KVM-FUZZ: *fuzzing* de binarios x86-64 con emulación y aceleración por hardware

David Mateos Romero
Universidad de Granada
Granada, España
davidmateos@correo.ugr.es

Gabriel Maciá Fernández
Dep. Teoría de la Señal, Telemática y Comunicaciones - CITIC
Universidad de Granada
Granada, España
gmacia@ugr.es

***Resumen*—El *fuzzing* es un método para descubrir *bugs* y vulnerabilidades en software comprobando su comportamiento ante entradas generadas de forma automática. En los últimos años, el *fuzzing* basado en cobertura de código está siendo ampliamente investigado y utilizado. La efectividad del *fuzzing* es proporcional a su velocidad de ejecución. Sin embargo, esta se ve reducida cuando no se dispone del código fuente del programa objetivo, ya que se suele recurrir a la instrumentación dinámica y a la emulación, lo cual conlleva una pérdida de eficiencia considerable.**

En este artículo presentamos KVM-FUZZ, un *fuzzer* guiado por cobertura de código que incorpora un hipervisor basado en KVM y un kernel propios. Hace uso de aceleración por hardware para conseguir velocidad de ejecución casi nativa, e incorpora un mecanismo de *snapshots* que permite restaurar el programa objetivo miles de veces por segundo.

***Index Terms*—*Fuzzing*, Emulación, Rendimiento**

Tipo de contribución: *Investigación original (límite 8 páginas)*

I. Introducción

El descubrimiento de vulnerabilidades en software es un problema crítico en seguridad cada vez más relevante. Google Project Zero llevó a cabo un análisis de las vulnerabilidades de día 0 (*0-days*) explotadas de forma activa en 2021 [1]. Se detectaron 58 fallos de seguridad explotados activamente, más del doble que el año anterior, de los cuales 39 fueron vulnerabilidades de corrupción de memoria, que incluyen clases como *use-after-free*, escritura o lectura fuera de límites, *buffer overflow* e *integer overflow*. Dichas vulnerabilidades fueron encontradas en proyectos conocidos, abarcando navegadores como Google Chrome, WebKit (Safari) o Internet Explorer, sistemas operativos como Windows, iOS/macOS y Android, y aplicaciones como Microsoft Exchange Server. Por otro lado, CVE Details enumera 421 diferentes vulnerabilidades de corrupción de memoria en *software* de uso común publicadas en 2022 [2].

Esto sugiere que hoy en día las vulnerabilidades están presentes en todo *software*, y encontrarlas supone un gran interés, tanto para desarrolladores como para investigadores de seguridad. Sin embargo, el descubrimiento de vulnerabilidades de forma manual es una tarea difícil, especialmente en *software* complejo.

El *fuzzing* es un método automático que ha sido estudiado como una forma efectiva de descubrimiento de vulnerabilidades. La idea principal consiste en ejecutar el programa objetivo con entradas generadas automáticamente con el objetivo de descubrir *bugs*. Un ejemplo para ilustrar su impacto es OSS-Fuzz [3], un proyecto de Google que utiliza diferentes herramientas de *fuzzing* de forma continua para encontrar y reportar vulnerabilidades de forma automática. Desde su creación, ha encontrado miles de fallos de seguridad en cientos de proyectos de código abierto.

En el proceso de *fuzzing*, con el objetivo de analizar el comportamiento del programa objetivo con cada entrada, es común compilarlo introduciendo pequeños fragmentos de código mediante un proceso llamado instrumentación. Sin embargo, esto no es posible cuando no se dispone del código fuente del programa objetivo, presentando retos adicionales. La solución más común es introducir la instrumentación en tiempo de ejecución con QEMU, pero esto resulta en una pérdida significante de rendimiento.

En este artículo presentamos KVM-FUZZ, un *fuzzer* de binarios x86-64 basado en cobertura de código que implementa un hipervisor y kernel propios y hace uso de diferentes mecanismos de aceleración hardware. El uso de un hipervisor y máquinas virtuales permite aislar el programa objetivo del sistema anfitrión, así como hacer uso de restauración de *snapshots* y diferentes instrumentos de cobertura de código de alto rendimiento. Por otra parte, el uso de un kernel propio permite la emulación de las llamadas al sistema, logrando así una velocidad superior a la nativa en muchos casos. Estas características permiten a KVM-FUZZ alcanzar un rendimiento y velocidad de ejecución superiores a alternativas similares.

El artículo está organizado de la siguiente forma. En primer lugar, se explican conceptos previos en la Sección II. A continuación, se presenta el estado del arte en *fuzzing* de binarios en la Sección III, discutiendo especialmente los *fuzzers* AFL++ [4] y NYX [5]. Luego se procede a la descripción de KVM-FUZZ y sus detalles de implementación en la Sección IV. Finalmente, se realiza una evaluación experimental de KVM-FUZZ y se analizan sus resultados en la Sección V, y se termina con unas conclusiones en la Sección VI.

II. Conceptos previos

II-A. *Fuzzing*

El *fuzzing* es el proceso de ejecutar un programa de forma repetida con entradas generadas automáticamente con el objetivo de descubrir *bugs* y vulnerabilidades.

Aunque incluso hacerlo de forma ciega es suficiente en muchos casos para descubrir *bugs* superficiales, en general es más efectivo llevar a cabo enfoques más inteligentes. Destaca entre ellos el *fuzzing* basado en cobertura de código, que se basa en analizar qué partes del código del programa bajo prueba se han ejecutado con cada entrada. De esta forma, cuando una entrada consigue ejecutar un trozo del programa

al que no se había llegado previamente, dicha entrada se guarda para continuar mutándola y generar nuevas entradas que pudieran ser efectivas para recorrer nuevas partes del código. Con este sencillo concepto se consigue que un *fuzzer* genérico que carezca de información del formato de entrada de un programa aprenda y genere entradas que alcancen fragmentos de códigos profundos, y por tanto descubra más *bugs*.

Para la obtención de información de cobertura de código se suele hacer uso de instrumentación. La instrumentación de código es una técnica consistente en modificar el código original de una aplicación con el objetivo de depurarla, medir su rendimiento o modificar su comportamiento. En el caso de la cobertura de código, la instrumentación inserta pequeños pedazos de código en diferentes partes del programa original con el objetivo de saber qué partes del código se han ejecutado en cada iteración. Si se del dispone código fuente del programa, se suele utilizar un *plugin* para el compilador que realiza el proceso descrito. En caso contrario, se puede optar por un enfoque estático (modificar el binario original) o dinámico (modificar el comportamiento del programa en tiempo de ejecución) para insertar la instrumentación. En particular, se explicará más a fondo el enfoque dinámico en la Sección III-A.

