(19) OFICINA ESPAÑOLA DE PATENTES Y MARCAS

ESPAÑA

(11) Número de publicación: **2 675 371**

(51) Int. CI.:

G06F 21/14 (2013.01)
G06F 21/53 (2013.01)
G06F 9/455 (2008.01)

(12) TRADUCCIÓN DE PATENTE EUROPEA T3

(86) Fecha de presentación y número de la solicitud internacional: **16.10.2007 PCT/US2007/081485**

(87) Fecha y número de publicación internacional: **25.09.2008 WO08115279**

(96) Fecha de presentación y número de la solicitud europea: **16.10.2007 E 07874416 (6)**

(97) Fecha y número de publicación de la concesión europea: **02.05.2018 EP 2076863**

(54) Título: **Virtualización para una resistencia a manipulación diversificada**

(30) Prioridad:

27.10.2006 US 553841

(45) Fecha de publicación y mención en BOPI de la traducción de la patente:
10.07.2018

(73) Titular/es:

MICROSOFT TECHNOLOGY LICENSING, LLC (100.0%)
One Microsoft Way
Redmond, WA 98052, US

(72) Inventor/es:

ANCKAERT, BERTRAND;
JAKUBOWSKI, MARIUSZ H. y
VENKATESAR, RAMARATHNAM

(74) Agente/Representante:

CARPINTERO LÓPEZ, Mario

DESCRIPCIÓN

Virtualización para una resistencia a manipulación diversificada

Antecedentes

A pesar de los enormes esfuerzos de los proveedores de software, los mecanismos de protección de software todavía se rompen de forma regular. Además, un ataque exitoso en una copia a menudo se puede replicar automáticamente en otras copias. Por ejemplo, si un proveedor de software distribuye versiones de evaluación de una pieza de software, también se puede aplicar una grieta que elimine la limitación de tiempo de una copia a todas las otras copias distribuidas. Además, los modelos de distribución convencionales pueden permitir ataques en serie que afectan rápidamente a miles de usuarios.

La diversificación es un concepto que puede usarse para mejorar la seguridad del software y confundir el ataque. Sin embargo, las técnicas de diversificación desarrolladas para la seguridad del software no siempre son transferibles a la protección del software ya que se pueden aplicar diferentes reglas. Por ejemplo, la mayoría de la diversificación en tiempo de ejecución introducida para la seguridad podría desactivarse fácilmente cuando un atacante tenga acceso físico al programa y al entorno de ejecución.

Como se describe en este documento, varias técnicas de diversidad para la protección del software proporcionan defensas renovables en el espacio y en el tiempo, por ejemplo, dando a cada usuario una copia diferente y renovando las defensas con actualizaciones a medida.

El documento US2005/0069131 A1 se refiere a un motor de renderizado y encriptación para la ofuscación del programa de aplicación. El proveedor del programa de aplicación comprende un ocultador para ofuscar un programa de aplicación ejecutable almacenado en el proveedor del programa de aplicación. El proveedor del programa de aplicación también comprende un descriptor de ofuscación que describe los datos ofuscados. El descriptor de ofuscación incluye una indicación del procedimiento de ofuscación utilizado por el ofuscador para crear un programa de aplicación ofuscado. El secreto se usa para encriptar el descriptor de ofuscación. El dispositivo de usuario comprende el despachador configurado para usar una de las múltiples tablas de envío para determinar una correspondencia entre un valor de código de operación y una referencia a un procedimiento de implementación de instrucciones. Cada tabla de envío utiliza un esquema de codificación de valor de código de operación diferente para al menos un código de operación en un conjunto de instrucciones. Si el despachador ejecuta un programa de aplicación ofuscado utilizando el esquema de codificación de valor de código de operación de la tabla de despacho permutada, la tabla de despacho permutada se usa para determinar una correspondencia entre un valor de código de operación y una referencia a un procedimiento de implementación de instrucción.

Collberg C. et al, "Rompiendo abstracciones y estructuras de datos no estructuradas", procedimientos. Conferencia internacional de 1998 sobre lenguajes de ordenador, IEEE COMPUT. SOC LOS ALAMITOS, CA, EUA, (1998), ISBN 978-0-8186-8454-8, páginas 28 - 38 se refiere a romper abstracciones y estructuras de datos no estructuradas. Un ofuscador es una herramienta que, mediante la aplicación de transformaciones de código, convierte un programa en uno equivalente que es más difícil de aplicar ingeniería inversa.

Cohen FB, "Protección del sistema operativo mediante la evolución del programa", COMPUTERS & SECURITY, ELSEVIER SCIENCE PUBLISHERS, AMSTERDAM, NL, vol. 12, n.º 6, ISSN 0167-4048, páginas 565 - 584 se relaciona con la protección del sistema operativo a través de la evolución del programa. La evolución se puede proporcionar de muchas maneras y en muchos lugares diferentes, desde un pequeño número finito de defensas proporcionadas por diferentes proveedores, y extendiéndose hacia un sistema defensivo que se desarrolla a sí mismo durante cada llamada al sistema. Con más evolución, obtenemos menos rendimiento, pero un mayor coste de ataque. Por lo tanto, como en todas las funciones de protección, hay un precio a pagar para una mayor protección.

El documento US2006/0136867 A1 se refiere a la diversificación del código. El código del ordenador de entrada puede incluir el código fuente y la generación automática de múltiples versiones de código informático diversificado puede incluir la generación automática de múltiples versiones de código fuente diversificado. El código del ordenador de entrada puede incluir código binario y la generación automática de múltiples versiones de código informático diversificado puede incluir la generación automática de múltiples versiones de código binario diversificado. La generación automática de código informático diversificado puede incluir la identificación de uno o más destinos de salto en el código del ordenador de entrada y la revisión de los destinos de salto en función de una o más operaciones de diversificación. La generación automática de código informático diversificado puede incluir la aplicación de múltiples operaciones de diversificación. Las operaciones de diversificación pueden incluir uno o más de inserción de código no funcional; insertar código duplicado (por ejemplo, duplicados de operaciones de asignación), donde el código duplicado duplica las operaciones especificadas en el código del ordenador de entrada; cambio de nombre de etiquetas; y redactando un código de ordenador.

Michael C.C. et al., "Dos sistemas para la diversificación automática de software", PROCEDIMIENTOS DE LA CONFERENCIA Y EXPOSICIÓN SOBRE SOBREVIVENCIA DE INFORMACIÓN DE DARPA 2000, vol. 2, páginas 220 - 230 se refiere a dos sistemas para la diversificación automática de software. La redundancia y la diversidad

artificial se utilizan en la seguridad de la información. Se discuten dos enfoques fundamentalmente diferentes para crear diversidad: la mutación del software basada en la sintaxis y las representaciones alternativas de la semántica del programa obtenidas por interpretación abstracta.

El documento US 7.051.200 B1 se refiere a un sistema y procedimiento para interconectar un proceso de software para proteger repositorios. Se proporciona una caja negra que es un repositorio seguro que usa técnicas criptográficas, preferiblemente técnicas de claves públicas/privadas, para realizar servicios de descifrado y autenticación de una manera segura que resiste el descubrimiento de claves secretas utilizadas por las técnicas criptográficas.

Linger RC, "Generación sistemática de diversidad estocástica como una barrera de intrusión en el software del sistema de supervivencia", PROCEDIMIENTOS DE LA 32ª CONFERENCIA INTERNACIONAL HAWAII ANUAL SOBRE CIENCIAS DEL SISTEMA se relaciona con la generación sistemática de diversidad estocástica como una barrera de intrusión en el software de sistemas con capacidad de supervivencia. Los intrusos a menudo confían en el análisis del código fuente para identificar y explotar vulnerabilidades en el software. La capacidad de los intrusos para comprender y analizar el código se puede reducir drásticamente a través de un proceso de desestructuración estocástica para aumentar la complejidad del software como una barrera de intrusión a la vez que se preservan la función y el rendimiento. Existe un proceso sistemático para transformar programas complejos y no estructurados en una estructura estructurada equivalente a la función para mejorar la comprensibilidad y el mantenimiento. Este proceso se invierte para introducir sistemáticamente la diversidad estocástica mediante la transformación de programas estructurados en programas no estructurados equivalentes a funciones de complejidad arbitraria que son prácticamente imposibles de comprender.

Sumario

El objeto de la presente invención es mejorar la resistencia a la manipulación de un software mientras se mantiene un buen rendimiento computacional del software.

Este objetivo se resuelve mediante el objeto de las reivindicaciones independientes.

Realizaciones preferidas se definen en las reivindicaciones dependientes.

Diversas técnicas ejemplares utilizan virtualización para la diversificación de código y/o máquinas virtuales (VM) para mejorar así la seguridad software. Por ejemplo, un procedimiento implementable por ordenador incluye proporcionar una arquitectura de conjunto de instrucciones (ISA) que comprende características para generar diversas copias de un programa, utilizando la arquitectura del conjunto de instrucciones para generar diversas copias de un programa y proporcionar una VM para la ejecución de una de las diversas copias del programa. También se describen otras diversas tecnologías ejemplares.

Dibujos

Ejemplos no limitativos y no exhaustivos se describen con referencia a las siguientes figuras:

La figura 1 es un diagrama de un sistema y una arquitectura general que incluye una capa de virtualización como una arquitectura de conjunto de instrucciones (ISA) personalizada/modificada y/o una máquina virtual (VM);
La figura 2 es un diagrama de bloques de un procedimiento ejemplar que incluye un módulo de seguridad para generar código personalizado y/o una máquina virtual personalizada.
La figura 3 es un diagrama de bloques de un procedimiento ejemplar para generar una máquina virtual personalizada.
La figura 4 es un diagrama de bloques de un procedimiento ejemplar para diversificar funciones en código para generar código personalizado.
La figura 5 es un diagrama de un ejemplo que se refiere al procedimiento de la figura 4;
La figura 6 es un diagrama de bloques de un procedimiento ejemplar para diversificar datos y/o estructura de datos.
La figura 7 es un diagrama de bloques de varias características del marco de trabajo que pueden usarse para la diversificación y la resistencia a la manipulación.
La figura 8 es un diagrama de un modelo de ejecución con características que pueden utilizarse para la diversificación y la resistencia a la manipulación.
La figura 9 es un diagrama de bloques de diversos enfoques que se pueden aplicar a la semántica de instrucciones con fines de diversificación y resistencia a la manipulación.
La figura 10 es un diagrama de bloques de diversos enfoques que se pueden aplicar a la codificación de instrucciones con fines de diversificación y resistencia a la manipulación.
La figura 11 es un diagrama de bloques de diversos enfoques que se pueden aplicar a un ciclo de recuperación para fines de diversificación y resistencia a la manipulación.
La figura 12 es un diagrama de bloques de diversos enfoques que se pueden aplicar a un contador de programa (PC) y/o representación de programa con fines de diversificación.
La figura 13 es un diagrama de un ejemplo de un fragmento de código en un árbol extendido, para fines de resistencia a la manipulación diversificada.

La figura 14 es un diagrama de bloques de diversos enfoques que pueden aplicarse a la implementación de una máquina virtual (VM) con fines de diversificación.
La figura 15 muestra un diagrama de bloques de un dispositivo de ordenador ejemplar.

Descripción detallada

Sumario

Ejemplos de técnicas utilizan la seguridad basada en software para código que se ejecuta en una máquina virtual u otra máquina que puede ser controlada (por ejemplo, operado de una manera particular). Varios ejemplos modifican el código, modifican datos y/o modifican el funcionamiento de la máquina virtual de una manera cooperativa para mejorar la seguridad. Por ejemplo, las características asociadas con el código se pueden identificar y usar para diversificar las instancias del código. Varios ejemplos incluyen el uso de una arquitectura de conjuntos de instrucciones (ISA) personalizada o modificada que se emula sobre una arquitectura existente o una arquitectura modificada. Cuando la arquitectura existente incluye una capa de virtualización (por ejemplo, una máquina virtual o motor de tiempo de ejecución o simplemente "tiempo de ejecución"), un enfoque ejemplar puede agregar otra capa de virtualización sobre la capa de virtualización existente. La implementación de varias técnicas puede ocurrir mediante el uso de una máquina virtual personalizada que opera sobre una máquina virtual subyacente o mediante el uso de una máquina virtual modificada (por ejemplo, la modificación de una máquina virtual subyacente). Para una mayor seguridad a expensas del rendimiento, dicho proceso puede repetirse para generar una pila de máquinas virtuales, cada una de las cuales virtualiza el conjunto de instrucciones de la máquina directamente debajo.

Con respecto a la diversificación, puede hacerse una analogía a la diversidad genética donde los componentes comunes (por ejemplo, bloques de construcción de ADN) se ensamblan en una variedad de maneras para mejorar de este modo la diversidad de una especie. A su vez, es poco probable que un agente malicioso (por ejemplo, un virus) afecte a todos los miembros de las especies genéticamente diversas. También existe una analogía para algunos parásitos donde cientos de genes pueden fabricar proteínas que se mezclan y combinan. Tal diversidad proteica ayuda a dicho parásito a evadir la detección del sistema inmune. Sin embargo, aunque la diversidad genética en las especies a menudo se asocia con la diversidad fenotípica (es decir, la diversidad de expresión o manifestación), como se describe en este documento, la diversificación del código no debe alterar la experiencia del usuario. En otras palabras, dadas las mismas entradas, todas las instancias de un código diversificado deberían ejecutarse para producir el mismo resultado.

Considere las siguientes ecuaciones generalizadas:

$$\text{genotipo} + \text{entorno} = \text{fenotipo} \quad (1)$$

$$\text{código/datos} + \text{máquina} = \text{resultado} \quad (2)$$

En la ecuación 2, con código y/o datos diversificados, la máquina puede ser una máquina personalizada o una máquina modificada o controlada que garantiza que el resultado sea sustancialmente el mismo. Cuando se usa una máquina virtual o máquinas virtuales, el hardware y/o sistema operativo subyacente normalmente no se ve afectado por los enfoques de seguridad ejemplares descritos en este documento; señalando, sin embargo, que tales enfoques pueden aumentar la demanda computacional.

Como se describe en el presente documento, la diversificación de código aumenta la seguridad confundiendo el ataque. Además, la diversidad del código puede confundir los ataques en serie, ya que es poco probable que la replicación de un ataque exitoso en una instancia del código tenga éxito en una instancia "genéticamente" diferente del código. Además, los datos (o la estructura de datos) pueden diversificarse para aumentar la seguridad. Además, las técnicas ejemplares para diversificar una máquina virtual personalizada o modificada se pueden usar para mejorar la seguridad con o sin codificación y/o diversificación de datos.

Aunque varios ejemplos se centran en la generación de bytecodes individualizados de los binarios MSIL, diversas técnicas ejemplares se pueden usar para los programas expresados en cualquier lenguaje de programación, lenguaje intermedio, bytecode o manera.

Como se describe aquí, el proceso de virtualización (por ejemplo, la ejecución de código en una máquina virtual o entorno virtual) facilita (i) la capacidad de hacer muchas versiones diferentes de un programa y (ii) la capacidad de hacer cada instancia de un programa resistente contra el ataque. Varias técnicas ejemplares utilizan la virtualización para emular una arquitectura de conjunto de instrucciones (ISA) personalizada sobre un entorno CLR gestionable, verificable y portátil. Se puede usar un tiempo de ejecución personalizado o un tiempo de ejecución modificado para emular la ISA personalizada, es decir, de alguna manera, el tiempo de ejecución subyacente debe ser capaz de administrar la diversidad introducida a través de la ISA personalizada.

La virtualización proporciona grados de libertad que pueden usarse en la diversificación. Por ejemplo, una arquitectura de conjuntos de instrucciones (ISA) (por ejemplo, y/o microarquitectura) típicamente incluye (1) semántica de instrucciones, (2) codificación de instrucciones, (3) codificación de código de operación, (4) representación de códigos y un contador de programas. y (5) una implementación interna correspondiente de una

máquina virtual. Dados estos grados de libertad, varios procedimientos ejemplares pueden generar muchas copias diversas de un programa con cualquiera de una variedad de mecanismos de protección.

Los grados de libertad asociados con una ISA proporcionan para el diseño y la selección de programas resistentes a la manipulación. Por ejemplo, la resistencia a la manipulación puede ser el resultado de (1) hacer las modificaciones locales más difíciles mediante (a) longitudes variables de instrucción, (b) conjuntos de instrucciones limitados, y fomentando (c) el solapamiento físico y (d) semántico; (2) hacer las modificaciones globales más difíciles por todo lo anterior y difuminar los límites entre el código, los datos y las direcciones; (3) hacer que la semántica de la instrucción sea variable; y (4) reubicar constantemente el código.

Ejemplos más específicos incluyen (i) la aleatorización de la semántica de instrucciones mediante la construcción de instrucciones a través de la combinación de instrucciones más pequeñas; (ii) elegir la semántica de instrucciones para permitir una mayor superposición semántica; (iii) apartarse de la representación de código lineal tradicional para representar el código como una estructura de datos tal como un árbol binario autoadaptable (bisel); (iv) asignar longitudes variables a los códigos de operación y operandos para complicar el desmontaje y dificultar las modificaciones locales; (v) limitar el conjunto de instrucciones para dar al atacante menos opciones para analizar y modificar el código; y (vi) hacer variable la correspondencia entre patrones de bits, códigos de operación y operandos. Algunos de estos ejemplos se describen con más detalle a continuación. La descripción detallada incluye el mejor modo actualmente contemplado.

