



UNIVERSIDAD DE MÁLAGA



Grado en Ingeniería Informática

Cortafuegos de Aplicaciones Blockchain

Blockchain Application Firewall

Realizado por
Antonio José González Castillo

Tutorizado por
Isaac Agudo Ruíz

Cotutorizado por
Francisco José Jaime Rodríguez

Departamento
Lenguajes y Ciencias de la Computación

MÁLAGA, SEPTIEMBRE DE 2025



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA INFORMÁTICA

Cortafuegos de Aplicaciones Blockchain

Blockchain Application Firewall

Realizado por
Antonio José González Castillo

Tutorizado por
Isaac Agudo Ruiz

Cotutorizado por
Francisco José Jaime Rodríguez

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2025

Agradecimientos

Finalizar esta etapa académica supone para mí la culminación de un largo camino, marcado por numerosos retos, sacrificios y aprendizajes. A lo largo de estos años, he contado con el apoyo incondicional de muchas personas, cuyo esfuerzo y dedicación han sido fundamentales para alcanzar este objetivo.

En primer lugar, quiero expresar mi gratitud a mi tutor, D. Isaac Agudo Ruiz y a mi cotutor, D. Francisco José Jaime Rodríguez, por su apoyo y su guía en este proyecto.

Asimismo, quiero agradecer a todos los profesores, quienes a lo largo de mi formación en la Escuela Técnica Superior de Ingeniería Informática, me han transmitido conocimientos esenciales y valores fundamentales para el ejercicio de mi profesión.

No puedo dejar de mencionar a mis amigos de clase, con quienes he compartido incontables horas de estudio, trabajo y esfuerzo. Gracias a ellos, los momentos de dificultad fueron más llevaderos y los éxitos más gratificantes. Nuestro apoyo, motivación y confianza nos ha impulsado mutuamente a seguir adelante.

El más especial de los agradecimientos, lo merece mi familia, cuyo respaldo ha sido inquebrantable a lo largo de estos años. A mis padres, por su amor incondicional, por inculcarme la importancia del esfuerzo y la perseverancia, y por brindarme la oportunidad de alcanzar este logro. A mi hermana, por su compañía y su soporte en los momentos más exigentes de esta etapa. Sin su apoyo, este camino habría sido infinitamente más difícil.

Finalmente, quiero agradecer a todas aquellas personas que, de una u otra manera, han contribuido a mi formación y crecimiento, ya sea mediante un consejo oportuno, una palabra de ánimo o simplemente su presencia en mi vida. Cada uno de ellos ha sido parte de este proceso y ha dejado una marca imborrable en mi trayectoria.

Este trabajo es el fruto de años de constancia, pero también de la colaboración y el apoyo de todas las personas que han estado a mi lado. A todos ellos, mi más profundo agradecimiento.

- Antonio José González Castillo

Resumen

En la era digital, la ciberseguridad se ha convertido en un aspecto crucial para la protección de la información y sistemas distribuidos. Con la creciente adopción de Blockchain en sectores como finanzas, logística e identidad digital, los ciberdelincuentes han comenzado a desarrollar estrategias para comprometer estas redes. Aunque la naturaleza descentralizada de Blockchain proporciona ventajas en términos de seguridad e integridad de datos, no es inmune a ataques.

Los ataques tradicionales como inyecciones SQL, MITM o malware basado en archivos han evolucionado en el contexto de Blockchain. Ahora se observan amenazas específicas como la manipulación de contratos inteligentes, el Sybil Attack y los ataques dirigidos a nodos y redes de comunicación. Además, el problema del spam en Blockchain, donde atacantes saturan la red con transacciones inútiles para degradar el rendimiento, es un desafío creciente.

En este contexto, el concepto de *Blockchain Application Firewall* (BAF) surge como una solución para filtrar el tráfico y garantizar que solo nodos autorizados puedan interactuar con la red de manera segura. A diferencia de los firewalls tradicionales, diseñados para proteger infraestructuras centralizadas, un BAF debe abordar amenazas específicas de Blockchain.

Este TFG se centra en diseñar e implementar un *Blockchain Application Firewall* básico, así como contextualizar la seguridad en la Blockchain y sentar las bases en un campo que aún está en desarrollo. Entre los pocos ejemplos comerciales, destaca *Kaleido* como solución privada ya existente.

Pese a presentar diversos desafíos, como la integración con redes descentralizadas sin afectar su rendimiento, la detección precisa de ataques sin generar falsos positivos y la escalabilidad para manejar grandes volúmenes de transacciones, se emplearán enfoques como pruebas en entornos controlados, diseño modular y buenas prácticas de programación para preservar la viabilidad del firewall.

Palabras clave: *Blockchain Application Firewall – Ataques – Ethereum – Proxy*

Abstract

In the digital age, cybersecurity has become crucial for protecting distributed information and systems. With the growing adoption of Blockchain in sectors such as finance, logistics, and digital identity, cybercriminals have begun to develop strategies to compromise these networks. Although the decentralised nature of Blockchain provides advantages in terms of security and data integrity, it is not immune to attacks.

Traditional attacks such as SQL injections, MITM, or file-based malware have evolved in the context of Blockchain. Specific threats such as smart contract manipulation, Sybil Attack, and attacks targeting nodes and communication networks are now being observed. In addition, the problem of spam on Blockchain, where attackers saturate the network with useless transactions to degrade performance, is a growing challenge.

In this context, the concept of a Blockchain Application Firewall (BAF) emerges as a solution to filter traffic and ensure that only authorised nodes can interact with the network securely. Unlike traditional firewalls, designed to protect centralised infrastructures, a BAF must address threats specific to Blockchain.

This thesis focuses on designing and implementing a basic Blockchain Application Firewall, as well as contextualising security in Blockchain and laying the foundations in a field that is still under development. With few commercial examples, Kaleido stands out as an existing private solution.

Despite presenting various challenges, such as integration with decentralised networks without affecting their performance, accurate detection of attacks without generating false positives, and scalability to handle large volumes of transactions, approaches such as testing in controlled environments, modular design, and good programming practices will be employed to preserve the firewall's viability.

Keywords: Blockchain Application Firewall – Attacks – Ethereum – Proxy

Índice

1. Introducción	13
1.1. Motivación	13
1.2. Objetivos	14
1.3. Metodología	16
1.4. Estructura de la Memoria	18
2. Estado del Arte	21
2.1. Introducción a la Blockchain	21
2.2. Fortalezas y debilidades	23
2.2.1. Ventajas más importantes de la tecnología Blockchain:	23
2.2.2. Desventajas más importantes de la tecnología Blockchain:	25
2.3. Conceptos clave	26
2.3.1. Smart Contracts	26
2.3.2. Tokens y Estándares de Tokenización	27
2.3.3. Tipos de redes	29
2.4. Plataformas Blockchain	31
2.5. Tecnologías Blockchain Relevantes	38
3. Estudio de la solución y análisis de amenazas	43

3.1.	Sistemas de seguridad existentes y análisis del gap tecnológico	43
3.1.1.	Análisis de soluciones comerciales: Kaleido BAF y competidores	47
3.2.	Ánalisis de amenazas Blockchain	48
3.2.1.	Riesgos generales al exponer un nodo Ethereum en producción	48
3.2.2.	Spam en la mempool (mempool flooding)	49
3.2.3.	Ataques de repetición (Replay attacks)	50
3.2.4.	Payloads maliciosos dirigidos a contratos	51
3.2.5.	Envío masivo desde identidades Sybil (Sybil Attack)	52
3.2.6.	Consultas abusivas y DoS	53
3.2.7.	MEV y front-running	54
3.2.8.	Impacto: cuantificación de riesgos	55
3.2.9.	Priorización: matriz de criticidad y frecuencia	56
4.	Especificación de requisitos y diseño de la arquitectura	57
4.1.	Especificación funcional	57
4.1.1.	Casos de uso: escenarios operacionales detallados	59
4.1.2.	Interfaces requeridas	62
4.2.	Especificación no funcional	64
4.3.	Restricciones y limitaciones	66
4.4.	Arquitectura general: principios de diseño y patrones	68
4.4.1.	Patrones arquitectónicos implementados	68

4.4.2. Arquitectura modular por componentes	69
4.4.3. Principios de diseño aplicados	71
4.5. Modelo arquitectónico: patrón proxy con análisis en tiempo real	72
4.5.1. Arquitectura de proxy transparente	72
4.6. Stack tecnológico seleccionado	73
4.6.1. Node.js como runtime principal	73
4.6.2. TypeScript para desarrollo type-safe	75
4.6.3. Redis para estado distribuido	75
4.6.4. Express.js para servidor HTTP	76
4.7. Bibliotecas especializadas y dependencias críticas	77
4.7.1. Ethers.js para interacción Blockchain	77
4.7.2. Bibliotecas criptográficas nativas	78
4.7.3. Bibliotecas de observabilidad	78
5. Implementación y Desarrollo	81
5.1. Proyecto completo: el BAF como producto	81
5.2. Módulos en detalle: función y composición	83
5.3. Análisis de componentes clave	88
6. Pruebas y Validación del Sistema	97
6.1. Metodología de Pruebas	97

