Towards the Detection of Isolation-Aware Malware

Ricardo J. Rodríguez, *Member, IEEE*, Iñaki Rodríguez-Gastón, and Javier Alonso, *Member, IEEE*

***Abstract*— Malware analysis tools have evolved in the last years providing tightly controlled sandbox and virtualised environments where malware is analysed minimising potential harmful consequences. Unfortunately, malware has advanced in parallel, being currently able to recognise when is running in sandbox or virtual environments and then, behaving as a non-harmful application or even not executing at all. This kind of malware is usually called *analysis-aware malware*. In this paper, we propose a tool to detect the evasion techniques used by analysis-aware malware within sandbox or virtualised environments. Our tool uses Dynamic Binary Instrumentation to maintain the binary functionality while executing arbitrary code. We evaluate the tool under a set of well-known analysis-aware malware showing its current effectiveness. Finally, we discuss limitations of our proposal and future directions.**

***Index Terms*— analysis-aware malware, binary analysis, dynamic binary instrumentation.**

I. Introducción

El volumen de aplicaciones software creadas con intenciones maliciosas o sospechosas (dícese *malware* por su nombre en inglés) ha crecido tanto en cantidad como en complejidad en la última década [18], [35], [36], [44]. La lucha contra el malware requiere una constante actualización de las técnicas defensivas, entender cómo actúa el malware, qué tipo de acciones dañinas realiza y cómo éstas pueden ser detectadas, eliminadas o prevenidas.

Expertos analistas de malware deben hacer frente cada día a un creciente número de muestras de malware. Por ejemplo, la compañía especializada en antivirus Kaspersky indicó que en 2013 analizó, aproximadamente, 350.000 muestras de malware diarias [30]. La ingeniería inversa es el proceso desarrollado para conocer y entender las capacidades y comportamiento del malware; sin embargo, requiere mucho tiempo y trabajo. Dado el incremento diario de retos a los que se enfrentan los analistas de malware, dos tendencias complementarias han emergido recientemente en el análisis de malware: (i) automatización de las tareas de análisis de muestras de malware; y (ii) aislamiento de los entornos usados para el análisis del malware [24].

La automatización del proceso de análisis del malware permite, por un lado, un conocimiento del comportamiento de la muestra de una forma ágil y rápida, y por tanto permite mantener actualizadas las bases de datos de anti-virus para proteger de forma efectiva a los usuarios finales ante nuevas amenazas. Por otro lado, el aislamiento del entorno de análisis de malware previene que durante el proceso de análisis el malware pueda infectar un sistema o una red legítima, o incluso llegue a interferir en el negocio de la empresa donde se está llevando el análisis. En este sentido, en este artículo sólo consideramos las soluciones software propuestas para aislar el entorno de análisis de malware, dejando de lado las soluciones basadas en máquinas físicas aisladas como entorno de análisis.

Sin embargo, los desarrolladores de malware han aprendido a su vez cómo se analizan las muestras de malware, y por ello han empezado a incorporar a sus desarrollos diversas técnicas de evasión que permiten reconocer cuándo el malware está siendo ejecutado en un entorno de análisis. Es importante hacer notar que cuanto más tiempo una muestra de malware permanece no detectada, más probable es que dicho malware infecte equipos, sistemas y redes, extendiendo así la capacidad operativa del criminal. Algunas de las técnicas más comunes usadas por el malware para evadir los entornos de análisis han sido ya descritas en la literatura [9], [12], [20], [22], [26]. Una muestra de malware que integre cualquiera de estas técnicas u otras con el objetivo de evitar ser detectada durante el proceso de análisis se conoce como malware consciente (*analysis-aware malware*, en inglés) o malware con doble personalidad (*split personality malware*, en inglés) [5], [31], [47]. Cuando un entorno de análisis (por ejemplo, herramientas de depuración, de sandbox o máquinas virtuales) se detecta, el malware consciente es capaz de cambiar su comportamiento dañino o perjudicial por otro aparentemente legítimo, o simplemente no ejecutarse. Además, se debe tener muy en cuenta que ser capaz de reconocer un entorno aislado de análisis puede ser el primer paso para escapar de él: una vez que el software malicioso ha sido incorrectamente clasificado como un software legítimo o cuando menos benigno, puede penetrar en el sistema objetivo sin mayores problemas [11].

En este artículo, primero hemos revisado las técnicas de evasión más relevantes usadas por el software malicioso consciente; es decir, aquellas técnicas que son usadas para detectar y reconocer un entorno aislado de análisis. Después, proponemos una herramienta de análisis binario dinámico (DBA, por sus siglas en inglés) que permite ofuscar el entorno

R. J. Rodríguez, Department of Computer Science and Systems Engineering, University of Zaragoza, Spain, rjrodriguez@unizar.es. Part of this work was done while R. J. Rodríguez was at Research Institute of Applied Sciences in Cybersecurity, University of León, Spain.

I. Rodríguez-Gastón, MLW.re NPO, Palma de Mallorca, Spain, inaki@virtualminds.es

J. Alonso, Research Institute of Applied Sciences in Cybersecurity, University of León, Spain, javier.alonso@unileon.es

aislado de análisis haciendo así que el software malicioso consciente no sea capaz de detectar dicho entorno, y éste muestre todo su comportamiento malicioso. Una herramienta DBA hace uso de instrumentación dinámica de binarios (DBI, por sus siglas en inglés) para analizar el comportamiento de los ejecutables mientras mantiene un control absoluto de su ejecución. Esta aproximación al problema ofrece dos ventajas principales: por un lado, es totalmente independiente del lenguaje de programación y compilador usado para generar el código binario; por otro lado, no es necesario recompilar cada vez que se cambia el código de instrumentación, ya que el código de instrumentación se añade en tiempo de ejecución. Finalmente, hemos evaluado la herramienta propuesta analizando un conjunto de software malicioso consciente conocido. Los resultados muestran claramente que la herramienta propuesta es capaz de engañar al software malicioso haciéndole creer que está en un entorno no aislado, ejecutando así todas sus funciones maliciosas.

El artículo se estructura como sigue. En la Sección II describimos y clasificamos las técnicas de evasión más comunes o relevantes usadas por el software malicioso consciente. La Sección III introduce la arquitectura de nuestra herramienta y del entorno de DBI donde se integra. Las técnicas actuales de evasión detectadas por nuestra herramienta, así como las contramedidas DBI, se introducen también en esta sección. La evaluación de la herramienta frente a malware real se presenta en la Sección IV. La Sección V hace una revisión de la literatura relacionada. Finalmente, la Sección VI concluye el trabajo y presenta el trabajo futuro.

II. Revisión de Técnicas de Evasión de Análisis

Las técnicas de evasión son usadas por el malware consciente para reconocer y evadir el proceso de análisis; mostrando así un comportamiento lo suficientemente legítimo (no malicioso) para ser ejecutado por cualquier usuario. Muchas de estas técnicas se pueden encontrar actualmente en numerosas familias de malware, como Conficker, Neutrino, o familias de bots IRC como Rbot, SDbot/Reptile, Mechbot, SpyBot, o AgoBot [21], [34]. De hecho, cualquier software protector comercial o personalizado normalmente ofrece algunas de estas técnicas de evasión como un mecanismos de legítima defensa contra la ingeniería inversa o la protección de copia de software [9], [13].

En este trabajo se han clasificado las técnicas de evasión considerando si la evasión se realiza por el código ejecutable directa o indirectamente. Así, las técnicas de evasión se pueden clasificar en dos categorías principales: *directas*, cuando hay trozos de código añadidos en el malware para específicamente reconocer y evadir el análisis, o *indirectas*, cuando el malware no incorpora explícitamente estos trozos de código para evadir el análisis si no que introducen otros códigos que hacen uso del tiempo (e.g., ejecución retrasada mediante funciones durmientes, bucles de espera, o tareas programadas), o se basan en eventos (e.g., ejecución condicionada según actividad del usuario, o ejecución al arrancar) con el mismo objetivo. Las técnicas directas, a su vez, se pueden subcategorizar considerando el entorno/ámbito de análisis del que la muestra de malware intenta evadirse: *reconocimiento de análisis del binario*, cuando un malware se da cuenta de que su código binario está siendo analizado; *reconocimiento de análisis en entorno aislado*, cuando un malware identifica características exclusivas de un entorno aislado; y *reconocimiento de análisis de la memoria*, cuando un malware de manera proactiva ejecuta código para evitar el análisis del espacio de memoria del propio proceso del malware. La Figura 1 muestra la clasificación de técnicas de evasión propuesta. En este artículo nos centramos en las técnicas de evasión directas contra el sistema operativo Windows, dado que actualmente es el principal objetivo del software malicioso [30].

A. Reconocimiento de análisis del binario

Esta categoría incluye las técnicas que previenen un binario de ser dinámica o estáticamente analizado [9], [12], [22], [47]. El código binario de un programa se puede analizar de manera estática usando un desensamblador, o dinámicamente usando un depurador o un framework de instrumentación de código binario. Así, esta categoría se puede redefinir en tres subcategorías: Anti-depuración, anti-desensamblado y anti-instrumentación dinámica. Las técnicas de anti-depuración se usan para evitar el análisis bajo un entorno de depuración. Estas técnicas intenta reconocer cuándo un proceso está siendo depurado, o incluso traceado. Por ejemplo, el sistema operativo Windows incorpora numerosas funciones propias (llamadas APIs) que pueden ejecutarse para verificar si un proceso está bajo ejecución de un depurador (e.g., `IsBeingDebugged`, `CheckRemoteDebuggerPresent`, o `NtQueryInformationProcess`, entre otras), y otros tantos elementos/artefactos de memoria que se pueden consultar también para este propósito (e.g., *NtGlobalFlags*, *DebugObject*, o el token *SeDebugPrivilege*). Una actividad de traceo se puede reconocer mediante la ejecución de ciertas instrucciones de ensamblador cuyo comportamiento varía cuando la bandera de *trap* está activada en el procesador (e.g., la instrucción **pop ss**), o bien mediante la ejecución controlada de excepciones (e.g., la excepción **SINGLE_STEP**), comprobando a su vez cómo son manejadas.

