplicación. En otros, el proceso de código implementa una biblioteca.

En otro aspecto de la divulgación, se proporciona un método para de generación de código protegido. Se identifica una o más funciones en el código a compilar para un primer proceso para migrar a un segundo proceso, en donde el uno del primer y segundo procesos es un depurador para el otro del primer y segundo procesos. A continuación, se lleva a cabo la migración y se modifica el primer proceso para permitir transferir el estado entre el primer y segundo

procesos. A continuación, el primer y segundo procesos generan código binario. El código binario en tiempo de ejecución puede hacer que un proceso de depuración se adjunte a un proceso de software, ejecutándose la función o funciones identificadas dentro del proceso de depurador

El código a compilar puede ser código fuente o código de bits. En general, puede ser código a un nivel superior al binario.

Se puede inyectar un inicializador en uno del primer y segundo proceso para invocar la ejecución del otro del primer y segundo procesos. Este inicializador puede invocar la ejecución del primer o segundo proceso que actúa como depurador del otro del primer y segundo procesos. De esta manera, el depurador se lanza automáticamente.

Uno o más inicializadores pueden inyectarse en el primer o segundo programa para registrar funciones presentes en el otro del primer y segundo procesos. Como tal, cada proceso puede tener en cuenta y tomar medidas en reconocimiento de las funciones llevadas a cabo en cualquier otra parte. Por ejemplo, los inicializadores pueden facilitar la generación de una estructura de datos para la uno o más funciones realizadas en el otro proceso.

En algunos ejemplos, cada uno del primer y segundo procesos es un depurador para el otro del primer y segundo procesos. Como tal, se proporciona una disposición de depuración "circular", en la que ambos procesos actúan para depurar al otro. Esto evita que un depurador ilícito se adjunte al programa de depurador.

El método puede proporcionar un tercer proceso que es un depurador para uno del primer y segundo procesos. Por ejemplo, el segundo proceso puede depurar el primero, y el tercero puede depurar el segundo. Se pueden proporcionar procesos adicionales para continuar la cascada, depurándose cada proceso por otro. De nuevo, puede proporcionarse una disposición circular. Por ejemplo, donde el segundo proceso depura el primero y el tercer proceso depura el segundo, el primer proceso puede depurar el tercero.

En otro aspecto de la divulgación, se proporciona un método para generar código protegido. Se pueden identificar fragmentos de código dentro de código objeto que han de migrarse a un depurador. A continuación, se puede generar código binario, donde el código binario en tiempo de ejecución hace que un proceso de depuración se adjunte a un proceso de software, ejecutándose los fragmentos de código identificados dentro del proceso de depurador. El proceso de software y el proceso de depurador pueden bifurcarse desde un único proceso. Por ejemplo, el proceso de software puede inicializar el proceso de depurador.

La etapa de generación puede comprender incorporar código predefinido correspondiente a la funcionalidad de depurador genérico dentro del código binario. La etapa de generación puede ser una etapa de enlace que incorpora algunos aspectos predefinidos del depurador, tales aspectos pueden denominarse un "minidepurador". Como tal, el depurador global incluye algunos aspectos genéricos, así como algunos aspectos específicos del código fuente por medio de la inclusión de los fragmentos de código identificados.

El método puede comprender extraer del código fuente una o más anotaciones que identifican fragmentos de código a migrar a un depurador. A continuación, el código fuente se compila para generar el código objeto. A continuación, puede generarse código binario a partir del código objeto, integrándose los fragmentos de código identificados con un depurador en el código binario. De esta manera, la generación de código binario puede ser una etapa de vinculación que incluye un elemento de reescritura para mover fragmentos identificados a otra ubicación. A continuación, se usa el código binario, se genera un depurador que comprende aspectos del código fuente original, que pueden ser pertinentes para la funcionalidad del código fuente.

En algunos ejemplos, el código binario comprende un primer archivo de código binario correspondiente al código fuente, pero que excluye los fragmentos de código identificados, y un segundo archivo de código binario correspondiente al depurador. Como alternativa, un único archivo de código binario puede incorporar tanto el código fuente como el depurador.

Aspectos adicionales de la divulgación se refieren a productos de programa ejecutables por ordenador que comprenden instrucciones ejecutables por ordenador para llevar a cabo los métodos de los aspectos descritos anteriormente. Los aspectos de la divulgación también se refieren a dispositivos configurados para llevar a cabo los métodos de los aspectos descritos anteriormente.

Algunas realizaciones específicas se describen ahora a modo de ilustración con referencia a los dibujos adjuntos en los que los números de referencia similares se refieren a características similares.

A través de, técnicas de reescritura binaria, la presente divulgación puede migrar trozos enteros de funcionalidad desde el software original a un autodepurador. Esto ofrece varias ventajas. En primer lugar, el comportamiento de entrada-salida del autodepurador ya no está más predeterminado: cada vez que interviene el autodepurador, ejecuta funcionalidad diferente que no está predeterminada, pero que, en su lugar, puede variar tanto como puede variar la funcionalidad de los programas protegidos. Esto hace que la protección sea mucho más resistente frente a los análisis automatizados, desofuscación y deconstrucción. En segundo lugar, incluso si el atacante puede averiguar el flujo de

control y el flujo de datos equivalente del programa original, es mucho más difícil para un atacante deshacer la protección y reconstruir el programa original. En combinación, estos dos puntos fuertes hacen que sea mucho más difícil para un atacante des-adjuntar el autodepurador mientras mantiene un programa en funcionamiento para ser rastreado o depurado en vivo.

DISEÑO GLOBAL DEL AUTODEPURADOR

La Figura 1 ilustra los conceptos básicos de un esquema de autodepuración de acuerdo con la presente divulgación. Esta realización tiene como objetivo Linux (y derivados tales como Android), los principios también pueden aplicarse a otros entornos tales como Windows y OS X.

A la izquierda de la Figura 1, se representa una aplicación desprotegida original, incluye un pequeño fragmento de gráfico de flujo de control. El código ensamblador mostrado es (pseudo) código ARMv7. Esta aplicación desprotegida se convierte en una aplicación protegida que consiste en dos partes: un elemento a depurar que corresponde en su mayor parte a la aplicación original, como se muestra en el medio de la figura, y un depurador como se muestra a la derecha. Aparte de algunos componentes nuevos inyectados en el elemento a depurar y el depurador, la principal diferencia con la aplicación original es que el fragmento de gráfico de flujo de control se ha migrado de la aplicación al depurador. Esta realización particular soporta todos los fragmentos de código de entrada única salida múltiple que no contienen flujo de control inter-procedural tal como llamadas de función.

La migración de tales fragmentos es más que una simple copia: las referencias de memoria tales como la instrucción de LDR deben transformarse debido a que, en la aplicación protegida, el código migrado que se ejecuta en el espacio de direcciones del depurador puede preferentemente acceder a datos que aún residen en el espacio de direcciones del elemento a depurar. Todos los componentes y transformaciones relevantes se analizarán con más detalle en secciones posteriores.

Los fragmentos migrados son preferentemente críticos para la operación de la aplicación. Es decir, la salida producida por esas operaciones llevadas migradas en el proceso de depurador sirve como entrada para la parte restante del proceso de código, y que esa entrada es necesaria para permitir que el proceso de código genere su salida correcta dada la otra entrada de ese proceso de código. Este requisito es fácil de pasar por alto en la práctica. Por ejemplo, un programador típico podría considerar ejecutar la inicialización de variables del proceso de código en el contexto de depurador. Sin embargo, en general, no es suficiente ejecutar la inicialización de variables desde el proceso de código en el proceso de depurador, debido a que, en la práctica, en los procesos ocurre con bastante frecuencia que la inicialización de variables (por ejemplo, de las variables locales tras la entrada a una función) se realiza como resultado de las buenas prácticas de programación y para cumplir los requisitos de definición del lenguaje de programación fuente, sin requerirse realmente necesarios para el correcto funcionamiento del proceso y para generar salidas correctas. Esto puede deberse a que las variables simplemente no se usan en las rutas ejecutadas en el proceso de código, o a que los valores iniciales se sobrescriben antes de que puedan afectar a la ejecución o salida del proceso de código.

En tiempo de ejecución, la operación de esta aplicación protegida es la siguiente. En primer lugar, el depurador se lanza en la etapa s21, como si fuera la aplicación original. A continuación, un inicializador recién inyectado bifurca un nuevo proceso para el depurador, en el que el inicializador del depurador se adjunta inmediatamente al proceso a depurar. Por lo tanto, el proceso de depurador se lanza y se adjunta al proceso a depurar en la etapa s22.