6.2. Pruebas Unitarias	98
6.3. Pruebas de Integración y Sistema	100
6.4. Pruebas de Seguridad	103
6.5. Análisis de resultados y eficacia	123
7. Conclusiones y líneas futuras	125
7.1. Conclusiones generales	125
7.2. Relevancia e implicaciones del proyecto	127
7.3. Líneas de trabajo futuro	128
7.4. Reflexiones finales	130
7.5. Cierre	131
Bibliografía	133
Apéndice A. Manual de Usuario	137
A.1. Requisitos del sistema	137
A.2. Instalación paso a paso	138
A.3. Compilar y arrancar	141
A.4. Pruebas de Funcionalidades	141
A.5. Enviar Transacciones	141
A.6. Comandos para ejecutar tests	142
A.7. Resolución de problemas (troubleshooting)	143

Apéndice B. Documentación API	145
B.1. Autenticación	145
B.2. Endpoints públicos (información y métricas)	146
B.3. JSON-RPC (proxy protegido)	147
B.4. Endpoints administrativos (API /admin)	148
B.5. Analytics	150
B.6. Seguridad y comportamiento esperado	151
B.7. Pruebas de estrés y casos límite	152
B.8. Flujos de trabajo recomendados	152
B.9. Soporte Unicode y caracteres especiales	153
B.10. Códigos de estado (resumen rápido)	153
B.11. Notas operativas y recomendaciones	154
Apéndice C. Bot de notificaciones de Telegram. Manual y guía de usuario.	155
C.1. Para qué sirve	155
C.2. Qué hace	155
C.3. Arquitectura básica	156
C.4. Beneficios	156
C.5. Paso a paso de creación y puesta en marcha	156
C.5.1. Crear el bot en Telegram y conseguir token del bot	157
C.5.2. Obtener el CHAT_ID (destino de las notificaciones)	157

C.5.3. Estructura del proyecto y variables de entorno	158
C.5.4. Puesta en marcha	158

1

Introducción

Esta es una sección introductoria para presentar la motivación a la hora de realizar este Trabajo de Fin de Grado (en lo sucesivo, TFG), los principales objetivos que se pretenden alcanzar con su ejecución, la metodología utilizada para llevarlo a cabo y la estructura empleada en el documento.

1.1. Motivación

El creciente interés por la ciberseguridad, impulsado por las estrategias tecnológicas autonómicas y la llegada de grandes compañías a Málaga, ha evidenciado la necesidad de profesionales especializados y de soluciones innovadoras para proteger los nuevos entornos digitales. En este contexto, el desarrollo de competencias en la defensa de sistemas y redes ha orientado mi interés y trayectoria profesional hacia la seguridad informática, reforzando el compromiso con la identificación y mitigación de riesgos en cualquier plataforma.

La amplia experiencia de mi tutor y cotutor en tecnologías Blockchain ha sido clave para la aproximación a esta temática y definir los objetivos de este Trabajo de Fin de Grado. Su conocimiento en este área, sumado a mi atracción por el tema, ha permitido identificar las principales carencias en la protección de aplicaciones descentralizadas y diseñar un enfoque que combine prácticas clásicas de ciberseguridad con técnicas propias del entorno Blockchain.

A pesar del rápido crecimiento de las redes y las aplicaciones descentralizadas (dApps), no existe a día de hoy un mecanismo estándar para inspeccionar y filtrar el tráfico a nivel de aplicación en la cadena de bloques. Frente a esta carencia, el proyecto “Blockchain Applica-

tion Firewall” propone el diseño de un cortafuegos especializado que incorpore detección de anomalías, análisis de transacciones y bloqueo de comportamientos maliciosos, adaptándose a la naturaleza distribuida y transparente de estas plataformas.

Con ello, este TFG busca no solo afianzar los conocimientos en ciberseguridad y Blockchain, sino también ofrecer una solución pionera de código abierto que continúe en constante desarrollo y democratice el acceso a herramientas de seguridad Blockchain avanzadas. ”NodeGuard Blockchain Application Firewall”(abreviado, BAF) aspira a mejorar la robustez de los entornos descentralizados, contribuir al establecimiento de nuevos estándares en la protección de la cadena de bloques, y proporcionar la capacidad de proteger infraestructuras contra las amenazas específicas de este ecosistema en constante evolución.

1.2. Objetivos

Si bien Blockchain aporta garantías de inmutabilidad e integridad, su interfaz de ejecución (contratos inteligentes y nodos que exponen APIs JSON-RPC) introduce superficies de ataque propias. En paralelo, la práctica de asegurar infraestructuras tradicionales mediante cortafuegos y sistemas perimetrales ha demostrado su eficacia; sin embargo, las características intrínsecas de las redes Blockchain requieren soluciones adaptadas que comprendan la semántica de las transacciones y los patrones de comportamiento específicos del ecosistema descentralizado.

El presente TFG tiene como objetivo aplicar los conocimientos adquiridos en diferentes campos del Grado en Ingeniería Informática para desarrollar NodeGuard BAF, un componente de inspección y control a nivel de aplicación diseñado para filtrar, analizar y mitigar tráfico malicioso dirigido a nodos y contratos en entornos Ethereum. La solución propuesta actuará como proxy inteligente entre clientes y nodos RPC. Implementará capacidades de detección multicapa, incluyendo reglas estáticas, análisis heurístico y técnicas de aprendizaje automático para identificar amenazas como ataques Sybil, DoS y manipulación de protocolos DeFi.

Objetivo general. Diseñar, implementar y validar un firewall de aplicaciones Blockchain de código abierto que permita la inspección en tiempo real del tráfico JSON-RPC, proporcionando mecanismos de detección y mitigación adaptativos para amenazas específicas del ecosistema Blockchain, con especial énfasis en la preservación del rendimiento y la facilidad de despliegue en entornos de desarrollo y producción.

Objetivos específicos sintetizados. Para alcanzar el objetivo general, el desarrollo del TFG se estructurará en las siguientes fases:

1. **Diseñar una arquitectura modular** que aporte al proyecto una estructura clara y organizada, facilitando la comprensión y el desarrollo del código, así como la posibilidad de ampliar funcionalidades, integrar nuevas reglas de seguridad y mantener el sistema de forma más eficiente en el tiempo.
2. **Implementar un prototipo funcional** en TypeScript/Node.js que actúe como proxy/interceptor transparente entre clientes y nodos Ethereum, incorporando un sistema de reglas estáticas configurables y mecanismos de detección heurística dinámica capaces de adaptarse a patrones de ataque emergentes. Asimismo, incluir capacidades de parsing de transacciones con soporte para los principales estándares EIP (155, 2718, 1559, 2930) y filtrado de acciones maliciosas.
3. **Desarrollar un entorno controlado** que posibilite la evaluación del sistema mediante redes de prueba y configuraciones locales, garantizando la reproducibilidad de los experimentos y permitiendo la simulación de ataques reales en Blockchain sin comprometer la seguridad ni la estabilidad de las redes principales.
4. **Definir y ejecutar una batería de experimentos** en distintos escenarios, orientados a mitigar las amenazas que detallaremos posteriormente en el transcurso de este documento. Se pretende también mantener una proporción de falsos positivos y negativos adecuada.
5. **Analizar los resultados y evaluar las limitaciones** del sistema, realizando una retrospección global que identifique áreas de mejora y establezca líneas futuras de investigación para ampliar las capacidades del firewall.

1.3. Metodología

La metodología adoptada para la elaboración de este Trabajo de Fin de Grado se fundamentó en un enfoque estructurado y ágil, orientado a lograr un avance progresivo, adaptable a los cambios y centrado en la obtención de resultados. El desarrollo se organizó en distintas etapas, abordadas de forma iterativa. No obstante, en el comienzo se partió de una idea basada en dos preguntas clave:

- **¿Qué se quiere hacer?**
- **¿Qué se puede hacer a partir de esto?**

A raíz de sus respuestas y el consenso alcanzado en las primeras reuniones con los tutores, se añadieron sucesivas ideas, mejoras y funcionalidades sin que éstas perjudicaran lo desarrollado hasta el momento. Sobre este punto de partida, se han utilizado diferentes herramientas como Notion y el archiconocido GitHub para llevar a cabo la organización de este proyecto.

Notion es una aplicación que permite la creación de notas, llamadas páginas, cuyo contenido puede ser variado: texto, imágenes, tablas, listas, archivos, enlaces, etc. Gracias a esto, las actividades principales fueron divididas en subtareas más pequeñas, específicas y manejables, clasificadas según su finalidad. El uso de las diferentes vistas que ofrece Notion para representar el progreso del proyecto (Kanban, Gantt, etc.) facilita una visualización temporal de las tareas y sus dependencias.