Cuando finaliza cada ejecución, es necesario restaurar el estado del programa antes de la siguiente ejecución. La forma más sencilla es simplemente lanzar el programa desde cero en cada ejecución y esperar a que finalice el proceso. Sin embargo, esto es muy poco eficiente y no escala apropiadamente (Sección IV-D3). Una manera más efectiva de hacer esto es mediante *snapshot fuzzing*: se captura el estado del proceso antes de comenzar una ejecución, y se restaura dicho estado cuando termina. De esta forma, al trabajar con un único proceso cuyo estado se restaura continuamente, se produce un ahorro en el coste de creación y finalización de procesos. Tomar y restaurar *snapshots* de un proceso requiere acceder a sus registros y memoria, entre otras cosas. Se puede realizar desde el espacio de usuario haciendo uso de interfaces del sistema operativo como `ptrace` y `/proc/pid/mem` [6]. Sin embargo, es más eficiente si se realiza directamente en espacio de kernel [7], o haciendo uso de máquinas virtuales [5].

En caso de que el programa carezca de un estado global que se pueda alterar en cada ejecución, es posible hacer uso de la técnica de *fuzzing* persistente. Consiste en tener una sola instancia del programa objetivo ejecutándose continuamente. Ya que no existe un estado global que pueda hacer que unas ejecuciones afecten a otras, no hay necesidad de restaurar el proceso. Por tanto, en cada iteración simplemente se establece una nueva entrada, se ejecuta el programa, y se comienza de nuevo, todo ello sin terminar el proceso. Para modificar el comportamiento del programa original y hacerlo adecuado para el *fuzzing* persistente, se hace también uso de la instrumentación. La idea es alterar el inicio y fin del programa para que se comunique con el *fuzzer*, y así obtener la nueva entrada y evitar que el proceso finalice.

II-B. QEMU

QEMU es un emulador que permite la ejecución de código de una arquitectura dada mediante la traducción dinámica de dicho código a la arquitectura del sistema anfitrión. El proceso de traducción recibe el nombre de TCG (*Tiny Code Generation*), y es similar a un proceso de compilación JIT (*Just In Time*). Primero, el código de la arquitectura emulada es traducido a instrucciones del lenguaje intermedio *Tiny Code*, que es independiente de la máquina. Finalmente, el código en lenguaje intermedio es traducido a instrucciones de la arquitectura anfitrión. La introducción de un lenguaje intermedio es una técnica ampliamente utilizada por los compiladores para simplificar el proceso de traducción.

La traducción se realiza a nivel de bloque básico de código, esto es, una secuencia de instrucciones que no tiene flujo de control, es decir, que se ejecuta de forma secuencial de principio a fin. Por tanto, una posible forma de modelar el flujo de ejecución de un programa sería mediante bloques básicos, que se encadenan entre sí mediante instrucciones de salto. De forma similar, QEMU realiza el proceso de traducción a cada bloque básico, y encadena los trozos de código resultantes para una mayor eficiencia.

QEMU dispone de dos modos principales de ejecución: modo usuario y modo sistema completo. El modo usuario permite ejecutar programas de usuario compilados para una arquitectura diferente a la del sistema anfitrión. En primer lugar, emula sus instrucciones mediante TCG, la técnica de traducción de código descrita anteriormente. En segundo lugar, redirige al sistema operativo anfitrión las llamadas al sistema, ajustando la *endianness* y el tamaño de la arquitectura.

Por otro lado, el modo sistema completo permite emular un sistema completo, incluyendo el sistema operativo y los periféricos, de forma similar a una máquina virtual. En este caso, las instrucciones son emuladas mediante TCG, y las operaciones de entrada-salida son emuladas mediante dispositivos virtuales desde el sistema anfitrión.

Cuando la arquitectura objetivo es la misma que la arquitectura del anfitrión, QEMU en modo sistema completo puede utilizar aceleración por hardware mediante el uso de KVM (ver Sección II-C). De esta forma, en vez de tener que realizar el proceso de traducción, el procesador ejecuta directamente las instrucciones del huésped, consiguiendo así una velocidad de ejecución casi nativa.

II-C. KVM

KVM (*Kernel-based Virtual Machine*) es un hipervisor implementado en el kernel de Linux que utiliza virtualización por hardware. Expone la interfaz `/dev/kvm`, que un programa en espacio de usuario puede usar para crear y administrar máquinas virtuales.

Para ello, KVM hace uso de las tecnologías de virtualización Intel VT-x y AMD-V. Estas extensiones implementan unos niveles de privilegio cuyo objetivo es discernir si un código se está ejecutando en modo huésped o en modo anfitrión (correspondiéndose a dentro o fuera de la máquina virtual, respectivamente), además de un conjunto de instrucciones para poder entrar en modo huésped. Cuando el procesador se está ejecutando dentro de una máquina virtual en modo huésped, algunas instrucciones privilegiadas provocan una salida de la máquina virtual y un cambio a modo anfitrión. Será entonces KVM o la aplicación de usuario los que deberán emular dicha instrucción.

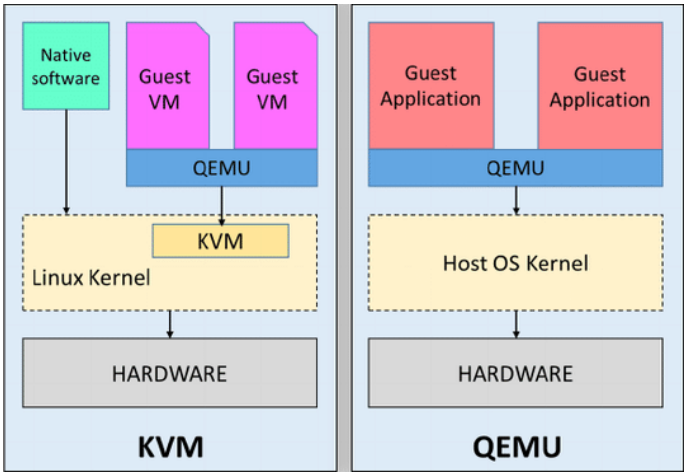

Figura 1. A la izquierda, QEMU en modo sistema completo con KVM. Las máquinas virtuales disponen de un sistema operativo propio. QEMU se encarga de emular los dispositivos y KVM de proporcionar la virtualización por hardware. A la derecha, QEMU en modo usuario. Emula las aplicaciones con el proceso de traducción dinámica (TCG) y redirige las llamadas al sistema al kernel anfitrión. Figura tomada de [8].