Las técnicas anti-desensamblado tratan de evitar el desensamblado del código del binario, bien en una herramienta de desensamblado o en un depurador. La ofuscación de código o el código basura (código adicional que se inserta para confundir durante un análisis ya que nunca se ejecuta) son ejemplos claros de este tipo de técnicas de evasión. Otras técnicas de anti-desensamblado explotan vulnerabilidades en herramientas de desensamblado o depuradores. Por ejemplo, dos conocidos depuradores (OllyDBG y SoftICE) son incapaces de desensamblar un ejecutable de Windows que tenga una cabecera de ejecutable mal formada.

Las técnicas de detección de instrumentación dinámica de binarios, es decir, técnicas que reconocen cuando un binario está siendo instrumentado durante su ejecución, también encajan dentro de esta categoría. Por lo que sabemos, el único trabajo dirigido hacia este tipo de técnicas es [19]. En él,

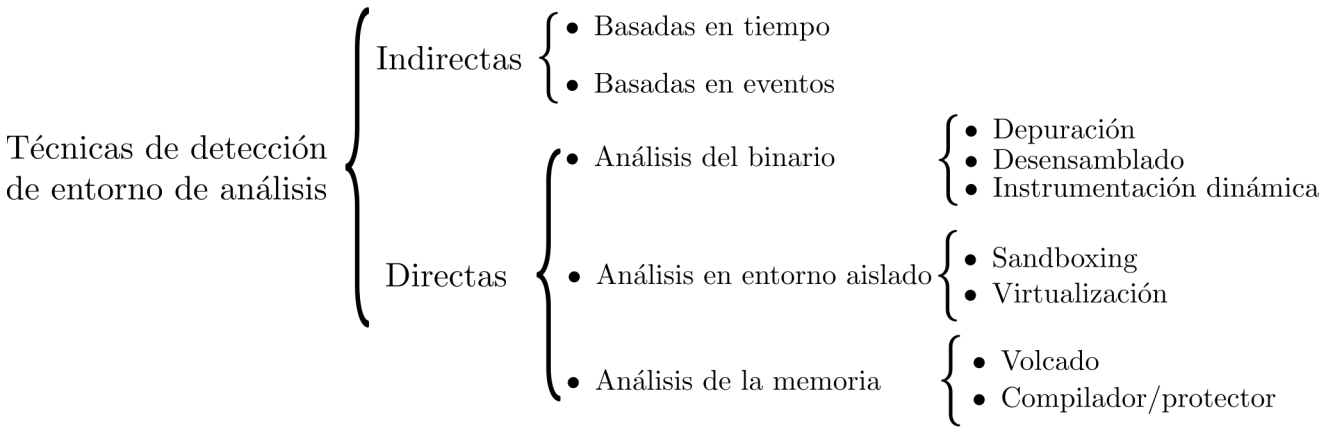

Figura 1. Clasificación de las técnicas de evasión.

Falcón y Riva proveen numerosas técnicas para detectar la ejecución en un entorno de instrumentación dinámica – concretamente, en el entorno Pin [33] y el sistema operativo Windows; aunque estas técnicas podrían extenderse para reconocer otros entornos de instrumentación. Como la herramienta que se propone en este artículo hace uso de Pin, a continuación explicamos en más detalle estas técnicas de detección. Nótese que algunas de estas técnicas de detección son evadidas actualmente por nuestra herramienta (véase la Sección III.C). Así, las técnicas de detección de instrumentación dinámica de ejecutables se pueden subdividir en:

- **Detección de la máquina virtual de Pin**: numerosas técnicas se detallan en [19] para reconocer la presencia de la librería dinámica de la máquina virtual de Pin (llamada *pinvm.dll*). Por ejemplo, realizando búsquedas de patrones de cadenas, de código, nombres de funciones exportadas o secciones de código, entre otras.
- **Detección de tuberías de comunicación**: mediante la búsqueda de las tuberías que usa Pin para comunicarse internamente entre sus procesos se puede detectar su presencia. Normalmente estas tuberías comienzan con la subcadena "Pin_IPC".
- **Detección por variaciones de tiempo**: es bien sabido que el uso de instrumentación dinámica induce un sobrecoste, en términos de ejecución. En [39] se llevó a cabo una cuantificación de este sobrecoste, en términos de tiempos de ejecución y memoria, en diferentes entornos de instrumentación dinámica. Esta sobrecarga se puede notar mediante la medición de instrucciones de manera consecutiva, y comparando su duración de ejecución con un valor de ejecución conocido. Sin embargo, el uso de una medida temporal como cota superior puede producir falsos positivos, dado que el tiempo de ejecución es muy dependiente del entorno donde se está ejecutando el binario.
- **Detección del compilador JIT de Pin**: el compilador de Pin (véase la Sección III.A) sobrescribe algunas de las funciones nativas usadas por el propio sistema operativo de Windows (i.e., de la librería *ntdll.dll*). El compilador JIT también usa numerosas páginas de memoria con un conocido conjunto de permisos (EXECUTE READ/WRITE, en particular). Nótese que esta técnica de detección también puede producir falsos positivos, ya que un binario por sí mismo puede ser/usar un compilador JIT propio.
- **Detección por el valor del puntero de siguiente instrucción (registro EIP de la CPU)**: Pin usa una cache de código donde guarda el código después de su ejecución instrumentada (véase la Sección III.A), así que el código original nunca se ejecuta. El puntero de siguiente instrucción a ejecutar de la CPU (registro EIP en arquitecturas Intel x86) se puede consultar mediante llamadas al entorno de ejecución de la unidad de coma flotante (con instrucciones en ensamblador como `fstenv`, `fsave` o `fxsave`), o mediante interrupciones como `0x2E`.
- **Detección por otras técnicas (miscelánea)**: existen otras técnicas válidas para la detección del entorno Pin, como por ejemplo, la comprobación de parámetros de entrada al programa, la jerarquía de procesos (comprobando el proceso padre del binario), o por emulación mediante `SYSENTER`. Nótese que este último método sólo es válido para las versiones de Pin anteriores a la versión 39599, de 2011.

B. Reconocimiento de análisis en entorno aislado

Las técnicas de reconocimiento de análisis en entorno aislado incluyen las técnicas que permiten a un ejecutable reconocer cuándo éste se está ejecutando dentro de una herramienta de *sandbox* o en una máquina virtual [9], [12], [20], [26]. Una sandbox provee un entorno donde los recursos del ordenador (es decir, controladores de red, teclado y ratón, controladores de disco, etc.) se controlan y monitorizan de manera muy estricta; mientras que una máquina virtual provee una capa de hardware virtual que puede ser usada en cualquier ordenador físico permitiendo así la emulación de cualquier

LISTADO I.
INSTRUCCIONES PARA RECONOCER EL CANAL DE COMUNICACIÓN DE VMWARE ENTRE MÁQUINAS VIRTUALES Y ANFITRIONA.

```
mov eax, 564D5868h ; VMXh
mov ebx, 0
mov ecx, 0Ah
mov edx, 5658      ; VX
in eax, dx         ; VMWare detectado
                   ; si ebx = 564D5868h
```

sistema operativo sobre esa capa virtual. En esta categoría, hemos distinguido principalmente entre técnicas anti-sandbox y anti-virtualización. Las técnicas de anti-sandboxing se usan para evadir el análisis realizado dentro de una sandbox. Existen diversas herramientas sandbox comerciales y libres basadas en el sistema operativo Windows, como Sandboxie [40], JoeBox [28], CWSandbox [15], Cuckoo Sandbox [14], o PyBox [17], por nombrar algunos. Normalmente, estos entornos se basan en un sistema operativo Windows modificado para registrar cualquier acción producida durante la ejecución de un binario. Así, estos entornos presentan algunas características (o huellas) que permiten reconocerlos fácilmente. Por ejemplo, JoeBox, CWSandbox, o Anubis [6] pueden ser reconocidos y así, evadidos, mediante la búsqueda de una clave de producto de Windows concreta en el registro, por nombres de usuario conocidos, o por la existencia de ventanas abiertas con manejadores conocidos. Del mismo modo, la presencia de una librería externa cargada junto con el proceso activo de nombre "*SbieDll.dll*" indica que la sandbox Sandboxie se está ejecutando.