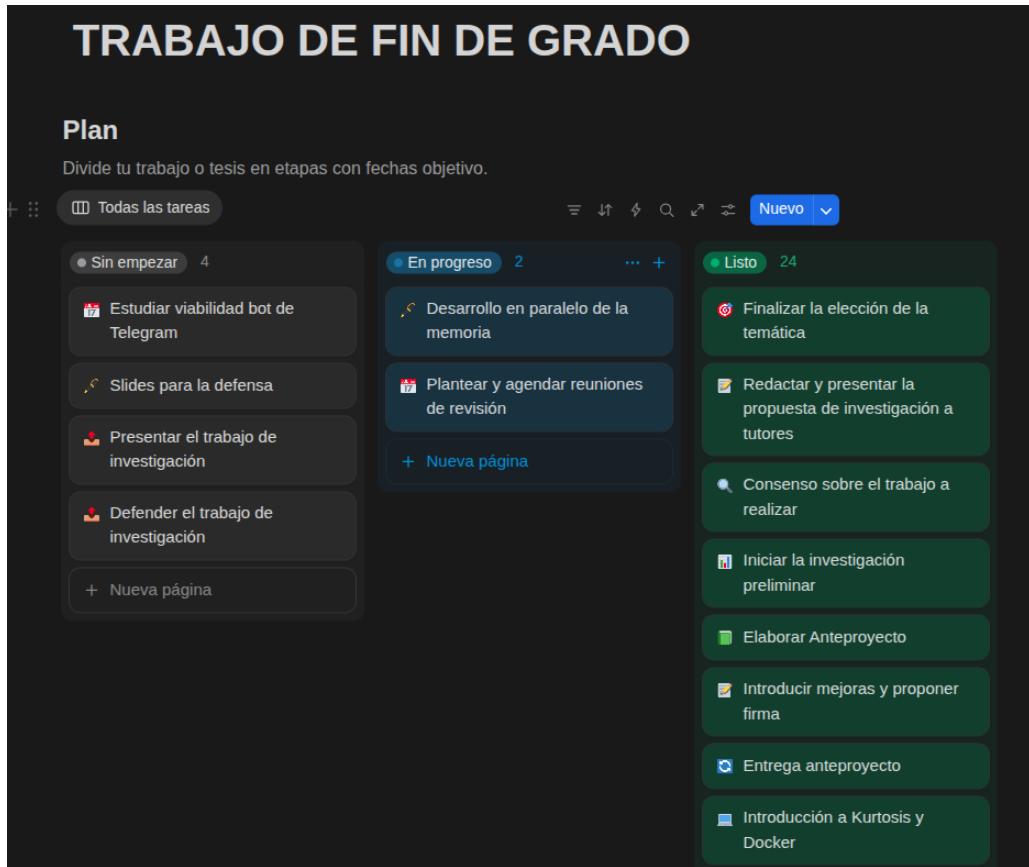


Figura 1: Vista de Kanban en Notion

Todo esto, ligado al enfoque escogido, permitió estructurar el trabajo en seis etapas principales, que se describen a continuación:

Etapa 1: Revisión bibliográfica y análisis inicial. En esta fase preliminar, se realizó un estudio del estado actual de la tecnología Blockchain, prestando especial atención a sus principios fundamentales y a las cuestiones relativas a su seguridad. Asimismo, se investigaron aspectos teóricos sobre el funcionamiento de mecanismos de seguridad activa como los firewalls y herramientas para la integración conjunta de ambas temáticas.

Etapa 2: Extracción de requisitos y configuración de frameworks. Con la base teórica ya consolidada, se definieron los requisitos funcionales y no funcionales a implementar posteriormente en nuestro BAF y se procedió a escoger las herramientas necesarias para su diseño y desarrollo. Tras esto, se llevó a cabo la familiarización con los diferentes entornos y su configuración para la puesta en marcha del diseño.

Etapa 3: Diseño arquitectónico e implementación modular. Esta fase se centra en la definición de la arquitectura del sistema y su posterior materialización. Se estableció una estructura modular que separa claramente las responsabilidades. Comenzando por los componentes fundamentales, se fue progresando hacia funcionalidades avanzadas, manteniendo la integridad del sistema en cada iteración y validando la correcta integración entre módulos.

Etapa 4: Validación experimental y evaluación de rendimiento. Se diseño e implementó una suite de pruebas que incluye pruebas unitarias, de integración y de sistema. Se realizaron experimentos controlados para evaluar la eficacia de detección mediante simulación de ataques conocidos, control del desempeño y análisis de falsos positivos/negativos.

Etapa 5: Análisis de resultados y documentación. Se llevó a cabo el análisis de los resultados experimentales, la comparación con soluciones ideales y la identificación de limitaciones del enfoque propuesto. Paralelamente, se desarrolló la documentación técnica completa del sistema, incluyendo manuales de instalación, configuración y operación.

Etapa 6: Redacción de la memoria. Esta fase final incluyó la elaboración de la memoria del TFG siguiendo los estándares académicos de la Universidad de Málaga, con especial atención a la cohesión argumental, la fundamentación técnica de las decisiones adoptadas y la presentación clara de los resultados obtenidos.

La metodología adoptada garantizó un desarrollo sistemático y reproducible, facilitando la trazabilidad de decisiones técnicas y la validación continua de los resultados obtenidos.

1.4. Estructura de la Memoria

En cierta concordancia con las etapas descritas en la metodología, el presente documento se organiza en siete capítulos principales, estructurados para conducir al lector desde la comprensión del problema hasta las conclusiones y recomendaciones de mejora. Esta disposición sigue una progresión lógica que facilita la comprensión tanto de los fundamentos teóricos como del desarrollo práctico del sistema NodeGuard BAF.

El **Capítulo 1 (Introducción)** presenta el contexto del trabajo, la motivación que impulsó su desarrollo, los objetivos planteados y la metodología empleada. También se describe la organización general del documento, proporcionando una hoja de ruta clara.

En el **Capítulo 2 (Estado del Arte)** se realiza una revisión sistemática de la literatura científica sobre seguridad Blockchain, se analizan los campos más relevantes del ecosistema descentralizado y se contextualiza el proyecto en relación a las plataformas y tecnologías actuales. Este capítulo establece el marco teórico que fundamenta las decisiones técnicas posteriores.

El **Capítulo 3 (Estudio de la solución y análisis de amenazas)** evalúa las soluciones existentes y caracteriza detalladamente las amenazas Blockchain que debe abordar el sistema para terminar de sentar las bases de la solución a desarrollar.

El **Capítulo 4 (Especificación de requisitos y diseño de la arquitectura)** define los requisitos funcionales y no funcionales del BAF, así como establece los criterios de evaluación que guiarán el desarrollo. Se incluye la especificación de casos de uso y flujos de trabajo. Por otra parte, expone la arquitectura modular propuesta, describe la interacción entre componentes principales, y justifica las decisiones de diseño adoptadas.

El **Capítulo 5 (Implementación y Desarrollo)** detalla la materialización del diseño mediante código, describiendo la implementación de cada módulo, las tecnologías empleadas y las funcionalidades desempeñadas. Se proporcionan fragmentos de código para mejorar la comprensión.

El **Capítulo 6 (Pruebas y Validación del Sistema)** describe la metodología de testing empleada y presenta los resultados de las pruebas y evalúa la eficacia del sistema mediante simulación de ataques reales. Se analizan además las distintas métricas consideradas por NodeGuard.

Finalmente, el **Capítulo 7 (Conclusiones y Líneas Futuras)** sintetiza los principales logros alcanzados, evalúa críticamente las limitaciones del sistema desarrollado, y propone direcciones para investigaciones futuras que puedan extender las capacidades defensivas del firewall hacia amenazas emergentes.

Además, se incluye un apartado de **Apéndices**, donde se proporciona documentación técnica complementaria como especificaciones detalladas de APIs y manuales de usuario.

2

Estado del Arte

Este capítulo proporciona una exposición detallada de los conceptos fundamentales y el panorama tecnológico actual en Blockchain, estableciendo el marco teórico fundamental para comprender las bases sobre las que se desarrolla NodeGuard BAF. El propósito es ofrecer una visión global y actualizada, que sirva como base para comprender y contextualizar los desarrollos técnicos llevados a cabo en los siguientes capítulos de este Trabajo de Fin de Grado. La selección de contenidos se ha orientado específicamente hacia aquellos aspectos que tienen implicaciones relevantes y que, por tanto, resultan pertinentes para el diseño e implementación de un firewall de aplicaciones Blockchain.

2.1. Introducción a la Blockchain

La tecnología *Blockchain*, también conocida como cadena de bloques, es una estructura de datos distribuida y descentralizada que funciona como un libro de contabilidad inmutable. Cada elemento de esta cadena, denominado bloque, agrupa un conjunto de transacciones o registros, junto con metadatos como la marca temporal, el hash criptográfico del bloque anterior y un valor aleatorio (nonce) que facilita la validación mediante mecanismos de consenso.

De este modo, los bloques quedan enlazados de manera secuencial y segura, formando una cadena inalterable: cualquier intento de modificar el contenido de un bloque invalida todos los hashes posteriores, lo que hace prácticamente imposible la manipulación sin el acuerdo de la mayoría de los participantes de la red [1].