Sin embargo, KVM no proporciona emulación de dispositivos. En su lugar, las operaciones de entrada-salida generan una salida de la máquina virtual, de forma que es el programa en espacio de usuario (por ejemplo, QEMU) el que debe emular esas instrucciones y los dispositivos a los que puedan acceder.

Al hacer uso de las tecnologías de virtualización de los procesadores, KVM sólo permite la ejecución de máquinas virtuales de la misma arquitectura que el sistema anfitrión.

III. Estado del arte

En esta sección se pretende ilustrar el estado del arte de las técnicas empleadas para el *fuzzing* de binarios. En el *fuzzing* de binarios, la obtención de cobertura de código se obtiene mediante uno de los siguientes mecanismos:

***Tracing* asistido por hardware**. Los procesadores más modernos están incluyendo mecanismos que permiten la obtención de cobertura de código mediante hardware con muy bajo sobrecoste, como es el caso de Intel PT. Sin embargo, las trazas necesitan una fase de post-procesado de un coste significativo. Es el método usado por *fuzzers* como NYX [5] o PTfuzz [9]. Una estrategia diferente es el uso de *breakpoints* para la obtención de las trazas. Se aplica un *breakpoint* en cada bloque básico de código y se elimina cuando es alcanzado. De esta forma, el sobrecoste tiende a ser nulo conforme los *breakpoints* van desapareciendo. Es el método implementado por Mesos [10].

Traducción dinámica. Consiste en introducir instrumentación al vuelo haciendo uso del proceso de traducción dinámica de herramientas como QEMU (Sección II-B), DynamoRIO [11] o PIN [12]. Permite una gran introspección, pero también conlleva una pérdida de rendimiento muy significativa, haciéndolo inviable en muchos casos.

Reescritura de binarios. La idea consiste en modificar el binario original e insertarle la instrumentación antes de su ejecución. De esta forma se consigue un resultado similar a disponer del código fuente e introducir la instrumentación mediante un compilador. Algunos proyectos que hacen uso de este mecanismo para *fuzzing* son ZAFL [13], RetroWrite [14] y AFL-Dyninst [15]. Sin embargo, reescribir un binario es complejo, y en muchos casos se restringen a binarios compilados de cierta forma o que dispongan de símbolos, o se termina alterando el funcionamiento del binario original.

A continuación se detallará el funcionamiento de dos *fuzzers* de binarios que destacan por su amplio uso y por su relevancia con respecto a la propuesta: AFL++, que usa traducción dinámica, y NYX, que usa *tracing* asistido por hardware.

III-A. AFL++

AFL++ [4] es la versión mantenida de AFL. Es un *fuzzer* que incluye numerosas aportaciones de la comunidad científica y que se suele usar como base para crear prototipos de ideas, muchas de las cuales terminan siendo integradas en el proyecto.

Para *fuzzear* programas de los que no se dispone su código fuente, AFL++ utiliza un *fork* de QEMU en modo usuario llamado `qemuafl`. Esto le permite aprovechar la traducción dinámica de código de QEMU para introducir instrumentación, y así medir la cobertura de código.

III-A1. Fork server: Para la ejecución del programa objetivo en un nuevo proceso, en Linux se debe hacer uso de las llamadas al sistema `fork()` (para duplicar el proceso actual) y `exec()` (para cambiar el proceso hijo por el programa objetivo). Sin embargo, esto es muy costoso de realizar en cada ejecución. Para solventarlo, AFL++ hace uso de la técnica *fork server*, que permite que solo sea necesario el uso de la llamada al sistema `fork()` en cada ejecución. La idea es ejecutar el programa objetivo bajo `qemuafl` en un nuevo proceso, que será el *fork server*. Cuando la ejecución llega al punto de entrada del programa objetivo, dicho proceso se parará. Ahora, en cada ejecución simplemente será necesario que este *fork server* realice la llamada al sistema `fork()`. No será necesario el uso de `exec()`, porque el *fork server* ya está ejecutando el programa objetivo.

Por otro lado, otra técnica consiste en parar la ejecución en un punto más avanzado que el de entrada, por ejemplo, después de las rutinas de inicialización. Esto permite que dichas rutinas no se tengan que ejecutar en cada iteración, efectivamente adelantando el inicio del programa. Dicha técnica se conoce como inicialización diferida y consigue aumentar considerablemente el rendimiento, reduciendo la cantidad de código que se tiene que ejecutar en cada iteración.

El uso del *fork server* reduce las llamadas al sistema necesarias por ejecución a sólo una: `fork()`. Sin embargo, esa llamada al sistema es servida por el sistema operativo anfitrión. Cuando se dispone de múltiples instancias del *fuzzer* ejecutándose en paralelo, el uso intensivo de `fork()` y su sincronización necesaria en el kernel hace de cuello de botella, ya que sólo se dispone de una instancia del kernel para satisfacer todas las llamadas a `fork()` por cada uno de los procesos. El resultado es un peor escalado conforme aumenta el número de núcleos disponibles, contrario a lo que se desea.

III-A2. Fuzzing persistente y snapshots: AFL++ también implementa el uso de *fuzzing* persistente y de *snapshots*. Mediante el uso de instrumentación dinámica, `qemuafl` puede modificar el flujo de ejecución del programa para que se repita en bucle entre dos puntos de ejecución dados, sin necesidad de la creación de nuevos procesos. Esto supone un

alto rendimiento y un mejor escalado, ya que elimina el uso de `fork()` y reduce las llamadas al sistema necesarias por ejecución a simplemente las involucradas para la comunicación entre procesos.