Antes de describir diversos detalles, a efectos de contexto, la figura 1 muestra un sistema general 100 y la arquitectura 105 que incluye la virtualización. La arquitectura 105 muestra además una ISA y/o máquina virtual personalizada y/o modificada en relación con una máquina virtual denominada tiempo de ejecución de lenguaje común (CLR). El CLR en la figura 1 es, por ejemplo, un CLR capaz de manejar un código de lenguaje intermedio derivado de cualquiera de una variedad de lenguajes de programación orientados a objetos (OOPL), por lo tanto, el término "común". Mientras que la figura 1 se discute con referencia al marco de trabajo .NET™ (Microsoft Corp, Redmond, WA), se pueden usar técnicas ejemplares con otras arquitecturas.

El sistema 100 incluye varios dispositivos informáticos 101, 101', 102, 102' en comunicación a través de una red 103. Los dispositivos 101, 101’ pueden ser clientes (por ejemplo, PC, estaciones de trabajo, dispositivos livianos, dispositivos inteligentes, etc.) mientras que los dispositivos 102, 102' son servidores. La arquitectura 105 se muestra con algunos enlaces al dispositivo 101, el servidor 102' y la red 103.

El marco de trabajo .NET™ tiene dos componentes principales: el tiempo de ejecución de lenguaje común (CLR) y la biblioteca de clases marco de trabajo .NET™. Estos se muestran como asociados con aplicaciones gestionadas. El CLR es una máquina virtual (VM) en la base del marco de trabajo .NET™. El CLR actúa como un agente que administra el código en el momento de la ejecución, proporcionando servicios básicos como administración de memoria, administración de hilos y comunicación remota, mientras aplica la seguridad de tipo estricto y otras formas de exactitud de código que promueven la seguridad y robustez. El concepto de administración de código es un principio fundamental del CLR. El código que se dirige al CLR se conoce como código gestionado, mientras que el código que no se dirige al tiempo de ejecución se conoce como código no gestionado (mitad derecha de la arquitectura).

En el .NET™, los programas de marco de trabajo de trabajo se ejecutan dentro de un entorno de ejecución gestionado proporcionado por el CLR. El CLR mejora en gran medida la interactividad en el tiempo de ejecución entre programas, la portabilidad, la seguridad, la simplicidad de desarrollo, la integración entre idiomas y proporciona una base excelente para un amplio conjunto de bibliotecas de clases. Cada lenguaje dirigido al marco de trabajo .NET™ CLR compila el código fuente y produce metadatos y el código de Microsoft® Intermediate Language (MSIL). Aunque varios ejemplos mencionan MSIL, se pueden usar varias técnicas de seguridad ejemplares con otro código de lenguaje. Por ejemplo, se pueden usar varias técnicas con la mayoría de cualquier lenguaje de ensamblado de bajo nivel (por ejemplo, cualquier código de lenguaje intermedio (IL)). Se pueden usar diversas técnicas con bytecodes tales como los del marco de trabajo JAVA™ (Sun Microsystem, Sunnyvale, CA).

En el marco de trabajo .NET™, el código de programa incluye típicamente información conocida como "metadatos", o datos sobre datos. Los metadatos a menudo incluyen una especificación completa para un programa que incluye todos sus tipos, aparte de la implementación real de cada función. Estas implementaciones se almacenan como MSIL, que es un código independiente de la máquina que describe las instrucciones de un programa. El CLR puede usar este "manual" para dar vida a un programa .NET™ en tiempo de ejecución, proporcionando servicios mucho más allá de lo que es posible con el enfoque tradicional que se basa en la compilación de código directamente al lenguaje de ensamblaje.

La biblioteca de clases, el otro componente principal del marco de trabajo .NET™, es una amplia colección, orientada a objetos de tipos reutilizables para desarrollar aplicaciones que van desde aplicaciones de línea de comandos tradicional o una interfaz gráfica de usuario (GUI) a aplicaciones basadas en las últimas innovaciones proporcionadas por ASP.NET, como Web Forms y XML Web services.

Aunque se muestra el CLR en el lado aplicación gestionada de la arquitectura 105, el marco de trabajo .NET™

puede ser alojado por los componentes no gestionados que cargan el CLR en sus procesos e inician la ejecución de código gestionado, creando de ese modo un entorno de software que puede explotar características gestionadas y no gestionadas. El marco de trabajo .NET™ no solo proporciona varios ordenadores anfitriones de tiempo de ejecución, sino que también es compatible con el desarrollo de ordenadores anfitriones de tiempo de ejecución de terceros.

Por ejemplo, ASP.NET aloja el tiempo de ejecución (RT) para proporcionar un entorno escalable, del lado del servidor para el código gestionado. ASP.NET trabaja directamente con el tiempo de ejecución para habilitar las aplicaciones ASP.NET y los servicios web XML.

La aplicación de navegador de Internet Explorer® (Microsoft Corp.) es un ejemplo de una aplicación no gestionada que aloja el tiempo de ejecución (por ejemplo, en la forma de una extensión de tipo MIME). El uso del software Internet Explorer® para alojar el tiempo de ejecución permite a un usuario incorporar componentes gestionados o controles de Windows Forms en documentos HTML. Hospedar el tiempo de ejecución de esta forma hace posible el código móvil gestionado (similar a los controles de Microsoft® ActiveX®), pero con importantes mejoras que solo el código gestionado puede ofrecer, como la ejecución semi-confiable y el almacenamiento de archivos aislados.

Como se describe aquí, el código gestionado puede ser cualquier código que se dirige a un tiempo de ejecución (por ejemplo, una máquina virtual, el motor de tiempo de ejecución, etc., donde las interfaces de tiempo de ejecución con un sistema operativo subyacente (OS) típicamente directamente asociados con el control del hardware (HW)). En el marco de trabajo de .NET™, el código gestionado se dirige al CLR y, por lo general, incluye información adicional conocida como metadatos que «se describe a sí mismo». Si bien el código gestionado y no gestionado se puede ejecutar en el CLR, el código gestionado incluye información que permite al CLR garantizar, por ejemplo, la ejecución segura y la interoperabilidad.

Además de código gestionado, los datos gestionados existen en la arquitectura del marco de trabajo .NET™. Algunos lenguajes .NET™ usan datos gestionados por defecto (por ejemplo, C #, Visual Basic.NET, JScript.NET) mientras que otros (por ejemplo, C ++) no lo hacen. Al igual que con el código gestionado y no gestionado, es posible el uso de datos gestionados y no gestionados en aplicaciones .NET™ (por ejemplo, datos que no se recogen basura, sino que son atendidos por código no gestionado).

Un programa o código se distribuye normalmente como un archivo ejecutable portátil (por ejemplo, un "PE"). En un .NET™ PE, el primer bloque de datos dentro del envoltorio PE es MSIL, que, como ya se mencionó, se ve aproximadamente como un código de lenguaje de ensamblaje de bajo nivel. El MSIL es convencionalmente lo que se compila y ejecuta en el marco de trabajo .NET™. Un segundo bloque de datos con el PE es convencionalmente los metadatos y describe los contenidos del PE (por ejemplo, qué procedimientos proporciona, qué parámetros toman y qué devuelven). Un tercer bloque de datos se conoce como el manifiesto, que describe convencionalmente qué otros componentes necesita el ejecutable para ejecutarse. El manifiesto también puede contener claves públicas de componentes externos, de modo que el CLR pueda garantizar que el componente externo esté identificado correctamente (es decir, el componente requerido por el ejecutable).

Cuando se ejecuta el ejecutable, el CLR .NET™ puede utilizar compilación en tiempo de ejecución (JIT). La compilación del JIT permite que todo el código gestionado se ejecute en el lenguaje de máquina nativo del sistema en el que se está ejecutando (OS/HW). Según el JIT, cuando se llama a cada procedimiento dentro del ejecutable, se compila en código nativo y, dependiendo de la configuración, las llamadas subsiguientes al mismo procedimiento no necesariamente tienen que someterse a la misma compilación, lo que puede reducir la sobrecarga (es decir, la sobrecarga es solo incurrida una vez por llamada de procedimiento). Aunque el CLR proporciona muchos servicios de tiempo de ejecución estándar, el código gestionado nunca se interpreta. Mientras tanto, el gestionador de memoria elimina las posibilidades de la memoria fragmentada y aumenta la memoria de la localidad de referencia para aumentar aún más el rendimiento.

Con referencia de nuevo a la arquitectura 105, la relación del CLR y la biblioteca de clases se muestra con respecto a las aplicaciones y al sistema en general (por ejemplo, el sistema 100) y muestra cómo el código gestionado opera dentro de una arquitectura más grande.

El CLR administra la memoria, la ejecución del hilo, la ejecución de código, verificación de seguridad de código, compilación y otros servicios del sistema. Estas características son intrínsecas al código gestionado que se ejecuta en el CLR.

En lo que respecta a la seguridad, componentes gestionados se pueden conceder diversos grados de confianza, dependiendo de una serie de factores que incluyen su origen (tales como Internet, una red de la empresa, o un ordenador local). Esto significa que un componente gestionado puede o no ser capaz de realizar operaciones de acceso a archivos, operaciones de acceso al registro u otras funciones confidenciales, incluso si se está utilizando en la misma aplicación activa.

El CLR se puede aplicar la seguridad de acceso al código. Por ejemplo, los usuarios pueden confiar en que un ejecutable incrustado en una página web puede reproducir una animación en la pantalla o cantar una canción, pero no puede acceder a sus datos personales, sistema de archivos o red. Las características de seguridad del CLR

pueden permitir que el software legítimo implementado en Internet sea excepcionalmente rico en funciones.

El CLR también puede ejecutar el código robustez mediante la aplicación de una estricta infraestructura de tipo y código de verificación de llamada del sistema de tipo común (CTS). El CTS garantiza que todo el código gestionado sea autodescriptivo. El código gestionado puede consumir otros tipos e instancias gestionadas, al tiempo que se aplica estrictamente la fidelidad de tipo y la seguridad de tipo.

El entorno gestionado de CLR tiene como objetivo eliminar muchos problemas de software común. Por ejemplo, el CLR puede manejar automáticamente el diseño de objetos y gestionar referencias a objetos, liberándolos cuando ya no se utilizan. Dicha gestión automática de la memoria resuelve los dos errores de aplicación más comunes, pérdidas de memoria y referencias de memoria no válidas. La interoperabilidad entre el código gestionado y el no gestionado puede permitir a los desarrolladores continuar utilizando componentes COM (modelo de objeto componente) y dlls (bibliotecas de vínculos dinámicos) necesarios.

El marco de trabajo CLR.NET™ se puede alojar en aplicaciones de alto rendimiento, del lado del servidor, como Microsoft SQL Server™ y Servicios de información de Internet (IIS). Esta infraestructura permite el uso de código gestionado para escribir lógica de negocios, al tiempo que utiliza servidores empresariales que admiten el alojamiento en tiempo de ejecución.

Las aplicaciones de servidor en el dominio gestionado se implementan a través de los ordenadores anfitriones de tiempo de ejecución. Las aplicaciones no gestionadas alojan el CLR, que permite que el código gestionado personalizado controle el comportamiento de un servidor. Tal modelo proporciona características del CLR y la biblioteca de clases al mismo tiempo que obtiene el rendimiento y la escalabilidad de un servidor ordenador anfitrión.

Como ya se ha mencionado, diversas técnicas ejemplares utilizar la virtualización para emular una ISA personalizada o modificada en la parte superior de un entorno CLR portátil, verificable, gestionado. Esto se muestra en la arquitectura 105 mediante un ejemplo de ISA personalizada/modificada y/o la VM 107 (arco punteado) que se encuentra sobre el límite del CLR y el espacio de aplicaciones gestionadas (teniendo en cuenta que varios ejemplos también discuten aplicaciones no gestionadas). Esto infiere que la ISA personalizada/modificada no interfiere con el funcionamiento del CLR y el entorno subyacente o el entorno de "ordenador anfitrión" (por ejemplo, OS/HW). Mientras que una ISA personalizada puede proporcionar el control del CLR o la "VM" en la medida necesaria para implementar diversas características de la ISA personalizada, varios ejemplos se basan en una VM personalizada, que puede ser una versión modificada de la VM subyacente (es decir, un CLR modificado). Por lo tanto, una arquitectura ejemplar puede incluir una única máquina virtual modificada o varias máquinas virtuales (por ejemplo, una máquina virtual personalizada en la parte superior de una máquina virtual específica). Para aumentar la seguridad, a un cierto coste de rendimiento, se puede apilar una pluralidad de máquinas virtuales de forma tal que todas menos la máquina virtual de nivel más bajo virtualice el conjunto de instrucciones de la máquina virtual directamente por debajo.

En la figura 1, una flecha señala un ejemplo donde un CLR convencional tiene una capa de virtualización con dos tipos de virtualización VM1 y VM2 en la parte superior de la misma y otro ejemplo donde un CLR convencional tiene dos capas de virtualización apiladas VM1 y VM3 en encima del mismo. En dichos ejemplos, el CLR subyacente podría ser un CLR personalizado o patentado con funciones de seguridad en el que las capas de virtualización adicionales mejoren aún más la seguridad. Un procedimiento ejemplar incluye múltiples VM apilados donde cada VM virtualiza el conjunto de instrucciones de la máquina directamente por debajo. Tomando nota de que la VM de nivel más bajo generalmente virtualiza un sistema operativo que controla el hardware, mientras que otras VM de nivel superior virtualizan otra VM. Como se indica en la figura 1, son posibles varias disposiciones (por ejemplo, dos VM en la parte superior de una VM, VM apiladas, etc.). Un enfoque de VM múltiple "personalizada" se puede ver como aprovechamiento de la virtualización para obtener más seguridad, con algún coste en el rendimiento.

Con referencia de nuevo a la analogía con la genética y el medio ambiente, la genética puede considerarse estática mientras que el entorno se puede considerar dinámico. De forma similar, diversos enfoques basados en la diversidad para la resistencia a la manipulación indebida pueden ser estáticos y/o dinámicos. En general, un enfoque estático diversifica las copias del código del programa, mientras que un enfoque dinámico diversifica las VM o la operación de la VM o la operación del programa en tiempo de ejecución. Por lo tanto, como se describe aquí, varias técnicas ejemplares incluyen una virtualización que funciona estáticamente, dinámicamente y/o tanto estáticamente (por ejemplo, para generar código de programa individualizado) como dinámicamente (por ejemplo, para variar el funcionamiento del programa en tiempo de ejecución).

Un ejemplo de procedimiento puede incluir proporcionar una arquitectura que incluye una primera capa de virtualización y proporcionar una segunda capa de virtualización en la parte superior de la primera capa de virtualización donde el segundo virtualización está configurado para recibir una copia diversificada de un programa y permitir la ejecución del programa utilizando la primera capa de virtualización. Tal procedimiento puede mejorar la seguridad del software mediante el uso de las técnicas de diversificación descritas en este documento.

Un ejemplo de procedimiento puede incluir la generación de copias individualizadas de un código de programa y proporcionar una máquina virtual para la ejecución de una copia individualizada del código de programa en el que la máquina virtual puede variar el funcionamiento del programa en tiempo de ejecución. Tal procedimiento puede

mejorar la seguridad del software mediante el uso de las técnicas de diversificación descritas en este documento.

Por razones de conveniencia, las ISA personalizadas y modificadas se denominan aquí como ISA personalizadas. Se puede usar una ISA personalizada para crear un conjunto de copias diferentes (o "versiones") de un programa con las siguientes propiedades: (i) Cada copia en el conjunto tiene un nivel razonable de defensa contra la manipulación y (ii) Es difícil reorientar un ataque existente contra una copia para trabajar contra otra copia. Las muchas opciones dan como resultado un gran espacio de programas semánticamente equivalentes que se pueden generar. Un enfoque puede considerar todo este espacio para permitir una mayor diversidad o, como alternativa, un enfoque puede considerar solo las partes de este espacio que se cree que son, o han demostrado ser, más resistentes a las alteraciones que otras partes.

Las propiedades resistentes a la manipulación incluyen: (i) Prevenir el análisis estático del programa; (ii) Prevenir el análisis dinámico del programa; (iii) Prevenir modificaciones locales; y (iv) Prevenir las modificaciones globales. Los primeros dos están estrechamente relacionados con el problema de la ofuscación, mientras que los dos últimos están más orientados a la resistencia a la manipulación. Sin embargo, la manipulación inteligente requiere al menos algún grado de comprensión del programa, que normalmente se obtiene al observar el binario estático, observar el ejecutable en ejecución o una combinación y/o repetición de las dos técnicas anteriores.

Existen varias ISA, por ejemplo, CISC, RISC y más recientemente Java™ bytecode y MSIL gestionado. Sin embargo, los dos últimos tienden a analizarse más fácilmente debido a una serie de razones. En primer lugar, los binarios generalmente no se ejecutan directamente en el hardware, sino que se deben emular o traducir a código nativo antes de la ejecución. Para habilitar esto, se deben conocer los límites entre el código y los datos, y no puede haber confusión entre los datos constantes y las direcciones reubicables. Esto, por supuesto, viene con la ventaja de la portabilidad. Además de la portabilidad, los principios de diseño incluyen soporte para administración de memoria tipada y verificabilidad. Para garantizar la verificabilidad, no se permite la aritmética de punteros, el flujo de control está restringido, etc. Para habilitar la gestión de memoria tipada, es necesario comunicar mucha información al entorno de ejecución sobre los tipos de objetos.