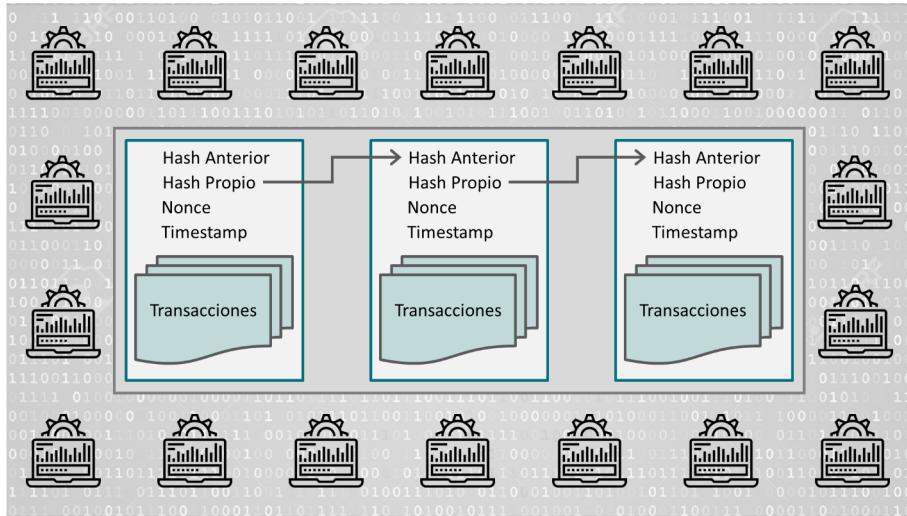


Figura 2: Representación gráfica de transacciones en Blockchain

Fuente: <https://xcoins.com/es/blog/prueba-de-trabajo-vs-prueba-de-participacion-los-pros-y-contras/>

El origen práctico de la Blockchain se remonta a 2008, cuando Satoshi Nakamoto publicó el célebre documento “*Bitcoin: A Peer-to-Peer Electronic Cash System*”. En él se propone una criptomoneda descentralizada capaz de registrar y validar transacciones directamente entre pares, sin intermediarios, gracias a un protocolo de consenso distribuido (*Proof-of-Work* en el caso de Bitcoin) [2]. Sin embargo, el potencial de esta tecnología ha trascendido con creces el ámbito monetario, extendiéndose a aplicaciones tan dispares como la gestión de la cadena de suministro, registros sanitarios, sistemas de identidad digital y ejecución de contratos inteligentes.

El funcionamiento básico de una Blockchain implica, primero, la creación de una transacción por parte de un usuario, quien la firma digitalmente con su clave privada para garantizar su autenticidad. Dicha transacción se propaga a través de la red de nodos —equipos que mantienen réplicas completas o parciales del libro de contabilidad—, donde cada nodo comprueba que el remitente posee los fondos o derechos que pretende transferir y que la firma es válida. Una vez validadas, las transacciones se agrupan en un nuevo bloque, que posteriormente es sometido a un proceso de consenso: la red compite por resolver un problema criptográfico (en *PoW*) o demuestra participación económica (en *Proof-of-Stake*), de modo que solo el bloque cuya resolución es aceptada por la mayoría se añade a la cadena [1, 3].

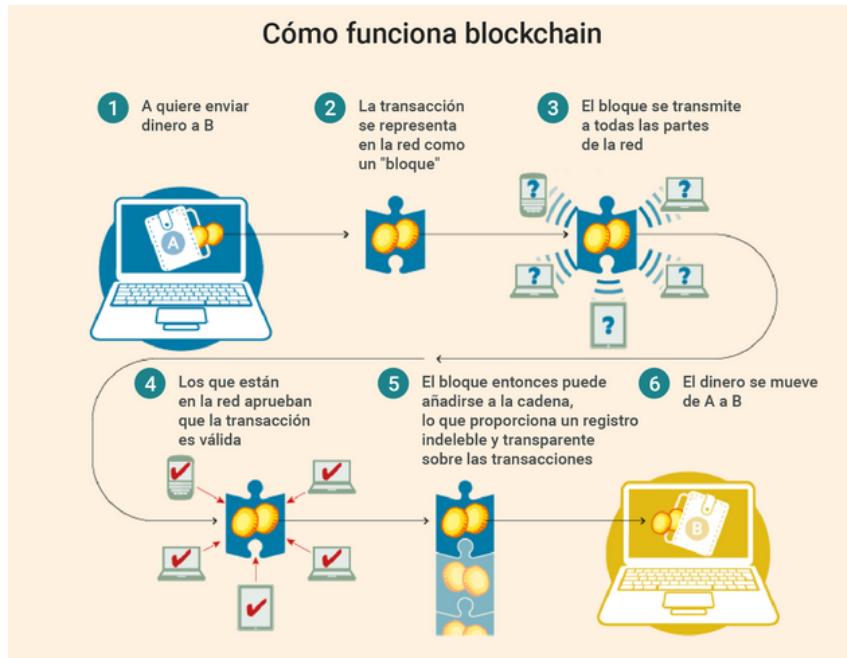


Figura 3: Funcionamiento esquematizado de Blockchain

Fuente: <https://www.xataka.com/especiales/que-es-Blockchain-la-explicacion-definitiva-para-la-tecnologia-mas-de-moda>

2.2. Fortalezas y debilidades

Al igual que ocurre con otras tecnologías emergentes, Blockchain presenta tanto características beneficiosas como retos, que es necesario analizar para apreciar plenamente su alcance y las consecuencias de su implementación. En las próximas líneas se abordarán sus principales fortalezas y limitaciones, con el propósito de ofrecer una visión objetiva que facilite una mejor comprensión de esta innovación transformadora [4].

2.2.1. Ventajas más importantes de la tecnología Blockchain:

- **Descentralización y confianza compartida.** La principal fortaleza de Blockchain es su capacidad para eliminar intermediarios gracias a su estructura descentralizada. No existe una autoridad central encargada de custodiar o validar los datos y cada nodo posee una copia del historial completo de transacciones, generando un entorno de confianza distribuida entre todos los participantes.

- **Seguridad criptográfica.** Utiliza técnicas criptográficas avanzadas para proteger la información y asegurar que los registros no puedan ser modificados una vez validados, lo que reduce significativamente el riesgo de fraude. Cada transacción y bloque emplea algoritmos de cifrado y funciones hash resistentes a colisiones, de modo que la integridad de la información y la privacidad de las claves privadas de los usuarios quedan protegidas.
- **Inmutabilidad.** Gracias a la vinculación criptográfica de bloques y al consenso distribuido, una vez que un bloque es aceptado, modificar sus datos (o los de cualquier bloque anterior) requeriría controlar más de la mitad de la potencia de cómputo de la red, un objetivo impracticable en redes de gran tamaño.
- **Reducción de costes operativos.** Al prescindir de terceros (como bancos o notarios), se disminuyen los costes asociados a procesos administrativos y de validación, lo que resulta atractivo para empresas y gobiernos.
- **Transparencia e integridad del sistema.** Todas las transacciones son visibles para los participantes de la red, lo cual fortalece la trazabilidad y permite auditorías automáticas y confiables.
- **Alta disponibilidad y resistencia a fallos.** Al estar replicada en múltiples nodos, la información no depende de un único punto de fallo, aumentando la resiliencia del sistema frente a ataques o caídas técnicas.

Estos atributos convierten a la Blockchain en una plataforma idónea para almacenar registros de cualquier naturaleza que requieran alta confianza: desde transacciones financieras y propiedad de activos digitales (como criptomonedas o NFTs) hasta históricos de contratos inteligentes.

El diseño modular de muchas implementaciones modernas de Blockchain permite, además, la incorporación de extensiones y mejoras. Por ejemplo, la capa de aplicación puede soportar *contratos inteligentes* que amplían la funcionalidad básica de transferencias de valor, mientras que distintas variantes de consenso (*Proof-of-Work*, *Proof-of-Stake*, BFT, entre otras) tratan de optimizar aspectos como eficiencia energética, escalabilidad o latencia de confirmación [5] .

2.2.2. Desventajas más importantes de la tecnología Blockchain:

Como no podía ser de otra forma, debido a la temática de este Trabajo de Fin de Grado, nos centramos más en las fortalezas que nos ofrece el ecosistema con el que trabajamos. No obstante, siempre es necesario conocer las limitaciones con las que contamos y los puntos no tan positivos de aquello en lo que se profundiza:

- **Problemas de escalabilidad.** A medida que crece el volumen de transacciones, también lo hace la cadena de bloques, lo que puede ralentizar los tiempos de procesamiento y afectar la eficiencia general del sistema.
- **Costes iniciales elevados y barreras técnicas.** Aunque reduce costes operativos a largo plazo, implementar una solución basada en Blockchain requiere inversión inicial en infraestructura, formación de personal y adaptación de procesos.
- **Desafíos regulatorios y legales.** La falta de un marco normativo claro sobre el uso y validez legal de las cadenas de bloques frena su adopción en sectores regulados como el financiero, sanitario o público.
- **Limitado control empresarial.** Al tratarse de redes descentralizadas, las empresas no tienen el control absoluto sobre el sistema, lo que puede generar fricciones en entornos donde se requiere gobernanza estricta.
- **Vulnerabilidades en Contratos Inteligentes.** Las vulnerabilidades de contratos inteligentes han resultado en pérdidas superiores a \$3 mil millones en 2022, destacando la importancia crítica de la auditoría y testing exhaustivo del código ejecutable en Blockchain [6, 7].