Las técnicas de anti-virtualización permiten evadir la ejecución dentro de una máquina virtual. Estas técnicas buscan características únicas en la memoria (e.g., procesos o servicios activos), en partes del sistema (e.g., registro de Windows, controladores de dispositivos, otros identificadores de hardware), o hace uso de instrucciones de ensamblador para poder reconocer estructuras del sistema específicas que son diferentes (sus valores) respecto a una máquina física. Por ejemplo, la tabla de descriptores de interrupciones (IDT, por sus siglas en inglés), que controla la respuesta a interrupciones hardware y excepciones, está localizada en direcciones de memoria diferentes cuando el sistema operativo está ejecutándose en una máquina virtual que cuando se está ejecutando en una máquina física. Del mismo modo, la tabla de descriptores locales (LDT, por sus siglas en inglés) y la tabla de descriptores global (GDT, por sus siglas en inglés) también se encuentran localizadas en diferentes zonas de memoria. Algunas instrucciones de ensamblador pueden usarse también para reconocer la presencia de máquinas virtuales, debido a implementaciones incorrectas (e.g., la instrucción `cmpxchg8b` [20]) o porque no se usan en la

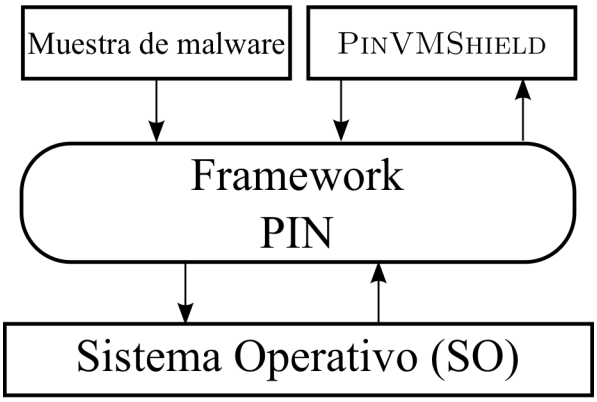

Figura 2. Arquitectura de la herramienta PINVMSHIELD.

máquina real. Por ejemplo, VMware usa las instrucciones que se muestran en el Listado 1 para conocer el canal de comunicación que se establece entre la máquina anfitriona y la máquina invitada (virtual). Así, estas instrucciones sirven para reconocer un entorno de VMware, consultando el valor de los registros de la CPU tras su ejecución.

C. Reconocimiento de análisis de la memoria

Esta categoría engloba las técnicas que tratan de evitar que la memoria de un proceso sea leída y/o volcada a disco total o parcialmente (una zona definida). Normalmente, el malware incorpora estas técnicas con el fin de prevenir que un analista acceda a su zona de memoria y pueda leer tanto su código binario como sus datos, ocultando así su comportamiento malicioso. Esta categoría se puede refinar en las subcategorías de volcado y reconocimiento de compilador o protector.

Las técnicas de anti-volcado evitan la lectura de memoria y posterior volcado (es decir, escritura a fichero de una zona de memoria). Normalmente usadas en Windows, estas técnicas hacen uso o bien de características propias de las cabeceras de los ejecutables de Windows, o eliminan el mapeado de la zona de memoria del proceso, o usan código intermedio que ha de ser interpretado por un compilador JIT. Nótese que el volcado de un proceso de memoria a disco es uno de los pasos necesarios para eliminar los protectores de binarios [48]. Actualmente, casi el 90% de las muestras de malware se distribuye de alguna forma protegida u ofuscada [29], [51].

Las técnicas de anti-reconocimiento de compilador o de protección permiten identificar un compilador o software de protección en base a ciertas firmas. Conocer de antemano el software de protección o el compilador de un binario permite determinar la metodología de análisis a utilizar. Estas técnicas normalmente realizan ciertas mezclas de las instrucciones iniciales de un binario con el fin de engañar a las herramientas de detección basadas en firmas.

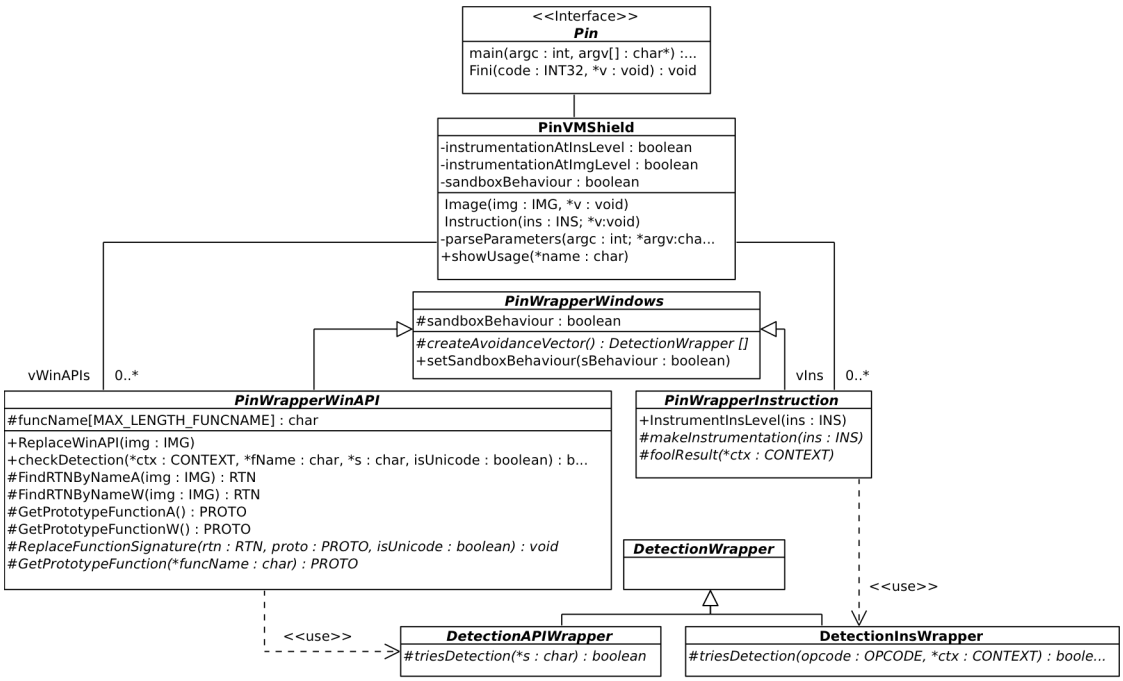

Figura 3. Diagrama de clases de PINVMSHIELD.

III. DESARROLLO DE UNA HERRAMIENTA PARA ANULAR TÉCNICAS DE EVASIÓN DE ANÁLISIS BASADAS EN DETECCIÓN DE ENTORNOS AISLADOS

En este artículo, nos hemos enfocado en muestras de malware que implementan técnicas de evasión de análisis basadas en la detección de entornos aislados. En esta sección presentamos la herramienta `PinVMShield`, desarrollada especialmente para anular este tipo de técnicas.

La herramienta hace uso de técnicas DBI para insertar código arbitrario en ciertas partes de la aplicación durante su ejecución. Nótese, sin embargo, que esta aproximación introduce una sobrecarga en términos de tiempos de ejecución y consumo de memoria. Existen diversos *frameworks* DBI en el mercado, como Pin [33], Valgrind [37] o DynamoRIO [10]. En este artículo se ha escogido Pin dado que tiene la menor sobrecarga en binarios de uso intensivo de CPU [39]. A continuación, se introduce brevemente el framework Pin [33] y después se presenta en detalle la arquitectura de la herramienta propuesta.

A. El framework DBI Pin

El *framework* DBI Pin (o simplemente Pin, para abreviar) posibilita escribir de forma sencilla herramientas software de instrumentación eficientes, portables y transparentes. Las herramientas desarrolladas con Pin se denominan *Pintools*. Una herramienta software de instrumentación permite realizar tareas como *profiling*, evaluación del rendimiento y detección de errores. Pin usa un compilador JIT para insertar y optimizar el código que se va a ejecutar. Dicho código compilado contiene no sólo el código original sino también el código de la instrumentación. La característica más relevante es que el código original de la aplicación nunca llega a ejecutarse. Únicamente se ejecuta el código que, almacenado en la cache de código, genera el compilador JIT después de la instrumentación.

Pin proporciona niveles distintos de granularidad:

- **Imagen**, que permite a Pin inspeccionar e instrumentar la imagen de un binario completo cuando se carga por primera vez.
- **Traza**, donde esta empieza normalmente al principio de una nueva rama y termina con una rama incondicional.
- **Rutina**, la cual permite a Pin instrumentar una rutina antes de que sea llamada.
- **Bloque**, definido como una secuencia de instrucciones con una sola entrada y una sola salida.
- **Instrucciones**, donde una instrucción concreta puede ser instrumentada.

Pin es usado ampliamente en la comunidad científica, publicándose numerosas *Pintools* [32], [38].

B. La herramienta PinVMShield

La Figura 2 muestra la arquitectura de `PinVMShield`. Como se observa, Pin interactúa directamente con el sistema operativo. A su vez, el binario instrumentado (la muestra de malware, en este caso) es gestionada por el Pin, mientras que nuestra herramienta indica a Pin qué parte del código está instrumentado (*puntos de instrumentación*) y qué código debería ser ejecutado en estos puntos de instrumentación (*código de instrumentación*). Actualmente, nuestra herramienta está enfocada al sistema operativo Windows dado que este sistema operativo es el objetivo de muchas muestras de malware actuales, aunque podría ser extendida para cubrir otros sistemas operativos soportados por Pin (e.g., Linux, o Android).

`PinVMShield` está diseñado siguiendo una arquitectura de *plug-ins* que nos permite extenderla fácilmente para cubrir nuevas técnicas de evasión. Nuestra herramienta puede instrumentar con dos niveles de granularidad: a nivel de rutina y a nivel de instrucción.

La Figura 3 muestra un extracto del diagrama de clases UML de `PinVMShield`. Se ha usado el patrón de software arquitectural *Facade* para abstraer la complejidad del sistema. Cada nivel de granularidad usa dos clases abstractas: *PinWrapperWinAPI* con *DetectionAPIWrapper*; y *PinWrapperInstruction* con *DetectionInsWrapper*.