Todos estos principios de diseño han llevado a los binarios que son fáciles de analizar por el entorno de ejecución, pero son igualmente fáciles de analizar por un atacante. Esto ha llevado a la creación de descompiladores tanto para Java™ como para los binarios MSIL gestionados.

En general, existe una tendencia donde los principios de diseño de las ISA están cada vez más en conflicto con los principios de diseño que facilitarían la protección del software.

Como se describe aquí, una técnica ejemplar para contrarrestar esta tendencia añade una capa adicional de la virtualización (u opcionalmente múltiples capas adicionales de virtualización). Más específicamente, la virtualización se puede usar para emular una ISA personalizada sobre un entorno de tiempo de ejecución gestionado, verificable y portátil. Considera la siguiente construcción: (i) Escriba un emulador (por ejemplo, una máquina virtual personalizada) para el entorno que se ejecuta en la parte superior de un CLR; (ii) Tome la representación binaria de un binario y agréguela como datos al emulador y (iii) Haga que el procedimiento principal comience la emulación en el punto de entrada del ejecutable original. Dada esta construcción, el resultado es un binario gestionado, portátil y verificable. Además, está tan protegido como un binario nativo, ya que el atacante de un binario nativo podría fácilmente tomar el binario nativo y seguir la construcción anterior.

En general, para los binarios, experiencia e intuición indican que el binario promedio IA32 es mucho más complejo de entender y manipular al binario gestionado promedio. Algunas razones subyacentes incluyen (i) Longitud de instrucción variable; (ii) No hay una distinción clara entre código y datos; y (iii) No hay una distinción clara entre datos constantes y direcciones reubicables. Dado que las instrucciones (opcode + operandos) pueden tener una longitud variable (por ejemplo, 1-17 bytes), las instrucciones solo deben alinearse en bytes, y pueden combinarse con datos de relleno o datos regulares en el IA32, los desensambladores pueden salirse fácilmente de la sincronización. Como no existe una separación explícita entre el código y los datos, ambos pueden leerse y escribirse de forma transparente y usarse de manera intercambiable. Esto permite el código de auto modificación, una característica que es reconocida por ser difícil de analizar y por confundir a los atacantes.

La característica de que la representación binaria del código puede ser fácilmente leída se ha utilizado para permitir que los mecanismos de autocomprobación mientras que la ausencia de restricciones en el flujo de control ha permitido a técnicas tales como aplanamiento flujo de control y la superposición de instrucciones.

El hecho de que las direcciones pueden ser calculadas y que no pueden fácilmente ser distinguidas de los datos regulares, complica la manipulación de los binarios. Por ejemplo, un atacante solo puede realizar modificaciones locales, ya que no tiene suficiente información para reubicar todo el binario. Tales observaciones se pueden usar para generar una ISA personalizada ejemplar que demuestre que es difícil analizar la ISA. Tal ISA puede agregar seguridad.

Mientras que las técnicas de ejemplo pueden incluir código automodificable en una ISA y/o aplanamiento del flujo de control y/o superposición de instrucciones, ejemplos específicos discutidos en el presente documento incluyen disposiciones para longitud variable de las instrucciones y el uso de la representación binaria de partes de un

programa para aumento de la resistencia a la manipulación, que puede considerarse relacionada con algunos mecanismos de autocontrol.

El software a menudo sabe cosas que no desea compartir de manera descontrolada. Por ejemplo, las versiones de prueba pueden contener la funcionalidad para realizar una tarea determinada, pero una limitación de tiempo puede evitar su uso durante demasiado tiempo. En los contenedores digitales, el software a menudo se usa para proporcionar acceso controlado a los contenidos. Los agentes móviles pueden contener claves criptográficas que deben permanecer en secreto.

Para confundir a un ataque, los enfoques ejemplares incluyen (i) hacer el programa diferente para diferentes instalaciones; (ii) hacer que el programa sea diferente a lo largo del tiempo a través de actualizaciones personalizadas; y (iii) hacer que el programa sea diferente para cada ejecución a través de aleatorizaciones en el tiempo de ejecución.

La figura 2 muestra un módulo 200 de seguridad ejemplar implementado junto con un marco de trabajo que ejecuta archivos ejecutables portátiles en una máquina virtual. El módulo 200 de seguridad incluye un módulo 210 de proceso frontal y un módulo 250 de proceso posterior. El módulo 210 de proceso frontal lee un archivo 110 binario ejecutable que se dirige a una máquina virtual y produce un archivo 150 binario ejecutable personalizado que se dirige a una versión modificada de la VM objetivo original o una VM personalizada. En cualquier caso, el módulo 210 de proceso frontal puede usar el archivo 110 para determinar información sobre la VM objetivo original, tal como una descripción 115 de la VM. El módulo 250 de proceso posterior puede usar la descripción 215 de la VM para generar código, un dll, etc., para una VM 170 personalizada. Por ejemplo, una VM convencional se puede enviar como una biblioteca compartida o dll (por ejemplo, una biblioteca "nativa") y dichas técnicas se pueden usar para una VM personalizada, teniendo en cuenta que cuando una VM opera en la parte superior de una MV, detalles del la VM subyacente se puede contabilizar en la forma y/o características de una VM personalizada. Por razones de conveniencia, el término "personalizado" tal como se aplica a una VM puede incluir una VM modificada (por ejemplo, una versión modificada de la VM objetivo original del código).

En un ejemplo dirigido al marco de trabajo .NET™, el módulo de proceso de extremo frontal 210 lee un MSIL 110 gestionado binario, se ejecuta varias veces durante el código para determinar su ISA, y produce una descripción XML 215 de su VM objetivo. Una vez que se ha determinado la ISA, el módulo 210 puede reescribir el binario original en el lenguaje 150 de códigos de bytes personalizado.

La figura 3 muestra un procedimiento 300 ejemplar en el que el módulo 250 de proceso posterior lee la descripción 215 del XML y crea un dll gestionado para una VM 172 personalizada. La separación en un extremo posterior y frontal es algo artificial, pero permite un diseño más modular y puede facilitar la depuración. Por ejemplo, el módulo 250 de proceso posterior puede ser instruido para dar salida al código 174 de C # en lugar de compilar un dll directamente (véase, por ejemplo, 172). El código 174 puede luego inspeccionarse y depurarse por separado.

La figura 4 muestra un procedimiento 400 ejemplar donde se retienen varias partes del binario original. Más específicamente, el binario 110 ejecutable de la VM original incluye un envoltorio 111 alrededor de las funciones 115 (1) - (N) y el módulo 210 de proceso de extremo frontal reescribe cada función 115 (1) - (N) en un envoltorio 113 que llama a la VM, pasando los argumentos necesarios. En dicho ejemplo, todos los argumentos se pueden pasar en una matriz de objetos. Para las funciones de "instancia", el procedimiento 400 también incluye el puntero "este". Como el procedimiento 400 opera en todas las funciones para colocar cada una de ellas como una única estructura, el módulo 210 de proceso frontal también puede proporcionar una identificación del punto de entrada de cada función. Además, el módulo 210 de proceso frontal puede proporcionar características para garantizar que un objeto devuelto se convierte al tipo de retorno de la función original, cuando corresponda.

La figura 5 muestra una implementación particular del procedimiento 400 en más detalle. En este ejemplo, el módulo 210 de proceso frontal ha convertido funciones en módulos ficticios usando un envoltorio que invoca una VM. Más especialmente, la función "foo" está envuelta con la llamada "InvokeVM".

Como ya se ha mencionado, los datos o estructura de datos puede proporcionar un medio para la diversificación. La figura 6 muestra un procedimiento 450 ejemplar en el que el módulo 210 de proceso de entrada frontal ejemplar recibe un binario 110 con datos 117 (1) - (N) en una estructura de datos original y emite un binario 150 personalizado con datos 117 (1) - (N) en una estructura 156 de datos modificada o personalizada. Aunque el procedimiento 450 también muestra el ajuste de las funciones 115 (1) - (N), un procedimiento ejemplar puede generar un binario personalizado diversificando el código, diversificando los datos, y/o diversificando el código y los datos. La frase "diversificación de datos" puede incluir la diversificación de datos y la diversificación basada en la estructura de datos.

Los procedimientos ejemplares que incluyen la reescritura del programa original por función solo tienen la ventaja de que cosas como la recolección de basura no se convierten en un problema, ya que las estructuras de datos todavía se tratan como en el programa original. Cuando las técnicas se aplican para ofuscar, diversificar y hacer que los datos sean más resistentes a las alteraciones, las modificaciones pueden proporcionar tareas como la recolección de basura.

Las figuras 1-6, descritas anteriormente, ilustran cómo se puede usar la virtualización para mejorar la seguridad. Más específicamente, un módulo de proceso de frontal puede usarse para generar un código binario personalizado y un proceso de posterior puede usarse para generar una VM personalizada para ejecutar el código binario personalizado. Las figuras 7-14, que se describen a continuación, ilustran cómo las características específicas de una ISA y/o una VM pueden usarse para mejorar la seguridad mediante la diversificación.

Con respecto a código gestionado y varias técnicas ejemplares presentadas en este documento, la elección entre MSIL gestionada para el código de bytes CLR y Java™ para el entorno de ejecución de Java (JRE) es algo arbitrario como diversas técnicas ejemplares se pueden transferir desde el .NET™ al dominio de Java™. Además, las técnicas para ofuscar el bytecode de Java™ se pueden aplicar a binarios gestionados de MSIL. La discusión que sigue se enfoca principalmente en técnicas ejemplares que se derivan de una capa de virtualización "añadida" o personalizada. La diversificación automatizada de copias distribuidas, por ejemplo, a través de la distribución por Internet, está ganando aceptación continuamente. Por lo tanto, la introducción por encima de cualquiera de las técnicas ejemplares es cada vez más viable desde el punto de vista económico.

Diversas técnicas ejemplares pueden introducir protección automáticamente en un punto donde ya no es necesaria la interacción humana. En teoría, es posible generar un número inmanejable de diversos programas semánticamente equivalentes: Considere un programa con 300 instrucciones y elija para cada instrucción si debe o no anteponerlo a un no-op. Esto produce 2^{300} programas semánticamente equivalentes diferentes y 2^{300} es mayor que 10^{87}, el número estimado de partículas en el universo.

Sin embargo, la singularidad no es necesariamente suficiente como los programas resultantes deben ser lo suficientemente diverso como para complicar el mapeo de la información obtenida a partir de un caso a otro caso. Además, los programas resultantes deberían ser preferiblemente no triviales para romperse. Si bien no es razonable esperar que el codominio de varias técnicas de diversificación ejemplares incluya cada programa semánticamente equivalente, se puede establecer un objetivo para maximizar el codominio de un diversificador ya que cuanto mayor sea el espacio, más fácil será obtener programas internamente diferentes.

Un enfoque ejemplar se inicia desde una implementación existente de la semántica, más que de la propia semántica. A través de una serie de transformaciones parametrizables se obtienen diferentes versiones. En varios ejemplos que siguen, se identifican varios componentes de una ISA que se pueden individualizar de forma independiente.

La figura 7 muestra características 500 de marco de trabajo ejemplares agrupadas como componentes 505 binarios y un componente 560 de implementación de VM. Los componentes 505 binarios incluyen la semántica 510 de instrucción, la codificación 520 de instrucción, la codificación 530 del operando, el ciclo 540 de búsqueda y el contador de programa (PC) y la representación 550 de programa. Estos componentes se pueden personalizar de forma ortogonal, siempre que se respeten las interfaces. Los componentes 505 son suficientes para generar un binario en un lenguaje de códigos de bytes personalizado; es decir, para determinar una ISA personalizada ejemplar. Además, puede producirse una diversificación diversificando la VM objetivo o una VM personalizada (véase, por ejemplo, el componente 560 de implementación de VM).

La figura 8 muestra un modelo de ejecución e interfaces 800. El modelo 800 incluye el código 802, la estructura 804 de datos, un marco 810 de trabajo de procedimientos y un binario 150 personalizado. En la figura 8, las flechas representan dependencias de interfaz y los asteriscos representan algunas partes diversificables. Un enfoque ejemplar que utiliza dicho modelo permite un diseño modular y un desarrollo independiente.

De acuerdo con el modelo 800 y el marco 500, la diversificación puede incluir (i) la aleatorización de la semántica de instrucciones mediante la construcción de instrucciones a través de la combinación de instrucciones más pequeñas, (ii) la elección de la semántica de instrucciones para permitir una mayor superposición semántica, (iii) apartarse del representación de código lineal tradicional para representar el código como una estructura de datos tal como un árbol binario auto-adaptable (bisel), (iv) asignar longitudes variables a códigos de operación y operandos para complicar el desensamblaje y hacer las modificaciones locales más difíciles, (v) limitar el conjunto de instrucciones para dar al atacante menos opciones para analizar y modificar el código, (vi) hacer variable la asignación entre patrones de bits, códigos de operación y operandos.

El código 802 en la figura 8 da una visión general de alto nivel de un motor de ejecución basado en un ciclo de búsqueda - ejecutar. Las principales estructuras internas de datos de la VM se muestran como el procedimiento del marco de trabajo 810. Como ya se mencionó, las flechas indican dependencia de la interfaz. Por ejemplo, DecodeOpcode espera poder recuperar una cantidad de bits.

La figura 9 muestra la semántica 510 de instrucciones de la figura 5 y algunas características que pueden usarse para la diversificación. El concepto de microoperaciones puede permitir la diversificación de la semántica de la instrucción. Por ejemplo, una instrucción en un lenguaje de códigos de bytes personalizado (por ejemplo, según una ISA personalizada) puede ser cualquier secuencia de un conjunto predeterminado de microoperaciones. Para el MSIL, el conjunto de microoperaciones actualmente incluye instrucciones de MSIL verificables y una serie de instrucciones adicionales para: (i) Comunicar la metainformación requerida para la ejecución correcta y (ii) Habilitar

funciones adicionales, como cambiar la semántica (se describe con más detalle más adelante).

Esto puede ser comparado con el concepto de micro-operaciones (μops) en la microarquitectura P6. Cada instrucción IA32 se traduce en una serie de operaciones que luego son ejecutadas por la tubería. Esto también podría ser comparado con super-operadores. Los súper operadores son operaciones de máquinas virtuales que se sintetizan automáticamente a partir de combinaciones de operaciones más pequeñas para evitar gastos por operación costosos y reducir el tamaño del ejecutable.

Un ejemplo de procedimiento puede incluir proporcionar módulos ficticios para emular cada una de las micro-operaciones y estos pueden ser concatenados para emular instrucciones más expresivas en un lenguaje de código de bytes de encargo (por ejemplo, una ISA personalizada), tomando nota de que muchas funciones de emulación pueden depender en gran medida de reflexión.

Consideremos un ejemplo utilizando las siguientes instrucciones del MSIL (además, el argumento de la carga y constantede carga) y sus módulos ficticios de emulación (que han sido simplificados):

```
Idarg Int32:
    EvaluationStack.Push (
        ArgsIn.Peek (getArgSpec (insNr));
Idc Int32:
    EvaluationStack.Push (
        getInt32Spec (insNr));
añadir:
    EvaluationStack.Push (
        (Int32) EvaluationStack.Pop () +
        (Int32) EvaluationStack.Pop ());
```

Supongamos que, durante la fase de selección de la instrucción, queremos crear una instrucción de código de bytes personalizada con la semántica siguiente:

CustomIns ni: carga el enésimo argumento, carga la constante i y agrega estos dos valores.

Esta instrucción se asigna luego a una declaración de "caso" (por ejemplo, 1) en una declaración grande de "cambio". La declaración de caso es la concatenación de los diferentes módulos ficticios de emulación de las microoperaciones:

```
switch (insNr) {
...
case 1:
    // Se rompe la concatenación de módulos ficticios;
...
}
```

Con respecto a las manipulaciones de resistencia, falta de conocimiento de la semántica de una instrucción complicará comprensión programa, en lugar de tener un manual en el que se especifica la semántica. Para ir un nivel más allá, una ISA resistente a alteraciones personalizada puede elegir la semántica de la instrucción como adherente a algunos principios de diseño.

Haciendo referencia nuevamente a la figura 9, se puede usar la ejecución 512 condicional, opcionalmente junto con los registros 513 de predicados para aumentar la resistencia a la manipulación indebida. La ejecución condicional puede promover aún más la fusión de piezas de código ligeramente diferentes. En presencia de ejecución condicional, las instrucciones pueden ser predicadas por registros de predicados. Si el registro de predicados se establece en falso, la instrucción se interpreta como no operativa (sin operación); de lo contrario, se emula la instrucción. Usando este enfoque, los registros se establecen en o a lo largo de diferentes rutas de ejecución para poder ejecutar fragmentos de código ligeramente diferentes.