Cuando más adelante, durante la ejecución del programa, se alcanza el punto de entrada del fragmento de código migrado, un posible flujo de control en la aplicación sigue las flechas en la Figura 1. En la aplicación/elemento a depurar, la instrucción que induce la excepción se ejecuta y provoca una excepción en la etapa s23 (etiquetada como 1 en la Figura 1). El depurador es notificado de esta excepción y la maneja en su bucle de depurador en la etapa s24 (etiquetada como 2 en la Figura 1). Entre otros, el código en este bucle es responsable de capturar el estado de proceso desde el elemento a depurar, buscando el correspondiente fragmento de código migrado, y transfiriendo el control al punto de entrada de ese fragmento en la etapa s25 (etiquetada como 3 en la Figura 1). Como se ha indicado, en que los accesos a la memoria de fragmentos no pueden realizarse tal cual. Por lo que son reemplazados por invocaciones 4 de funciones de soporte de memoria 5 que acceden a la memoria en el espacio de direcciones del elemento a depurar en la etapa s26. Cuando se alcanza un punto de salida 6 en el fragmento de código migrado, el control se transfiere al punto correspondiente del bucle de depurador 7 en la etapa s27, que actualiza el estado del elemento a depurar con los datos calculados en el depurador en la etapa s28, y 8 se transfiere de nuevo el control al elemento a depurar en la etapa s29. Para fragmentos de código con múltiples salidas, tal como el ejemplo de la figura, el control puede transferirse de nuevo a múltiples puntos de continuación en el elemento a depurar. En este sentido, el depurador de la presente divulgación se comporta de manera más compleja que los autodepuradores existentes, que implementan un mapeo uno a uno entre transferencias de flujo de control hacia adelante y hacia atrás entre el elemento a depurar y el depurador.

Eventualmente, cuando se sale de la aplicación, los finalizadores integrados realizarán las operaciones para des-adjuntar necesarias.

Es importante señalar que este esquema no únicamente puede desplegarse para proteger ejecutables (es decir, binarios con una función principal y un punto de entrada), sino también para proteger las bibliotecas compartidas. Igual que los ejecutables, las bibliotecas pueden contener inicializadores y finalizadores que se ejecutan cuando son cargadas o descargadas por el cargador del SO. En ese tiempo, todas las bifurcaciones necesarias, también se pueden adjuntar y des-adjuntar.

Aunque la siguiente descripción se refiere principalmente a proteger aplicaciones, implícitamente la enseñanza se aplica por igual a aplicaciones y bibliotecas. Un aspecto particularmente relevante para las bibliotecas es la necesidad de inicializar y finalizar apropiadamente el depurador. Esto es necesario porque no es raro que las bibliotecas se carguen y descarguen múltiples veces dentro de una única ejecución de un programa. Por ejemplo, la carga y descarga repetitivas son frecuentes en los módulos de extensión de reproductores multimedia y navegadores. Además, mientras que los programas principales consisten únicamente un solo hilo cuando se lanzan ellos mismos, pueden consistir en múltiples hilos cuando se cargan y descargan las bibliotecas.

SOPORTE DE HERRAMIENTAS

La Figura 3 muestra un posible flujo de herramientas conceptual.

Anotaciones de código fuente

Para determinar los fragmentos de código a migrar al depurador, existe un número de opciones. Una, representada en la figura -y también lo que usamos en nuestra implementación- es anotar el código fuente en la etapa s31 con pragmas, comentarios o cualquier otra forma de anotaciones que marcan los comienzos y extremos de las regiones de código a migrar al proceso de depurador. Un simple *grep* es suficiente para extraer las anotaciones y sus números de línea y almacenar esa información en un fichero de anotaciones en la etapa s32.

Otras opciones alternativas serían elaborar una lista de los procedimientos o archivos de código fuente que han de protegerse, o recopilar trazas o perfiles para seleccionar fragmentos interesantes de forma semiautomática.

A este respecto, es importante tener en cuenta que los fragmentos a migrar al depurador no deben ser necesariamente fragmentos muy críticos. Para lograr un fuerte acople entre el depurador y el elemento a depurar, es suficiente plantear excepciones con relativa frecuencia, pero esto no necesita estar en las rutas de código más críticas. A continuación, se detallan consideraciones adicionales para la selección de fragmentos. Dado que cada excepción introducirá una cantidad significativa de sobrecarga (cambio de contexto, muchas llamadas ptrace, ...) es importante minimizar su número sin comprometer el nivel de protección.

Compiladores y herramientas convencionales

Para que se despliegue el enfoque de autodepuración divulgado, en la etapa s33 se puede usar cualquier compilador "convencional". Esta técnica no impone ninguna restricción al código generado por el compilador. En las evaluaciones experimentales, se ha usado tanto GCC como LLVM, en los que no era necesario adaptar o perfeccionar la generación de código.

Sin embargo, un requisito, es que el compilador y las utilidades binarias (el ensamblador y el enlazador) proporcionen al reescritor en tiempo de enlace información suficientemente precisa sobre los símbolos y la reubicación. Esto es necesario para permitir análisis y transformaciones de código en tiempo de enlace fiables y conservadores para implementar la totalidad del esquema de autodepuración, incluyendo la migración y transformación de los fragmentos de código seleccionados. Proporcionar información suficientemente precisa está, ciertamente, al alcance de herramientas de uso común. Los compiladores propietarios de ARM, por ejemplo, lo han hecho durante mucho tiempo de manera predeterminada, y para las utilidades binarias (binutils) de GNU, GCC y LLVM, son suficiente parches muy sencillos para evitar que esas herramientas realicen una relajación de símbolos y una simplificación de la reubicación demasiado agresivas, y para obligarlas a insertar símbolos de mapeo para marcar datos en el código. Estos requisitos ya se han documentado anteriormente y se ha demostrado que son suficientes para realizar una reescritura de tiempo de enlace conservadora fiable de código tan complejo y poco convencional como ambas versiones CISC (x86) y RISC (ARMv7) del núcleo Linux y las bibliotecas C, que están llenas de código ensamblador escrito manualmente.

Una parte grande genérica del depurador -el "minidepurador"- puede precompilarse con el compilador convencional y, a continuación, vincularse simplemente a la aplicación a proteger. Otras partes, tal como los prólogos y epílogos del bucle de depuración para cada uno de los fragmentos migrados, se generan por la reescritura de tiempo de enlace, ya que se personalizan para sus fragmentos específicos.

Para permitir que el reescritor de tiempo de enlace identifique los fragmentos que se anotaron en el código fuente, es suficiente con pasarle la información de número de línea extraída de los archivos de código fuente, y dejar que los compiladores generen ficheros objeto con información de depuración. A continuación, esa información de depuración mapea todas las direcciones del código binario a números de línea de código fuente, que el reescritor puede enlazar a los números de línea de las anotaciones.

Binarios, bibliotecas y procesos

El reescritor de tiempo de enlace tiene dos opciones para generar una aplicación protegida en la etapa s35. Una primera opción es generar dos binarios, uno para la aplicación/elemento a depurar, y uno para el depurador. Desde una perspectiva de seguridad, esto puede ser preferible, debido a que la semántica de la aplicación y su implementación se distribuyen a continuación a través de múltiples binarios, lo que probablemente hace aún más difícil para un atacante deshacer la protección, es decir, para parchear el elemento a depurar en la aplicación original. Esta opción introduce sobrecarga en tiempo de ejecución adicional, sin embargo, ya que el lanzamiento del depurador también requiere a continuación cargar el segundo binario.

La opción alternativa -usada en los ejemplos adicionales a continuación- es incrustar todo el código del elemento a depurar y todo el código de depurador en un binario. En ese caso, una simple bifurcación será suficiente para lanzar el depurador. Si es así o no, y en qué medida, esto facilita los ataques a la protección proporcionada por la autodepuración es una cuestión de investigación abierta.

IMPLEMENTACIÓN

Inicialización y finalización

Se puede añadir una rutina de inicialización adicional a un binario protegido. Esta rutina se invoca tan pronto como se ha cargado el binario (porque tiene asignada una prioridad alta), después de lo cual se ejecutan todas las demás rutinas enumeradas en la sección .init del binario.

Esta rutina de inicialización invoca fork(), creando por lo tanto dos procesos llamados padre e hijo. Una vez finalizada la rutina de inicialización, el proceso padre continuará su ejecución, invocando típicamente la siguiente rutina de inicialización.

Existen dos opciones para asignar las funciones de depurador y elemento a depurar: Después de la bifurcación, o bien el proceso hijo se adjunta al proceso padre, o viceversa. En el primer caso, el hijo se convierte en el depurador y el padre se convierte en el elemento a depurar, en este último caso las funciones se invierten obviamente.

Se prefiere la primera opción. El proceso padre (es decir, el elemento a depurar) sigue siendo el proceso de aplicación principal, y mantiene el mismo ID de proceso (PID). Esto facilita la ejecución continua o el uso de todas las aplicaciones externas y canales de comunicación inter-proceso que se basan en el PID original, por ejemplo, debido a que se configuraron antes de la carga y bifurcación de una biblioteca protegida.

Este sistema conlleva sus propios problemas, sin embargo. Como ya se ha mencionado, las bibliotecas compartidas pueden cargarse y descargarse (usando dlopen() y dlclose()) en cualquier momento durante la ejecución de un programa. Por lo tanto, existe el problema potencial de que una biblioteca compartida protegida puede descargarse y cargarse de nuevo mientras que el depurador originalmente cargado y bifurcado aún no ha terminado su inicialización. Esto puede dar como resultado la existencia simultánea de dos procesos de depurador, ambos intentando (y uno fallando) adjuntarse al elemento a depurar. Para evitar esta situación, bloqueamos la ejecución del hilo que llamó a dlopen(). Por lo que, hasta ese momento, ese hilo no puede invocar dlclose() usando el manejador que obtuvo con dlopen() y tampoco puede pasar el manejador a otro hilo. Un bucle infinito en la rutina de inicialización del elemento a depurar evita que el subproceso de carga salga de la rutina de inicialización antes de que el depurador le permita continuar.