PinWrapperWinAPI y *PinWrapperInstruction* definen la instrumentación que se ha de realizar a nivel de la API de Windows y a nivel de instrucción, respectivamente. Se pueden extender de forma sencilla para interceptar una nueva función o instrumentar una instrucción determinada. Las otras clases, *DetectionAPIWrapper* y *DetectionInsWrapper*, extienden las capacidades de detección de técnicas de evasión a nivel de la API de Windows y a nivel de instrucción, respectivamente.

A continuación, describimos brevemente cómo extender nuestra herramienta para poner de manifiesto cuán fácil es. Considérese a un analista de malware interesado en extender las capacidades de `PinVMShield` a nivel de la API de Windows. Los pasos para extender nuestra herramienta serán los siguientes: Primero, se crea una clase concreta extendiendo *DetectionAPIWrapper*; y segundo, se implementa el método *triesDetection* (método abstracto) el cual recibe una cadena y devuelve un valor booleano que indica si la cadena se usa como patrón de reconocimiento (la implementación son cerca de 12 líneas de código, a las que hay que añadir la lógica para del método de comparación de la cadena). Por ejemplo, para reconocer el software de virtualización comercial Parallels [1] el cuerpo del método de esta nueva clase puede comprobar, sin distinguir entre mayúsculas o minúsculas, cuando la cadena de entrada contiene la palabra "*parallel*".

Del mismo modo, cuando un analista de malware quiere instrumentar una nueva llamada al sistema de Windows para anular una técnica de detección, debe crear una clase concreta extendiendo *PinWrapperWinAPI* e implementar el método de la clase abstracta. Finalmente, también es necesario implementar una nueva función que invalide la llamada al sistema (la implementación se acerca a las 40 líneas de código, además de la lógica necesaria para sustituir la API de Windows).

Recuérdese que nuestra herramienta actúa sobre técnicas de evasión que emplean las funciones API de Windows y una cadena como parámetro, dado que la mayoría de las técnicas de evasión basadas en API documentadas se basan en comprobar los parámetros de entrada/salida frente a un conjunto de cadenas bien conocidas.

Sin embargo, la arquitectura de `PinVMShield` también permite extender la herramienta para cubrir otras APIs de Windows que no se basan en cadenas. Por ejemplo, la función *GetCursorPos* (devuelve la posición del cursor en pantalla) puede ser usada para el reconocimiento de un entorno de análisis automatizado obteniendo valores con llamadas consecutivas y comprobando si la posición del cursor ha variado. A pesar de que este método puede devolver falsos positivos, es comúnmente utilizada por algunas muestras de malware que esperan algún tipo de interacción humana con la máquina a infectar. Así, se puede interceptar las llamadas a la función *GetCursorPos* y devolver valores aleatorios como coordenadas para evitar la detección.

La versión actual de `PinVMShield` evita el reconocimiento de entornos virtuales (en concreto, VirtualPC, VMWare y Virtualbox), depuradores (WinDBG, OllyDBG e ImmunityDebugger) y sandboxes (WinJail, Cuckoo Sandbox, Norman, Sandboxie, CWSandbox, JoeSandbox y Anubis). Las funciones APIs de Windows interceptadas por nuestra herramienta son las siguientes (en ambas versiones ASCII y UNICODE):

- Gestión de ficheros: *CreateFile*, *GetFileAttributes*.
- Gestión del registro: *RegOpenKey*, *RegOpenKeyEx*, *RegQueryValue*, *RegQueryValueEx*.
- Manejo de tuberías: *CallNamedPipe*, *WaitNamedPipe*, *PeekNamedPipe*.
- Gestión de procesos: *GetModuleHandle*, *GetModuleHandleEx*, *FindWindow*, *FindWindowEx*, *Process32First* y *Process32Next*.
- Comparación de cadenas: *lstrcmp*, *CompareString*, *CompareStringEx*.
- Otros datos: *GetUserName*, *GetUserNameEx*.

`PinVMShield` también trata el reconocimiento de entornos virtuales usando artefactos en memoria como se explica en la Sección II.B. Por ejemplo, las instrucción `sidt`, `sgdt` y `sldt` que devuelven respectivamente las direcciones de las tablas IDT, GDT y LDT, son detectadas por la herramienta. Recuérdese que estas direcciones tienen valores distintos dependiendo del entorno en el que se ejecuta el sistema operativo. De igual forma, nuestra herramienta anula la detección del canal de comunicación de VMWare mediante la instrucción `in`.

El nivel de instrumentación de `PinVMShield` se puede cambiar durante la inicialización a través de unos valores de entrada. Por defecto, `PinVMShield` instrumenta a nivel de API de Windows. Durante la ejecución, se crea un fichero de registro llamado *executionLog.pvs* que contiene los parámetros de entrada y el lugar donde se ha detectado cada técnica de evasión.

Otro parámetro permite registrar explícitamente en un fichero (llamado *fullLog.pvs*) cada una de las instrucciones o llamadas a la API de Windows que podrían ser usadas potencialmente para reconocimiento. Este parámetro puede ser útil para modificar manualmente la muestra de malware con el fin de evadir siempre estas técnicas si fuera necesario. Finalmente, la clase *PinVMShieldDetection* es la encargada de anular la detección de nuestra herramienta (por ejemplo, un binario podría buscar si la biblioteca "*PinVMShield.dll*" está cargada, funciones exportadas o nombres de ficheros, entre otros).

Finalmente, queremos remarcar que nuestra herramienta se basa en técnicas de detección de entornos de análisis mediante métodos directos. Sin embargo, el malware también usa métodos indirectos para evadir estos entornos de análisis, como tiempos de espera muy largos, bucles, disparadores externos que comienzan la ejecución o ejecución diferida creando tareas programadas. Nótese que nuestra herramienta se centra en las técnicas de detección de entorno de análisis detalladas previamente, dejando estas técnicas indirectas fuera del ámbito de aplicación. Se plantea continuar la investigación para proveer una solución integrada que cubra todos estos problemas.

`PinVMShield` está distribuida bajo la licencia GNU GPL versión 3, y está disponible para descarga en:

https://bitbucket.org/rjrodriguez/pinvmshield/

C. Detección de la Instrumentación Dinámica de Ejecutables

En la Sección A se ha hecho una revisión en profundidad de cómo un binario puede reconocer si está siendo ejecutado por un framework DBI. Un resumen de cuáles de estas técnicas de detección de DBI están cubiertas por `PinVMShield` se muestra en la Tabla I. Nótese que el reconocimiento a través de la instrucción `SYSENTER` no es aplicable en este caso, ya que afecta a versiones anteriores a la versión de Pin que se ha usado para los experimentos (Pin versión 62141 de 2013).

Como trabajo futuro, se pretende extender la herramienta para reconocer la detección mediante las técnicas no cubiertas actualmente.

IV. Casos de Estudio

En esta sección comprobamos la eficiencia de `PinVMShield` con tres casos de estudio. En primer lugar, hemos considerado un software especialmente diseñado para detectar entornos aislados. En segundo lugar, hemos estudiado una muestra de malware consciente de su entorno en detalle. Por último, se han analizado un conjunto de muestras de malware consciente que ponen en evidencia las fortalezas y debilidades actuales de nuestra herramienta. En todos los casos, `PinVMShield` se ha ejecutado para evitar tanto las técnicas de detección basadas en funciones API como las basadas en instrucciones simples, con un registro de actividad total (véase la Sección III.B).

TABLA I.
Soporte de `PinVMShield` de contramedidas de DBI.

Técnica de reconocimiento de entorno DBI	¿Actualmente soportada por `PinVMShield`?
Detección de Pin VM	
Patrones de cadenas	✓
Patrones de código	✖
Funciones exportadas	✓
Nombres de secciones	✓
Detección de tuberías de comunicación	
Inspección de manejadores	✓
Detección de variaciones de tiempo	
Sobrecarga introducida por Pin	✖
Detección de compilador JIT de Pin	
Sobreescritura de funciones de Windows	✖
Permisos de las páginas de memoria	✖
Detección por valor del puntero de siguiente instrucción (registro EIP)	
Entorno de contexto FPU	✓
APIs de Windows (*VirtualQuery*)	✖
Detección por otras técnicas	
Parámetros de entrada del programa	✓
Jerarquía de procesos	✖
Emulación instrucción `SYSENTER`	(n/a)

(n/a): No aplicable (en este caso)

Como máquinas de pruebas se han usado una máquina anfitriona con un procesador Intel Core i7 a 2GHz y 8GiB 1600MHz DDR3 de memoria RAM, ejecutando VirtualBox. Como máquina invitada, se ha ejecutado un sistema operativo Windows XP SP3, con Pin versión 2.13-62141 VC9 instalada. Después de cada ronda de ejecución, los resultados del experimento se han recogido del fichero de registro creado por `PinVMShield`, siendo además el disco duro de la máquina virtual restaurado a una copia limpia, eliminando así cualquier modificación realizada por las muestras de malware.

Con el fin de que los experimentos sean reproducibles, tanto la versión de la herramienta `Pafish` como las muestras de malware conscientes usadas en este estudio se encuentran disponibles para su descarga en:
http://webdiis.unizar.es/~ricardo/software-tools/pinvmshield

En la misma página web también se pueden encontrar disponibles las diferentes versiones de `PinVMShield`, ya compiladas.