Un ejemplo de procedimiento puede incluir proporcionar secuencias de código a, b, c y a, d, c en dos contextos diferentes en un programa original y luego "fusionar" el código a un, [p1]b, [p2]d, c donde p1 se establece en

"verdadero" y p2 se establece en "falso" para ejecutar el código en el primer contexto y viceversa para ejecutar el código en el segundo contexto. Como resultado de la configuración de uno o más registros de predicados de forma diferente, se pueden ejecutar diferentes partes de código (por ejemplo, a, b, no-op, c y a, no-op, d, c).

Un conjunto 514 de instrucción limitado se puede usar para aumentar la resistencia de manipulación indebida. Por ejemplo, una ISA personalizada puede carecer de las no-op 515, limitar los operandos 516 representables y/o eliminar al menos algunas ramas 517 condicionales. Otro enfoque puede adaptar una VM personalizada a un programa(s) específico; por lo tanto, un enfoque ejemplar puede garantizar que la máquina virtual solo pueda emular las operaciones requeridas por ese programa.

En referencia de nuevo a una ISA personalizada sin no-ops 515, este enfoque de las cuentas de una técnica común de ataque que elimina la funcionalidad "no deseada" (por ejemplo, una comprobación de la licencia o la disminución de la salud de un personaje del juego heridos) al sobrescribir dicha funcionalidad con no-ops. En muchos casos, hay pocas razones para incluir una instrucción no operativa en una ISA personalizada y no tener esta instrucción complicará el intento de un atacante de rellenar el código no deseado.

Con respecto a la limitación de operandos 516 representables, las estadísticas muestran que, por ejemplo, de los literales de número entero de alrededor de 600 programas de Java™ (1,4 millones de líneas en total) 80 % tiene entre 0-99, 95 % tienen entre 0 y 999 y el 92 % son potencias de dos o potencias de dos más o menos 1. Por lo tanto, una ISA personalizada ejemplar puede limitar el número de operandos representables, limitando de nuevo la libertad de un ataque.

Otro enfoque ejemplar puede modificar o restringir el uso de al menos algunas ramas 517 condicionales. Por ejemplo, generalmente, hay dos versiones para cada condición: "Ramificar si la condición está establecida y ramificar si la condición no está establecida". Como el uso de dos ramas es redundante, una ISA personalizada podría incluir un código de reescritura de modo que solo se use una versión y su contraparte no esté incluida en la ISA. Esta técnica ejemplar puede ser útil, por ejemplo, cuando una verificación de licencia se ramifica de manera condicional dependiendo de la validez del número de serie: Evitará que el atacante simplemente voltee la condición de rama.

La figura 10 muestra el bloque 520 de codificación de instrucciones de la figura 5 con diversos aspectos de la codificación de instrucciones que pueden usarse para la diversificación. Más específicamente, los aspectos incluyen tamaños 522 variables de instrucción, codificación unaria para solapamiento 524 físico, semántica 526 no local y reorganización de la estructura 529 de decodificación.

Una vez se ha determinado la semántica de instrucciones, existe una necesidad de determinar una codificación de código de operación para esas instrucciones. El tamaño de todos los códigos de operación para las arquitecturas tradicionales suele ser constante o solo ligeramente variable. Por ejemplo, los códigos de operación de MSIL son típicamente de un byte, con un valor de escape (0xfe) para habilitar los códigos de operación de dos bytes para instrucciones menos frecuentes. La variabilidad limitada facilita la búsqueda rápida a través de la interpretación de la tabla. Pero, de manera más general, cualquier código de prefijo (ninguna palabra de código es un prefijo de ninguna otra palabra de código) permite una interpretación inequívoca.

En su forma más general, los códigos de operación de decodificación para la semántica se pueden hacer a través de un recorrido de árbol binario. La decodificación normalmente comienza en el nodo raíz; cuando se lee un bit 0, se produce un movimiento hacia el nodo secundario izquierdo; cuando se lee un bit, se produce un movimiento al nodo secundario derecho. Cuando se alcanza un nodo hoja, el código de operación se ha decodificado con éxito (por ejemplo, considere un nodo hoja que contiene una referencia a la declaración de caso que emula la semántica de la instrucción).

Si una ISA personalizada permite tamaños de código de operación arbitrarios, sin opcodes ilegales, el número de posibles codificaciones para n instrucciones está dada por la siguiente ecuación:

$$\frac{\binom{2(n-1)}{n-1}}{n} n! \qquad (3)$$

En la ecuación 3, la fracción representa el número de árboles binarios planar con n hojas (número catalán), mientras que el factorial representa la asignación de códigos de operación a las hojas. Si se eligieran tamaños de opcode fijos con la codificación más corta posible, es decir, el bit log2(n), podría introducir códigos de operación ilegales. En este caso, el número de codificaciones posibles está dado por la siguiente representación (Ecuación 4):

$$\binom{2^{[\log_2(n)]}}{n} \qquad (4)$$

Se presentarían muchas más posibilidades si la ISA personalizada permitiera códigos de operación ilegales por

otras razones que no fueran los tamaños mínimos de código de operación fijos. Sin embargo, esto puede aumentar el tamaño de un binario escrito en la ISA personalizada sin ofrecer ventajas.

En varios ejemplos presentados en este documento, dirigido al marco de trabajo .NET™ (por ejemplo, MSIL), se pueden soportar los siguientes modos: (i) códigos de operación de longitud fija con búsqueda de tabla; (ii) codificación de tabla de niveles múltiples para permitir tamaños de instrucción ligeramente variables (los códigos de escape se usan para códigos de operación más largos) y (iii) códigos de operación de longitud arbitraria con recorrido de árbol binario para la decodificación. Dichos modos se pueden aplicar a otros marcos de trabajo según corresponda.

Con respecto a alterar la resistencia, de nuevo, no saber la asignación de secuencias de bits a la semántica introduce una curva de aprendizaje para un atacante, por ejemplo, en comparación con tener esa información en un manual. Existen varios enfoques adicionales para elegir una asignación de tal manera que permita las propiedades de resistencia a la manipulación.

Como ya se ha mencionado, las instrucciones que codifican el bloque 520 de la figura 10 incluye un enfoque 522 de tamaño instrucción variable. Por ejemplo, la ISA personalizada puede introducir aún más varianzas en la longitud de los códigos de operación que los permitidos en un binario CISC. Los tamaños de instrucción variable también se pueden usar para hacer que las modificaciones locales sean más complicadas. En general, una instrucción más grande no puede simplemente reemplazar una instrucción más pequeña, porque sobrescribiría la siguiente instrucción. Una ISA personalizada también puede garantizar que las instrucciones más pequeñas de transferencia sin control no puedan reemplazar las instrucciones más grandes. Por ejemplo, esto se puede lograr asegurándose de que dichas instrucciones no se puedan rellenar para permitir que el control fluya a la siguiente instrucción.

Considere un código o ISA con 64 instrucciones donde cada una de las instrucciones pueden ser asignadas un tamaño único en un rango de bits (por ejemplo, entre alrededor de 64 bits y de 127 bits). Claramente, las instrucciones más grandes no caben en el espacio de las instrucciones más pequeñas. Además, las instrucciones más pequeñas caben en el espacio de las instrucciones más grandes. Sin embargo, cuando el control cae al siguiente bit, no hay instrucciones disponibles para rellenar los bits restantes con no-ops para asegurar que el control fluya a la siguiente instrucción. Por lo tanto, bajo este esquema, es útil hacer que las instrucciones de transferencia de control sean las más largas, para evitar que un atacante se escape a otra ubicación donde pueda hacer lo que quiera.

El bloque 520 de instrucción de codificación también incluye un enfoque 524 unario de codificación para lograr, por ejemplo, la superposición física. Un enfoque de codificación unaria puede enredar un programa al aumentar o maximizar la superposición física. Por ejemplo, tal enfoque puede saltar al medio de otra instrucción y comenzar a decodificar otra instrucción. Este enfoque se puede facilitar eligiendo una buena codificación. Por ejemplo, la codificación unaria se puede usar para codificar los códigos de operación (0, 01, 001, ..., 0^{63} 1). En este ejemplo, hay una buena posibilidad de que uno encuentre otra instrucción al saltar un bit después del comienzo de una instrucción:

```
1:     Add
01:    mul
001:   mub
0001:  div
```

Arriba, a cuatro instrucciones se les ha asignado un código de operación usando codificación unaria. En este ejemplo, si la decodificación comienza en el segundo bit de la instrucción de división (div), se revela la instrucción de resta (sub). Del mismo modo, mirando la última parte de la instrucción de dividir, restar y multiplicar (mul) se revela instrucción adicional.

Otro enfoque para una ISA personalizada relacionado con la codificación de instrucciones usa semántica 526 no local. Tener un lenguaje de bytecode único para cada copia distribuida erige una barrera significativa contra los atacantes.

En general, para una ISA, no hay documentación disponible en: (i) La asignación de patrones de bits a instrucciones; (ii) La semántica de las instrucciones; (iii) La asignación de patrones de bits a operandos; (iv) La representación de estructuras de datos; etc. Sin embargo, tales asignaciones o representaciones eventualmente pueden ser aprendidas por un atacante a través de una inspección estática o dinámica. Para confundir un ataque, una ISA personalizada puede usar semántica 524 no local para garantizar que un patrón de bits tenga un significado diferente a lo largo de diferentes rutas de ejecución.

Un programa binario es simplemente una secuencia de "1"s y "0"s, que tiene un significado dado por un procesador. El significado entre los patrones de bits y la interpretación generalmente lo fija la ISA. En las arquitecturas tradicionales, si el código de operación de una determinada instrucción se representa mediante un patrón de bits

dado, este patrón es constante para cada binario, dondequiera que ocurra. Una ISA personalizada puede hacer que cualquier patrón de bits particular sea variable, teniendo en cuenta que no todos los patrones de bits de instrucciones deben ser variables para erigir una barrera significativa contra el ataque. Por lo tanto, el bloque de semántica no local 526 incluye un enfoque de bloque de patrón de bits variable 527, por ejemplo, para un código de operación para una instrucción.

En una ISA personalizada, un patrón de bits solo puede tener significado dependiendo del código ejecutado previamente. Para hacer que la interpretación dependa del código ejecutado previamente, dependiendo de la entrada (totalmente especificada), una ISA personalizada puede permitir llegar a un punto del programa a lo largo de diferentes rutas de ejecución. Sin embargo, tal ISA personalizada puede querer controlar la interpretación de bits en un punto de programa dado. Para acomodar esta variabilidad, una ISA personalizada puede hacer que los cambios de interpretación sean explícitos en lugar de implícitos como un efecto secundario de algún otro evento. Por lo tanto, el bloque de semántica no local 526 incluye un enfoque de bloque de patrón de bits 528, por ejemplo, para asignar significado en base a la ejecución de código anterior. Además, la semántica no local bloquea enlaces 526 a cambios de interpretación explícitos en la ISA personalizada, por ejemplo, para llegar a un punto de programa a lo largo de diferentes rutas de ejecución.

Un ejemplo de procedimiento incluye la generación de diversas copias de un programa utilizando la codificación de instrucciones para reorganizar una estructura de decodificación para permitir de ese modo para llegar a un punto en el programa a lo largo de dos o más rutas de ejecución donde un significado asignado de un patrón de bits en el punto depende en la ruta de ejecución al punto. Por ejemplo, dicho procedimiento puede asignar significado en función de la ejecución previa del código, que puede diferir para diferentes rutas de ejecución.

Una ISA personalizada puede tener como objetivo no limitar la complejidad con respecto a obtener un entorno de ejecución en un estado de interpretación específico. En otras palabras, tal enfoque puede garantizar que, si se permite llegar a un punto de programa desde diferentes rutas de ejecución en diferentes estados de interpretación, puede ser relativamente fácil migrar a un solo estado de interpretación objetivo, sin importar cuáles sean los diferentes estados de interpretación.

Un enfoque particular implica reordenar la estructura 529. Por ejemplo, cambiar la interpretación puede equivaler a nada más que reorganizar un árbol de decodificación. Teniendo en cuenta las observaciones anteriores, una ISA personalizada solo puede permitir una forma limitada de diversificación. Con este fin, una ISA personalizada puede tener un nivel elegido alrededor del cual los subárboles (u otras subestructuras) pueden moverse. Tal elección es una compensación entre cuántas interpretaciones diferentes son posibles y qué tan fácil es ir a una interpretación fija desde un conjunto de estados de interpretación posiblemente diferentes.

En un ejemplo, considere elegir el tercer nivel de una estructura de árbol. Suponiendo que el código de operación más corto es de 3 bits, esto permite 8! estados de interpretación, mientras que cualquier estado de interpretación es alcanzable en un máximo de 8 microoperaciones. Tal enfoque se puede aplicar a un conjunto de microoperaciones de MSIL. Por ejemplo, considere las siguientes microoperaciones:

Swap (UInt3 position1, UInt3 position2), que intercambia los nodos en posición position1 y position2 y

Set (etiqueta UInt3, posición UInt3), que intercambia el nodo con la etiqueta (donde sea que esté) y el nodo en la posición de posición.

En el caso de la interpretación de tablas, esto se implementa como una interpretación de tabla de dos niveles. El primer nivel puede referirse a otras tablas que pueden intercambiarse.

En el ejemplo anterior, las microoperaciones corresponden en gran medida a las instrucciones de MSIL y los tipos de operandos corresponden en gran medida a los tipos de operandos de MSIL. Los módulos ficticios de emulación de microoperaciones que usan operandos utilizan llamadas a funciones para garantizar que la codificación del código de operación se pueda diversificar ortogonalmente. Tales rellamadas además pasan un argumento "insNr" que identifica una instrucción VM personalizada a partir de la cual se llamó (ver, por ejemplo, ejemplo de semántica 510 de instrucción). Esto permite codificar los operandos de forma diferente para diferentes instrucciones de VM personalizadas. Tenga en cuenta que debido a la concatenación de módulos ficticios, un número arbitrario de operandos puede seguir el código de operación. Por lo tanto, el enfoque para la codificación 530 de operandos puede incluir tales técnicas. Por lo tanto, se pueden realizar enfoques similares para diversificar la codificación de código de operación como para la codificación de instrucciones.

La diversificación del ciclo de búsqueda puede considerarse una forma de diversificación "artificial". La figura 11 muestra el bloque 540 de ciclo de búsqueda que incluye diversos enfoques que usan "filtros" 542. Un ciclo de búsqueda básico "no personalizado" simplemente obtiene una cantidad de bits de un código binario de byte personalizado, según el contador de programa (PC) actual. Sin embargo, el uso de uno o más filtros 542 permite un ciclo de búsqueda personalizado que mejora la resistencia a la manipulación. Dichos filtros o filtros pueden transformar los bits reales en el binario en los bits que interpretará la VM.

Los filtros 542 del ciclo de búsqueda pueden añadir complejidad al combinar uno o más bits solicitados con otra

información. Por ejemplo, los bits solicitados reales pueden combinarse con otras partes de un programa 543. De esta manera, un programa se vuelve más interdependiente ya que cambiar una parte del programa también puede afectar a otras partes. Otros enfoques de filtro incluyen un filtro que combina uno o más bits con un valor 544 aleatorio (por ejemplo, derivado de una clave secreta) y un filtro que combina uno o más bits con el contador 545 de programa (PC) para complicar las técnicas de coincidencia de patrones.

La representación más tradicional de código es como una secuencia lineal de bytes. En tal enfoque, un contador de programa (PC) simplemente apunta al siguiente byte para ejecutar, y las transferencias de control típicamente especifican el byte para continuar la ejecución como un desplazamiento relativo o una dirección absoluta. Esto se puede ver esencialmente como una estructura que representa el código como una matriz de bytes.

La figura 12 muestra el contador de programa y el bloque 550 de representación de programa junto con diversos enfoques estructurales, que incluyen la matriz 553, el árbol 554, la lista 555 vinculada y la tabla 556 de verificación. Una ISA personalizada puede representar el código como un árbol extendido tal como los árboles 1320 y 1330 extendidos de la figura 13. Si bien el código se puede representar como un árbol de distribución, un enfoque ejemplar para una ISA personalizada puede, alternativamente o adicionalmente, representar datos en un árbol extendido u otra estructura seleccionada para mejorar la seguridad. En general, tales enfoques pueden proporcionar la diversificación más fácilmente que una representación lineal tradicional (véase, por ejemplo, el enfoque 1310 lineal de la figura 13).

Los árboles extendidos tienen una serie de ventajas: Se autoequilibran, lo que permitirá la reubicación automática del código. Además, son casi óptimos en términos de coste amortizado para secuencias arbitrarias. Finalmente, los nodos accedidos recientemente tienden a estar cerca de la raíz del árbol, lo que permite el aprovechamiento parcial de la localidad espacial y temporal presente en la mayoría de los ejecutables.

Debido a la propiedad de autoequilibrado, un fragmento de código podría estar en muchas ubicaciones diferentes en la memoria, dependiendo de la ruta de ejecución que condujo a un cierto fragmento de código. Los fragmentos de código se pueden mover, siempre que haya una forma de referirse a ellos para las transferencias de flujo de control, y para que puedan recuperarse cuando se les transfiere el control. Un enfoque de estructura ejemplar utiliza claves de nodos en el árbol de distribución donde las transferencias de control especifican la clave del nodo al que se debe transferir el control. En dicho ejemplo, se requiere que los objetivos del flujo de control sean nodos (es decir, que no puedan saltar fácilmente al centro del código contenido dentro de un nodo). En la práctica, esto significa que la ejecución inicia un nuevo nodo para cada bloque básico. Las rutas de paso pueden manejarse haciendo que todo el flujo de control sea explícito. En dicho ejemplo, todos los objetivos de flujo de control se pueden especificar como las claves del nodo que contiene el código objetivo. Además, el tamaño del código en un nodo puede ser constante. Además, si un nodo es demasiado pequeño para contener un bloque básico completo, puede desbordarse a otro nodo y continuar la ejecución allí.