Sin embargo, la mayoría de programas con un mínimo de complejidad no pueden hacer uso de *fuzzing* persistente por sí solo, ya que es necesario restaurar la memoria antes de cada ejecución. AFL++ lo soluciona mediante el uso de *snapshots*. De forma similar a lo descrito, puede guardar el estado del programa antes de entrar al bucle de ejecución, y restaurarlo después de cada iteración. Sin embargo, esto es complicado, ya que se realiza desde espacio de usuario. No se dispone de un mecanismo adecuado para llevar el recuento de páginas de memoria modificadas, por lo que se debe restaurar el contenido de todas las páginas que tengan permisos de escritura, aunque no hayan sido modificadas. La comunidad de AFL++ desarrolló un módulo para el kernel de Linux [7] basado en [16] que implementaba *snapshots* y que solucionaba parcialmente el problema de tener que restaurar todas las páginas de memoria, pero el proyecto fue discontinuado.

III-B. Nyx

NYX [5] es un *fuzzer* guiado por cobertura de código que destaca por su capacidad para *fuzzear* objetivos complejos: hipervisores, navegadores, kernels, etc.

NYX ejecuta el programa objetivo dentro de máquinas virtuales. Para ello, hace uso de KVM-PT y QEMU-PT, versiones modificadas del hipervisor KVM y de QEMU respectivamente. En este caso, y al contrario que AFL++, utiliza QEMU en su modo de emulación de sistema completo haciendo uso de máquinas virtuales. Esto les permite no sólo *fuzzear* aplicaciones de usuario de Linux como AFL++, sino también de otros sistemas operativos como Windows, e incluso código que se ejecute en modo privilegiado (*ring 0*), como pueden ser drivers o el propio kernel del sistema operativo.

NYX extiende KVM y QEMU con la habilidad de realizar *snapshots* que permiten guardar y restaurar de forma muy rápida el estado de las máquinas virtuales, incluyendo el hardware emulado. Para ello, hacen uso de una característica de KVM llamada *Page Modification Logging* (PML), que utiliza aceleración hardware para identificar de forma eficiente qué páginas de la memoria de la máquina virtual deben ser restauradas. Junto a un mecanismo similar para el estado del hardware emulado en QEMU, esto les permite restaurar el estado de la máquina virtual de forma diferencial: sólo se restaura la memoria que ha sido modificada en cada ejecución. Poder realizar y restaurar *snapshots* a nivel de máquina virtual permite a NYX *fuzzear* programas que debido a su alta complejidad sería imposible hacerlo de otra manera, como por ejemplo los mecanismos de comunicación entre procesos de navegadores como Firefox.

Para la cobertura de código, una de las opciones que propone NYX es el uso de Intel PT (*Intel Processor Trace*). Intel PT es una tecnología que incorporan los procesadores modernos de Intel, y que permite obtener información del flujo de ejecución de un programa, produciendo trazas muy comprimidas. Al ser una capacidad incorporada en hardware, tiene muy poco sobrecoste (*overhead*) en términos de rendimiento. Esto permite obtener cobertura de código de forma genérica, ya sea en binarios que no hayan sido instrumentados o en código del kernel del sistema operativo que se ejecute en ring 0.

Intel PT produce unas trazas en formato de paquetes muy comprimidos, que deben ser analizadas y decodificadas posteriormente. Para ello, QEMU-PT hace uso de `libxdc` [17], una librería desarrollada por los mismos autores que consigue dicho objetivo con alto rendimiento.

IV. DESCRIPCIÓN DE LA PROPUESTA

En este artículo presentamos KVM-FUZZ, un *fuzzer* de aplicaciones de usuario x86-64 que hace uso de emulación y diferentes tecnologías de aceleración hardware, como ejecución con KVM, cobertura de código mediante Intel PT y *breakpoints*, y *snapshots*.

KVM-FUZZ consigue una mejora de rendimiento y escalado significativa frente a AFL++ gracias al uso de la aceleración hardware y las *snapshots*. Las mejoras de rendimiento frente a NYX también son significativas, en este caso debido al uso de un kernel propio para la emulación de llamadas al sistema.

IV-A. Fundamentación de la propuesta

Como se ha descrito en la Sección III-A, para *fuzzear* binarios de los que no se dispone el código fuente AFL++ utiliza QEMU en modo usuario, que hace uso de traducción dinámica (TCG), lo que reduce el rendimiento. Por otro lado, cuando la arquitectura objetivo es la misma que la del anfitrión, QEMU en modo de emulación de sistema completo dispone de un modo de aceleración hardware que hace uso de KVM. El objetivo de KVM-FUZZ es emular binarios aprovechando dicha aceleración hardware, evitando la traducción dinámica y consiguiendo una mejora de rendimiento significativa en el proceso de *fuzzing*. Para ello, se implementa un programa de usuario que hace uso de KVM, al que a partir de ahora nos referiremos como el hipervisor.

El problema que presenta KVM es que requiere el uso de máquinas virtuales de sistema completo. Es decir, además del programa de usuario objetivo, necesitan un kernel. NYX opta por usar máquinas virtuales tradicionales, con su respectivo sistema operativo completo (Linux, Windows, etc) y la emulación de los dispositivos mediante QEMU. Sin embargo, nuestro objetivo es simplemente *fuzzear* una aplicación de usuario. Por tanto, no necesitamos el sistema operativo entero, sino simplemente un mecanismo que nos permita emular las llamadas al sistema.

La idea inicial es que las llamadas al sistema sean emuladas desde el hipervisor fuera de la máquina virtual. De esta forma, el kernel de la máquina virtual es simplemente un trozo de código que se ejecuta cuando la aplicación objetivo realiza una llamada al sistema, y que se encarga de ejecutar una instrucción privilegiada que provoca una salida de la máquina virtual al hipervisor. El hipervisor, al detectar el uso de dicha instrucción, emulará la llamada al sistema y continuará con la ejecución. Este método de comunicación entre el kernel huésped y el hipervisor se conoce como *hypercall* (llamada al hipervisor).

Esta forma de emulación tiene la ventaja de que las instrucciones de la aplicación objetivo se ejecutan dentro de la máquina virtual con KVM, y por tanto a velocidad nativa, ya que evita la traducción dinámica y la consecuente pérdida

de rendimiento. El inconveniente de este método es el elevado coste de las llamadas al sistema. Ya que es el hipervisor el que se encarga de emularlas, cada vez que la aplicación realiza una llamada al sistema ocurren muchos cambios de contexto: usuario huésped (aplicación) → kernel huésped (simplemente realiza salida de la VM) → kernel anfitrión (KVM) → usuario anfitrión (hipervisor, emula la llamada al sistema), y vuelta.