A. Herramienta `Pafish`

`Pafish` es una herramienta de código abierto para la plataforma Windows que incorpora las técnicas más comunes y conocidas para el reconocimiento de ejecución dentro de un entorno virtual o herramienta de sandbox. En concreto, en este artículo se ha usado `Pafish` en su versión 0.25, siendo su valor hash MD5 `7662cb4b1abc4ccb30b3682acc3dae24`. Tanto el código fuente como el binario de la herramienta se pueden descargar de la página web https://github.com/a0rtega/pafish.

Como se ha comentado, este software incorpora diversas técnicas de reconocimiento de software de depuración, de virtualización o de sandbox. Cuando reconoce algún entorno, crea un fichero vacío, indicando en su nombre la detección positiva realizada. En concreto, reconoce los siguientes elementos:

- Un software de depuración es detectado mediante las funciones de Windows *IsDebuggerPresent*, *CheckRemoteDebuggerPresent* y *OutputDebugString*.
- Una herramienta de sandbox es detectada usando diferentes métodos: mediante el nombre del usuario de la máquina (comprobando si este nombre coincide con “SANDBOX”, “VIRUS”, o “MALWARE”); o mediante la ruta del fichero (comprobando si este nombre coincide con “SAMPLE”, “VIRUS”, o “SANDBOX”).
- El software de emulación Wine es detectado mediante la comprobación de existencia de una

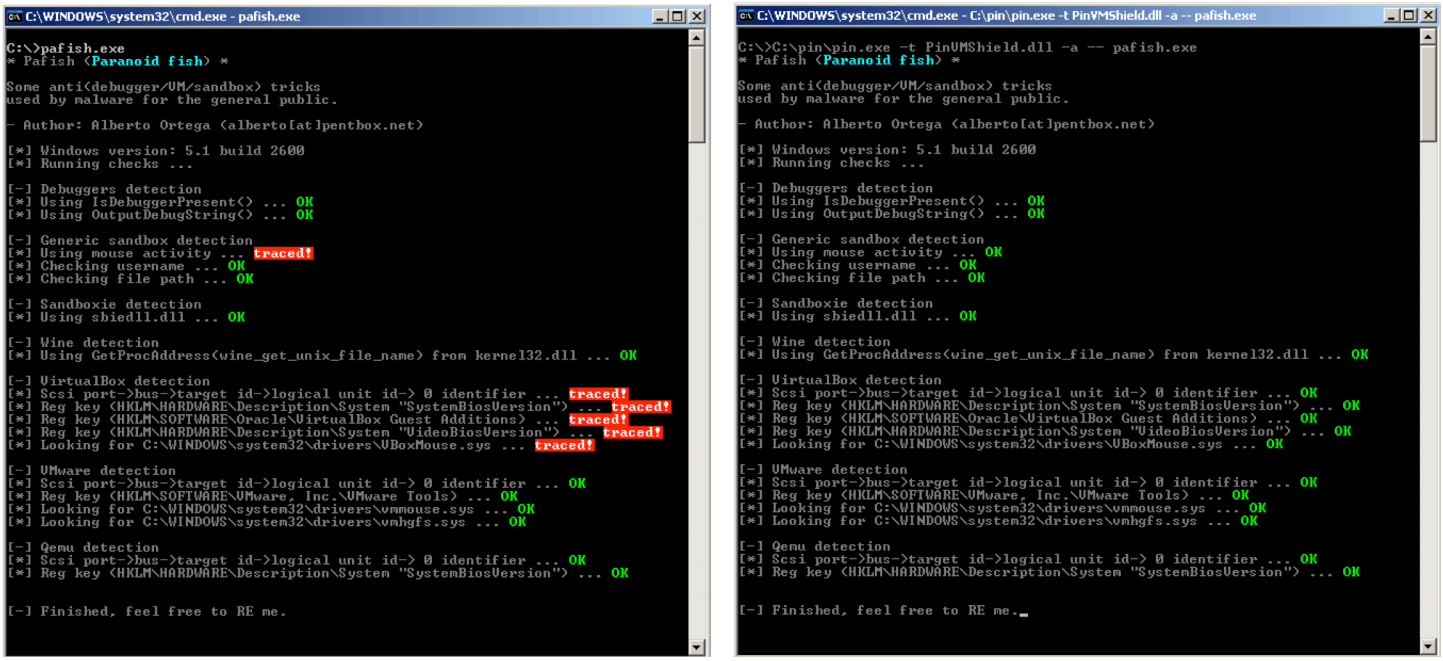

Figura 4. Ejecución de `Pafish` en la máquina virtual de forma aislada (izquierda), e instrumentada por `PinVMShield` (derecha).

función de nombre *wine_get_unix_file_name*. Esta función es única de la librería "*kernel32.dll*" que incorpora Wine.

- QEMU [7] se reconoce mediante la consulta en el registro de Windows de ciertas claves de registro específicas. En particular, se consultan las claves relativas a la versión de BIOS del sistema y al controlador del puerto SCSI. Un valor conteniendo "QEMU" en estas claves indica que es altamente probable que el sistema se esté ejecutando sobre un entorno QEMU.
- La herramienta de sandbox Sandboxie se reconoce mediante la identificación de un módulo cargado en el espacio de memoria del proceso con el nombre "*sbiedll.dll*".
- Virtualbox también es reconocible a partir de claves del registro de Windows. Concretamente, es posible identificarlo mediante claves relativas al controlador del puerto SCSI, a la versión de BIOS del sistema y a la de vídeo. También la instalación de herramientas adicionales de VirtualBox (conocidas como "VirtualBox Guest Addition") dejan rastros en el registro y en los controladores del sistema (por ejemplo, el controlador del ratón) que identifican a la máquina como un entorno virtualizado.
- El entorno virtual VMWare se puede reconocer similarmente al entorno Virtualbox. Del mismo modo, la instalación de herramientas adicionales ("VMWare tools" en este caso) deja indicios tanto en el registro como en los controladores del sistema (e.g., "*vmmouse.sys*" and "*vmhgfs.sys*").

En este caso de estudio, hemos ejecutado `Pafish` en la máquina virtual de dos maneras diferentes. En primer lugar, se ha ejecutado de manera aislada. En segundo lugar, se ha ejecutado de manera instrumentada con nuestra herramienta desarrollada (véase la Sección III.B). Dado que esta herramienta se desarrolló para detectar de manera explícita los entornos virtuales, esperamos observar diferencias entre ambas ejecuciones.

Unas capturas de pantalla de ambas ejecuciones se muestra en la Figura 4. Como se puede observar, la ejecución de manera aislada (izquierda) detecta de manera positiva un entorno virtual (el mensaje "traced!", sobre fondo rojo, indica una detección de entorno de análisis). Sin embargo, la ejecución instrumentada no levanta ninguna alarma.

Del mismo modo, se han comprobado el número de ficheros creados, el número de ficheros accedidos en lectura/escritura, el número de claves de registro de Windows creadas o accedidas para lectura o escritura, y el número de procesos creados o accedidos por el programa a analizar. Todos estos valores se han recogido de manera automática mediante la ejecución desatendida de `PinVMShield` dentro del entorno de sandbox de Cuckoo Sandbox (versión 0.6) [14], una herramienta de código abierta ampliamente usada por la comunidad de análisis de software malicioso [23][45][46].

Durante la ejecución instrumentada se crea y se accede a un menor número de ficheros. Esto se debe a que, como se explicó anteriormente, `Pafish` crea un fichero para avisar sobre la detección positiva de algún entorno de análisis. El resto de parámetros estudiados es idéntico en ambas ejecuciones.

En resumen, los resultados evidencian que `Pafish` es incapaz de reconocer el entorno de análisis aislado cuando se ejecuta mediante `PinVMShield`.

B. Análisis destallado de una muestra de malware consciente del entorno de análisis

En este caso de estudio analizamos en detalle una muestra real de software malicioso consciente del entorno de análisis usando la misma metodología y entorno de análisis que en el

TABLA II.
RESULTADOS DE ANÁLISIS DE LA MUESTRA MALICIOSA `4FAFAEC2A6ED080FC5D8E28657D59E1`. EN GRIS SE MUESTRAN LAS DIFERENCIAS OBSERVADAS DURANTE LA EJECUCIÓN INSTRUMENTADA CON `PinVMShield`.

Ficheros creados
FriendsKeep.txt
iexplorer
Ficheros accedidos
C:\FriendsKeep.txt
SICE\xe5\xf4\x85\xe3\xee\xa0\xe8\xbb\x9do
SIWVIDNTICE}
NTICE}SICE
ICEEXTSIWVID
SYSERBOOT\xe3\xee\xa0ICEEXTSI
%LOCALAPPDATA%\iexplorer*
%LOCALAPPDATA%\iexplorer
%LOCALAPPDATA%\Google\Chrome\User Data\Default\Extensions\fckenojfmfijmbkigoajddgondmfhefd*
%LOCALAPPDATA%\Google\Chrome\User Data\Default\Extensions\fckenojfmfijmbkigoajddgondmfhefd
%LOCALAPPDATA%\Mozilla\FireFox\profiles.ini
C:\onewebsearch
GeniusXXAddon*
GeniusXXAddon
onewebsearch*
C:\onewebsearch
Claves de registro
HKLM\System\CurrentControlSet\Services\Tcpip\Parameters
HKLM\System\CurrentControlSet\Services\DnsCache\Parameters
HKLM\Software\Policies\Microsoft\Windows NT\DnsClient
HKLM\Software\Policies\Microsoft\System\DNSClient
HKLM\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0
HKLM\HARDWARE\Description\System
HKLM\SOFTWARE\Oracle\VirtualBox Guest Additions
HKLM\SOFTWARE\VMware, Inc.\VMware Tools
HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders
HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders
HKLM\Software\Microsoft\Windows\CurrentVersion

caso anterior. La firma MD5 de esta muestra es, en concreto, `4fafaec2a6ed080fc5d8e28657d59e10`.