La figura 13 ilustra un enfoque ejemplar que usa árboles extendidos 1320, 1330 para un enfoque lineal dado 1310 para una función factorial "Fac". Cuando, por ejemplo, se llama por primera vez a la función "Fac", el nodo con la clave 1 será referenciado y filtrado a la raíz, como se muestra en la parte (2). Otra cosa que vale la pena destacar en este ejemplo es que las llamadas ya no necesitan especificar la firma de la función, ya que este código no estará sujeto a verificación.

Si tal técnica se implementa ingenuamente, solo se moverán los punteros, y el código real permanecerá en el mismo lugar en el montón. Para complicar aún más esto, puede ocurrir un intercambio explícito de los contenidos reales (de tipos primitivos) de los nodos, o alternativamente, puede ocurrir una asignación de una nueva memoria intermedia de código junto con la copia de la memoria intermedia de código al espacio asignado, posiblemente con re-encriptación y/o con diferente relleno de basura.

Con referencia de nuevo a las características de estructura 500 de la figura 7, se pueden aplicar enfoques ejemplares a la implementación 560 de la VM. Algunos enfoques se muestran en la figura 12. Para una implementación interna dada, una pila de evaluación no se determina sobre la base de la ISA (por ejemplo, considere los componentes 505). En un ejemplo de este tipo, los módulos ficticios de emulación para microoperaciones pueden basarse únicamente en una interfaz que admite varias operaciones como "pop" y "push". Un enfoque ejemplar para una implementación interna de una estructura 562 de datos de pila introduce diversidad independiente a través, por ejemplo, de una matriz, una lista vinculada, etc. Un enfoque ejemplar puede proporcionar opcionalmente una cantidad de implementaciones diferentes de tales interfaces.

Otro enfoque tiene como objetivo diversificar la generación 564 de VM. Por ejemplo, una vez que los parámetros para las formas de diversificación especificadas anteriormente se especifican completamente, un proceso final ejemplar puede combinar fragmentos de código de varias ubicaciones junto con algún código autogenerado para ensamblar una representación de C# gestionada para la implementación de la máquina virtual personalizada. Alternativamente, un proceso final ejemplar puede generar directamente un dll.

Otro enfoque ejemplar implica la diversificación de dll 566, por ejemplo, el uso de versiones aleatorizable de transformaciones de código existentes de diversos dominios, tales como la optimización de software, ofuscación software, la diversificación de software, etc. (enfoques para no virtualización de base)

Si bien diversas técnicas ejemplares descritas en este documento generalmente introducen cierta sobrecarga, donde la gestión de derechos digitales, la información sensible (por ejemplo, gobierno, propietario, etc.), las licencias, etc., están involucradas, entonces tal sobrecarga puede ser tolerada, dada la mejorada seguridad introducida a través de la diversificación. En tales casos, las técnicas de diversificación se pueden aplicar a las áreas normalmente identificadas y no se pueden aplicar a las características de tiempo de ejecución que requieren algún grado de ejecución simultánea o de "tiempo real". Por ejemplo, la diversificación puede aplicarse al código asociado con la gestión de derechos digitales y no al código asociado que requiere algún tipo de autorización digital antes de la ejecución.

La virtualización abre una amplia gama de posibilidades tanto para la diversidad como para la resistencia a la manipulación. Controlar un entorno de ejecución proporciona una influencia significativa para complicar la tarea de un atacante. Si bien varios ejemplos se refieren a un marco de trabajo particular para la protección de software basado en el concepto de virtualización, también se han identificado varios enfoques en los que se pueden introducir características de diversidad y/o resistentes a la manipulación de una manera en gran parte independiente. Se puede usar desarrollo modular y/o implementación.

La figura 15 ilustra un dispositivo 1500 informático ejemplar que se puede usar para implementar diversos componentes ejemplares y para formar un sistema ejemplar. Por ejemplo, los servidores y clientes del sistema de la figura 1 pueden incluir diversas características del dispositivo 1500.

En una configuración muy básica, el dispositivo 1500 informático típicamente incluye al menos una unidad 1502 de procesamiento y la memoria 1504 del sistema. Dependiendo de la configuración exacta y del tipo de dispositivo informático, la memoria 1504 del sistema puede ser volátil (como RAM), no volátil (como ROM, memoria flash, etc.) o alguna combinación de ambas. La memoria 1504 del sistema incluye típicamente un sistema 1505 operativo, uno o más módulos 806 de programa, y puede incluir datos 1507 del programa. El sistema 1506 operativo incluye un marco de trabajo 1520 basado en componentes que admite componentes (incluyendo propiedades y eventos), objetos, herencia, polimorfismo, reflexión y proporciona una interfaz de programación de aplicaciones (API) basada en componentes y orientada a objetos, como la del marco de trabajo .NET™ fabricado por Microsoft Corporation, Redmond, WA. El sistema 1505 operativo también incluye un marco de trabajo 1600 de ejemplo, tal como, pero no limitado a, un marco de trabajo ejemplar con una personalizada ISA y/o una VM personalizada. Además, el dispositivo 1500 informático puede incluir un módulo de software para generar una ISA personalizada y/o una VM personalizada. Además, el dispositivo 1500 informático puede incluir un módulo de software para probar una ISA personalizada y/o una VM personalizada. El dispositivo 1500 informático puede incluir un módulo de software para generar un código personalizado y/o una VM personalizada para, al menos en parte, ejecutar un código personalizado. El dispositivo 1500 tiene una configuración muy básica demarcada por una línea 1508 discontinua. Nuevamente, un terminal puede tener menos componentes, pero interactuará con un dispositivo informático que pueda tener una configuración básica.

El dispositivo 1500 informático puede tener características o funcionalidades adicionales. Por ejemplo, el dispositivo 1500 informático también puede incluir dispositivos de almacenamiento de datos adicionales (extraíbles y/o no extraíbles) tales como, por ejemplo, discos magnéticos, discos ópticos o cinta. Tal almacenamiento adicional se ilustra en la figura 15 mediante el almacenamiento 1509 extraíble y el almacenamiento 1510 no extraíble. Los medios de almacenamiento informático pueden incluir medios volátiles y no volátiles, extraíbles y no extraíbles implementados en cualquier procedimiento o tecnología para el almacenamiento de información, tales como instrucciones legibles por ordenador, estructuras de datos, módulos de programa u otros datos. La memoria1504 del sistema, el almacenamiento 1509 extraíble y el almacenamiento 1510 no extraíble son todos ejemplos de medios de almacenamiento informático. Los medios de almacenamiento informático incluyen, entre otros, RAM, ROM, EEPROM, memoria flash u otra tecnología de memoria, CD-ROM, discos versátiles digitales (DVD) u otro tipo de almacenamiento óptico, casetes magnéticos, cinta magnética, almacenamiento en disco magnético u otro dispositivo de almacenamiento magnético, o cualquier otro medio que pueda usarse para almacenar la información deseada y a la que se pueda acceder mediante el dispositivo 1500 informático. Cualquier medio de almacenamiento informático de este tipo puede ser parte del dispositivo 1500. El dispositivo 1500 informático también puede tener dispositivo(s) 1512 de entrada, como teclado, ratón, lápiz, dispositivo de entrada de voz, dispositivo de entrada táctil, etc. Los dispositivos 1514 de salida, como una pantalla, altavoces, impresora, etc., también pueden ser incluido. Estos dispositivos son bien conocidos en la técnica y no necesitan discutirse extensamente aquí.

El dispositivo 1500 informático también puede contener conexiones 1516 de comunicación que permiten que el dispositivo se comunique con otros dispositivos 1518 informáticos, tal como a través de una red (por ejemplo, considere la red web o internet 103 mencionada anteriormente). Las conexiones 1516 de comunicación son un ejemplo de medios de comunicación. Los medios de comunicación pueden incorporarse típicamente mediante instrucciones legibles por ordenador, estructuras de datos, módulos de programa u otros datos en una señal de datos modulada, tal como una onda portadora u otro mecanismo de transporte, e incluye cualquier medio de entrega de información. El término "señal de datos modulada" significa una señal que tiene una o más de sus características establecidas o cambiadas de tal manera que codifican información en la señal. A modo de ejemplo, y no de limitación, los medios de comunicación incluyen medios cableados tales como una red cableada o conexión de cableado directo, y medios inalámbricos tales como medios acústicos, de RF, infrarrojos y otros medios inalámbricos. El término medio legible por ordenador como se usa en el presente documento incluye tanto medios

de almacenamiento como medios de comunicación.

Aunque el tema se ha descrito en un lenguaje específico para las características estructurales y/o los actos metodológicos, se debe entender que el tema definido en las reivindicaciones adjuntas no está necesariamente limitado a las características o actos específicos descritos anteriormente. Por el contrario, las características y actos específicos descritos anteriormente se describen como formas ejemplares de implementación de las reivindicaciones.

REIVINDICACIONES

1. Un procedimiento implementado por ordenador, que comprende:

proporcionar una arquitectura (200) de conjunto de instrucciones que comprende características para generar diversas copias de un programa; utilizar la arquitectura del conjunto de instrucciones para generar diversas copias de un programa (150); proporcionar una máquina (170) virtual para la ejecución de una de las diversas copias del programa,
caracterizado porque

las características para generar diversas copias de un programa comprenden semántica (510) de instrucción que proporcionan la ejecución (512) condicional usando registros (513) de predicados, y
las características para generar diversas copias de un programa comprenden semántica (510) de instrucción con un conjunto (514) de instrucciones limitadas.

2. El procedimiento de la reivindicación 1, en el que proporcionar una máquina virtual comprende generar una biblioteca (172) de enlaces dinámicos de máquina virtual.

3. El procedimiento de la reivindicación 1, en el que el conjunto de instrucciones limitadas es uno o más de:

un conjunto de instrucciones que no incluye una instrucción (515) de "no operación";
un conjunto de instrucciones que tiene una representación limitada de operandos (516); y
un conjunto de instrucciones que limita al menos algunas ramificaciones (517) condicionales.

4. El procedimiento de la reivindicación 1, en el que las características para generar diversas copias de un programa comprenden uno o más de:

codificación (520) de instrucciones para tamaños (522) de instrucciones variables;
codificación de instrucciones para asignar un código de operación usando codificación unaria para introducir superposición (524) física;
codificación de instrucciones para introducir un patrón de bits variable para un código de operación para una instrucción (527);
codificación de instrucciones para asignar un patrón de bits en función de la ejecución previa de un código de operación (528); y
codificación de instrucciones para reorganizar una estructura (529) de decodificación para permitir así llegar a un punto en el programa a lo largo de dos o más rutas de ejecución en el que el significado asignado de un patrón de bits en el punto depende de la ruta de ejecución al punto.

5. El procedimiento de la reivindicación 1, en el que las características para generar diversas copias de un programa comprenden uno o más filtros (542) de ciclo de búsqueda.

6. El procedimiento de la reivindicación 5, en el que uno o más filtros (542) de ciclo de búsqueda comprenden un filtro que agrega información a un bit o bits solicitados de un código (543).

7. El procedimiento de la reivindicación 6, en el que la información comprende uno o más de:

un valor (544) aleatorio; y
un valor (545) de contador de programa o información basada al menos en parte en un valor de contador de programa.

8. El procedimiento de la reivindicación 1, en el que las características para generar diversas copias de un programa comprenden al menos una estructura, seleccionada de un grupo (550) que consiste en árboles extendidos, listas vinculadas y tablas de verificación, para representar el programa.

9. El procedimiento de la reivindicación 1, en el que la máquina virtual se genera como una máquina virtual diversificada combinando partes del código de varias ubicaciones y algunos códigos autogenerados para ensamblar una representación de código gestionado para la implementación de máquinas virtuales o para generar una dll de máquina virtual diversificada.

10. El procedimiento de la reivindicación 1, en el que al menos una segunda capa de virtualización está provista encima de una primera capa de virtualización, en el que la segunda capa de virtualización está configurada para recibir la copia diversificada del programa y permitir la ejecución del programa usando la primera capa de virtualización.

11. El procedimiento de la reivindicación 1, en el que la máquina virtual puede variar el funcionamiento del programa en tiempo de ejecución.

12. Un medio legible por ordenador que comprende instrucciones que, cuando se ejecutan mediante un procesador, hacen que el procesador realice el procedimiento según las reivindicaciones 1 a 11.

13. Un dispositivo informático adaptado para realizar el procedimiento según las reivindicaciones 1 a 11.

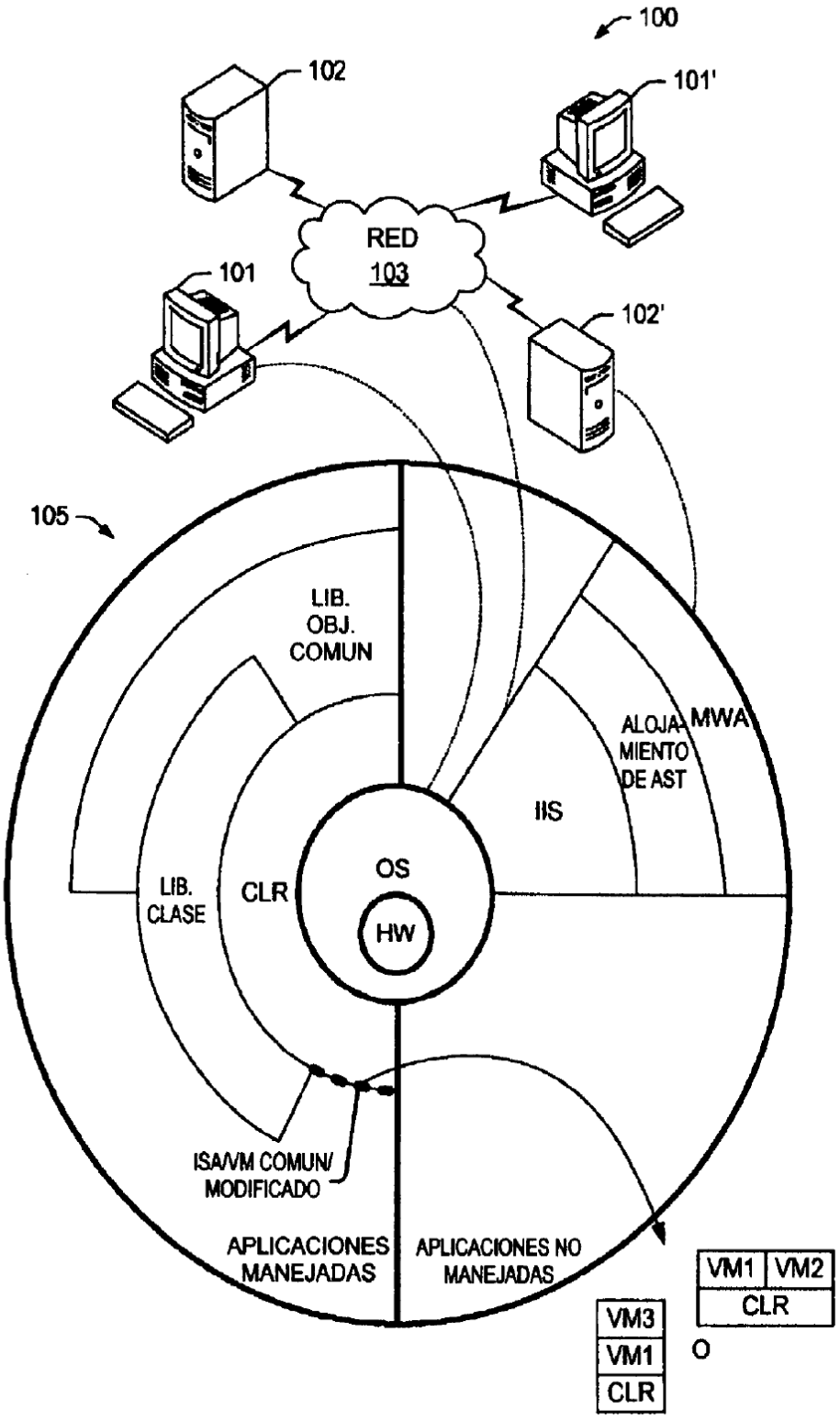
100
102
101'
RED
103
101
102'
105
LIB.
OBJ.
COMUN
ALOJA-
MIENTO
DE AST
MWA
IIS
LIB.
CLASE
CLR
OS
HW
ISA/VM COMUN/
MODIFICADO
APLICACIONES
MANEJADAS
APLICACIONES NO
MANEJADAS
VM1
VM2
CLR
O
VM3
VM1
CLR