La rutina de inicialización también instala un finalizador en el depurador. Este finalizador no hace mucho. A la salida del programa (o cuando se descarga la biblioteca compartida) simplemente informa al minidepurador de este hecho elevando una señal SIGUSR1, lo que hace que el minidepurador se des-adjunte de todos los hilos del elemento a depurar y cierre el proceso de depuración.

Soporte de múltiples hilos

Adjuntar el depurador no es trivial, en particular en el caso de bibliotecas compartidas protegidas. Cuando se carga una biblioteca, la aplicación puede consistir en varios hilos. Únicamente uno de ellos ejecutará la rutina de inicialización del elemento a depurar durante su llamada a dlopen. Esto es bueno, ya que únicamente se ejecutará una bifurcación, pero también tiene el inconveniente de que únicamente un hilo entrará en el bucle infinito mencionado en la sección anterior. Los otros hilos en el proceso del elemento a depurar continuarán ejecutándose, y podrían crear nuevos hilos en cualquier punto durante la ejecución de la rutina de inicialización del elemento a depurar o de la rutina de inicialización del depurador.

Para garantizar una protección apropiada, el depurador debe adjuntarse a cada hilo en el proceso del elemento a depurar como parte de su inicialización. Para garantizarse de que el depurador no se pierda ningún hilo creado en el elemento a depurar mientras tanto, usamos el directorio /proc/[pid]/task, que contiene una entrada para cada hilo de

un proceso. El proceso de depuración se adjunta a todos los hilos iterando a través de las entradas de este directorio, y manteniendo la iteración hasta que no se encuentren nuevas entradas. Tras adjuntarlo al hilo, que se produce por medio de una solicitud PTRACE_ATTACH, el hilo también se detiene (y el depurador es notificado de este evento por el SO), lo que significa que a partir de entonces ya no podrá generar nuevos hilos. Por lo que, para cualquier programa que genere un número finito de hilos, el procedimiento iterativo para adjuntar a todos los hilos está garantizado para terminar. Una vez que todos los hilos se han unido al bucle infinito en el depurador, se termina y se permite que sus hilos detenidos continúen.

Cuando se crean hilos adicionales posteriormente durante la ejecución del programa, el depurador se adjunta automáticamente a ellos por el SO, y obtiene una señal de tal manera que se puede realizar todo el mantenimiento de registros necesario.

Flujo de control

Transformar el flujo de control dentro y fuera de los fragmentos de código migrados consiste en varias partes. Analizamos la elevación de excepciones para notificar al depurador, la transferencia del ID que informa al depurador de qué fragmento se va a ejecutar, y los prólogos y epílogos personalizados que se añaden a cada fragmento de código.

Elevación de excepciones

La notificación real del depurador puede producirse a través de cualquier instrucción que provoque que se eleve una excepción. En nuestra implementación, usamos un punto de interrupción de software (es decir, una instrucción BKPT en ARMv7) para simplificar. Por supuesto, se pueden usar otras excepciones menos llamativas, tales como las causadas por instrucciones ilegales o indefinidas. Cuando tales instrucciones son accesibles a través del flujo de control directo (bifurcación directa o ruta de caída), por supuesto, pueden detectarse fácilmente de forma estática. Pero cuando se usan transferencias de flujo de control indirecto para saltar a datos en las secciones de código, y los bits de datos corresponden a una instrucción ilegal o indefinida, la detección estática puede hacerse mucho más difícil. Análogamente, pueden usarse instrucciones legales que lanzan excepciones únicamente cuando sus operandos son "inválidos" para ocultar el objetivo de las instrucciones. Tales instrucciones incluyen la división por cero, accesos a memoria no válidos (es decir, un fallo de segmentación), o la des-referenciación de un puntero no válido (que provoca un error de bus).

Transferir ID

Solicitamos al hilo en el elemento a depurar que eleva una excepción en el hilo solicitante, ya que prácticamente está pidiendo al depurador que ejecute algún fragmento de código.

El depurador, después de que el SO le notifique la solicitud, necesita averiguar qué fragmento ejecutar. Para activarlo, el depurador puede pasar un ID del fragmento en un número de maneras. Una opción es simplemente usar la dirección de la instrucción que induce la excepción como ID. Otra opción es pasar el ID colocándolo en un registro fijo justo antes de lanzar la excepción, o en una ubicación de memoria fija. En nuestra implementación, usamos esta última opción. Como múltiples hilos en el elemento a depurar pueden solicitar un fragmento diferente al mismo tiempo, la ubicación de memoria no puede ser una ubicación global. En su lugar, necesita ser hilo-local. Como cada hilo tiene su propia pila, optamos por pasar el ID del fragmento a través de la parte superior de la pila del hilo solicitante.

Dependiendo del tipo de instrucción usada para elevar la excepción, también se pueden prever otros métodos. Por ejemplo, también podría usarse el operando divisor de una instrucción de división (por cero) para pasar el ID.

Prólogos y epílogos

El bucle de depurador del minidepurador es responsable de extraer el estado del programa del elemento a depurar antes de que se ejecute un fragmento, y de transferirlo de vuelta después de su ejecución. Para hacer esto se usa la funcionalidad convencional de ptrace.

Por cada fragmento de código migrado, el bucle de depuración contiene también un prólogo y un epílogo personalizados a ejecutar antes y después del fragmento de código resp. El prólogo carga los valores necesarios de la estructura (struct) en registros, el epílogo vuelve a escribir los valores necesarios en la estructura. El prólogo está personalizado en el sentido de que únicamente carga los registros que realmente se usan en el fragmento (los llamados registros residentes). El epílogo únicamente almacena los valores que no son residentes (es decir, que se consumirán en la depuración) y que se sobrescribieron en el fragmento de código.

Accesos a memoria

Para cada operación de carga o almacenamiento en un fragmento de código migrado, se necesita un acceso a la memoria del elemento a depurar. Existen múltiples opciones para implementar tales accesos.

La primera es simplemente usar la funcionalidad ptrace: el depurador puede realizar solicitudes PTRACE_PEEKDATA y PTRACE_POKEDATA para leer y escribir en el espacio de direcciones del elemento a depurar. En este caso, por palabra' a leer o escribir, se necesita una llamada de sistema ptrace, que da como resultado una sobrecarga significativa. Algunas versiones recientes de Linux soportan accesos más amplios, pero aún no están disponibles en todas partes, tal como en Android.

La segunda opción es abrir el fichero /proc/[pid]/mem del elemento a depurar en el depurador, y, a continuación, simplemente leer o escribir en este fichero. Esto es más fácil de implementar, y se pueden leer o escribir datos más amplios con una única llamada de sistema, por lo que, en ocasiones, este método es más rápido. Escribir en /proc/[pid]/mem de otro proceso no está soportado en cada versión de los núcleos Linux/Android, sin embargo, por lo que, en nuestro prototipo, las solicitudes de escritura se siguen implementando con la primera opción.

Una tercera opción se basa en la segunda: si el reescritor binario puede determinar a qué páginas de memoria se accederá en un fragmento de código migrado, el bucle de depuración puede copiar esas páginas en el espacio de direcciones del depurador usando la opción 2. A continuación, el fragmento en el depurador simplemente ejecuta operaciones de carga y almacenamiento regulares para acceder a las páginas copiadas, y después de que el fragmento se haya ejecutado, las páginas actualizadas se copian en el elemento a depurar. Esta opción puede ser más rápida si, por ejemplo, el fragmento de código contiene un bucle para acceder a un búfer en la pila. Los experimentos que realizamos para comparar la tercera opción con las dos opciones anteriores revelaron que esta técnica podría valer la pena para tan sólo 8 accesos a memoria. No implementamos un soporte fiable para ello en nuestro prototipo, sin embargo: Un análisis del tiempo de enlace conservador para determinar a qué páginas accederá un fragmento de código sigue siendo trabajo futuro en este momento.

Una cuarta opción posible es adaptar el elemento a depurar, por ejemplo, proporcionando una biblioteca de gestión de memoria de almacenamiento dinámico tipo heap personalizada (malloc, free, ...) de tal manera que toda la memoria asignada (o al menos la de almacenamiento dinámico tipo heap) se asigne como memoria compartida entre los procesos del elemento a depurar y el depurador. A continuación, los fragmentos de código del depurador pueden acceder directamente a los datos. Por supuesto, aún es necesario reescribir los fragmentos para incluir una traducción de direcciones entre los dos espacios de direcciones, pero es probable que la sobrecarga de esta opción sea mucho menor que los de las otras opciones. La implementación y evaluación de esta opción sigue siendo, por el momento, un trabajo futuro.

En cuanto a la seguridad, es probable que las distintas opciones tengan también un impacto diferente, en el sentido de que impactarán en la dificultad de un atacante para aplicar ingeniería inversa a la semántica original del programa y deconstruir la versión autodepuración en un equivalente del programa original.

Combinar la autodepuración con otras protecciones

Para proporcionar una sólida protección de software contra los ataques MATE, pueden emplearse técnicas de protección adicionales. Por ejemplo, además de la autodepuración, puede emplearse la ofuscación para evitar el análisis estático, junto con técnicas antimanipulación para evitar todo tipo de ataques.