2.3. Conceptos clave

En el ámbito de la tecnología Blockchain resulta fundamental comprender ciertos conceptos que constituyen la base de su funcionamiento y de las aplicaciones que se construyen sobre ella. Entre los más relevantes destacan los *smart contracts*, los *tokens* y los distintos tipos de redes.

2.3.1. Smart Contracts

Los contratos inteligentes son aplicaciones informáticas diseñadas para ejecutarse automáticamente dentro de una red Blockchain. Su propósito es hacer efectivos acuerdos previamente definidos entre distintas partes, sin requerir la intervención de intermediarios, siempre que se cumplan determinadas condiciones establecidas de antemano. Su valor principal reside en garantizar la ejecución segura, transparente e inalterable de los términos acordados.

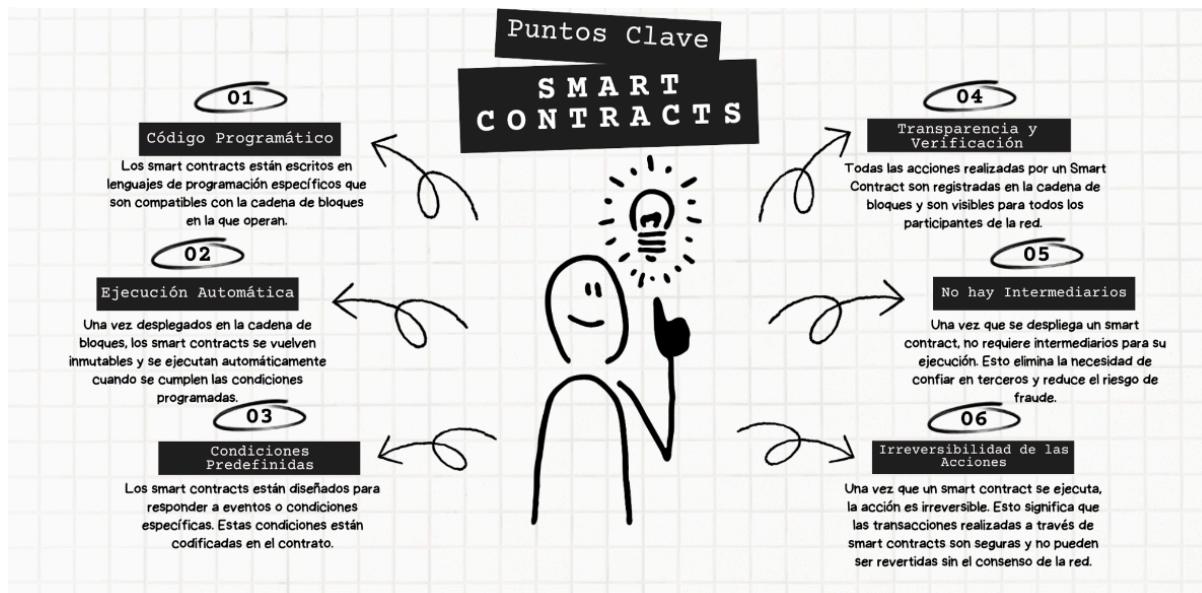


Figura 4: Características de los contratos inteligentes

Fuente: <https://www.unknowngravity.com/articulos/que-son-los-smart-contracts-que-utilidad-tienen>

A diferencia de los contratos convencionales, los smart contracts se alojan directamente en una cadena de bloques, lo cual impide cualquier modificación del código una vez ha sido desplegado. Esto brinda a las partes la certeza de que el acuerdo funcionará exactamente según lo establecido, sin riesgo de manipulación o interferencias externas. Para su desarrollo, se emplean lenguajes como Solidity, que permite codificar instrucciones comprensibles por la Ethereum Virtual Machine (EVM), responsable de ejecutar dichos contratos [8, 9].

El ciclo de vida de un contrato inteligente incluye distintas etapas: la escritura del código fuente, su compilación a bytecode, el despliegue en la red a través de una transacción y, por último, la interacción de usuarios o dApps. Una vez implementado, cualquier interacción que cumpla las condiciones definidas, desencadena su ejecución automática.

Entre sus principales ventajas se encuentran la automatización de procesos, la reducción de costes operativos, la eficiencia en la ejecución de tareas y una sólida protección frente a fraudes. Estas propiedades los hacen especialmente relevantes en ámbitos como las finanzas descentralizadas (DeFi), la trazabilidad en cadenas de suministro, los seguros, los entornos legales o la gestión de derechos de propiedad intelectual.

2.3.2. Tokens y Estándares de Tokenización

Los tokens son representaciones digitales de valor que operan sobre una red Blockchain, donde su existencia y transferencia se registran de forma descentralizada e inmutable. A diferencia del dinero convencional, emitido por gobiernos o bancos, los tokens se crean y gestionan mediante contratos inteligentes, lo que les confiere una gran flexibilidad e interoperabilidad. Desde el punto de vista funcional, los tokens se dividen en varias categorías:

- **Criptomonedas.** Diseñadas como medio de intercambio y reserva de valor. Ejemplos notables incluyen Bitcoin, Ethereum o Polkadot, las cuales pueden transferirse libremente dentro de sus respectivas redes.
- **Tokens de utilidad.** Otorgan derechos de acceso o ventajas dentro de un ecosistema determinado, como funcionalidades premium, descuentos o poder de voto.

Un caso conocido es el token BNB de Binance, que permite reducir comisiones de transacción y participar en lanzamientos de nuevos activos.

- **Tokens de seguridad.** Representan participaciones en un proyecto o activo subyacente, ofreciendo derechos económicos —por ejemplo, dividendos— y sujetándose a regulaciones financieras, al funcionar como instrumentos de inversión.
- **Tokens no fungibles (NFTs).** Activos únicos e indivisibles, empleados para certificar la autenticidad de obras de arte digitales, colecciónables o bienes virtuales, muy apreciados en el arte digital y los videojuegos online.

La creación de un token en una red Blockchain se fundamenta en el uso de estándares de protocolos que definen su comportamiento y aseguran su compatibilidad. Estos estándares proporcionan un marco común que garantiza la interoperabilidad entre aplicaciones descentralizadas, billeteras digitales y plataformas de intercambio. Entre los más relevantes se encuentran el ERC-20, empleado en la emisión de criptomonedas y tokens fungibles; el ERC-721, diseñado para la creación de tokens no fungibles (NFTs) que representan activos digitales únicos; y el ERC-1400, orientado a la emisión de tokens de seguridad, que incorpora requisitos adicionales de cumplimiento normativo y control de acceso [10, 11, 12].



Figura 5: Infografía de estándares de tokens de la Blockchain

Fuente: <https://www.criptonoticias.com/criptopedia/que-son-estandares-token/>

Cada estándar establece un conjunto mínimo de funciones que deben implementarse en el contrato inteligente correspondiente, como la transferencia de tokens entre direcciones, la consulta de saldos, la autorización de terceros para operar en nombre del propietario o la emisión y quema de unidades.

El contrato inteligente, escrito habitualmente en lenguajes como Solidity, no solo incorpora estas funciones básicas, sino que también define parámetros fundamentales como la oferta total de tokens, los mecanismos de distribución inicial y las reglas de transacción que regularán su circulación. Una vez desplegado en la Blockchain, adquiere un carácter autónomo e inmutable: gestiona de manera descentralizada la emisión, transferencia y control de los tokens, mientras que cada operación queda registrada en el libro mayor distribuido y validada mediante el consenso de los nodos de la red.

Para facilitar este proceso, han surgido plataformas gráficas que abstraen la programación, permitiendo a usuarios sin conocimientos técnicos definir parámetros como el nombre, el símbolo o el volumen máximo de emisión. De este modo, se democratiza el acceso a la tokenización de activos.

Finalmente, una vez en circulación, los tokens se almacenan en billeteras digitales asociadas a direcciones únicas. Todas las operaciones, ya sea emitir, enviar o recibir, quedan registradas en bloques validados mediante mecanismos de consenso como la prueba de trabajo (PoW) o la prueba de participación (PoS), lo que asegura transparencia y reduce el riesgo de fraude. Gracias a estas propiedades, los tokens están transformando sectores tan diversos como las finanzas descentralizadas (DeFi), la cadena de suministro, los seguros o la gestión de derechos de autor, al habilitar modelos de negocio más inclusivos, automáticos y seguros.

2.3.3. Tipos de redes

Las redes de Blockchain se clasifican en distintas categorías según su acceso, control y grado de descentralización. Esta clasificación es clave para entender sus diferencias y elegir la red más adecuada para cada caso de uso o proyecto.