FIG. 1

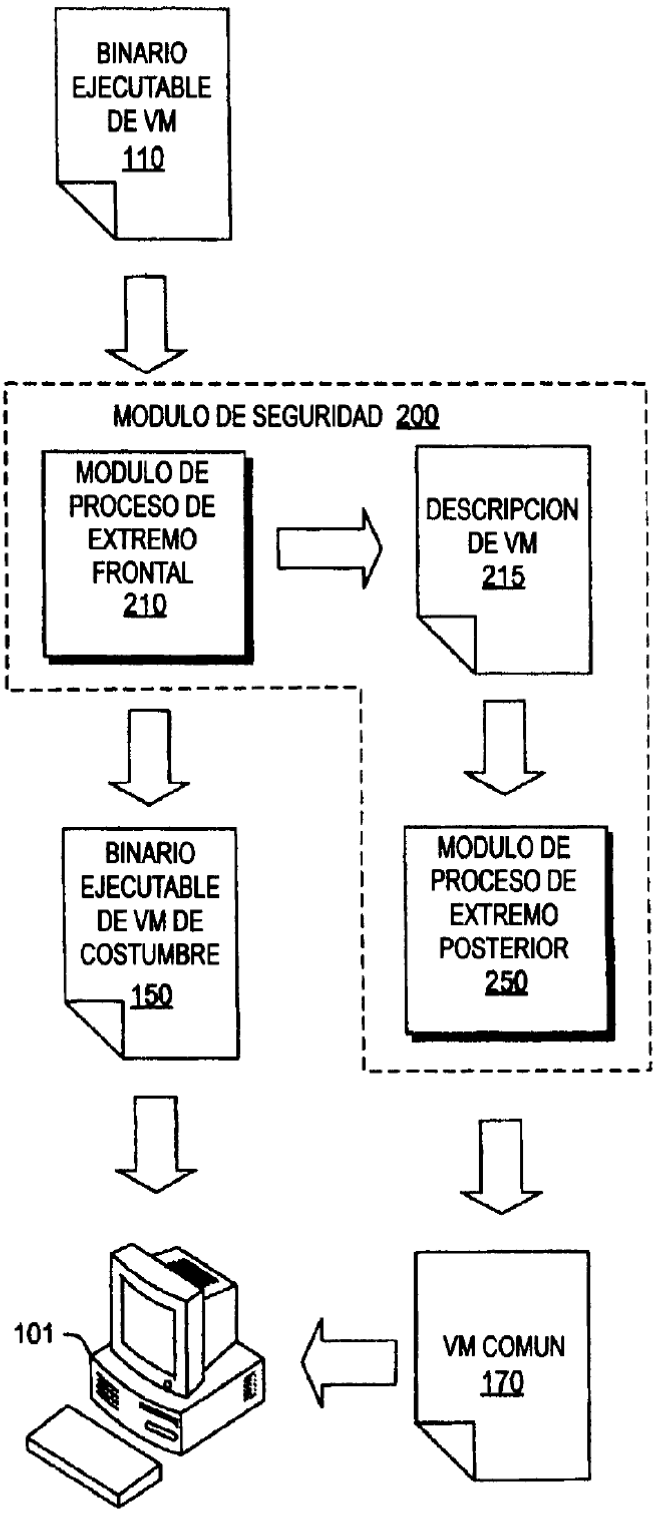
BINARIO EJECUTABLE DE VM 110
MODULO DE SEGURIDAD 200
MODULO DE PROCESO DE EXTREMO FRONTAL 210
DESCRIPCION DE VM 215
BINARIO EJECUTABLE DE VM DE COSTUMBRE 150
MODULO DE PROCESO DE EXTREMO POSTERIOR 250
101
VM COMUN 170

FIG. 2

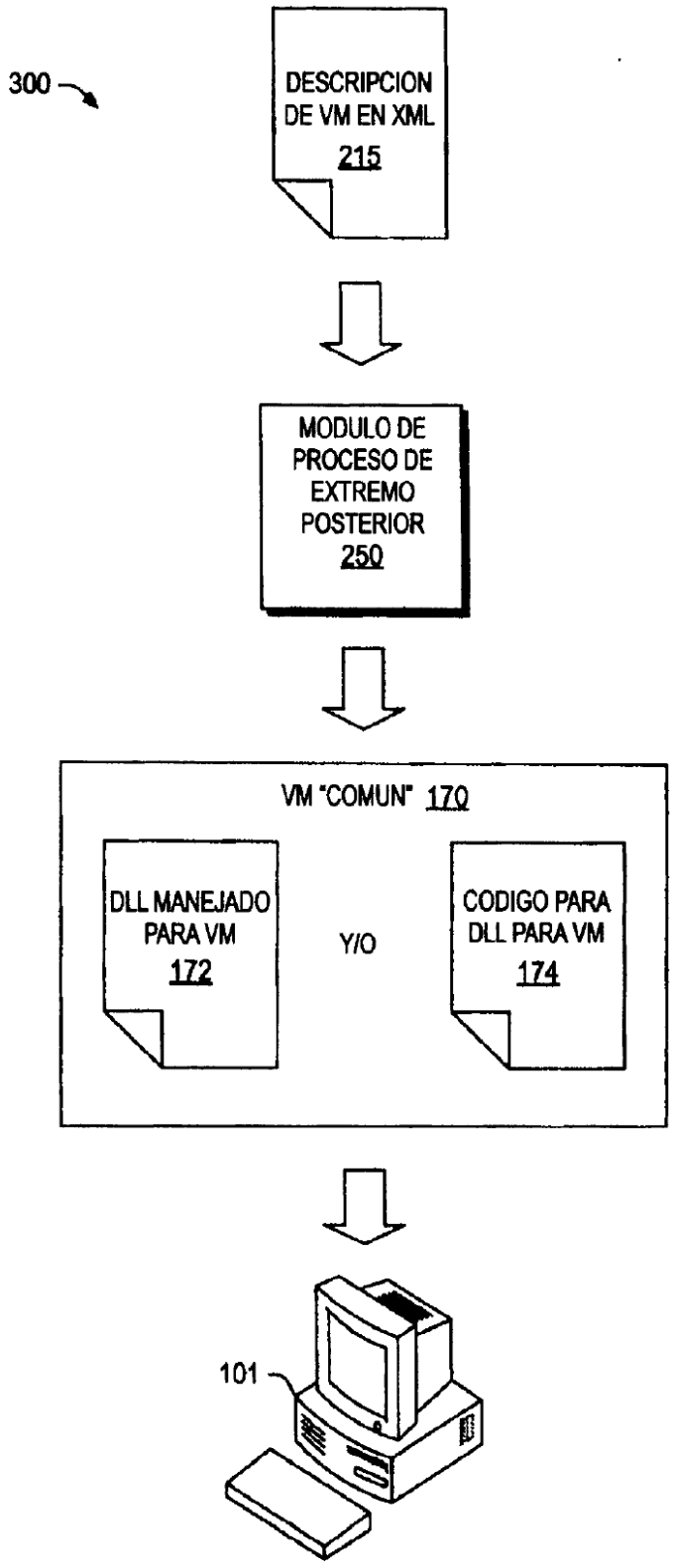
300
DESCRIPCION DE VM EN XML
215
MODULO DE PROCESO DE EXTREMO POSTERIOR
250
VM "COMUN" 170
DLL MANEJADO PARA VM
172
Y/O
CODIGO PARA DLL PARA VM
174
101

FIG. 3

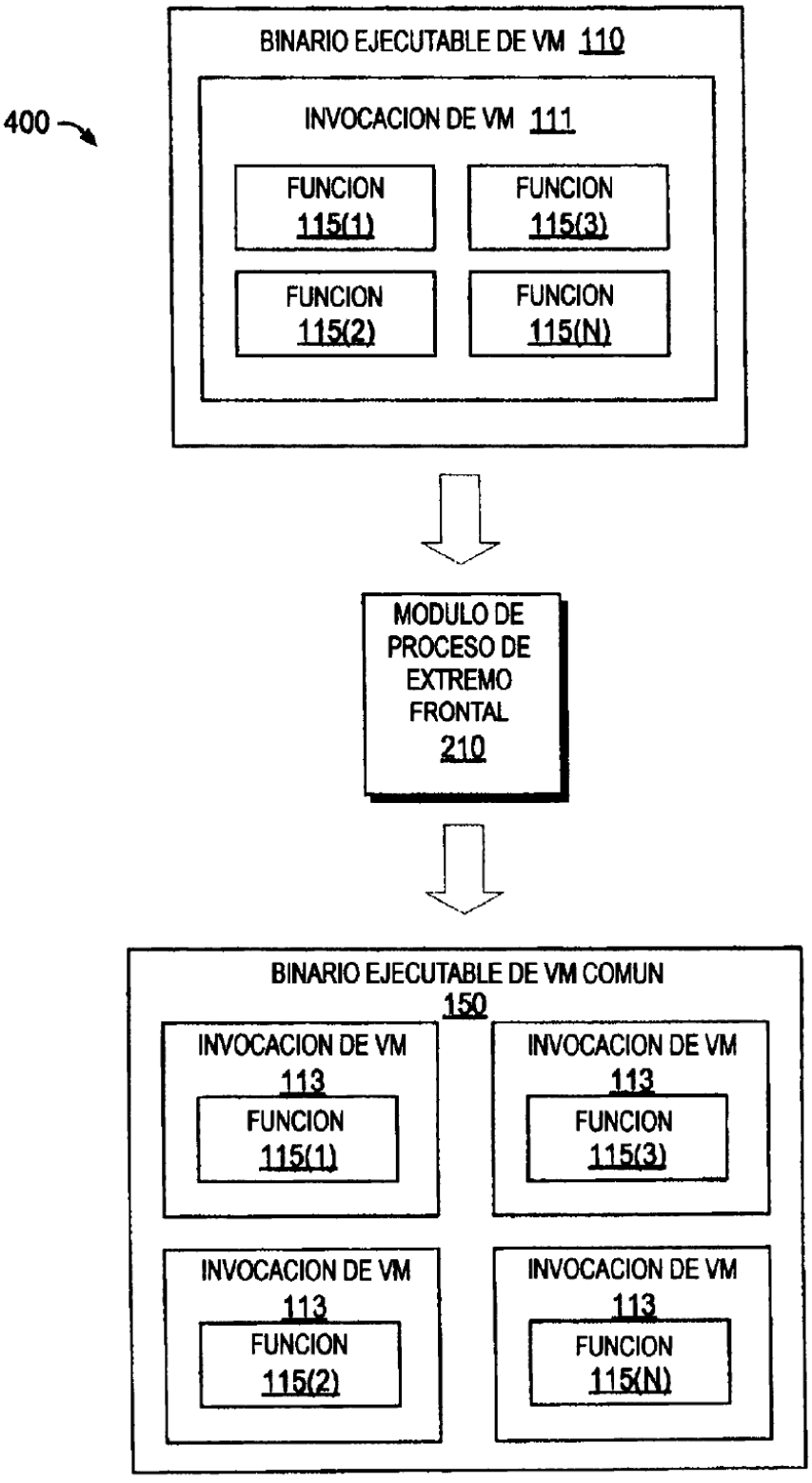
400
BINARIO EJECUTABLE DE VM 110
INVOCACION DE VM 111
FUNCION 115(1)
FUNCION 115(3)
FUNCION 115(2)
FUNCION 115(N)
MODULO DE PROCESO DE EXTREMO FRONTAL 210
BINARIO EJECUTABLE DE VM COMUN 150
INVOCACION DE VM 113
FUNCION 115(1)
INVOCACION DE VM 113
FUNCION 115(3)
INVOCACION DE VM 113
FUNCION 115(2)
INVOCACION DE VM 113
FUNCION 115(N)

FIG. 4

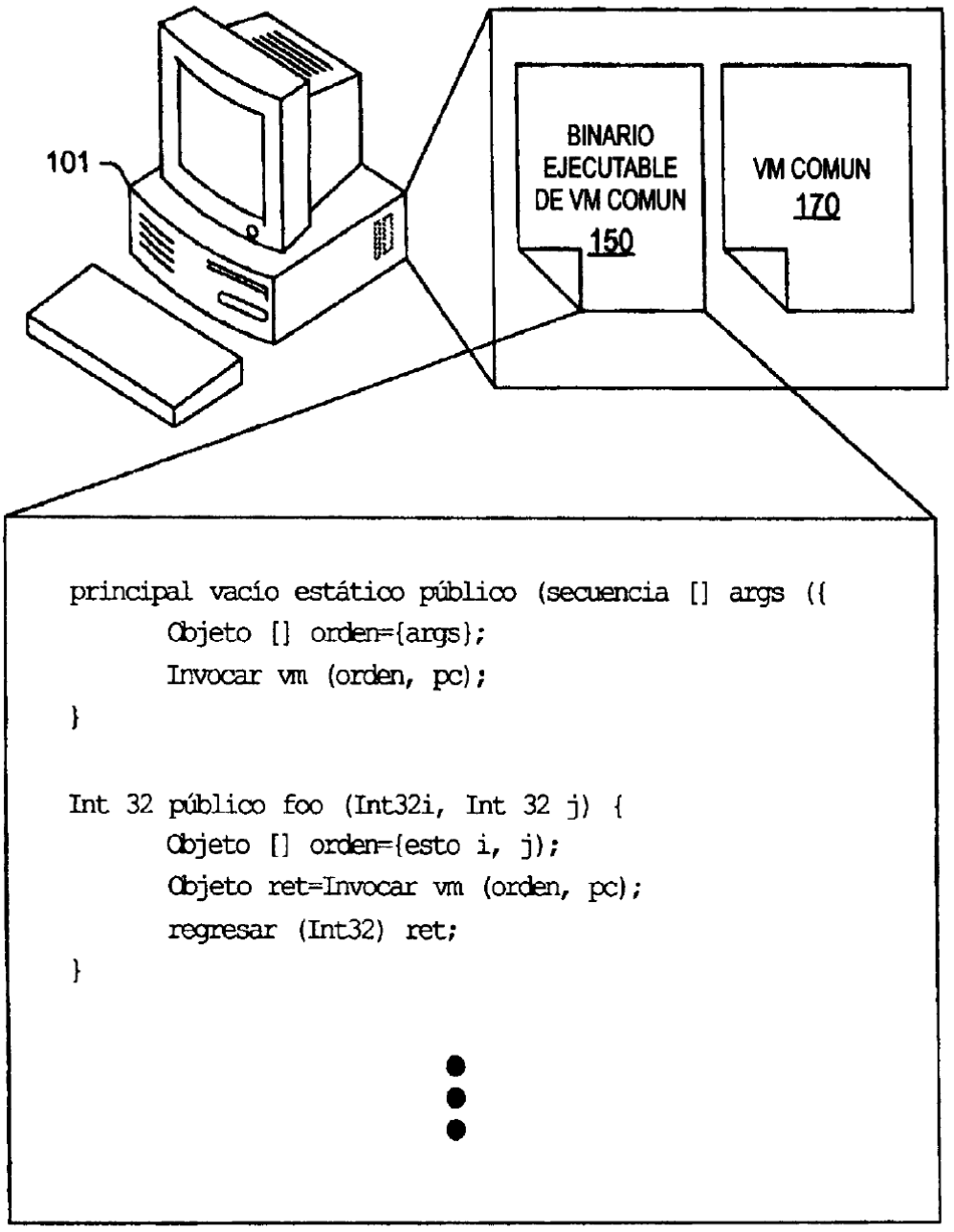
101
BINARIO EJECUTABLE DE VM COMUN
150
VM COMUN
170
principal vacío estático público (secuencia [] args ({
Objeto [] orden={args};
Invocar vm (orden, pc);
}
Int 32 público foo (Int32i, Int 32 j) {
Objeto [] orden={esto i, j);
Objeto ret=Invocar vm (orden, pc);
regresar (Int32) ret;
}