Para solucionarlo, la propuesta incorpora un kernel propio, más complejo que el sugerido anteriormente, que se encarga de emular las llamadas al sistema en lugar de enviarlas a emular al hipervisor. De esta forma, los cambios de contexto necesarios en cada llamada al sistema son los mismos que habría en un sistema real: usuario huésped (aplicación) → kernel huésped (emula la llamada al sistema), y vuelta. Ya que ahora la emulación ocurre dentro de la máquina virtual, hay un ahorro en el coste de las salidas y en la comunicación con el hipervisor.

Por ejemplo, usando `readelf` como programa objetivo, la implementación inicial que emulaba las llamadas al sistema desde el hipervisor alcanzaba una media de 2.800 ejecuciones por segundo, realizando un total de 76 salidas de la máquina virtual (una por cada llamada al sistema) por ejecución. La mayoría del tiempo de ejecución se perdía en los cambios de contexto. En comparación, la implementación actual que emula las llamadas al sistema desde el kernel dentro de la máquina virtual, alcanza una media de 15.000 ejecuciones por segundo, con una sola salida de la máquina virtual para finalizar cada ejecución.

Adicionalmente, ya que el kernel está pensado para ejecutarse en el propio hipervisor, puede hacer uso de llamadas al hipervisor (*hypercalls*) para operaciones que de otra forma serían complicadas de realizar desde espacio de kernel, como puede ser leer archivos del disco. De esta forma se simplifica el kernel y se alivia su dificultad de desarrollo, ya que no son necesarios drivers y otros elementos que proporciona el hipervisor.

A continuación se detallan las funciones de los dos componentes principales de KVM-FUZZ: el hipervisor y el kernel.

IV-B. Hipervisor

El hipervisor hace uso de KVM para crear y manejar máquinas virtuales. A grandes rasgos, el funcionamiento es como sigue. En primer lugar, crea una máquina virtual y la configura para su ejecución en *long mode* (el modo de 64 bits de x86). Esto incluye crear una tabla de páginas, establecer los valores iniciales de los registros y alguna configuración más relativa a KVM. A continuación, el hipervisor carga los binarios necesarios (kernel, programa objetivo, e intérprete en caso de que el programa esté enlazado dinámicamente) en la máquina virtual. Una vez hecho esto, comienza la ejecución de la máquina virtual en el kernel, que inicializa diferentes componentes e inicia la ejecución del programa objetivo. La ejecución de la máquina virtual para cuando se llega a un cierto punto de ejecución (por ejemplo, la función `main()` del binario).

Una vez la máquina virtual está inicializada y se ha llegado al punto de entrada del programa objetivo, el hipervisor crea diferentes hilos de ejecución. Cada hilo clona la máquina virtual, y usa esa copia para ejecutar en paralelo el bucle principal del proceso de *fuzzing*:

1. Obtener una nueva entrada. El *fuzzer* tiene un conjunto de entradas guardadas, y las nuevas entradas se obtienen a partir de estas mediante mutaciones.
2. Escribir la entrada en la memoria de la máquina virtual. Se puede escribir en la memoria del kernel, si el programa objetivo lee la entrada desde un archivo, o bien directamente en la memoria del programa objetivo, si lee la entrada de un buffer.
3. Ejecutar la máquina virtual hasta que el kernel indique que la ejecución ha finalizado, ya sea porque el programa objetivo ha llamado a la *syscall* `exit()` o bien porque ha sufrido un *crash*.
4. Obtener la cobertura de código que se ha obtenido en la ejecución. Si la entrada ha ejecutado código interesante (por ejemplo, código que no se había ejecutado anteriormente con ninguna otra entrada), se guarda para generar nuevas entradas a partir de ella.
5. Restaurar el estado de la máquina virtual al estado original.

Este bucle se ejecuta de forma indefinida hasta que el proceso sea interrumpido.

IV-C. Kernel

El kernel se ejecuta dentro de la máquina virtual. Se encarga de satisfacer las llamadas al sistema que ejecuta el programa objetivo y de reportar los crashes al hipervisor.

Cuando la máquina virtual se inicia, el kernel primero inicializa diferentes componentes que requiere para su correcto funcionamiento. En particular, el manejador de archivos del kernel se comunica con el hipervisor por medio de *hypercalls* para cargar en memoria los archivos que el programa objetivo pueda necesitar, incluyendo librerías (si el programa está enlazado dinámicamente), archivos de configuración, o el archivo de entrada que será modificado en cada ejecución.

Una vez inicializado, el kernel cambia a espacio de usuario, salta al punto de entrada del programa objetivo y comienza su ejecución. Cuando el programa objetivo ejecuta la instrucción `syscall`, la ejecución se transfiere de nuevo al kernel, que se encargará de emular el comportamiento de la llamada al sistema. Finalmente, cuando se ejecuta la llamada al sistema `exit` o el programa objetivo *crashea*, el kernel notifica al hipervisor, que restaura el estado de la máquina virtual y comienza una nueva ejecución.

IV-D. Características de KVM-FUZZ

IV-D1. Velocidad de ejecución: Gracias al uso de la virtualización por hardware de KVM, la velocidad de ejecución es casi nativa. Sin embargo, las máquinas virtuales no ejecutan Linux, sino un kernel propio más pequeño que lo emula, y que no realiza acciones costosas como accesos a disco o a red. En algunos casos, esto resulta en una velocidad de ejecución mayor que nativa, ya que se ejecutan un menor número de instrucciones que en Linux, como se ve ilustrado en los experimentos de la Sección V-A.

IV-D2. Snapshots: KVM-FUZZ implementa un mecanismo de *snapshots* similar a `fork()` y al de NYX (Sección III-B). Permite ejecutar la máquina virtual hasta que el programa objetivo llega a `main()` o cualquier otro punto avanzado de la ejecución, y clonar la máquina en dicho punto. El proceso de *fuzzing* consistirá en ejecutar esta nueva

máquina virtual. Cuando finalice, se restaurará su estado al que tenía en el punto de clonado usando la máquina padre como *snapshot*, y comenzará una nueva ejecución. De forma similar a la inicialización diferida de AFL++ (Sección III-A1), modificar el punto de inicio de las ejecuciones elimina los costes de inicialización del programa, pudiendo implicar en algunos casos una mejora del rendimiento de hasta 10 veces.