DIFERENCIA ENTRE BLOCKCHAIN PRIVADA, PÚBLICA Y DE CONSORCIO

Característica	Blockchain Privada	Blockchain Pública	Consorcio Blockchain
Acceso	Privada	Pública	Pública/Privada
Consenso	Basada en la Organización	Pública	Nodos Seleccionados
Eficiencia	Alta	Baja	Alta
Centralización	SI	No	Parcial
Proceso de consenso	Basada en permisos	Basada en permisos	Sin permiso
Inmutabilidad	No completamente a prueba de manipulaciones	Completamente a prueba de manipulaciones	No completamente a prueba de manipulaciones

Figura 6: Comparativa visual de los diferentes tipos de redes

Fuente: <https://101blockchains.com/es/blockchain-para-principiantes/>

- **Redes públicas.** También conocidas como redes no permissionadas, están abiertas a cualquier usuario sin necesidad de autenticación previa. En ellas, la descentralización es completa, y todos los nodos tienen iguales privilegios para validar transacciones y mantener el libro contable. La información es accesible y verificable por cualquier participante, lo que garantiza un alto nivel de transparencia. Ejemplos destacados incluyen Bitcoin y Ethereum. Su mayor fortaleza es la resistencia a la censura, aunque requieren un considerable poder computacional para su mantenimiento.
- **Redes privadas.** Restringen el acceso exclusivamente a usuarios autorizados, bajo el control de una entidad central que administra permisos y gestiona la red. Ofrecen mayor privacidad y eficiencia, siendo ideales para entornos empresariales donde la confidencialidad y el control son prioritarios (trato de datos sensibles, procesos comerciales internos, etc.). Sin embargo, su centralización implica menor resistencia a la censura.

- **Redes permisionadas o de consorcio.** Redes permisionadas o de consorcio. Se sitúan entre las redes públicas y privadas, donde múltiples organizaciones comparten la administración y control de la red. Este modelo equilibra transparencia y descentralización relativa con un control restringido a participantes autorizados. Es común en colaboraciones interempresariales como cadenas de suministro o el sector financiero.
- **Redes híbridas.** Combinan características de las redes públicas y privadas, permitiendo que ciertas partes de la red sean accesibles públicamente, mientras otras permanezcan privadas. Esta configuración ofrece flexibilidad para ajustar los niveles de acceso y control según las necesidades específicas del proyecto.

En conjunto, estos tipos de redes permiten diseñar soluciones Blockchain adaptadas a distintos escenarios, buscando un equilibrio entre transparencia, seguridad, control y eficiencia operativa [3, 13].

2.4. Plataformas Blockchain

En este apartado revisamos las plataformas Blockchain principales por popularidad y adopción, describiendo sus propiedades técnicas, su compatibilidad con el *EVM* (cuando aplique), sus casos de uso y su idoneidad para ser objeto de protección por un *Blockchain Application Firewall* (BAF) como *NodeGuard*. En este caso, destacaremos explícitamente Ethereum, plataforma con la que trabaja NodeGuard, por lo que la exposición sobre otras redes busca ofrecer un marco comparativo y justificar la elección.

Ethereum

Ethereum es la plataforma de contratos inteligentes líder y el ecosistema más amplio para dApps y finanzas descentralizadas. Está basada en una cadena de bloques pública que emplea un mecanismo de consenso Proof-of-Stake (PoS) para la validación de bloques, y expone de forma pública una API JSON-RPC usada por la mayoría de dApps y herramientas de desarrollo.

La arquitectura de Ethereum incluye conceptos clave como transacciones RLP, gas, mempool, y una máquina virtual (EVM) que ejecuta bytecode generado por compiladores de Solidity y otros lenguajes compatibles. La robustez del ecosistema y la gran cantidad de infraestructuras y APIs disponibles hacen de Ethereum el objetivo principal de muchos ataques dirigidos a la capa de aplicación, por ejemplo, inyección de payloads en `eth_sendRawTransaction` o ataques de spam en la mempool, lo que motiva el diseño de capas de herramientas como NodeGuard para inspeccionar y filtrar tráfico JSON-RPC [14, 9, 15].

Ethereum Virtual Machine (EVM)

- **Definición y objetivo:** La EVM es la máquina virtual responsable de ejecutar el bytecode de contratos de forma determinista en todos los nodos de la red, garantizando que la misma entrada y estado produzcan el mismo resultado en cada nodo honesto. Esta propiedad de determinismo es la base de la consistencia del estado global y permite la replicación segura de la cadena de bloques [9]
- **Modelo de ejecución:** Máquina de pila (palabras de 256 bits) con tres espacios principales de trabajo:
 - *stack*: estructura LIFO para parámetros y resultados intermedios;
 - *memory*: área volátil durante la ejecución;
 - *storage*: mapa persistente asociado a la cuenta del contrato.

Los opcodes (p. ej. *SLOAD*, *SSTORE*, *CALL*, *DELEGATECALL*, *STATICCALL*) determinan efectos sobre estos espacios y condicionan tanto el coste como las superficies de riesgo (lecturas intensivas, escrituras persistentes, llamadas delegadas).

- **Gas como control de recursos:** Cada opcode tiene un coste en gas fijado por la especificación; la suma de esos costes determina el gasto asociado a una ejecución. El gas actúa como mecanismo contable y limitador técnico (evita bucles infinitos y abuso de CPU/IO). Para el diseño de un BAF es crucial distinguir operaciones de lectura y escritura (las escrituras en *storage* suelen ser las más costosas) y priorizar la instrumentación de rutas que detecten caminos de ejecución de alto coste [16, 17, 9].

Nodos, transacciones y formatos

- **Estructura esencial de una transacción:** remitente (implícito por la firma), destinatario (vacío para creación de contrato), valor en wei, datos (payload con selector de función y argumentos) y parámetros de gas (límite y precios). La firma autentica al emisor y posibilita la recuperación de su dirección.
- **Formatos y versiones:** el formato de las transacciones está en proceso de evolución constante; los detalles protocolarios y las implicaciones de EIPs concretos (p. ej. EIP-2718, EIP-2930, EIP-1559) se abordan en la sección 2.5. Aquí basta señalar que dicha evolución obliga al perímetro (que en este caso será nuestro BAF) a implementar parsers y extractores de metadatos tolerantes a versiones múltiples.
- **Ejecución simulada (*eth_call*) vs inclusión on-chain:** las llamadas simuladas no alteran el estado global pero consumen recursos del nodo que las atiende; constituyen un vector de sobrecarga cuando se usan para desencadenar ejecuciones costosas. NodeGuard debe poder limitar la frecuencia, el gas simulado y el tiempo de ejecución de estas llamadas, así como aplicar cacheado para respuestas repetitivas de lectura [14].
- **Validaciones en el perímetro:**
 1. Comprobaciones sintácticas y límites de tamaño (payload, campos obligatorios).
 2. Verificaciones criptográficas ligeras (p. ej. coherencia de firma con remitente recuperado).
 3. Políticas operativas (cuotas por IP/cliente, listas negras/blancas, bloqueo de métodos sensibles como *debug_** o *trace_**).
 4. Escalado controlado: reenvío a validación completa o sandbox de ejecución sólo si la política lo requiere.

- **Señales operativas de abuso detectables por RPC:** ráfagas de *eth_call* costosos, envío masivo de transacciones mal formadas, uso indebido de métodos de depuración por clientes no autorizados, patrones de acceso a *storage* (indicativos de posible scraping o borrados masivos) y parámetros de gas anómalos que busquen manipular prioridad o congestionar el nodo, constituyen indicadores críticos para la detección temprana de ataques y la implementación de contramedidas proactivas en la capa de consenso.

Estructura de firmas y validación

- **Firma y autenticidad:** las transacciones emplean esquemas criptográficos (p. ej. ECD-SA) cuyos campos *v,r,s* permiten recuperar la dirección remitente; la inclusión de *chainId* en la firma (EIP-155) mitiga ataques de replay entre cadenas y es relevante para validar coherencia con la red destino [18].
- **Nonce y ordenación:** el nonce asegura la ordenación de transacciones por remitente y previene replays locales. En el perímetro, observar nonces atípicos (saltos, repeticiones o nonces adelantados) aporta una señal de anomalía sin requerir lecturas de estado completo.
- **Comprobación de fondos y coste de las validaciones:** comprobar saldo exacto exige consultar el estado del ledger, operación relativamente costosa. El BAF deberá aplicar heurísticas (umbral mínimo de gas, remitentes verificados, limitación temporal ante patrones sospechosos) y reservar consultas al estado para casos que justifiquen escalado.
- **Criterio de diseño. Balance validación/eficiencia:** realizar en el BAF verificaciones ligeras y determinísticas que descarten entradas manifiestamente inválidas y delegar en el nodo las comprobaciones más costosas (saldo, inclusión en mempool, validación completa), de modo que la protección preserve la disponibilidad y no se convierta en cuello de botella.