FIG. 5

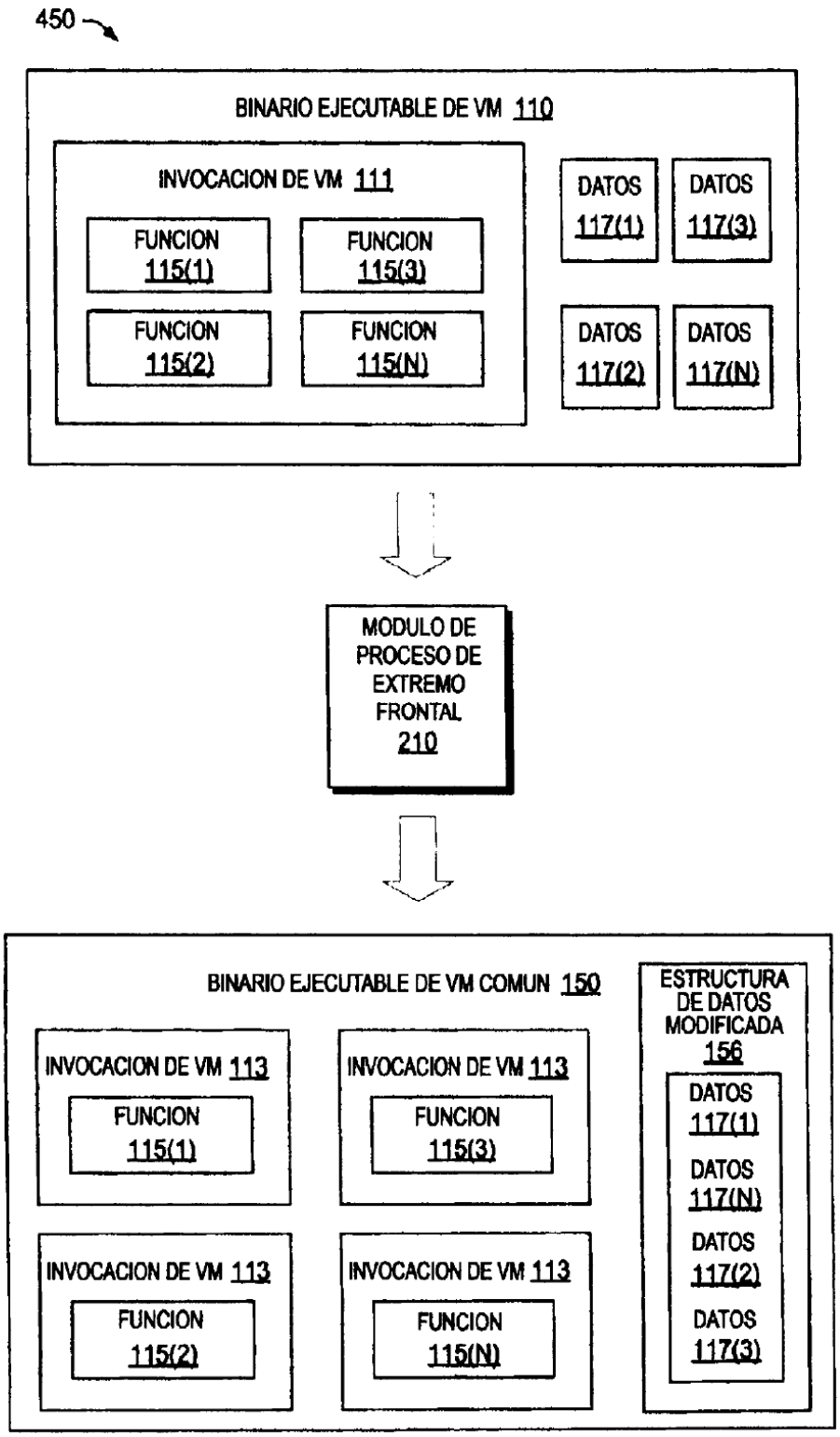
450
BINARIO EJECUTABLE DE VM 110
INVOCACION DE VM 111
FUNCION 115(1)
FUNCION 115(3)
FUNCION 115(2)
FUNCION 115(N)
DATOS 117(1)
DATOS 117(3)
DATOS 117(2)
DATOS 117(N)
MODULO DE PROCESO DE EXTREMO FRONTAL 210
BINARIO EJECUTABLE DE VM COMUN 150
INVOCACION DE VM 113
FUNCION 115(1)
INVOCACION DE VM 113
FUNCION 115(3)
INVOCACION DE VM 113
FUNCION 115(2)
INVOCACION DE VM 113
FUNCION 115(N)
ESTRUCTURA DE DATOS MODIFICADA 156
DATOS 117(1)
DATOS 117(N)
DATOS 117(2)
DATOS 117(3)

FIG. 6

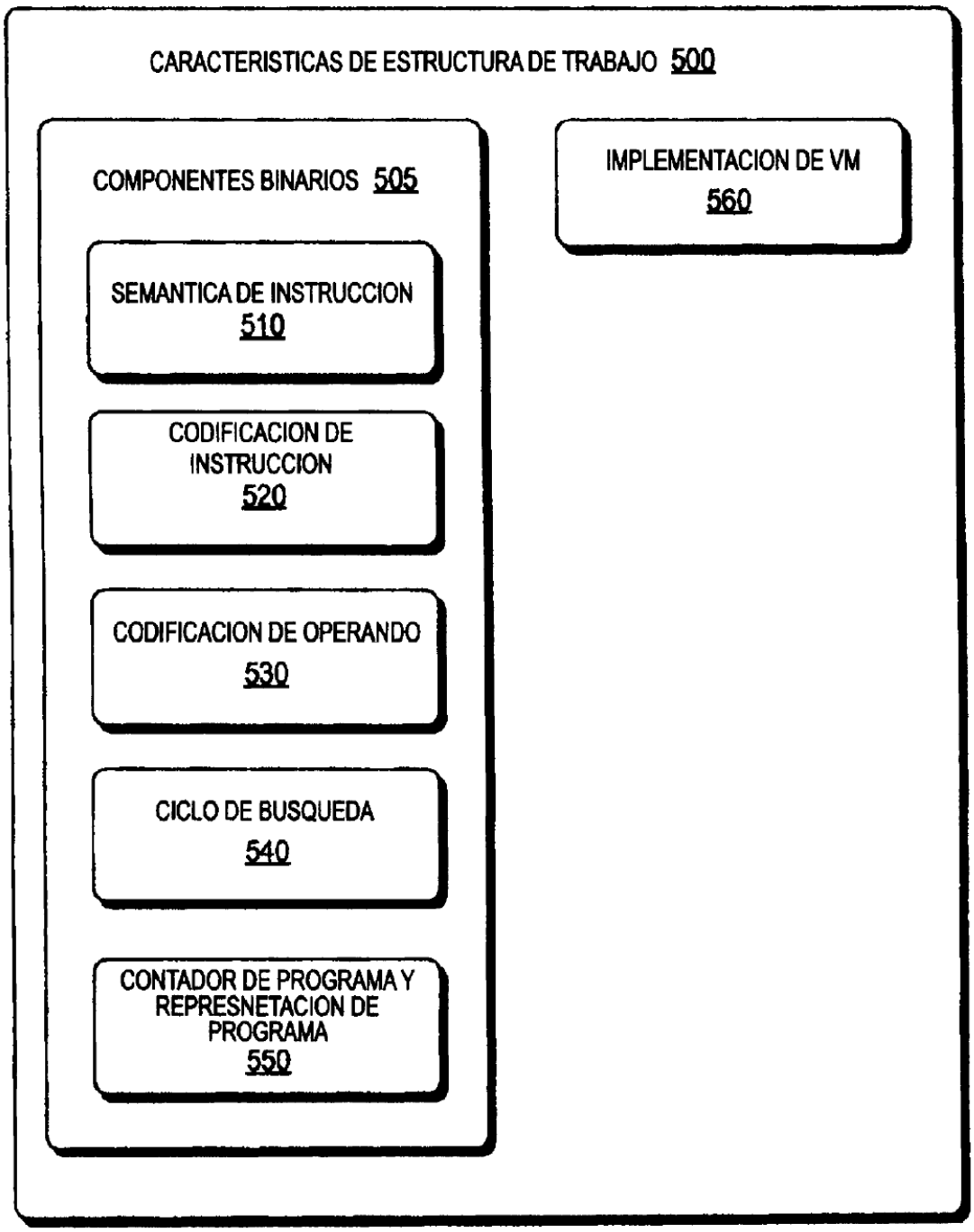
CARACTERISTICAS DE ESTRUCTURA DE TRABAJO 500
COMPONENTES BINARIOS 505
SEMANTICA DE INSTRUCCION 510
CODIFICACION DE INSTRUCCION 520
CODIFICACION DE OPERANDO 530
CICLO DE BUSQUEDA 540
CONTADOR DE PROGRAMA Y REPRESNETACION DE PROGRAMA 550
IMPLEMENTACION DE VM 560

FIG. 7

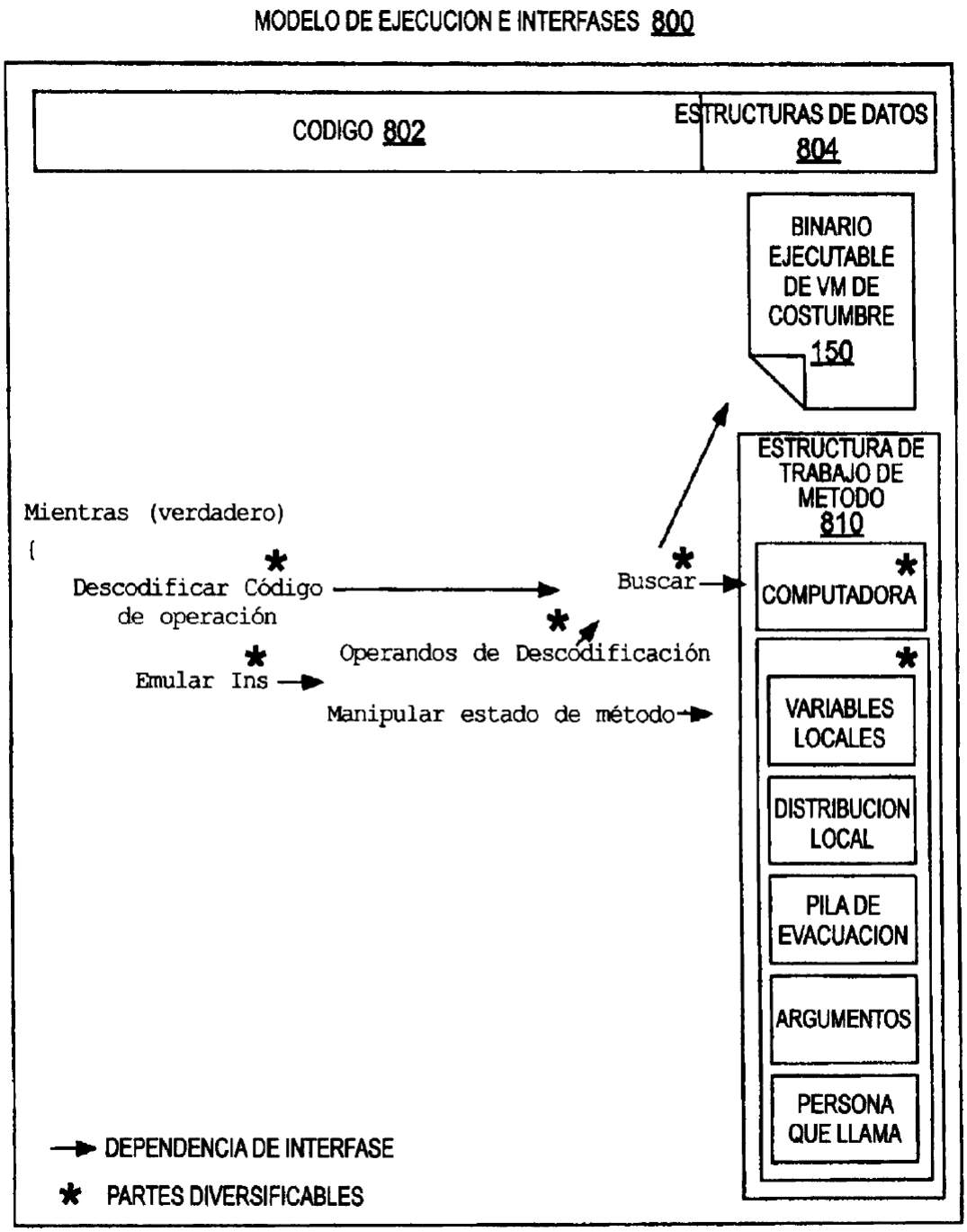
MODELO DE EJECUCION E INTERFASES 800
CODIGO 802
ESTRUCTURAS DE DATOS 804
BINARIO EJECUTABLE DE VM DE COSTUMBRE 150
ESTRUCTURA DE TRABAJO DE METODO 810
Mientras (verdadero)
{
Descodificar Código de operación
Buscar
COMPUTADORA
Operandos de Descodificación
Emular Ins
Manipular estado de método
VARIABLES LOCALES
DISTRIBUCION LOCAL
PILA DE EVACUACION
ARGUMENTOS
PERSONA QUE LLAMA
DEPENDENCIA DE INTERFASE
PARTES DIVERSIFICABLES

FIG. 8

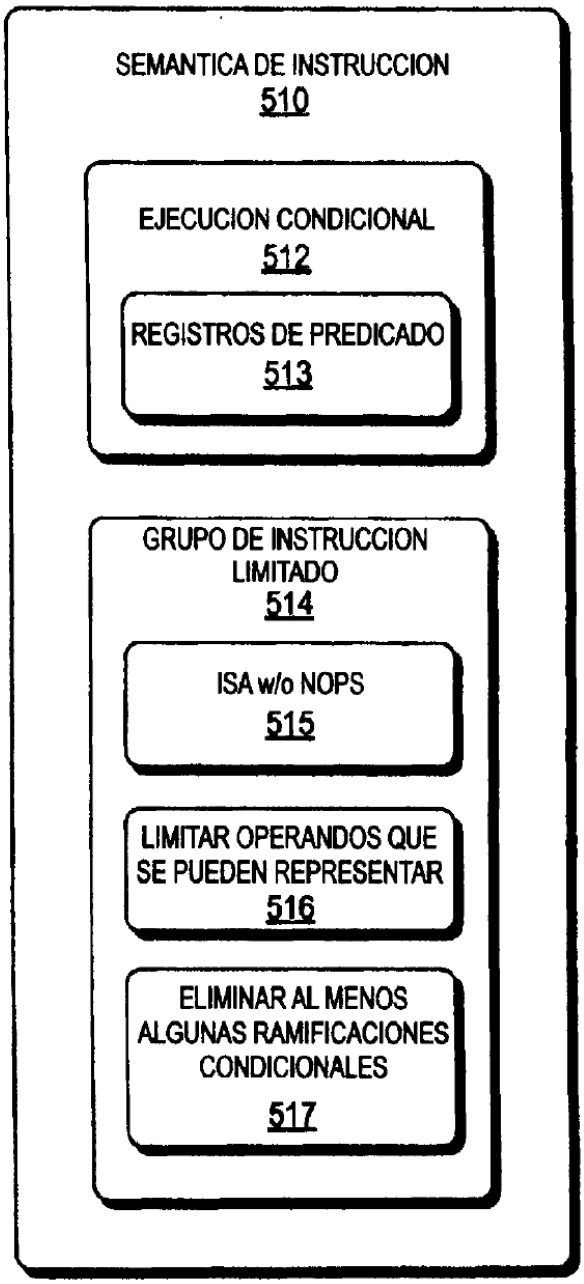
SEMANTICA DE INSTRUCCION
510
EJECUCION CONDICIONAL
512
REGISTROS DE PREDICADO
513
GRUPO DE INSTRUCCION LIMITADO
514
ISA w/o NOPS
515
LIMITAR OPERANDOS QUE SE PUEDEN REPRESENTAR
516
ELIMINAR AL MENOS ALGUNAS RAMIFICACIONES CONDICIONALES
517

FIG. 9

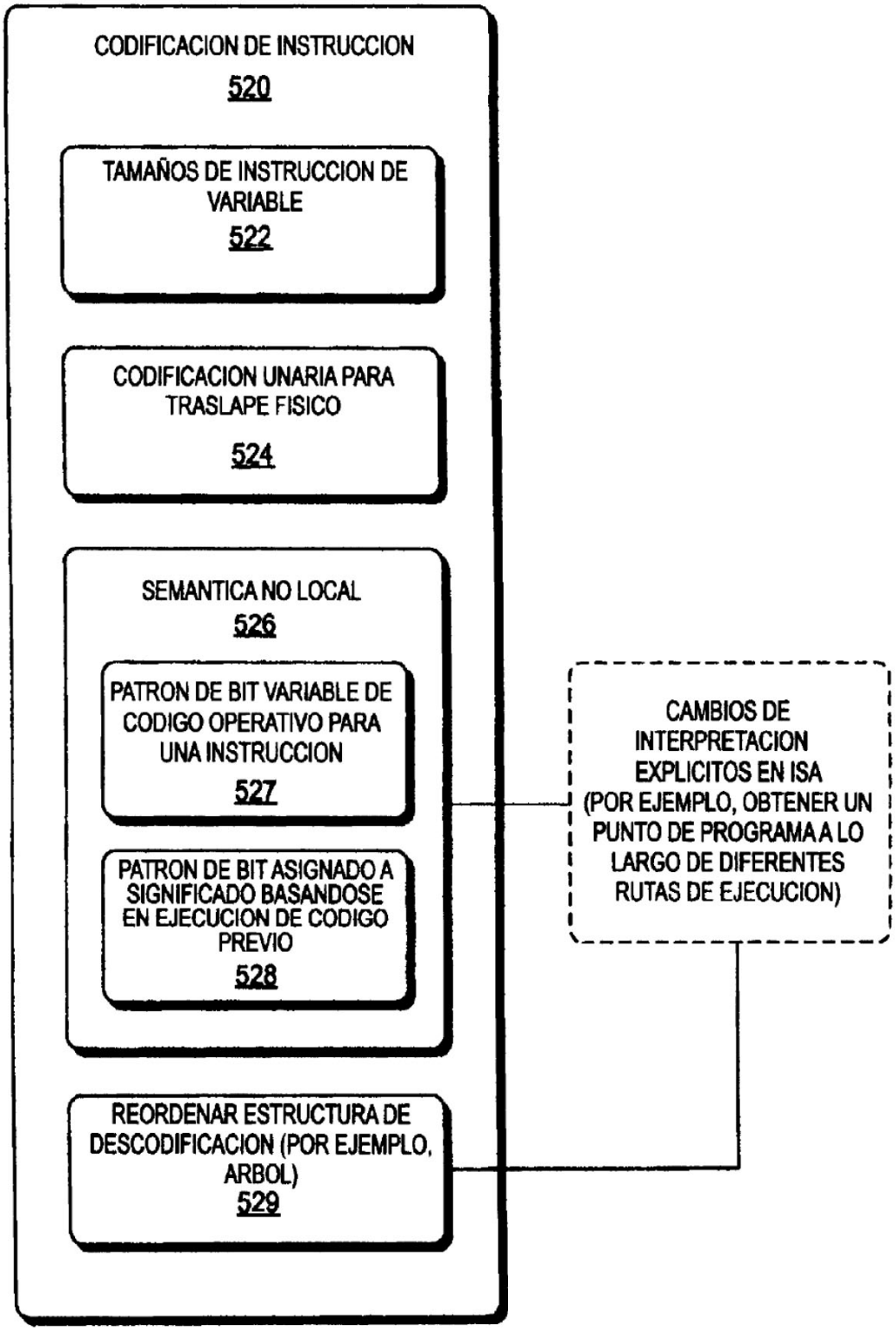
CODIFICACION DE INSTRUCCION
520
TAMAÑOS DE INSTRUCCION DE VARIABLE
522
CODIFICACION UNARIA PARA TRASLAPE FISICO
524
SEMANTICA NO LOCAL
526
PATRON DE BIT VARIABLE DE CODIGO OPERATIVO PARA UNA INSTRUCCION
527
PATRON DE BIT ASIGNADO A SIGNIFICADO BASANDOSE EN EJECUCION DE CODIGO PREVIO
528
REORDENAR ESTRUCTURA DE DESCODIFICACION (POR EJEMPLO, ARBOL)
529
CAMBIOS DE INTERPRETACION EXPLICITOS EN ISA (POR EJEMPLO, OBTENER UN PUNTO DE PROGRAMA A LO LARGO DE DIFERENTES RUTAS DE EJECUCION)