Para implementar las *snapshots*, el hipervisor hace uso de *Page Modification Logging* (PML), un mecanismo de aceleración hardware que permite a KVM obtener la memoria modificada por el huésped. PML nos permite restaurar sólo las páginas de memoria que hayan sido modificadas, lo que tiene un gran impacto en el rendimiento, especialmente en programas que necesitan máquinas virtuales con mayor memoria.

PML está implementado a nivel hardware en los procesadores modernos, lo que hace que su impacto sobre el rendimiento sea mínimo. Por tanto, es un enfoque superior a alternativas software, como recorrer la tabla de páginas en busca de qué páginas han sido modificadas (tienen el *dirty bit* activo).

El rendimiento del mecanismo de *snapshots* de KVM-FUZZ se discute y se compara al de otros *fuzzers* en la Sección V-C.

IV-D3. Escalado lineal: Como cada máquina virtual tiene su propia instancia del kernel y está totalmente aislada del resto, es posible crear una máquina virtual por núcleo de ejecución. Esto escala linealmente con el número de núcleos, ya que las máquinas virtuales no interactúan entre sí, y apenas interactúan con el kernel anfitrión.

Tener una instancia del kernel por cada instancia del *fuzzer* soluciona el problema que tiene el *fork server* de AFL++ (Sección III-A1), donde cada una de las instancias del *fuzzer* realiza la llamada al sistema `fork()` una vez por ejecución, haciendo cuello de botella. Esto se ve reflejado en el experimento de la Sección V-B.

IV-D4. Introspección de máquina virtual: El hipervisor dispone de las funcionalidades esperadas de un *framework* de emulación, que permiten examinar el estado de la máquina virtual y modificar su comportamiento. Destacan: *a)* Inspección y modificación del estado: el hipervisor tiene acceso al estado del programa objetivo en todo momento, incluyendo la memoria, los registros, y los archivos cargados en memoria, y proporciona una interfaz cómoda para su acceso. *b) Breakpoints*: permiten parar la ejecución del programa objetivo cuando llegue a cierto punto, y se usan también como base para implementar *hooks* y cobertura de código. *c) Hooks*: consisten en ejecutar *callbacks* en el hipervisor cuando el programa objetivo llegue a un punto de la ejecución determinado. Entre otras cosas, los *hooks* permiten modificar funcionalidades del programa objetivo o emular otras que el kernel no soporte. *d)* Símbolos y *stacktraces*: el hipervisor implementa resolución de símbolos y obtención de *stacktraces* a partir del estado actual de la máquina virtual. Esto es especialmente útil para la depuración de *crashes*, tanto del kernel como del programa objetivo o sus librerías.

Estas funcionalidades, junto a la emulación de llamadas al sistema del kernel (Sección IV-D6), hacen que KVM-FUZZ no sirva solo como un *fuzzer*, sino también como un emulador de alto rendimiento.

IV-D5. Cobertura de código: KVM-FUZZ implementa dos formas de obtener cobertura de código: *a)* Intel Processor Trace: es una solución hardware implementada en CPUs modernas de Intel que permite obtener información del flujo de ejecución. Se hace uso del trabajo de NYX (Sección III-B) con KVM-PT (modificación de KVM que permite la ejecución de Intel PT en máquinas virtuales) y `libxdc` [17] (librería que implementa decodificación de las trazas generadas por Intel PT de forma eficiente). *b) Breakpoints*: implementación que consiste en establecer un breakpoint al inicio de cada bloque básico del programa objetivo. Cuando se ejecuta un *breakpoint*, se guarda como interesante la entrada actual y se elimina el *breakpoint*. De esta forma, el sobrecoste tiende a ser nulo conforme los *breakpoints* van desapareciendo, pero la información obtenida sobre el flujo de ejecución es también menor. La lista de bloques básicos del programa objetivo se puede obtener mediante frameworks de análisis de binarios como `angr` o `radare2`.

IV-D6. Simulación de syscalls: Al disponer de un kernel propio, se puede modificar y simplificar el comportamiento de las llamadas al sistema. Un ejemplo es el de los archivos. En la inicialización de las máquinas virtuales, el kernel obtiene del hipervisor una lista de archivos y sus contenidos, que almacena en memoria. De esta forma, cuando el programa objetivo intenta leer de un archivo, el kernel simplemente copia el contenido de su memoria a la del programa, sin necesidad de acceder a disco. Así, leer de un archivo es poco más que realizar un `memcpy()`.

Otro ejemplo es el de las llamadas al sistema relacionadas con la red. Para *fuzzear* servidores se pueden emular la mayoría de syscalls de forma sencilla, evitando así complicaciones que debe tener en cuenta un sistema operativo normal, como accesos al hardware de red, o implementación de las diferentes capas de la pila de red. Por ejemplo, llamar a `bind()` o `listen()` en un socket simplemente consiste en cambiar un atributo. La syscall `recv()`, que bajo condiciones normales tendría que recibir algo por la red, se puede emular como si estuviera leyendo de un archivo, resultando en la ejecución de `memcpy()` como en el caso anterior.

Además, el comportamiento y la implementación de las llamadas al sistema se puede alterar dependiendo del programa objetivo y de la funcionalidad que este necesite para su correcto funcionamiento, aportando una gran flexibilidad.

V. EXPERIMENTOS

La implementación de KVM-FUZZ se ha desarrollado en el lenguaje de programación C++ para el hipervisor y Zig para el kernel, y está disponible en https://github.com/klecko/kvm-fuzz/. Los experimentos del número de instrucciones ejecutadas (Sección V-A) y de la velocidad de restauración de snapshots (Sección V-C) han sido realizados en una máquina con un procesador Intel(R) Core(TM) i7-6700K de 8 núcleos lógicos y 32GB de RAM. Por otra parte, los experimentos de escalado (Sección V-B) han sido realizados en una máquina con un procesador Intel(R) Xeon(R) Silver 4314 de 32 núcleos físicos y 64 lógicos.