BNB Chain

BNB Chain (antes Binance Smart Chain, BSC) es una cadena de alto rendimiento diseñada para baja latencia y tarifas reducidas, manteniendo compatibilidad con la EVM para facilitar la portabilidad de proyectos desde Ethereum. BNB Chain opera como un ecosistema dual (Beacon Chain + Smart Chain históricamente), y ha conseguido una amplia adopción por su coste y rapidez. Para el diseño de un BAF, BNB Chain se comportaría de forma muy similar a Ethereum en cuanto a los vectores de ataque a nivel de aplicación, ya que también expone JSON-RPC y reproduce las mismas superficies (transacciones, mempool, calls) [19, 20].

Características relevantes

- **Compatibilidad:** EVM-compatible. Permite usar las mismas herramientas de desarrollo que en Ethereum.
- **Rendimiento:** orientada a rendimiento alto y costes reducidos.
- **Casos de uso:** dApps con necesidades de bajo coste por transacción.

Solana

Solana es una cadena diseñada también para alto rendimiento y latencias muy bajas, pero su modelo arquitectónico es distinto al EVM. Solana ejecuta programas en una máquina virtual propia (BPF programs) y emplea un conjunto de optimizaciones como Proof-of-History (PoH) complementario al consenso para ordenar rápidamente eventos. Por su diseño, muchas herramientas y vectores de ataque de Ethereum no aplican directamente, pero aparece una superficie distinta (programas BPF, cuentas con estado, mecanismos de confirmación y políticas de reintentos) que debe considerarse al desplegar un BAF en entornos heterogéneos. En general, NodeGuard, orientado inicialmente a entornos EVM y JSON-RPC, necesitaría adaptaciones semánticas importantes para comprender y filtrar peticiones en Solana. [21]

Características relevantes

- **Consenso:** diseño con Proof-of-History + PoS (arquitectura propia).

- **Contratos:** programas compilados a BPF (Rust, C).
- **Interfaz:** RPC con semántica y tipos diferentes al JSON-RPC EVM.
- **Casos de uso:** aplicaciones de muy alta frecuencia (juegos, micropagos, trading de alta frecuencia).

Polygon

Polygon es un conjunto de soluciones de escalado para Ethereum (PoS chain, zkEVM, Optimistic rollups). Su objetivo es reducir costes y aumentar rendimiento heredando la seguridad o compatibilidad del ecosistema Ethereum cuando corresponde (por ejemplo, Polygon PoS y Polygon zkEVM son compatibles con EVM). Para pruebas en devnet y la evaluación de un BAF, Polygon ofrece escenarios prácticos: L2s que conservan el mismo modelo JSON-RPC/EVM, pero con mayores tasas de transacción que permiten estresar reglas de rate-limiting y heurísticas de spam [22].

Características relevantes

- **Compatibilidad:** EVM (según variante, p. ej. zkEVM es EVM-equivalente).
- **Casos de uso:** escalado para dApps existentes en Ethereum.
- **Idoneidad para pruebas:** útil para validar reglas de alta carga y L2-specific behaviour.

Avalanche

Avalanche es un framework *network-of-networks* que proporciona cadenas L1 rápidas y la posibilidad de crear subnets personalizadas. Uno de sus componentes (C-Chain) es compatible con EVM y permite ejecutar contratos Solidity con rendimiento elevado y baja latencia. La existencia de subnets haría de Avalanche una plataforma interesante para evaluar BAFs en contextos multi-chain y para experimentar políticas por-subnet. [23]

Características relevantes

- **Compatibilidad:** C-Chain es EVM-compatible.
- **Particularidad:** subnets y configuración flexible de consenso y validación.
- **Casos de uso:** proyectos que requieren redes privadas o L1s customizadas.

Plataforma	Arquitectura	EVM Compatibile	Adaptación Requerida	Idoneidad
Ethereum	EVM Nativa	Compatible	Adaptado	5
BNB Chain	EVM Compatible	Si	Mínima	4
Polygon	EVM L2	Parcial / Sí (según variante)	Moderada	3
Avalanche	C-Chain EVM	Sí (C-Chain)	Moderada	3
Solana	BPF Propio	No (RPC diferente)	Completa	2

Cuadro 1: Evaluación de compatibilidad de NodeGuard con plataformas Blockchain

Leyenda: *Arquitectura* indica el tipo de diseño de consenso y ejecución de la plataforma. *EVM compatible* indica si la plataforma ejecuta contratos en la EVM o ofrece compatibilidad equivalente. *Adaptabilidad* = esfuerzo estimado para adaptar NodeGuard. *Idoneidad* = adecuación técnica para aplicar el BAF (5 = idóneo, 1 = inviable).

2.5. Tecnologías Blockchain Relevantes

En esta sección se sintetizan las tecnologías y protocolos que resultan especialmente relevantes para el diseño, la implementación y la evaluación de un *Blockchain Application Firewall* (BAF). El objetivo no es realizar una revisión enciclopédica, sino destacar aquellos elementos que condicionan de forma directa la interfaz, la lógica de filtrado y las métricas operativas de un BAF orientado a entornos Ethereum y EVM-compatibles.

Protocolos fundamentales y EIPs de referencia

La evolución de Ethereum se articula a través del proceso de *Ethereum Improvement Proposals* (EIPs), que actúan como el registro oficial de cambios y estandarizaciones del protocolo [24]. De entre las propuestas formales y los cambios de alto impacto, conviene subrayar, por su incidencia práctica en el tráfico RPC y en la semántica de las transacciones, las siguientes:

- **EIP-155 — Replay protection.** Introduce el uso del *chainId* en la firma de transacciones para mitigar ataques de *replay* entre cadenas. Desde la óptica de un BAF esto añade un campo de verificación ligera (coherencia entre firma y *chainId*) que puede descartarse a nivel de perímetro de manera rápida antes de aceptar el reenvío al nodo. La existencia de este campo reduce vectores de error al operar NodeGuard en entornos multi-chain. [18]
- **EIP-2718 — Typed Transaction Envelope.** Establece un contenedor tipado para transacciones que permite la coexistencia de varios formatos de transacción. Para un BAF implica la necesidad de parsers modulares capaces de reconocer y deserializar transacciones con distintos tipos, y de mantener una tabla de compatibilidad que evolucione con la especificación. [25]
- **EIP-2930 — Optional Access Lists.** Introduce las *access lists* (listas anticipadas de cuentas y slots de almacenamiento que la transacción planea tocar) para optimizar costes. Esto aporta metadatos útiles para heurísticas de coste y de riesgo: una transacción con *access list* bien formada puede predecir su impacto en gas y memoria, y por tanto permite al BAF decidir si procede un análisis profundo o un rechazo preventivo. [26, 27]

- **EIP-1559 – Fee market change.** Redefinió la economía de las tarifas al introducir la *base fee* y separar la propina al validador. Desde el punto de vista operativo del BAF, los modelos de detección de abuso basados en precios han de contemplar la nueva contabilidad (base fee dinámica, máximo dispuesto a pagar por remitente) y evitar reglas simples del tipo «baja tarifa = spam», sustituyéndolas por modelos que crucen precio, complejidad estimada de ejecución y reputación del remitente [eip2718 , eip2930 , 18, 28, 24].

Estos EIPs no sólo modifican formatos y costes: condicionan la forma en que un proxy debe extraer metadatos, priorizar la ejecución y decidir cuándo escalar una transacción a una verificación completa.

Mecanismos de consenso: implicaciones para la seguridad

El mecanismo de consenso subyacente determina propiedades críticas relativas a finalidad, reversibilidad, ventanas de ataque y el perfil de incentivos de los validadores. El cambio de Ethereum desde Proof-of-Work (PoW) a Proof-of-Stake (PoS), conocido como "The Merge", tuvo consecuencias notables: reducción del coste energético, cambios en la emisión y, sobre todo, alteración del modelo de incentivos y de las superficies de ataque asociadas a validadores en lugar de mineros [15, 29].

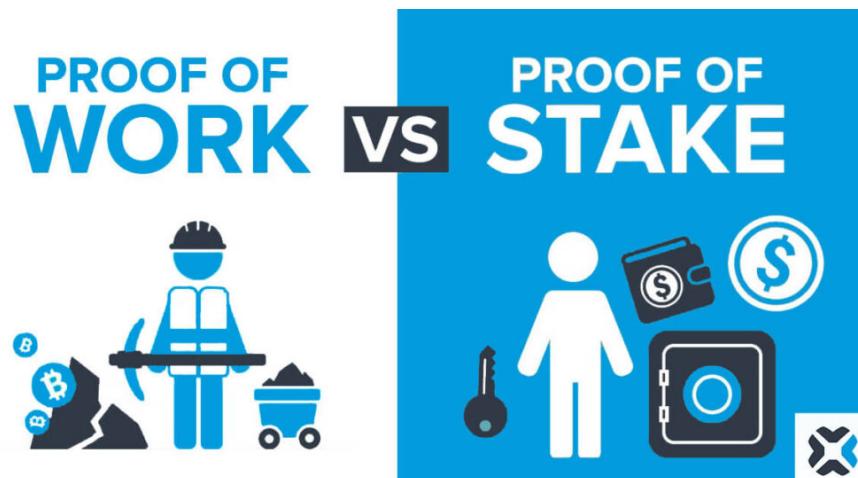


Figura 7: Representación gráfica para la dinamización de los mecanismos de consenso.