Por ejemplo, el reescritor de binarios que implementa el enfoque de autodepuración también puede aplicar un número de otras protecciones, tal como una o más de:

- Ofuscaciones de flujo de control: las conocidas ofuscaciones de predicados opacos, aplanamiento de flujo de control y funciones de ramificación;
- Aleatorización de distribución de código: durante la distribución de código, el código de todas las funciones se mezcla y se aleatoriza la distribución;
- Movilidad de código: técnica en la que se eliminan fragmentos de código del binario estático y únicamente se descargan, como el llamado código móvil, en la aplicación en tiempo de ejecución;
- Guardas de código: implementaciones en línea y fuera de línea de técnicas en las que se calculan funciones de troceo sobre el código en el espacio de direcciones del proceso para comprobar que el código no ha sido modificado.
- Integridad de flujo de control: técnica ligera en la que se comprueban las direcciones de retorno para impedir que se invoquen funciones internas desde código externo.
- Virtualización de conjunto de instrucciones: técnica con la que el código nativo se traduce a código de bytes bytecode que es interpretado por una máquina virtual integrada en lugar de ejecutarse de forma nativa.

Combinar la técnica de autodepuración con todas esas protecciones no plantea ningún problema en la práctica. En el reescritor de tiempo de enlace, no es difícil determinar un buen orden para realizar todas las transformaciones para todas las protecciones, y evitar que se apliquen múltiples técnicas sobre los mismos fragmentos de código cuando, en realidad, no están compuestas esas técnicas. Por ejemplo, el código móvil se reubica en ubicaciones aleatorizadas. Manejar correctamente todas las protecciones requiere cierto mantenimiento de registros, pero nada complejo.

En cuanto al comportamiento en tiempo de ejecución, las técnicas también están compuestas. Múltiples técnicas requieren inicializadores y finalizadores, pero, en el proceso de depuración no deseamos ejecutar los inicializadores de las otras protecciones, ya que ese proceso de depuración debería ser únicamente un depurador, y no otro cliente de movilidad de código o cualquier otra técnica. Para impedir que se ejecuten los otros inicializadores, a los inicializadores del autodepurador se les da la prioridad más alta. Se ejecutan en primer lugar cuando se carga un binario o biblioteca, y la rutina de inicialización del depurador implementa de hecho tanto el inicializador real, así como el bucle de depuración. Por lo tanto, la rutina nunca finaliza (es decir, siempre y cuando no se invoque al finalizador), y por lo tanto nunca se transfiere el control a los otros inicializadores que puedan estar presentes en el binario.

EVALUACIÓN

Plataforma de evaluación

Una implementación del autodepurador tiene como objetivo plataformas ARMv7. En concreto, esta implementación se orientó y evaluó exhaustivamente la implementación en Linux 3.15 y Android 4.3+4.4 (sin permisos de superusuario (unrooted)). Además, se ha confirmado además que las técnicas siguen funcionando en las últimas versiones de Linux (4.7) y Android (7.0), y, de hecho, es así.

El hardware de pruebas consistía en varias placas de desarrollo. Para Linux, se usó una Panda Board con un procesador Texas Instruments OMAP4 de un solo núcleo, una placa Arndale con un procesador Samsung Exynos de doble núcleo, y una placa Boundary Devices Nitrogen6X/SABRE Lite con un procesador Freescale i.MX6q de un solo núcleo. Esta última placa también se usó para las versiones de Android.

En la cadena de herramientas, se usó GCC 4.8, LLVM 3.4 y GNU binutils 2.23. El código se compiló con las siguientes banderas: - Os -march=armv7-a -marm -mfloat-abi=softfp -mfpu=neon -msoft- float.

Casos de uso

Se ha demostrado que el esquema de autodepuración funciona en múltiples casos de uso. Por ejemplo, en una situación de gestión de derechos digitales, se encontraron las siguientes consideraciones prácticas.

Este caso de uso consiste en dos módulos de extensión, escritos en C y C++, para la arquitectura de desarrollo de Android y la arquitectura de desarrollo DRM de Android. Estas bibliotecas son necesarias para acceder a las películas cifradas y para descifrarlas. Se usa una aplicación de vídeo programada en Java como GUI para acceder a los vídeos. Esta aplicación se comunica con las arquitecturas de desarrollo mediaserver y DRM de Android, informando a las arquitecturas de desarrollo del proveedor del que necesita módulos de extensión. Bajo demanda, estas arquitecturas de desarrollo cargan a continuación los módulos de extensión. En concreto, estos servidores son los procesos mediaserver y drmserver que se ejecutan en Android.

Durante los experimentos y el desarrollo, se observaron varias características que hacen de este caso de uso una prueba de resistencia perfecta para esta técnica. En primer lugar, el mediaserver es de múltiples hilos, y crea y mata nuevos hilos todo el tiempo. En segundo lugar, las bibliotecas de módulos de extensión se cargan y descargan con frecuencia. En ocasiones, la descarga se inicia incluso antes de que finalice la inicialización de la biblioteca. En tercer lugar, tan pronto como el proceso se bloquee, se lanza una nueva instancia. En ocasiones, esto permite que el reproductor de vídeo Java siga funcionando sin interrupciones, en ocasiones no. Esto hace que la depuración de la implementación de nuestra técnica sea aún más compleja de lo que ya es para aplicaciones sencillas. En cuarto lugar, el mediaserver y el drmserver participan en frecuentes comunicaciones entre procesos. No obstante, el éxito de la aplicación se basó en los principios descritos anteriormente.

Las técnicas de la presente divulgación pueden aplicarse en muchas otras situaciones de uso. Por ejemplo, en la banca móvil en cualquier otra situación en la que la seguridad sea deseable.

Segunda realización

En los ejemplos presentados anteriormente con respecto a las Figuras 1 a 3, el archivo binario se reescribe para transferir elementos al proceso de depurador. En una segunda realización, la técnica puede desplegarse a nivel de código fuente o a otro nivel superior al binario (por ejemplo, a nivel de código de bits) durante el proceso de construcción del software. Esto se describirá a continuación haciendo referencia a las Figuras 4 y 5. Este proceso transfiere el estado de programa entre el depurador y el elemento a depurar de una manera diferente al ejemplo presentado anteriormente.

En este enfoque, la aplicación puede segmentarse en dos o más partes en el nivel de fuente, o de código de bits, usando una herramienta de reescritura. Esta segmentación puede llevarse a cabo a nivel de función, tal manera que el proceso de reescritura transfiere ciertas funciones del programa inicial a otro programa y el programa inicial se modifica para poder transferir estado al otro programa. El programa inicial puede tomar la función del elemento a depurar durante la ejecución posterior, mientras que el otro programa puede tomar la función de depurador. Como

alternativa, la separación de funciones de depurador y elemento a depurar puede invertirse.

Además, se inyecta código adicional en el programa que se va a lanzar en primer lugar (que puede ser cualquier programa dentro de la aplicación segmentada) para que actúe como un inicializador que permite a la aplicación bifurcarse o lanzar otro proceso para permitir que el depurador se adjunte al elemento a depurar. Además, se pueden incorporar inicializadores adicionales en el programa que se va a lanzar en primer lugar para registrar aquellas funciones que va a llevar a cabo el otro programa.

La operación en tiempo de ejecución de la aplicación segmentada puede entenderse con referencia a las Figuras 4 y 5. En particular, La Figura 4 es una ilustración esquemática del proceso de código acoplado y del proceso de depurador de una segunda realización, mientras que la Figura 5 es un diagrama de flujo que muestra las etapas de tiempo de ejecución de acuerdo con la segunda realización.

En el ejemplo mostrado en las Figuras 4 y 5, el programa lanzado en primer lugar vez lleva a cabo el proceso del elemento a depurar. Como tal, en tiempo de ejecución, el depurador se lanza en la etapa s51, como si fuera la aplicación original. Los inicializadores inyectados durante el proceso de construcción registran a continuación las funciones que tiene como objetivo el depurador en la etapa s52 y bifurcan un nuevo proceso para el depurador. Dado que los inicializadores se pueden generar por encima del nivel binario en el conocimiento de la división de funciones entre los procesos de depurador y elemento a depurar, de manera beneficiosa el registro de funciones puede dirigirse adecuadamente.

Una vez que el nuevo proceso de depurador se bifurca, el inicializador del depurador se adjunta inmediatamente al proceso a depurar. Por lo tanto, el proceso de depurador se lanza y se adjunta al proceso a depurar en la etapa s53.

Cuando, más adelante, durante la ejecución del programa, se alcanza el punto de entrada de una función que se ha colocado en el depurador, un posible flujo de control en la aplicación sigue las flechas en la Figura 4. En primer lugar, el código de prólogo (arquitectura preferentemente independiente) puede serializar los parámetros de la función en una estructura de estado en la etapa s54 (etiquetada 1 en la Figura 4). A continuación, se ejecuta una instrucción de inducción de excepción, que provoca una excepción en la etapa s55 (es decir, un punto de interrupción, etiquetada como "bkpt" en la Figura 4). Esto activa el depurador en la etapa s56 (etiquetado como 2 en la Figura 4), que identifica la ubicación del elemento a depurar de la excepción/punto de interrupción.