Esta muestra ha sido seleccionada tras comprobar empíricamente que usa técnicas para detección de entornos de análisis aislado basadas en el registro, en los procesos y en los ficheros. Sorprendentemente, esta muestra maliciosa es indetectable por el conjunto de anti-virus de VirusTotal; sin embargo, un informe completo y detallado mostrando claramente su actividad perniciosa se puede consultar en [3].

El comportamiento de esta muestra maliciosa, basándose en los nombres de ficheros y las claves de registro creadas/accedidas, coincide con el comportamiento del malware identificado como Adware:Win32/Voontera[1]. Esta muestra de malware instala una barra adicional en el navegador web, permitiéndole mostrar anuncios de manera frecuente durante una navegación por Internet, la instalación de otras barras adicionales en el navegador e incluso la modificación de la configuración del propio navegador web.

Siguiendo la metodología del caso anterior, se ha analizado la muestra sin instrumentar e instrumentada con nuestra herramienta. La Tabla II muestra los resultados del análisis. En concreto, se detallan los ficheros y claves de registro creadas o accedidas en los dos escenarios. Se ha destacado con un color gris aquella actividad observada únicamente durante la ejecución instrumentada con `PinVMShield`. Del mismo modo, se ha omitido de esta tabla de resultados la actividad observada perteneciente a nuestra herramienta. Por brevedad, se han abreviado las rutas de carpetas comunes. En concreto, "%LOCALAPPDATA%" significa "*C:\ Documents and Settings\{usuario Windows}\Local Settings\Application Data*".

Los resultados muestran una mayor actividad durante la ejecución instrumentada con `PinVMShield`: el número de ficheros creados, así como el de ficheros y claves de registro accedidos, es mayor durante la ejecución instrumentada que durante la ejecución aislada. Este hecho indica claramente `PinVMShield` es efectivo en ocultar el entorno de análisis, ya que la muestra es consciente del mismo, realizando poca o nada de actividad cuando lo detecta. Como ya se ha descrito, durante la ejecución instrumentada la muestra maliciosa no detecta el entorno de análisis, empezando así su proceso de infección de la máquina donde se está ejecutando.

C. Conjunto de muestra de malware conscientes del entorno

Por último, hemos recogido de la literatura [50], de un repositorio comunitario de malware (https://zoo.mlw.re) y de investigadores de seguridad muestras de familias de malware conocidas por incorporar algunas técnicas de detección de entornos de análisis.

Estas muestras se han ejecutado instrumentadas por nuestra herramienta de manera manual en la máquina virtual creada para los experimentos. Los resultados de los experimentos se han recogido del fichero de registro creado por `PinVMShield` (véase la Sección III.B). De las 30 muestras analizadas, 8 de ellas se han ejecutado de manera incorrecta. En concreto, estas muestras pertenecen a las familias de malware Agobot y Rbot. Únicamente dos de ellas han mostrado un error de ejecución de Pin, relativo a liberaciones de zonas de memoria inesperadas o la transferencia del control

[1] Para saber más sobre esta muestra en particular, se recomienda la lectura de http://www.enigmasoftware.com/adwarewin32vonteera-removal/.

TABLA III.
CONJUNTO DE MUESTRAS DE SOFTWARE MALICIOSO REAL ANALIZADAS CON `PinVMShield`.

Hash MD5	Etiqueta identificadora (según Kaspersky)	Ratio de detección (según VirusTotal)	Técnicas de detección de entorno aislado detectadas
0b8b2c0926630c69a6c75bba67b24a3e	Trojan-Proxy.Win32.Bypass.a	44 sobre 55	—
106a1be7d04cab37b21af1a8d9c743d9	(no detectado)	1 sobre 57	Registro
14d294fbfef36c063b96fcfb0d849d46	Trojan-Spy.MSIL.Zbot.btc	49 sobre 57	Registro
2c1a7509b389858310ffbc72ee64d501	HEUR:Trojan.Win32.Generic	48 sobre 56	Registro
36e5fdcdbe0bcdc59ea001b162bfb97d	Net-Worm.Win32.Kolab.aefe	53 sobre 55	—
4fafaec2a6ed080fc5d8e28657d59e10	(no detectado)	0 sobre 54	Registro, ficheros, y procesos
687a06131feb8ba95ba5a27ef2450e1d	Trojan-Spy.Win32.Zbot.trfx	50 sobre 57	—
6d721c5980e1fc26ef92b3f0746681c5	Trojan-Spy.Win32.Zbot.sbdq	52 sobre 56	Registro
7b9ef183ea33387c8dbc3997f70cc5fa	HEUR:Trojan.Win32.Generic	50 sobre 56	Registro
7ce6cd9837e1a7837c2b491c21ff5b69	Backdoor.Win32.Agobot.nq	43 sobre 55	—
8863d38db188796e32c822dcc42a82ae	Trojan-Spy.Win32.Zbot.gen	51 sobre 57	—
bb42fce5d9cb73561ec4e3c343c10d52	Backdoor.Win32.Androm.drzc	40 sobre 55	Registro
bdcd4ff82d6894156b945c96ac45b9ec	HEUR:Trojan.Win32.Generic	45 sobre 56	—
c1a66699820fdeb7242e884e6d2f8bcb	Backdoor.Win32.Rbot.gen	47 sobre 55	—
ce5c86fb4c44a7655ed6caaf42a688b3	HEUR:Trojan.Win32.Generic	35 sobre 54	Registro
d062d420e2ac73b0211afe30063807fa	Backdoor.Win32.Androm.bkei	43 sobre 56	—
dab012115fa267d95c1145a1eb41d38d	Trojan.Win32.Badur.igah	40 sobre 54	Registro
dabec78d489f1e783fb23d6e726bd1a4	Backdoor.Win32.Rbot.gen	46 sobre 54	—
e6ea45deca7e9dd9afeb276ec1d4509c	HEUR:Trojan.Win32.Generic	39 sobre 55	Registro y procesos
ef0458e196fbd1b4cc1613ba2ca3c43b	Backdoor.Win32.Rbot.gen	48 sobre 55	—
f01cdf6e5052aeb5c6510bd8f8d88636	Backdoor.Win32.Agobot.aiw	46 sobre 55	—
f085395253a40ce8ca077228c2322010	Backdoor.Win32.Zegost.abzh	43 sobre 56	Registro

(—): No detectadas

de flujo a códigos de segmento diferentes. El análisis de las otras seis restantes fue parado tras cinco minutos. Empíricamente comprobamos que estas muestras estaban, además, consumiendo CPU sin realizan ninguna actividad aparente. Por lo tanto, creemos que estas muestras estaban realizando algún tipo de evasión indirecto, como bucles de consumo de tiempo, procesos durmientes, o ejecución activada por eventos; los cuales están fuera del ámbito de este trabajo. De las restantes 22 muestras, la mitad de ellos mostró una o más técnicas de evasión de análisis detectadas por nuestra herramienta; mientras que en la otra mitad no se detectó ninguna.

El conjunto final de muestras analizadas se resume en la Tabla III. Para cada una, se muestra su firma MD5, su etiqueta identificadora según el anti-virus Kaspersky, la ratio de detección de VirusTotal (a fecha de 21 de agosto de 2015) y las técnicas de detección de entorno aislado de análisis descubiertas por `PinVMShield`. Un "—" en la última columna indica que no se ha detectado ninguna técnica por parte de nuestra herramienta.

Los resultados muestran que la mayoría de muestras usan técnicas de reconocimiento de análisis basadas en el registro de Windows, es decir, hacen uso de las funciones habituales de Windows para manejo de registro como *RegOpenKey*, *RegOpenKeyEx*, *RegQueryValue*, y *RegQueryValueEx*. Sorprendentemente, sólo dos muestran usan técnicas relativas al manejo de procesos, y únicamente una muestra hace uso de técnicas relacionadas con registro, procesos y ficheros de manera conjunta. Conviene mencionar que ninguna de estas muestras hacía uso de técnicas de detección basadas en instrucciones (véase la Sección III.B). Por último, hay que destacar que `PinVMShield` no ha detectado ninguna actividad de evasión en algunas de las muestras. Esto se debe a que o bien las muestras hacen uso de algunas de las técnicas no cubiertas actualmente por `PinVMShield` (por ejemplo, el uso de tráfico de red para detectar desviaciones temporales), o bien porque usan otras técnicas de detección de entorno de análisis, como la detección de ejecución bajo depurador (es decir, son muestras conscientes del análisis binario; véase la Sección II.A).

En nuestra opinión, las muestras de malware conscientes irán en aumento en los próximos años hasta convertirse en una característica común. Nótese que cuanto más tiempo permanece indetectable una muestra de malware, más tiempo está generando beneficio el negocio criminal. Así, creemos firmemente que las muestras de malware conscientes del análisis empezarán a usar cada vez más las técnicas conocidas para detección de entornos aislados, así como otras técnicas actualmente desconocidas. Por ello, son necesarias nuevas alternativas de análisis de malware, como el uso de técnicas DBI que hemos propuesto en este trabajo. Pretendemos seguir estudiando este tipo de malware y anticipar las posibles técnicas de detección que usen.