FIG. 10

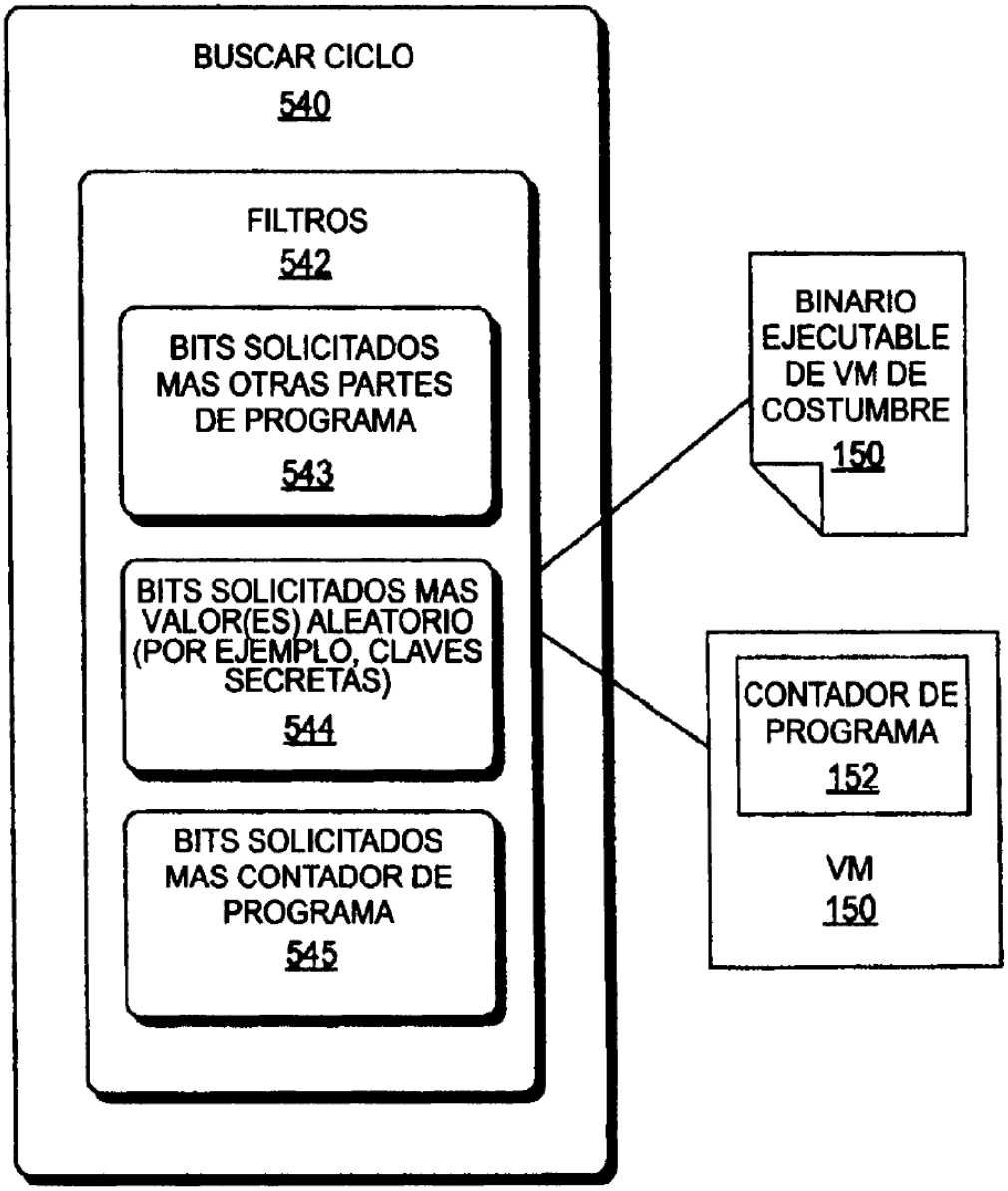
BUSCAR CICLO
540
FILTROS
542
BITS SOLICITADOS MAS OTRAS PARTES DE PROGRAMA
543
BITS SOLICITADOS MAS VALOR(ES) ALEATORIO (POR EJEMPLO, CLAVES SECRETAS)
544
BITS SOLICITADOS MAS CONTADOR DE PROGRAMA
545
BINARIO EJECUTABLE DE VM DE COSTUMBRE
150
CONTADOR DE PROGRAMA
152
VM
150

FIG. 11


CONTADOR DE PROGRAMA Y
REPRESENTACION DE PROGRAMA
550
ESTRUCTURAS
552
ORDEN
553
ARBOL (POR EJEMPLO CONVENCIONAL,
EXTENDIDO, ETC.)
554
LISTA VINCULADA
555
TABLA DE VERIFICACION
556

FIG. 12

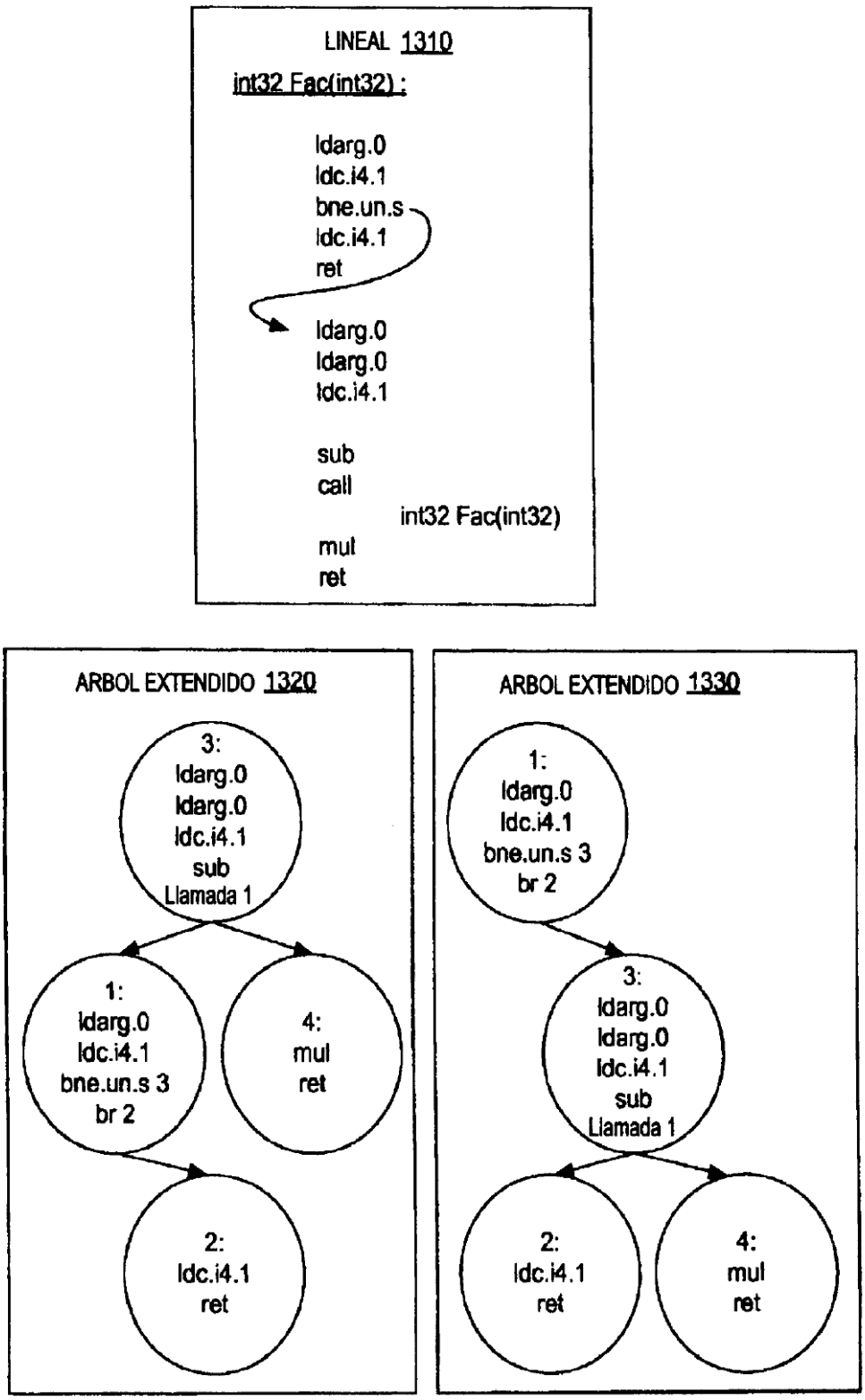
LINEAL 1310
int32 Fac(int32) :
ldarg.0
ldc.i4.1
bne.un.s
ldc.i4.1
ret
ldarg.0
ldarg.0
ldc.i4.1
sub
call
int32 Fac(int32)
mul
ret
ARBOL EXTENDIDO 1320
3:
ldarg.0
ldarg.0
ldc.i4.1
sub
Llamada 1
1:
ldarg.0
ldc.i4.1
bne.un.s 3
br 2
4:
mul
ret
2:
ldc.i4.1
ret
ARBOL EXTENDIDO 1330
1:
ldarg.0
ldc.i4.1
bne.un.s 3
br 2
3:
ldarg.0
ldarg.0
ldc.i4.1
sub
Llamada 1
2:
ldc.i4.1
ret
4:
mul
ret

FIG. 13

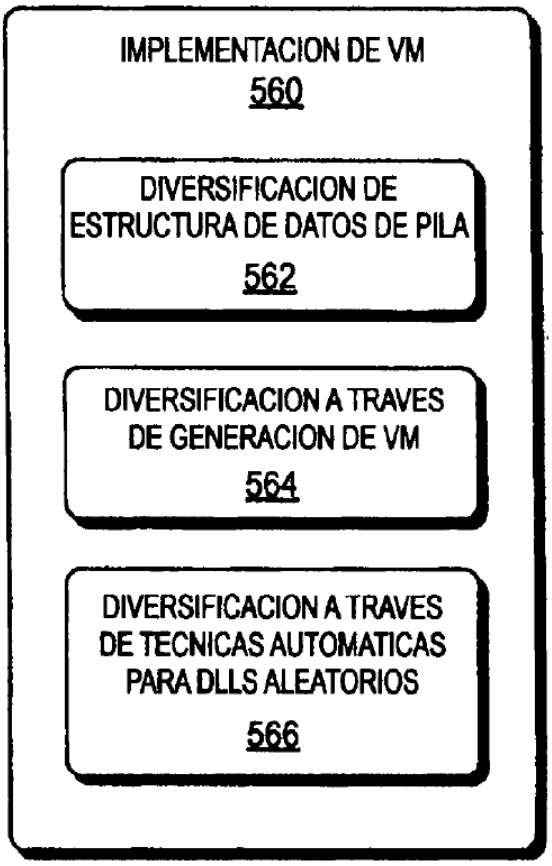
IMPLEMENTACION DE VM
560
DIVERSIFICACION DE ESTRUCTURA DE DATOS DE PILA
562
DIVERSIFICACION A TRAVES DE GENERACION DE VM
564
DIVERSIFICACION A TRAVES DE TECNICAS AUTOMATICAS PARA DLLS ALEATORIOS
566

FIG. 14

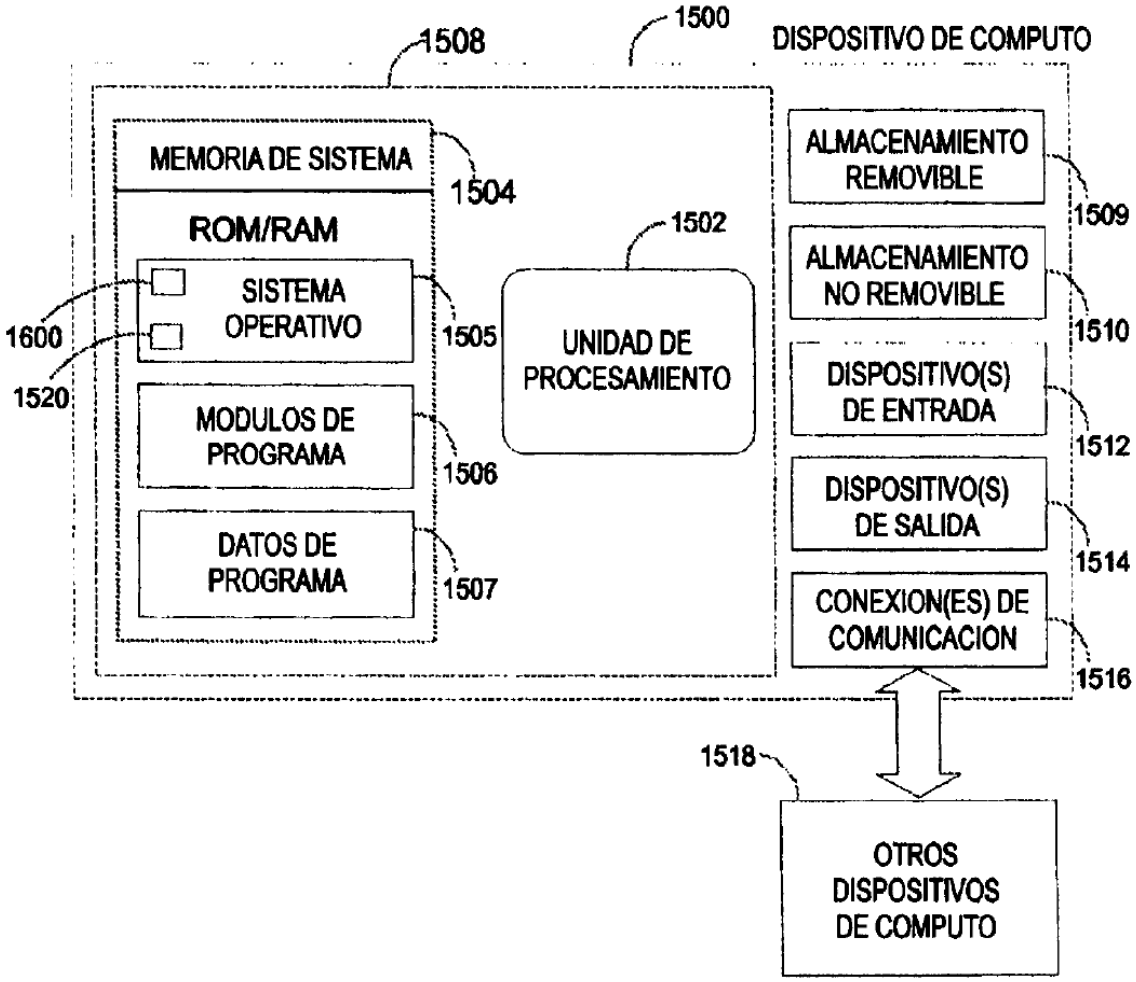
1500
DISPOSITIVO DE COMPUTO
1508
MEMORIA DE SISTEMA
1504
ROM/RAM
SISTEMA OPERATIVO
1505
1600
1520
MODULOS DE PROGRAMA
1506
DATOS DE PROGRAMA
1507
1502
UNIDAD DE PROCESAMIENTO
ALMACENAMIENTO REMOVIBLE
1509
ALMACENAMIENTO NO REMOVIBLE
1510
DISPOSITIVO(S) DE ENTRADA
1512
DISPOSITIVO(S) DE SALIDA
1514
CONEXION(ES) DE COMUNICACION
1516
1518
OTROS DISPOSITIVOS DE COMPUTO

FIG. 15