La rutina/bucle del minidepurador en el proceso de depurador puede inferir desde la ubicación del elemento a depurar de la excepción/punto de ruptura tanto el código objetivo (es decir, la función "f" a llevar a cabo dentro del proceso de depurador) como la forma de recuperar los parámetros serializados a la estructura de estado desde la memoria del elemento a depurar. A continuación, extrae los parámetros del elemento a depurar en la etapa s57 (etiquetada como 3 en la Figura 4). Además, dado que la estructura de estado puede hacer referencia a elementos de cualquier otra parte en la memoria del elemento a depurar, estos elementos de estado extendidos pueden ser identificados a nivel de fuente de tal manera que también pueden recuperarse de la memoria del elemento a depurar para estar disponibles para el depurador.

Como la ejecución a través del minidepurador de la función correcta y la recuperación de los parámetros de estado está condicionada a la ubicación del punto de interrupción/excepción, esto proporciona seguridad adicional, ya que un punto de activación incorrecto no causaría la ejecución apropiada.

Con los parámetros recuperados satisfactoriamente, el minidepurador puede invocar la función "f" en la etapa s58 (etiquetada 4 en la Figura 4). Esta función "f" fue migrada desde la aplicación del elemento a depurar durante la reescritura del código fuente como se ha descrito anteriormente. Se realiza la función "f" y devuelve los resultados al minidepurador en la etapa s59 (etiquetada 5 en la Figura 4). A continuación, los parámetros pueden actualizarse en la estructura de estado, incluyendo los parámetros de estado ampliados en la etapa s60 (etiquetada 6 en la Figura 4). El mini-depurador devuelve a continuación el control al elemento a depurar en la etapa s61 (etiquetada 7 en la Figura 4), y el elemento a depurar escribe los parámetros de estado y de estado ampliado de nuevo en la memoria del elemento a depurar a través de un proceso de des-serialización en la etapa s62 (etiquetada 8 en la Figura 4). En la etapa s63, el depurador puede ejecutar código de epílogo para restaurar parámetros y/o variables de sistema. Aunque tal código puede depender de la arquitectura actual, pueden formularse basándose en aspectos intrínsecos del compilador portátil.

En el ejemplo descrito anteriormente con referencia a las Figuras 4 y 5, un programa inicial está asociado con uno del depurador o elemento a depurar y un segundo programa está asociado con el otro. Sin embargo, el experto reconocerá que, en algunas arquitecturas, puede no ser necesario asignar procesos de esta manera. Por ejemplo, un único programa puede lanzar dos procesos, uno de los cuales depura al otro.

Además, se reconoce que, en algunos entornos, tales como Linux o Android, por ejemplo, puede adoptarse un procedimiento de "bifurcación". Por ejemplo, Fork() es una llamada de sistema que duplicará el proceso desde el que se llama. Copiará la memoria de proceso, y, a continuación, ambos procesos continuarán en paralelo. Como tal, en este enfoque, un único programa se ejecuta dos veces. El programa puede implementarse de tal manera que tenga

un comportamiento diferente en caso de que sea el padre o en caso de que sea el hijo. Por ejemplo, el proceso hijo puede actuar como depurador del proceso padre, o viceversa.

En consecuencia, dentro del ámbito de la presente divulgación se proporciona la posibilidad de que distintos programas estén asociados a procesos de depurador y elemento a depurar, un único programa que genera procesos de depurador y elemento a depurar independientes, y un único programa que se bifurca en múltiples procesos (idénticos) donde uno de ellos asume la función de depurador y otro el de elemento a depurar. En otra alternativa, se puede crear un nuevo proceso (idéntico) desde cero sin usar el procedimiento de bifurcación, de nuevo con un proceso que lleva a cabo funciones de depuración para el otro. También puede adoptarse cualquier división alternativa entre programas y procesos, según sea apropiado.

CARACTERÍSTICAS ADICIONALES

Las características que figuran a continuación como "Depuración circular", "Depuración anidada", "Re-entrada", "Re-adjuntar", ""Detección de des-adjuntar" y "comprobación mutua" pueden proporcionarse en combinación con cualquier ejemplo o realización descritos anteriormente y en cualquier combinación entre sí.

Depuración circular

Adjuntar un depurador al elemento a depurar no impide que el depurador sea depurado por un tercero. Para evitarlo, el elemento a depurar puede depurar a su vez al depurador, creando un bucle de depuración, donde cada proceso evita que otros depuradores se adjunten a su depurador. Esto se debe a que cada proceso tendrá un depurador adjunto, y, por lo tanto, no hay oportunidad para que un depurador externo se adjunte.

Depuración anidada

En el ejemplo de depuración circular anterior, únicamente hay dos procesos en operación, con el proceso P1 depurando al proceso P2, y el proceso P2 depurando al proceso P1. Sin embargo, se pueden proporcionar procesos adicionales, para proporcionar una cascada de relaciones de depuración entre un proceso Pn y un proceso posterior Pn+1. Por ejemplo, donde n es la secuencia de números enteros de 1 a N, el proceso Pn puede depurar el proceso Pn+1 donde n no es igual a N. Además, el proceso PN puede depurar el proceso P1, cerrando por lo tanto el bucle y garantizando que cada proceso tenga un depurador adjunto.

Reentrada

Tras la realización de una función, el código del elemento a depurar puede transferir el control al elemento a depurar en ubicaciones distintas a la cercana a la instrucción que induce la excepción, por ejemplo, llamando a funciones en el contexto del elemento a depurar. Esto tiene el beneficio de ocultar el flujo de control del análisis estático, ya que la decisión del flujo se delega en el proceso de depurador. Por ejemplo, La función Debuggee1 activa la función Debugger2, que llama a Debuggee2; ninguna llamada desde Debuggee1 a Debugee2 es visible por el análisis estático del elemento a depurar en solitario.

Re-adjutnar

Para complicar el análisis, el proceso de depurador puede intentar continuamente re-adjuntarse al elemento a depurar. Esto garantizaría que, si el depurador se des-adjuntara alguna vez del elemento a depurar, es probable que se re-adjunte antes de que se adjunte otro depurador.

Detección de des-adjunte

Para complicar el análisis, si el proceso depurador no puede adjuntarse al depurador porque hay otro depurador adjunto, puede intentar des-adjuntar/desactivar este otro depurador.

Comprobación mutua

El depurador y el elemento a depurar, una vez adjuntos, pueden comprobar que sus respectivos id de proceso son coherentes. Esto detectaría la sustitución y/o inserción del depurador en la cadena. Las comprobaciones de coherencia podrían incluir si el pid de depurador/elemento a depurar cambia, si el pid de depurador/elemento a depurar son padre/hijo (extendido a anidamiento)

La Figura 6 ilustra un diagrama de bloques de una implementación de un dispositivo informático 400 dentro del que puede ejecutarse un conjunto de instrucciones, para hacer que el dispositivo informático realice una cualquiera o más de las metodologías analizadas en el presente documento. En implementaciones alternativas, el dispositivo informático puede estar conectado (por ejemplo, en red) a otras máquinas en una red de área local (LAN), una intranet, una extranet o Internet. El dispositivo informático puede operar en la capacidad de un servidor o una máquina cliente en un entorno de red cliente-servidor, o como una máquina de pares en un entorno de red entre pares (o distribuido). El

dispositivo informático puede ser un ordenador personal (PC), un ordenador de tableta, un decodificador de salón (STB), un asistente personal digital (PDA), un teléfono celular, un dispositivo web, un servidor, un enrutador de red, conmutador o puente, o cualquier máquina que pueda ejecutar un conjunto de instrucciones (secuenciales o de otra manera) que especifican acciones a realizar por esa máquina. Además, mientras que únicamente se ilustra un único dispositivo informático, la expresión "dispositivo informático" también se considerará que incluye cualquier colección de máquinas (por ejemplo, ordenadores) que ejecutan individual o conjuntamente un conjunto (o múltiples conjuntos) de instrucciones para realizar una cualquiera o más de las metodologías analizadas en el presente documento.

El dispositivo informático 400 de ejemplo incluye un dispositivo de procesamiento 402, una memoria principal 404 (por ejemplo, memoria de solo lectura (ROM), memoria flash, memoria de acceso aleatorio dinámica (DRAM) tal como DRAM síncrona (SDRAM) o DRAM Rambus (RDRAM), etc.), una memoria estática 406 (por ejemplo, memoria flash, una memoria de acceso aleatorio estática (SRAM), etc.), y una memoria secundaria (por ejemplo, un dispositivo de almacenamiento de datos 418), que se comunican entre sí a través de un bus 430.

El dispositivo de procesamiento 402 representa uno o más procesadores de propósito general, tales como un microprocesador, unidad central de procesamiento, o similares. Más particularmente, el dispositivo de procesamiento 402 puede ser un microprocesador de cálculo de conjunto de instrucciones complejo (CISC), microprocesadores de cálculo de conjunto de instrucciones reducido (RISC), microprocesador de palabra de instrucción muy larga (VLIW), procesador que implementa otros conjuntos de instrucciones, o procesadores que implementan una combinación de conjuntos de instrucciones. El dispositivo de procesamiento 402 también puede ser uno o más dispositivos de procesamiento de propósito especial, tales como un circuito integrado específico de la aplicación (ASIC), una matriz de puertas programables en campo (FPGA), un procesador de señales digitales (DSP), procesador de red o similar. El dispositivo de procesamiento 402 está configurado para ejecutar la lógica de procesamiento (instrucciones 422) para realizar las operaciones y etapas analizadas en el presente documento.