Aunque los resultados iniciales son prometedores, hemos encontrado muchas limitaciones en nuestra aproximación, principalmente derivadas por el uso del framework DBI de Pin [33]. Por ejemplo, Pin maneja incorrectamente liberaciones de memoria no esperadas o transferencias del control de flujo a códigos de segmento diferentes. Empíricamente, hemos estudiado el comportamiento dinámico de las muestras que fallaban en su ejecución en este caso de estudio; y encontrado que incorporaban numerosas capas de descifrado de código como técnica de evasión de análisis del binario. Consideramos que esta característica puede estar siendo manejada incorrectamente por Pin, aunque merece un estudio posterior.

V. TRABAJO RELACIONADO

CWSandbox [15] es una herramienta de ejecución aislada diseñada bajo tres criterios principales: Automatización, efectividad y corrección [49]. Permite desarrollar análisis dinámico de muestras de software malicioso, pero no permite instrumentarlo. Por ello, algunas de las técnicas de evasión

descritas en la Sección III detectan el entorno de análisis gestionado por CWSandbox. Del mismo modo, una herramienta que integra la ejecución en una sandbox junto con técnicas de DBI se describe en [4]. En este caso, los autores usan el entorno de instrumentación de Pin y definen dos entornos de ejecución: testeo y entorno real. En el caso de testeo se refieren a un entorno donde la ejecución del código binario es traceado; siendo luego estas trazas comparadas con respecto a varias reglas de seguridad. En el caso del entorno real, el código binario en ejecución es monitorizado y se impide cualquier tipo de comportamiento prohibido.

Mucho mas cercano a nuestro trabajo y por ello de obligada referencia esta VMDetectGuard [31], [47]. Es un software desarrollado para enmascarar el uso de máquinas virtuales (como son VMWare, VirtualPC, o Virtualbox) y evitar así que sean detectadas. Para ello, también usa el entorno de Pin. Sin embargo, este software, a diferencia del nuestro, se centra en el análisis de muestras de software malicioso que usan las funciones de sistema de Windows con formato ASCII. Además, nuestra herramienta ha sido diseñada de forma que sea fácilmente extensible, y no sólo evita técnicas de detección de máquinas virtuales, sino también la detección de entornos de ejecución aislada. Divergence Detector [25] es otro entorno software diseñado para detectar software malicioso consciente de ser ejecutado en máquinas virtuales. Esta herramienta compara la traza de ejecución de una muestra en tres plataformas diferentes (QEMU, Bochs, y XEN) y detecta cualquier desviación de comportamiento entre las trazas. Otra herramienta a tener en cuenta es VMscope [27], desarrollada especialmente para ser integrada en QEMU [7] y que provee un sistema de monitorización para *honeypots* basado en un entorno virtual capaz de inspeccionar e interpretar eventos internos del sistema. Nuestro objetivo es diferente, aunque nuestra herramienta también es capaz de inspeccionar e interpretar (a nivel de usuario) eventos internos del sistema ya que las muestras se ejecutan en un entorno virtualizado. Finalmente, la mayor limitación de QEMU es la dificultad de identificar las estructuras internas alojadas en memoria del sistema operativo durante el proceso de análisis, lo cual sí es posible hacer con nuestra propuesta al trabajar ésta al nivel del sistema operativo.

Es importante también mencionar otros trabajos centrados en el análisis transparente de muestras de software malicioso como pueden ser Ether [16], Secure In-VM Monitoring (SIM) [41] o V2E [50]. Ether [16] es un analizador de software malicioso basado en extensiones de virtualización hardware (como por ejemplo Intel VT) que consigue buenos niveles de transparencia, pero tiene limitaciones importantes respecto al soporte de instrumentación. SIM [41] también se basa en extensiones de virtualización hardware. Sin embargo, estas técnicas de virtualización hardware implican un importante coste en cuanto al rendimiento cuando se pretende llevar a cabo instrumentación a bajo nivel, ya que es necesario activar el modo de un solo paso (*single-step mode*) en el entorno virtualizado. Por último, V2E [50] combina virtualización por hardware con emulación por software para llevar a cabo un análisis de muestras de software malicioso de forma transparente. V2E se ha desarrollado para *Kernel-based Virtual Machine* (KVM), y la emulación por software se consigue mediante el uso de un sistema construido sobre QEMU para el análisis dinámico binario llamado TEMU [8], [43]. A diferencia de nuestra herramienta, V2E permite ser usado para detectar software malicioso a nivel de núcleo del sistema operativo.

VI. Conclusiones y trabajo futuro

El software malicioso (o malware, del inglés) se ha incrementado en número y complejidad desde la última década. Es por esto que las muestras de malware se analizan en máquinas aisladas para prevenir infecciones o malfuncionamientos de otros sistemas. En este sentido, las *sandbox* y la virtualización se han convertido en tendencia dentro del análisis de malware. Cuanto más tiempo pasa la muestra sin ser detectada, mayor es el beneficio para el criminal puesto que su mercado ilícito funciona durante más tiempo. Así, para mantener el negocio criminal, los desarrolladores de malware han empezado a incorporar código para reconocer cuándo el programa está siendo ejecutado en un entorno de análisis. Este malware se define como malware consciente del entorno de análisis (en inglés, *analysis-aware malware*). Estas muestras cambian su comportamiento o incluso paran su ejecución por completo cuando reconocen estos entornos de análisis.

En este artículo, hemos revisado las técnicas que se usan para reconocer los entornos de análisis. Centrándonos en las técnicas de reconocimiento de entornos aislados, hemos propuesto `PinVMShield`. Nuestra herramienta hace uso de la Instrumentación Dinámica de Binarios, la cual permite analizar el comportamiento en tiempo de ejecución de un binario y ejecutar código arbitrario. Hemos evaluado nuestra herramienta con muestras de malware conscientes, mostrando los resultados preliminares que las técnicas DBI son útiles para detectar y evitar este tipo de evasiones.

Obsérvese que a pesar de que hemos sustituido el problema de reconocer los entornos aislados por los *frameworks* DBI, por lo que sabemos, actualmente el malware ignora las técnicas DBI. Sin embargo, el juego del gato y el ratón nunca acaba y el malware acabará mejorando sus técnicas de evasión para tratar estos nuevos entornos de análisis.

Como trabajo futuro, se pretende extender nuestra herramienta para burlar más técnicas de detección. También se pretende integrar nuestro enfoque con entornos de análisis automatizados como Cuckoo Sandbox, Anubis o sistemas con hipervisores nativos.

Agradecimientos

Los autores quieren mostrar su agradecimiento a Alberto Ortega por su gran trabajo desarrollando la herramienta `Pafish`, así como a MLW.RE, a Stefano Zanero y a su equipo de Politecnico di Milano por facilitarnos muestras de malware consciente para los experimentos. Este trabajo ha sido parcialmente financiado por el Instituto Nacional de Ciberseguridad (INCIBE) de acuerdo con la Regla 19 del Plan de Confianza Digital (Agenda Digital de España), la Universidad de León bajo el acuerdo X43, y el proyecto MINECO CyCriSec (TIN2014-58457-R).

REFERENCIAS

[1] Parallels. [Online]. http://www.parallels.com.

[2] VirusTotal. [Online]. https://www.virustotal.com.

[3] Joe Sandbox desktop analysis report of malware sample *4fafaec2a6ed080fc5d8e28657d59e10*. [Online], January 2014. http://www.file-analyzer.net/analysis/998/5193/0/html.

[4] N. Aaraj, A. Raghunathan, and N. K. Jha, "Dynamic Binary Instrumentation-Based Framework for Malware Defense," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5137 of Lecture Notes in Computer Science, pages 64–87. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-70541-3.

[5] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, "Efficient Detection of Split Personalities in Malware," in *Proceedings of the Network and Distributed System Security Symposium* (NDSS), 2010.

[6] U. Bayer, I. Habibi, D. Balzarotti, and E. Kirda, "A View on Current Malware Behaviors," in Proceedings of the *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats* (LEET). USENIX Association, 2009.

[7] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference* (ATEC), pages 41–46, Berkeley, CA, USA, 2005. USENIX Association.

[8] BitBlaze: Binary Analysis for Computer Security. [Online]. http://bitblaze.cs.berkeley.edu/.

[9] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and AntiVM Technologies," in *BlackHat USA'12*, 2012.

[10] D. Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," PhD thesis, Massachusetts Institute of Technology (MIT), 2004.

[11] M. Carpenter, T. Liston, and E. Skoudis, "Hiding Virtualization from Attackers and Malware," *IEEE Security & Privacy*, 5(3):62–65, May 2007. ISSN 1540-7993.

[12] X. Chen, J. Andersen, Z. Mao, M. Bailey, and J. Nazario, "Towards an Understanding of Anti-virtualization and Antidebugging Behavior in Modern Malware," in *Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC* (DSN), pages 177–186, 2008.

[13] C. S. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection," *IEEE Trans. Soft. Eng.*, 28(8):735–746, August 2002.

[14] Cuckoo Sandbox. [Online]. http://www.cuckoosandbox.org/.

[15] CWSandbox. [Online]. https://www.mwanalysis.org/.

[16] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware Analysis via Hardware Virtualization Extensions.," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 51–62. ACM, 2008. ISBN 978-1-59593-810-7.

[17] M. Engelberth, J. G¨obel, C. Schönbein, and F. C. Freiling, "PyBox – A Python Sandbox,", in *Sicherheit 2012: Sicherheit, Schutz und Zuverl ässigkeit, Beiträge der 6. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V.* (GI), 7.-9.März 2012 in Darmstadt, volume 195 of LNI, pages 137–148. GI, 2012. ISBN 978-3-88579-289-5.