V-A. Número de instrucciones ejecutadas

Este experimento pretende ilustrar la ventaja en rendimiento que supone disponer de un kernel propio con la posibilidad de emular las llamadas al sistema. Se han realizado dos ejecuciones diferentes de `readelf`, un programa perteneciente a la

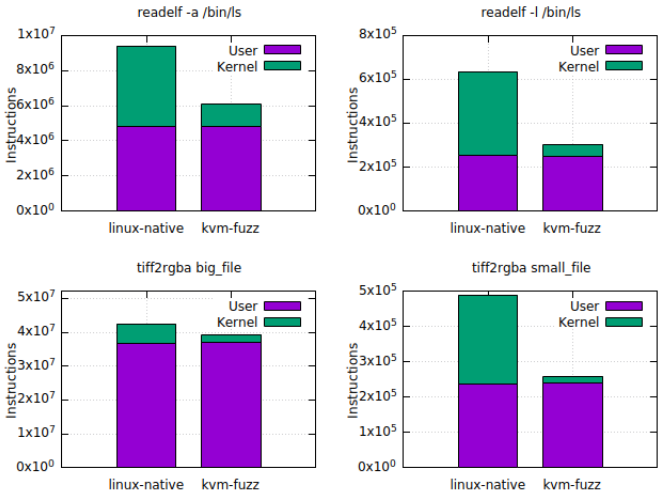

Figura 2. Comparación del número de instrucciones ejecutadas en diferentes ejecuciones bajo Linux y bajo el kernel de KVM-FUZZ.

colección de herramientas `binutils`, y de `tiff2rgba`, un programa que hace uso de la librería `libtiff` para procesamiento de imágenes. El objetivo es ver cómo de significativa es la mejora de rendimiento en ejecuciones largas (`readelf -a` y `tiff2rgba` con un archivo de entrada grande) frente a ejecuciones más cortas (`readelf -l` y `tiff2rgba` con un archivo de entrada pequeño).

En cada ejecución se ha medido el número de instrucciones ejecutadas desde `main()` hasta `exit()`, tanto en espacio de usuario como de kernel. Para obtener las mediciones en KVM-FUZZ se ha hecho uso de los contadores de rendimiento (*performance counters*), unos registros hardware que permiten medir eventos del procesador. Para obtener las mediciones en Linux se ha hecho uso de `perf`, una herramienta que permite acceder a los mismos contadores de rendimiento.

Los resultados se pueden observar en la Figura 2. Como se esperaba, el número de instrucciones ejecutadas por la aplicación en espacio de usuario es independiente del kernel, asumiendo que este se comporte correctamente. Sin embargo, se puede apreciar una reducción significativa del número de instrucciones ejecutadas en espacio de kernel cuando la aplicación se ejecuta bajo el kernel desarrollado frente a cuando lo hace en Linux de forma nativa.

De media, en cada ejecución KVM-FUZZ ejecutó un 78 % menos de instrucciones en espacio de kernel en comparación con Linux, y un 37 % menos de instrucciones totales. La diferencia en instrucciones totales es menos significativa en ejecuciones largas que pasan la mayor parte del tiempo en espacio de usuario, como es el caso de `readelf -a` o `tiff2rgba` con un archivo grande. Sin embargo, en ejecuciones más cortas que puedan pasar una mayor porción del tiempo en el kernel, dicha reducción del número de instrucciones en el kernel se refleja en una reducción de más del 50 % en el número de instrucciones totales, como ocurre en las ejecuciones de `readelf -l` y `tiff2rgba` con un archivo pequeño.

Se concluye que el kernel de KVM-FUZZ, al ser más simple, ejecuta menos instrucciones que Linux para satisfacer las llamadas al sistema, lo que causa una mejora en el rendimiento significativa.

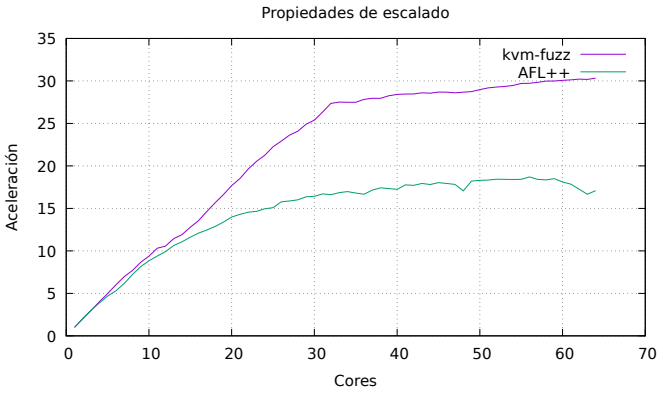

Figura 3. Comparación de aceleración en la velocidad de ejecución en función del número de cores con KVM-FUZZ y AFL++.

V-B. Escalado

Se desean medir las propiedades de escalado de un *fuzzer* como KVM-FUZZ, en el que cada instancia hace uso de una máquina virtual aislada, frente a un *fuzzer* tradicional como AFL++, en el que cada instancia hace uso de `fork()` para ejecutar el programa objetivo bajo el kernel anfitrión. Para ello, se ha medido la aceleración (definida como velocidad de ejecución paralela entre velocidad de ejecución secuencial) de ambos *fuzzers* con un programa de prueba en función del número de cores.

Para la ejecución con múltiples cores, KVM-FUZZ dispone de una opción de ejecución para activar multithreading con un determinado número de threads. AFL++, por su parte, no dispone de un método automático para ello, sino que requiere lanzar múltiples procesos por separado y usar otro programa para obtener las estadísticas.

Los resultados se pueden observar en la Figura 3. Se puede apreciar que KVM-FUZZ presenta un escalado lineal hasta los 32 cores. A partir de los 32 cores, la ganancia de aceleración disminuye debido al Hyper-Threading, pero sigue siendo positiva. Este escalado lineal permite la máxima utilización de los recursos de la CPU, ya que no hay apenas contención entre núcleos que pueda limitar el rendimiento.

Por otro lado, el rendimiento de AFL++ no escala linealmente. Podemos ver que conforme aumenta el número de cores disminuye la ganancia de aceleración, incluso antes de llegar al umbral en el que se comienza a usar *Hyper-Threading* [18]. Esto conlleva un significativo desperdicio de los recursos de procesadores con un número de núcleos considerable, como el usado para el experimento o como los que son ampliamente utilizados en campañas de *fuzzing*.