El dispositivo informático 400 puede incluir además un dispositivo de interfaz de red 408. El dispositivo informático 400 también puede incluir una unidad de visualización de vídeo 410 (por ejemplo, una pantalla de cristal líquido (LCD) o un tubo de rayos catódicos (CRT)), un dispositivo de entrada alfanumérica 412 (por ejemplo, un teclado o pantalla táctil), un dispositivo de control de cursor 414 (por ejemplo, un ratón o pantalla táctil) y un dispositivo de audio 416 (por ejemplo, un altavoz).

El dispositivo de almacenamiento de datos 418 puede incluir uno o más medios de almacenamiento legibles por máquina (o más específicamente uno o más medios de almacenamiento legibles por ordenador no transitorios) 428 en los que se almacenan uno o más conjuntos de instrucciones 422 que incorporan una cualquiera o más de las metodologías o funciones descritas en el presente documento. Las instrucciones 422 también pueden residir, total o al menos parcialmente, dentro de la memoria principal 404 y/o dentro del dispositivo de procesamiento 402 durante la ejecución de la misma por el sistema informático 400, constituyendo también la memoria principal 404 y el dispositivo de procesamiento 402 medios de almacenamiento legibles por ordenador.

Los diversos métodos descritos anteriormente pueden implementarse mediante un programa informático. El programa informático puede incluir código informático dispuesto para dar instrucciones a un ordenador para realizar las funciones de uno o más de los diversos métodos descritos anteriormente. El programa informático y/o el código para realizar tales métodos pueden proporcionarse a un aparato, tal como un ordenador, en uno o más medios legibles por ordenador o, más en general, un producto de programa informático. Los medios legibles por ordenador pueden ser transitorios o no transitorios. El uno o más medios legibles por ordenador podrían ser, por ejemplo, un medio electrónico, magnético, óptico, electromagnético, infrarrojo o semiconductor, o de propagación para la transmisión de datos, por ejemplo, para descargar el código a través de Internet. Como alternativa, el uno o más medios legibles por ordenador podrían tomar la forma de uno o más medios legibles por ordenador físicos, tales como memoria de semiconductores o de estado sólido, cinta magnética, un disquete informático extraíble, una memoria de acceso aleatorio (RAM), una memoria de solo lectura (ROM), un disco magnético rígido y un disco óptico, tal como un CD-ROM, CD-R/W o DVD.

En una implementación, los módulos, componentes y otras características descritas en el presente documento (por ejemplo, la unidad de control 410 en relación con la Figura 6) pueden implementarse como componentes discretos o integrarse en la funcionalidad de componentes de hardware tales como ASICS, FPGA, DSP o dispositivos similares como parte de un servidor de individualización.

Un "componente de hardware" es un componente tangible (por ejemplo, no transitorio) físico (por ejemplo, un conjunto de uno o más procesadores) que puede de realizar ciertas operaciones y puede configurarse o disponerse de una cierta manera física. Un componente de hardware puede incluir circuitería o lógica especializados que están configurados permanentemente para realizar ciertas operaciones. Un componente de hardware puede ser o incluir un procesador de propósito especial, tal como una matriz de puertas programables en campo (FPGA) o un ASIC. Un componente de hardware también puede incluir circuitería o lógica programable que están configurados temporalmente por software para realizar ciertas operaciones.

En consecuencia, la expresión "componente de hardware" debe entenderse que abarca una entidad tangible que

puede estar construida físicamente, configurada permanentemente (por ejemplo, de cableado permanente), o configurada temporalmente (por ejemplo, programada) para operar de cierta manera o para realizar ciertas operaciones descritas en el presente documento.

Además, los módulos y componentes pueden implementarse como firmware o circuitería funcional dentro de dispositivos de hardware. Además, los módulos y componentes pueden implementarse en cualquier combinación de dispositivos de hardware y componentes de software, o únicamente en software (por ejemplo, código almacenado o incorporado de otra manera en un medio legible por máquina o en un medio de transmisión).

A menos que se indique específicamente lo contrario, como es evidente a partir del siguiente análisis, se aprecia que, a través de toda la descripción, los análisis que utilizan términos tales como "recibir", "determinar", "comparar", "habilitar", "mantener", "identificar", "remplazar", o similares, se refieren a las acciones y procesos de un sistema informático o dispositivo informático electrónico similar, que manipula y transforma datos representados como cantidades físicas (electrónicas) dentro de los registros y memorias del sistema informático en otros datos representados de manera similar como cantidades físicas dentro de las memorias o registros del sistema informático u otros dispositivos de almacenamiento, transmisión o visualización de este tipo.

REIVINDICACIONES

1. Un método para asegurar el software, que comprende:

lanzar un proceso de software;
adjuntar un proceso de depurador al proceso de software;
ejecutar el proceso de software de tal manera que el proceso de depurador sea invocado al menos una vez;
realizar una o más funciones dentro del proceso de depurador en respuesta a la invocación del proceso de depurador, teniendo la una o más funciones una salida dependiente de los datos asociados con el proceso de software,
en donde el proceso de software genera una estructura de datos que comprende parámetros requeridos para la realización de la una o más funciones dentro del proceso de depurador antes de la invocación del proceso de depurador, **caracterizado por que**
los parámetros requeridos para la realización son parámetros de función y la generación de la estructura de datos comprende, cuando se alcanza el punto de entrada de una función colocada en el depurador mientras se ejecuta el proceso de software, serializar los parámetros de función en la estructura de datos (s54), y,
después de serializar los parámetros de la función en la estructura de datos, ejecutar una instrucción que induce la excepción y, por lo tanto, provocar una excepción (s55),
activar el depurador (s56) que identifica la ubicación del elemento a depurar de la excepción, infiriendo de esta manera, por el depurador, la función a llevar a cabo dentro del proceso de depurador,
recuperar, por el depurador, los parámetros de función serializados a la estructura de datos desde la memoria del elemento a depurar (s57)
realizar la función dentro del proceso de depurador (s58) y devolver los resultados (s59), actualizar, por el depurador, parámetros en la estructura de datos (s60) y devolver el control al proceso de software (s61).

2. Un método de acuerdo con la reivindicación 1, en donde la salida comprende una salida de datos para su uso por el proceso de software.

3. Un método de acuerdo con una cualquiera de las reivindicaciones 1 a 2, en donde el proceso de software actúa para depurar el proceso de depurador.

4. Un método de acuerdo con una cualquiera de las reivindicaciones 1 a 3, que comprende además lanzar un proceso adicional para depurar el proceso de depurador.

5. Un método de acuerdo con una cualquiera de las reivindicaciones 1 a 4, en donde la salida de una función dada puede indicar múltiples puntos de retorno dentro del proceso de software para continuar la ejecución.

6. Un método de acuerdo con una cualquiera de las reivindicaciones 1 a 5, en donde el proceso de depurador proporciona capacidades de soporte de memoria para permitir que la una o más funciones recuperen datos desde la memoria dentro del espacio de direcciones del proceso de software.

7. Un método de acuerdo con una cualquiera de las reivindicaciones 1 a 6, en donde el proceso de depurador se invoca cuando se alcanza un punto de interrupción dentro del proceso de software.

8. Un método de acuerdo con una cualquiera de las reivindicaciones 1 a 7, que comprende además des-adjuntar el proceso de depurador del proceso de software cuando el proceso de software se haya completado.

9. Un método de acuerdo con una cualquiera de las reivindicaciones 1 a 8, en donde el proceso de software implementa un ejecutable, tal como una aplicación.

10. Un método de acuerdo con una cualquiera de las reivindicaciones 1 a 9, en donde el proceso de software implementa una biblioteca.

11. Un producto de programa ejecutable por ordenador que comprende código ejecutable por ordenador para llevar a cabo el método de una cualquiera de las reivindicaciones 1 a 10.