[18] ESET, "Global Threat Report: ThreatBlogger FootSloggers Review 2012". [Online]. http://www.eset.com/us/resources/threat-trends/Global_Threat_Trends_December_2012.pdf.

[19] F. Falcón and N. Riva, "Dynamic Binary Instrumentation Frameworks: I know you're there syping on me," in REcon, 2012.

[20] P. Ferrie, "Attacks on More Virtual Machine Emulators," in Symantec Advanced Research Threat Research, pages 1–17, 2007.

[21] FireEye, "The Dead Giveaways of VM-Aware Malware," [Online]. http://www.fireeye.com/blog/technical/malware-research/2011/01/the-dead-giveaways-of-vm-aware-malware.html.

[22] M. Gagnon, S. Taylor, and A. Ghosh, "Software Protection through Anti-Debugging," *IEEE Security & Privacy*, 5(3):82–84, may-june 2007. ISSN 1540-7993.

[23] M. Graziano, C. Leita, and D. Balzarotti, "Towards Network Containment in Malware Analysis Systems," In *Proceedings of the 28th Annual Computer Security Applications Conference* (ACSAC), pages 339–348, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1312-4.

[24] C. Greamo and A. Ghosh, "Sandboxing and Virtualization: Modern Tools for Combating Malware," *IEEE Security & Privacy*, 9(2):79–82, 2011. ISSN 1540-7993.

[25] C.-W. Hsu, F.-S. Shih, C.-W. Wang, and S. Winston, "Divergence Detector: A Fine-Grained Approach to Detecting VM Awarenes Malware," in *Proceedings of the IEEE 7th International Conference on Software Security and Reliability* (SERE), pages 80–89, June 2013.

[26] A. Issa, "Anti-virtual machines and emulations," *Journal in Computer Virology*, 8:141–149, 2012. ISSN 1772-9890.

[27] X. Jiang and X. Wang, ""Out-of-the-Box" Monitoring of VM-based High-interaction Honeypots," in *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection* (RAID), pages 198–218. Springer-Verlag, 2007. ISBN 3-540-74319-7, 978-3-540-74319-4.

[28] JoeSandbox Desktop. [Online]. http://www.joesecurity.org/joe-sandbox-desktop/.

[29] S. Josse, "Secure and advanced unpacking using computer emulation," *Journal in Computer Virology*, 3(3):221–236, 2007. ISSN 1772-9890.

[30] Kaspersky. Security Bulletin 2013. [Online]. http://media.kaspersky.com/pdf/KSB_2013_EN.pdf.

[31] A. V. Kumar, K. Vishnani, and K. V. Kumar, "Split Personality Malware Detection and Defeating in Popular Virtual Machines," in *Proceedings of the 5th International Conference on Security of Information and Networks* (SIN), pages 20–26. ACM, 2012. ISBN 978-1-4503-1668-2.

[32] G. Lueck, H. Patil, and C. Pereira, "PinADX: An Interface for Customizable Debugging with Dynamic Instrumentation," in *Proceedings of the 10th International Symposium on Code Generation and Optimization* (CGO), pages 114–123, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1206-6.

[33] C.-K. Luk, R. Cohn, R.Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation* (PLDI), PLDI'05, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. .

[34] Malwarebytes, "A Look at Malware with Virtual Machine Detection". [Online]. https://blog.malwarebytes.org/intelligence/2014/02/a-look-at-malware-with-virtual-machine-detection/.

[35] McAfee Labs, "2013 Threats Predictions". [Online]. http://www.mcafee.com/us/resources/reports/rp-threat-predictions-2013.pdf.

[36] Microsoft. Microsoft Security Intelligence Report: July through December, 2012. [Online]. http://download.microsoft.com/download/E/0/F/E0F59BE7-E553-4888-9220-1C79CBD14B4F/Microsoft_Security_Intelligence_Report_Volume_14_English.pdf.

[37] N. Nethercote and J. Seward, "Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation," *SIGPLAN Not.*, 42(6):89–100, June 2007. ISSN 0362-1340.

[38] H. Pan, K. Asanović, R. Cohn, and C.-K. Luk, "Controlling Program Execution through Binary Instrumentation," *SIGARCH Comput. Archit. News*, 33(5):45–50, December 2005. ISSN 0163-5964.

[39] R. J. Rodríguez, J. A. Artal, and J. Merseguer, "Performance Evaluation of Dynamic Binary Instrumentation Frameworks," *IEEE Latin America Transactions*, 12(8):1572–1580, December 2014. ISSN 1548-0992.

[40] Sandboxie. [Online]. http://www.sandboxie.com/.

[41] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure In-VM Monitoring Using Hardware Virtualization," in *Proceedings of the 16th ACM Conference on Computer and Communications Security* (CCS), pages 477–487. ACM, 2009. ISBN 978-1-60558-894-0.

[42] T. Shields, "Anti-Debugging – A Developers View. Technical report", *SOURCE Boston*, 2009.

[43] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security* (ISS). Keynote invited paper, volume 5352 of Lecture Notes in Computer Science, pages 1–25. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-89861-0.

[44] Symantec, "Internet Security Threat report 2013", [Online]. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf.

[45] Y.-L. Tsai, L.-Y. Yeh, B.-Y. Lee, and J.-G. Chang, "Automated Malware Analysis Framework with Honeynet Technology in Taiwan Campuses," in *Proceedings of the IEEE 18th International Conference on Parallel and Distributed Systems* (ICPADS), pages 724–725, 2012.

[46] K. Tsyganok, E. Tumoyan, L. Babenko, and M. Anikeev, "Classification of Polymorphic and Metamorphic Malware Samples based on their

Behavior," in *Proceedings of the 5th International Conference on Security of Information and Networks*, SIN '12, pages 111–116, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1668-2.

[47] K. Vishnani, A. R. Pais, and R. Mohandas, "Detecting & Defeating Split Personality Malware," in *Proocedings of the 5th International Conference on Emerging Security Information, Systems and Technologies* (SECURWARE), pages 7–13, 2011.

[48] M. Vnuk and P. Návrat, "Decompression of Run-Time Compressed PE-Files," *Studies in Informatics and Control*, 15(2), 2006.

[49] C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *IEEE Security & Privacy*, 5(2):32–39, 2007. ISSN 1540-7993.

[50] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, "V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis," in *Proceedings of the 8th International Conference on Virtual Execution Environments* (VEE), pages 227–238. ACM, 2012.

[51] W. Yan, Z. Zhang, and N. Ansari, "Revealing Packed Malware," *IEEE Security & Privacy*, 6(5):65–69, Sept 2008. ISSN 1540-7993.

Ricardo J. Rodríguez (M'13) received the M.S. and Ph.D. degrees in computer science from the University of Zaragoza, Zaragoza, Spain, in 2010 and 2013, respectively, where his Ph.D. dissertation was focused on performance analysis and resource optimization in critical systems, with special interest in Petri net modeling techniques.

He was a Visiting Researcher with the School of Computer Science and Informatics, Cardiff University, Cardiff, U.K., in 2011 and 2012, and the School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden, in 2014. He is currently an Assistant Professor at the University of Zaragoza, Zaragoza, Spain. His current research interests include performance analysis and optimization of large and distributed systems, program binary analysis, and critical infrastructures security.

Dr. Rodríguez was involved in reviewing tasks for international conferences and journals, such as IEEE Transactions on Computers, IEEE Transactions on Information Forensics and Security, and IEEE Latin American Transactions, among others.

Iñaki Rodríguez-Gastón was involved on penetration testing and incident response companies in Spain and U.K. He is currently part of the Security Operations team at King Digital Entertainment PLC and staff member of MLW.re, a non-profit organization dedicated to malware research.

Mr. Rodríguez-Gastón holds several certifications, such as *Certified Information Systems Security Professional* (CISSP), *GIAC Web Application Penetration Tester* (GWAPT), *Certified Ethical Hacking* (CEH). He is at the moment working toward the *GIAC Certified Incident Handler* (GCIH) and *GIAC Reverse Engineering Malware* (GREM) certifications.

Javier Alonso received the master's degree in Computer Science in 2004 and the Ph.D. degree from the Technical University of Catalonia (Universitat Politecnica de Catalunya, UPC) in 2011. From 2006 to 2011, he held an assistant lecturer position in the Computer Architecture Department of UPC. From 2011 to 2014 he held a Postdoctoral Associate position under the mentoring of Professor K.S. Trivedi, in the Electrical and Computer Engineering Department, Duke University, Durham, NC.

Currently, Dr. Alonso is the Research Manager and Acting Research Director at the Research Institute of Applied Sciences in Cybersecurity - University of Leon, Spain. He also holds a visiting Assistant Professor position at Duke University, USA. Dr. Alonso has published several papers about different aspects of software engineering with special interest on dependability, high availability, performance, software security and software aging in premier conferences and journals. He has also served as a reviewer for IEEE Transactions on Computers, IEEE Transactions on Dependability and Security Computing, Performance Evaluation, and Cluster Computing, and several international conferences. His research interests are in software engineering focusing on high performance and high available large scale distributed software systems as well as mobile/cloud computing. He is also interested on applying data analytics, analytical models to improve the availability, and performance of large scale and distributed software systems during design, test and operation phases. He is or has been involved in NASA, JPL/NASA, NEC, NATO, Huawei, WiPro funded projects. He is a IEEE member.