V-C. Velocidad de restauración de snapshots

Para medir el rendimiento del mecanismo de restauración de *snapshots*, se ha creado un pequeño programa que reserva un cierto número de páginas de memoria y escribe en ellas. El objetivo es medir la velocidad a la que se puede restaurar una *snapshot* en función de la cantidad de memoria que modifique el programa objetivo. Para ello, KVM-FUZZ y NYX toman una *snapshot* después de que el programa haya reservado la memoria, y ejecutan sólo hasta después de escribir en ella. AFL++, por su parte, hace uso del *fork server* con inicialización diferida (Sección III-A1) en el mismo punto. A pesar de que el *fork server* de AFL++ no usa estrictamente

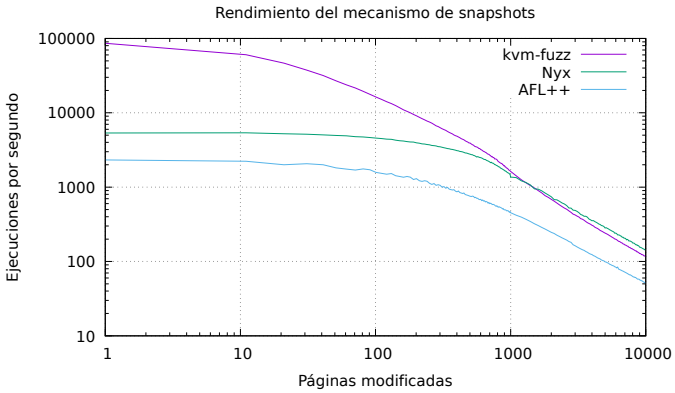

Figura 4. Comparación de velocidad de ejecución de un programa que modifica N páginas de memoria con KVM-FUZZ, NYX y AFL++.

hablando un mecanismo de *snapshots*, Linux hace uso de *copy on write* en `fork()`. De esta forma, las páginas de memoria del proceso hijo sólo se duplican cuando son modificadas, asemejándose al proceso de restauración diferencial de memoria de los mecanismos de *snapshots*. Se ha decidido hacer uso del *fork server* en lugar del mecanismo de *snapshots* que incorpora AFL++ porque su implementación (Sección III-A2) restaura todas las páginas, independientemente de si han sido modificadas o no.

Los resultados se ilustran en la Figura 4. Como se puede apreciar, KVM-FUZZ presenta una mejora de rendimiento de hasta un orden de magnitud para programas que modifican pocas páginas de memoria. La diferencia frente a NYX se debe al uso de memoria del kernel. Según los autores de NYX, *"we observe that when the target only dirties ten pages, we reload almost a 100 pages in the kernel"* [5]. Además, NYX también restaura el estado de los dispositivos emulados por QEMU. Por su parte, el kernel de KVM-FUZZ, al ser más simple, hace un menor uso de memoria, dando lugar a esas diferencias cuando el programa objetivo modifica pocas páginas.

Sin embargo, dicha diferencia de uso de memoria del kernel se vuelve poco significativa cuando el programa objetivo modifica una cantidad más elevada de páginas de memoria. En estos casos, el rendimiento se ve limitado por la velocidad hardware de la memoria, ya que hay que copiar un gran número de páginas para restaurar el estado del programa objetivo. Por ello, bajo estas condiciones el mecanismo de *snapshots* de KVM-FUZZ alcanza un rendimiento similar al de NYX.

Por su parte, AFL++ sigue una curva similar a NYX, pero con un notable peor rendimiento. Esto se debe principalmente a que el *fork server* debe esperar a que el proceso hijo termine en cada ejecución. Mientras tanto, los otros dos *fuzzers* finalizan la ejecución justo después de que el programa haya modificado las páginas indicadas, ahorrándose la pérdida de rendimiento que puede suponer que el programa ejecute las rutinas de salida y finalice.

VI. CONCLUSIONES

En este artículo, hemos introducido un enfoque para *fuzzear* programas de usuario x86-64 de los que no se dispone código fuente haciendo uso de KVM. Nuestra implementación KVM-FUZZ, que incluye un kernel y un hipervisor propios, hace uso de técnicas punteras como cobertura de código asistida por hardware, restauración de *snapshots* y emulación de llamadas al sistema. Como se ha mostrado en los experimentos, esto le permite alcanzar un rendimiento superior al de otros *fuzzers* similares, manteniendo mecanismos esenciales como la introspección del programa objetivo y el escalado lineal.

AGRADECIMIENTOS

Este proyecto ha sido financiado parcialmente por el Gobierno de España (Ministerio de Ciencia e Innovación) a través del proyecto SICRAC (PID2020-114495RB-I00).

REFERENCIAS

[1] M. Stone, "The more you know, the more you know you don't know: A year in review of 0-days used in-the-wild in 2021," https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html, Accedido: 25 de marzo, 2023.

[2] CVE Details, "Vulnerabilidades de corrupción de memoria publicadas en 2022," https://www.cvedetails.com/vulnerability-list/year-2022/opmemc-1/memory-corruption.html, Accedido: 25 de marzo, 2023.

[3] Google, "OSS-Fuzz: continuous fuzzing for open source software," https://github.com/google/oss-fuzz, Accedido: 25 de marzo, 2023.

[4] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, ser. WOOT'20. USA: USENIX Association, 2020.

[5] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[6] h0mbre, "Fuzzing like a caveman 4: Snapshot/code coverage fuzzer!" https://h0mbre.github.io/Fuzzing-Like-A-Caveman-4/, Accedido: 25 de marzo, 2023.

[7] A. Fioraldi and M. Heuse, "AFL++ Snapshot LKM," https://github.com/AFLplusplus/AFL-Snapshot-LKM, Accedido: 25 de marzo, 2023.

[8] Y. Song, H. Wang, and T. Soyata, *Hardware and Software Aspects of VM-Based Mobile-Cloud Offloading*, 08 2015, pp. 247–271.

[9] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "Ptfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. PP, pp. 1–1, 06 2018.

[10] B. Falk, "Mesos," https://github.com/gamozolabs/mesos, Accedido: 25 de marzo, 2023.

[11] "DynamoRIO," https://github.com/DynamoRIO/dynamorio, Accedido: 25 de marzo, 2023.

[12] 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*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[13] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1683–1700. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/nagy

[14] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1497–1511.

[15] M. Heuse, "AFL-Dyninst," https://github.com/vanhauser-thc/afl-dyninst, Accedido: 25 de marzo, 2023.

[16] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2313–2328. [Online]. Available: https://doi.org/10.1145/3133956.3134046

[17] S. Schumilo and C. Aschermann, "libxdc," https://github.com/AFLplusplus/AFL-Snapshot-LKM, Accedido: 25 de marzo, 2023.

[18] B. Falk, "Tweet sobre el rendimiento de fork," https://twitter.com/gamozolabs/status/1265292822292230144, Accedido: 25 de marzo, 2023.