12. Un dispositivo configurado para llevar a cabo el método de una cualquiera de las reivindicaciones 1 a 10.

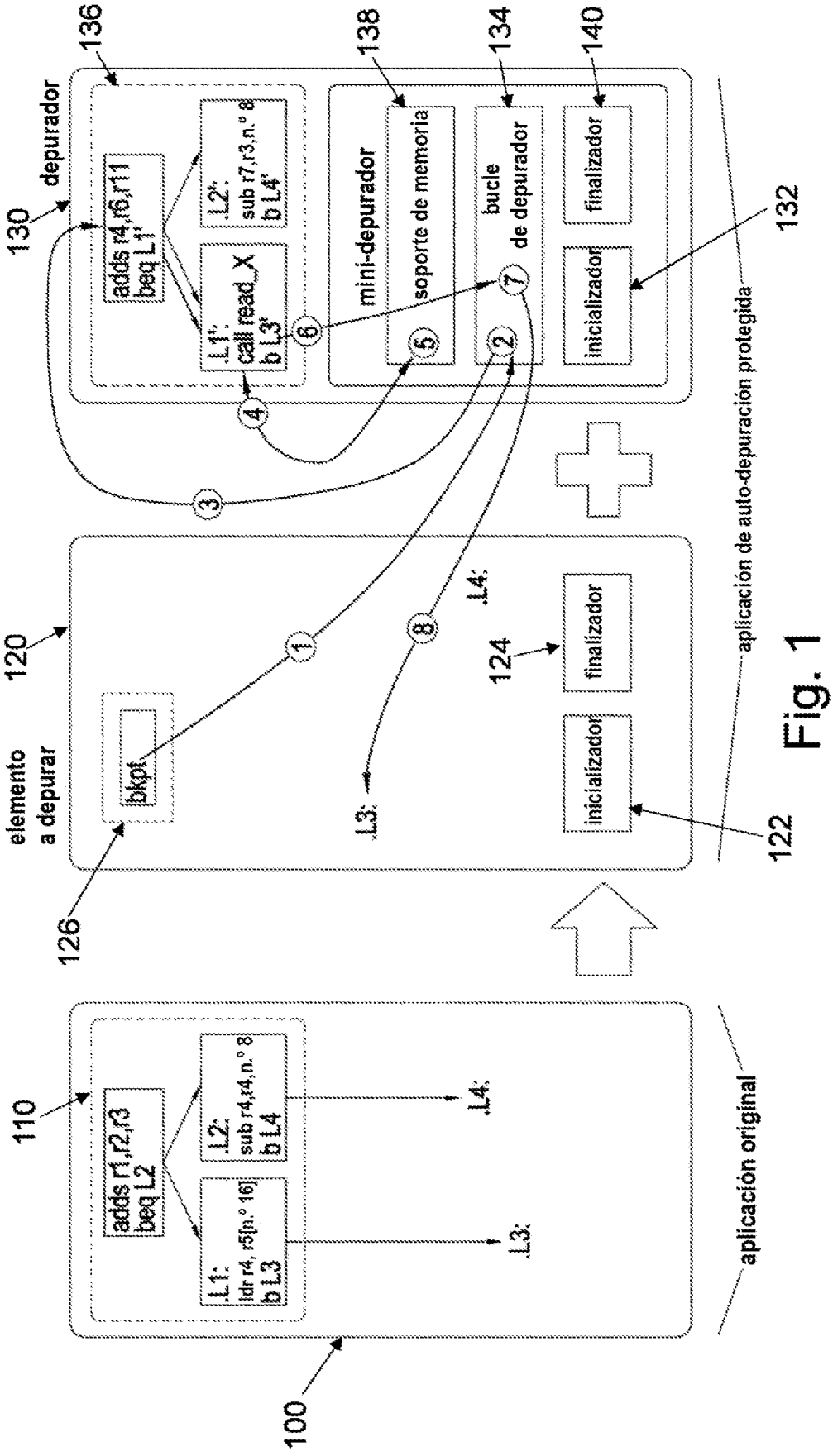
100
110
adds r1,r2,r3
beq L2
.L1:
ldr r4, r5[n.º 16]
b L3
.L2:
sub r4,r4,n.º 8
b L4
.L3:
.L4:
aplicación original
120
elemento
a depurar
bkpt
126
inicializador
finalizador
122
124
.L3:
.L4:
aplicación de auto-depuración protegida
130
depurador
136
adds r4,r6,r11
beq L1'
.L1':
call read_X
b L3'
.L2':
sub r7,r3,n.º 8
b L4'
138
mini-depurador
soporte de memoria
134
bucle
de depurador
140
finalizador
inicializador
132
1
2
3
4
5
6
7
8

Fig. 1

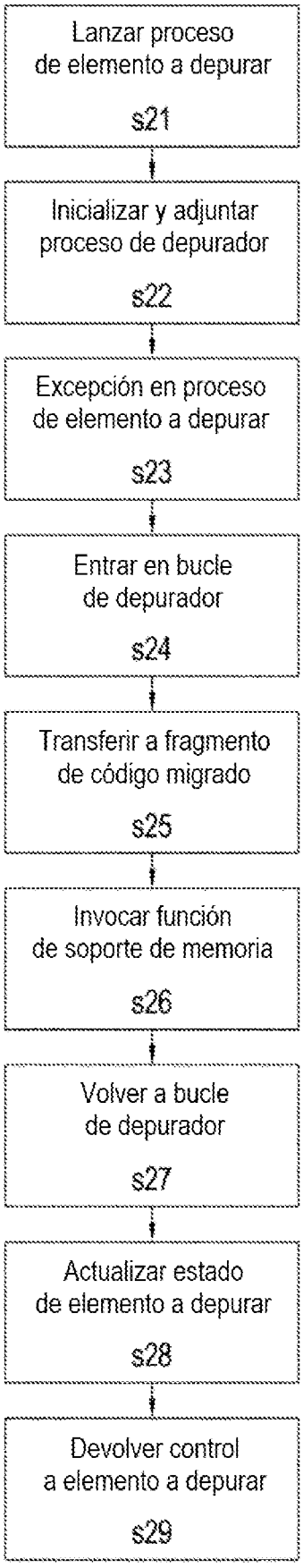
Lanzar proceso de elemento a depurar
s21
Inicializar y adjuntar proceso de depurador
s22
Excepción en proceso de elemento a depurar
s23
Entrar en bucle de depurador
s24
Transferir a fragmento de código migrado
s25
Invocar función de soporte de memoria
s26
Volver a bucle de depurador
s27
Actualizar estado de elemento a depurar
s28
Devolver control a elemento a depurar
s29

Fig. 2

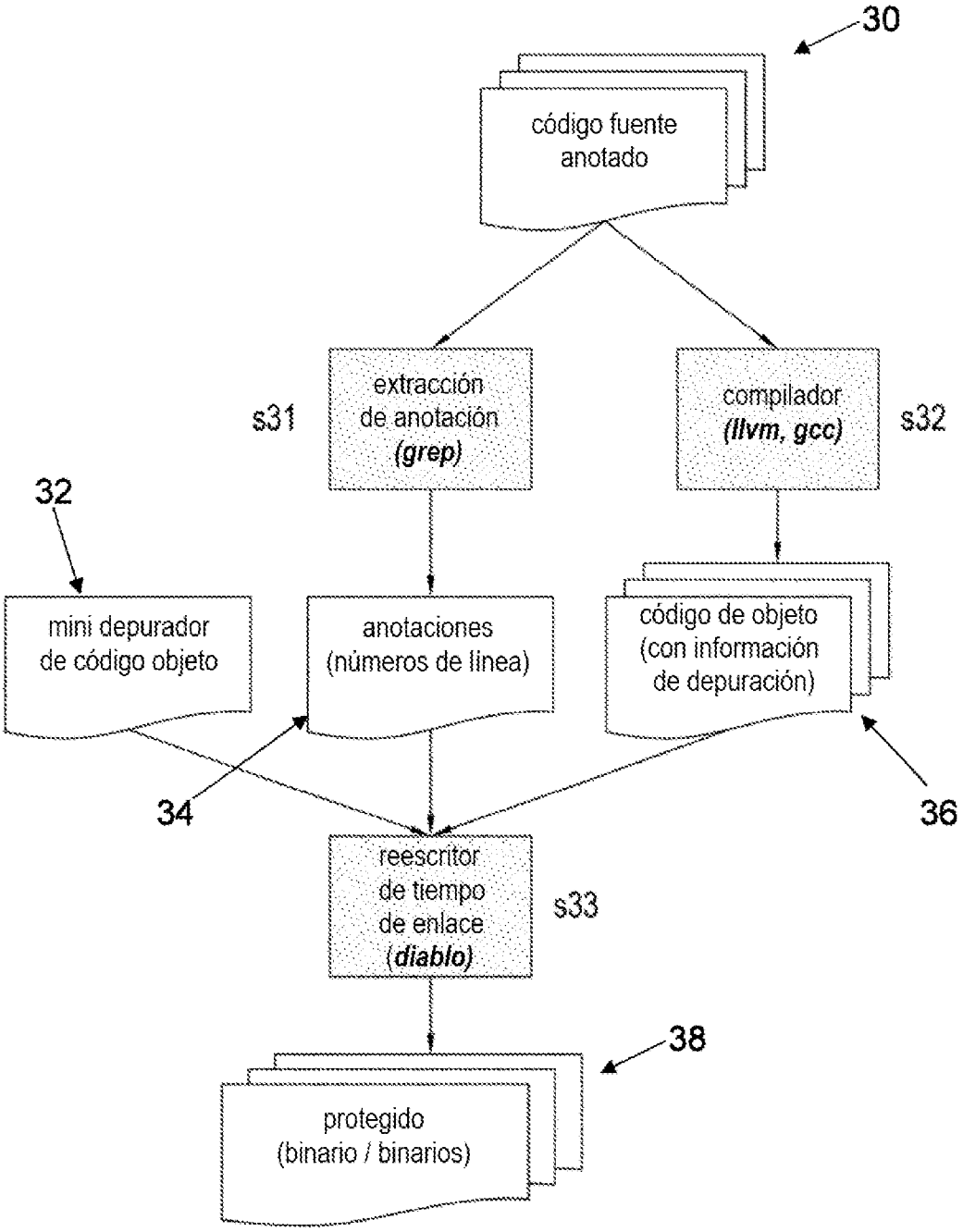
30
código fuente anotado
s31
extracción de anotación (grep)
compilador (llvm, gcc)
s32
32
mini depurador de código objeto
anotaciones (números de línea)
código de objeto (con información de depuración)
34
36
reescritor de tiempo de enlace (diablo)
s33
38
protegido (binario / binarios)

Fig. 3

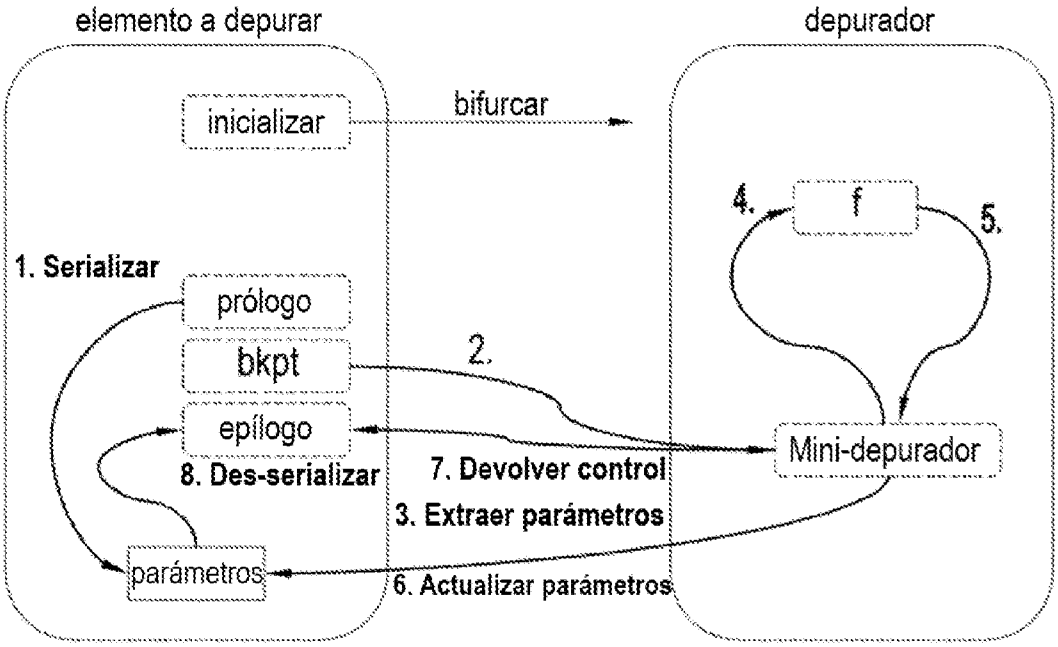
elemento a depurar
depurador
inicializar
bifurcar
4.
f
5.
1. Serializar
prólogo
bkpt
2.
epílogo
Mini-depurador
8. Des-serializar
7. Devolver control
3. Extraer parámetros
parámetros
6. Actualizar parámetros

Fig. 4

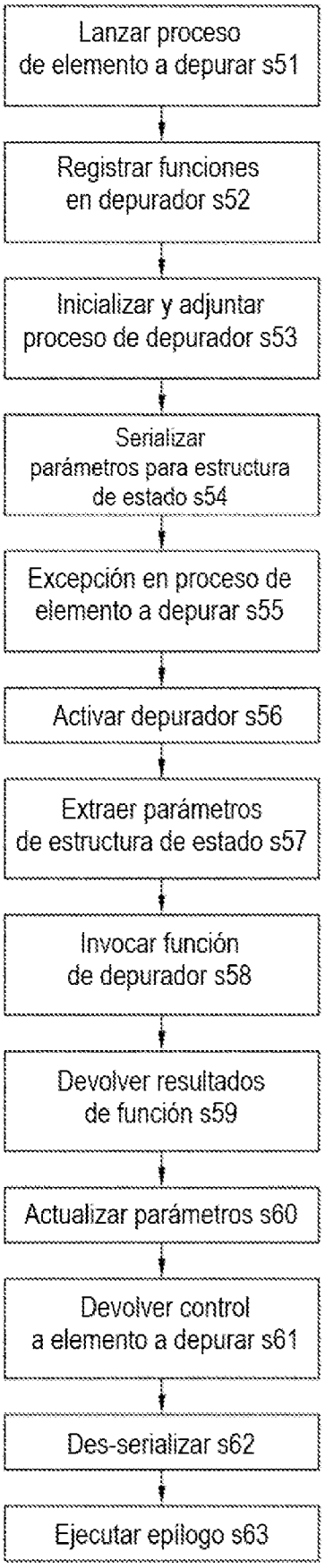
Lanzar proceso de elemento a depurar s51
Registrar funciones en depurador s52
Inicializar y adjuntar proceso de depurador s53
Serializar parámetros para estructura de estado s54
Excepción en proceso de elemento a depurar s55
Activar depurador s56
Extraer parámetros de estructura de estado s57
Invocar función de depurador s58
Devolver resultados de función s59
Actualizar parámetros s60
Devolver control a elemento a depurar s61
Des-serializar s62
Ejecutar epílogo s63

Fig. 5

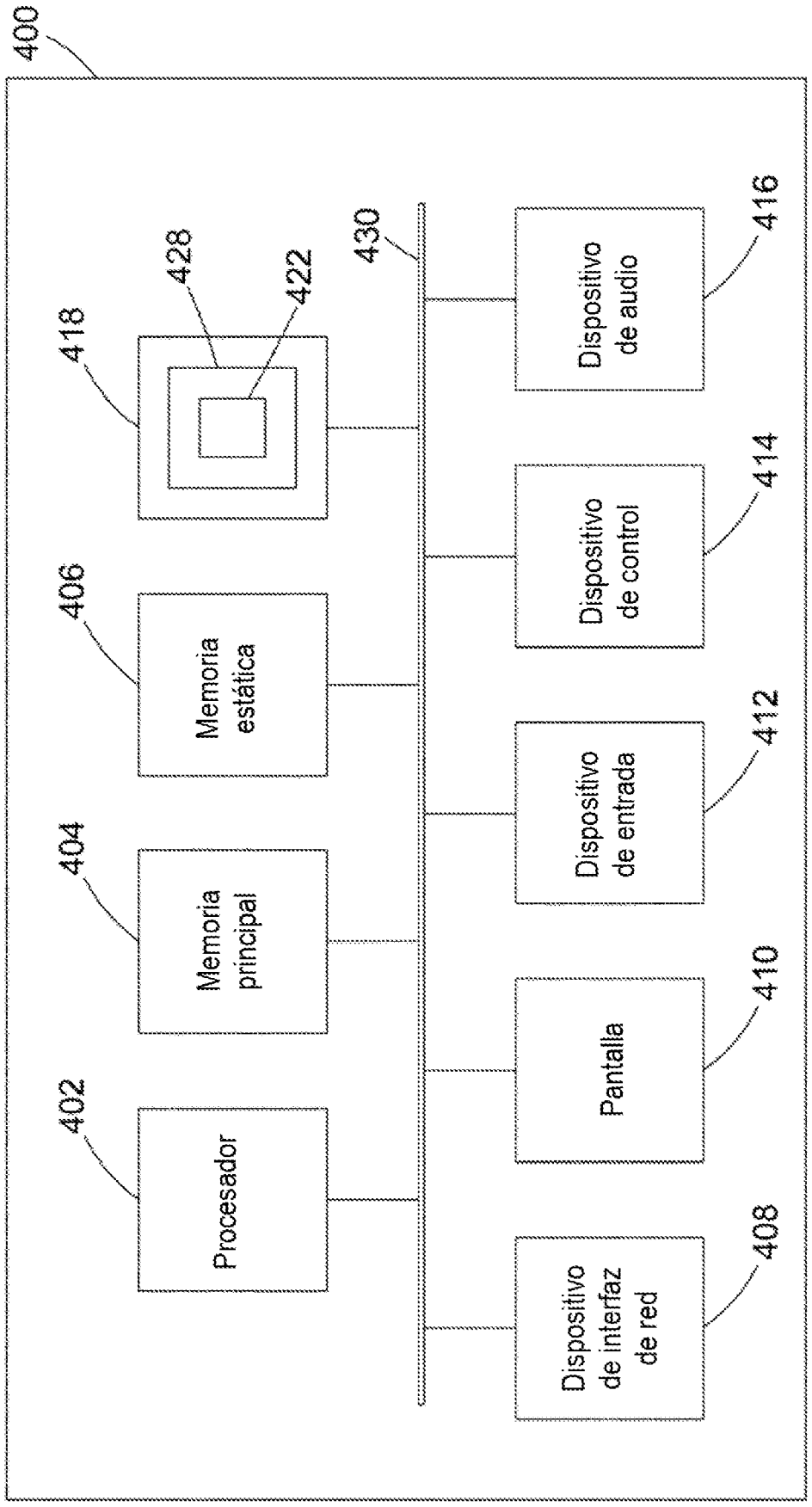
400
Procesador
402
Memoria principal
404
Memoria estática
406
418
428
422
430
Dispositivo de interfaz de red
408
Pantalla
410
Dispositivo de entrada
412
Dispositivo de control
414
Dispositivo de audio
416

Fig. 6