Fuente: <https://xcoins.com/es/blog/prueba-de-trabajo-vs-prueba-de-participacion-los-pros-y-contras/>

- **Ventana de reversibilidad y detección temprana:** los distintos consensos determinan cuánto tiempo una transacción puede ser revertida; en redes con finalidad rápida conviene priorizar la intercepción en perímetro frente a remedios tras la inclusión en bloque.
- **Incentivos y MEV:** la ordenación económica de transacciones genera vectores de explotación (frontrunning, sandwiching); el BAF debe incluir señales simples de MEV y políticas de enrutamiento que reduzcan la exposición a manipulaciones [30, 31].
- **Censura por validadores:** la concentración de poder en validadores facilita censura selectiva; contramedidas prácticas son la retransmisión multi-endpoint y el enmascaramiento de orígenes [15, 32].

Estas consideraciones muestran que la seguridad a nivel de consenso no se soluciona exclusivamente con controles en el nodo; el BAF debe diseñarse como una pieza coordinada con la política de validadores, estrategias de publicación y herramientas de privacidad.

Evolución tecnológica: tendencias y desarrollos futuros

El ecosistema Blockchain progresó rápidamente, muchas de las innovaciones tienen impacto directo y deben tenerse en cuenta ciertas consideraciones para desarrollar nuevos productos:

- **L2 y rollups:** los rollups cambian puntos de observación y latencias de finalización; es recomendable instrumentar tanto L1 como L2 y ajustar políticas según el tipo de rollup (optimista vs ZK) [33, 34, 35].
- **Privacidad ZK (Zero Knowledge):** las pruebas de conocimiento cero reducen la capacidad de inspección directa; ante ello se debe confiar en metadatos y telemetría en lugar de inspección de payloads [35].
- **Automatización de ataques:** la proliferación de bots y herramientas automatizadas exige reglas adaptativas y firmas de comportamiento para detectar ataques emergentes.
- **Interoperabilidad:** la heterogeneidad cross-chain requiere una arquitectura modular con adaptadores ABI/RPC para mantener políticas coherentes en entornos multi-chain.

La conjunción de estas tendencias nos exige adoptar estrategias de arquitectura evolutiva: parsers y motores de reglas modulares, telemetría rica y normalizada, y mecanismos de actualización segura de reglas (hot-reload) respaldados por pruebas de regresión [5, 36].

3

Estudio de la solución y análisis de amenazas

Este capítulo profundiza en el análisis comparativo de soluciones de seguridad existentes y expone las bases esenciales para formar una idea clara del proyecto. A diferencia del estado del arte general presentado anteriormente, aquí nos centramos específicamente en tecnologías de seguridad perimetral aplicables al contexto Blockchain, realizando un análisis crítico del gap tecnológico que motiva el desarrollo de NodeGuard. Por otra parte, realiza un análisis riguroso de las amenazas relevantes para la infraestructura que soporta nodos Ethereum y redes EVM-compatibles desde un punto de vista técnico y operativo: partimos de vectores de ataque concretos, describimos metodologías de explotación, y proponemos criterios, métricas y prioridades que guían la arquitectura defensiva. Además, se procede a cuantificar los riesgos para ayudar a una correcta especificación de requisitos (funcionales y no funcionales).

3.1. Sistemas de seguridad existentes y análisis del gap tecnológico

El panorama actual de seguridad perimetral presenta soluciones maduras para entornos web tradicionales, pero evidencia carencias significativas cuando se aplica al contexto Blockchain. Esta sección examina las tecnologías existentes, identifica sus limitaciones estructurales y establece el fundamento técnico para una solución especializada.

Firewalls tradicionales: fundamentos y limitaciones en Blockchain

Un firewall es un mecanismo de control de acceso que filtra el tráfico entre dos dominios de confianza distintos (por ejemplo, una red interna y la Internet pública) según un conjunto de políticas predefinidas. Los firewalls convencionales operan fundamentalmente en las capas 3 y 4 del modelo OSI, realizando filtrado basado en direcciones IP, puertos y protocolos de transporte. Su arquitectura se fundamenta en reglas estáticas que evalúan cabeceras de paquetes sin considerar el contenido aplicativo [37]. Esta aproximación, aunque efectiva para controlar acceso de red, presenta limitaciones estructurales al enfrentarse a las características del tráfico Blockchain.

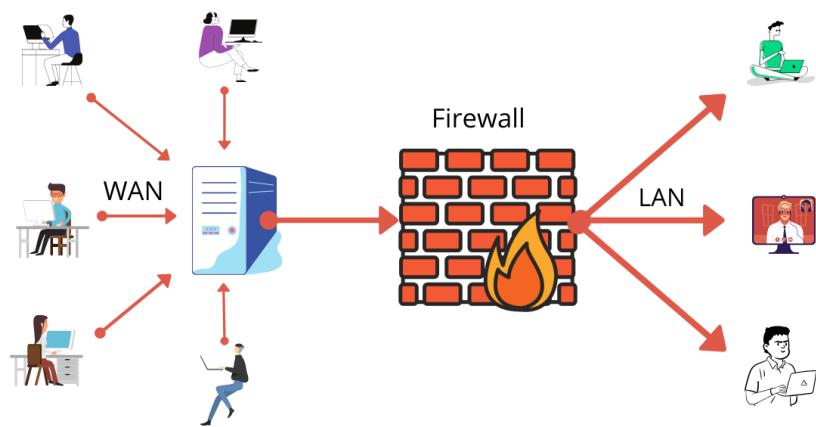


Figura 8: Abstracción gráfica de un firewall tradicional

Fuente: <https://geekflare.com/es/firewall-introduction/>

Estas transacciones encapsulan lógica de negocio compleja dentro de payloads JSON-RPC que requieren análisis semántico profundo. Un firewall tradicional puede detectar anomalías en patrones de tráfico HTTP, pero carece de la capacidad para interpretar la semántica de una transacción `eth_sendRawTransaction` o identificar patrones MEV en secuencias de llamadas `eth_call`. Además, la naturaleza descentralizada de las redes Blockchain introduce vectores de ataque inexistentes en arquitecturas cliente-servidor tradicionales. Los firewalls perimetrales asumen un modelo de confianza binario (interno/externo), mientras que los nodos Blockchain deben operar en entornos zero-trust donde cada transacción, independientemente de su origen, requiere validación exhaustiva.

La temporalidad constituye otra dimensión crítica. Los firewalls tradicionales evalúan paquetes de forma aislada, mientras que las amenazas Blockchain frecuentemente se manifiestan como patrones temporales: ataques de front-running que requieren correlación entre transacciones sucesivas, o flooding coordinado desde múltiples identidades Sybil.

Web Application Firewalls: arquitectura y adaptabilidad

A mitad de camino de nuestra transición hacia el BAF, encontramos los Web Application Firewalls (WAF), que actúan como proxies de aplicación que analizan en profundidad el tráfico HTTP mediante motores de reglas orientados a detectar vulnerabilidades típicas de la web (inyecciones, XSS, etc.). Su funcionamiento está organizado en capas: validación sintáctica de peticiones, detección por firmas/expresiones regulares, limitación de tasa a nivel de sesión y correlación básica de eventos. [38]

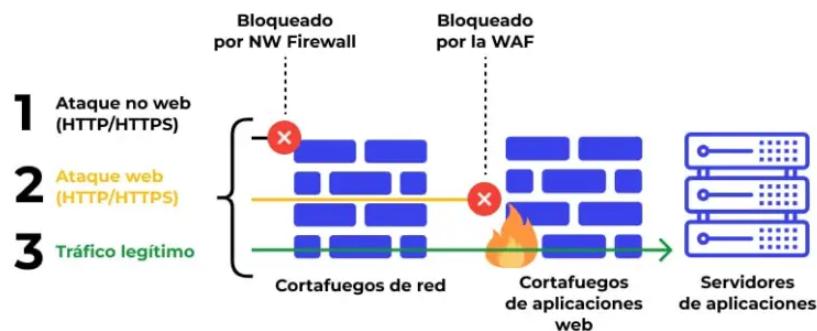


Figura 9: Comparativa WAF vs firewall tradicional

Fuente: <https://lab.wallarm.com/what/waf-firewall-de-aplicaciones-web/?lang=es>

No obstante, su adaptación directa al tráfico Blockchain presenta, de nuevo, limitaciones estructurales. El protocolo JSON-RPC y las transacciones RLP contienen semánticas específicas que exceden el ámbito de las reglas típicas de un WAF. Además, las amenazas relevantes en entornos Blockchain (replays multichain, MEV, coordinación Sybil, ataques sobre oráculos) requieren análisis temporal, correlación mempool y lógica transaccional que no están contempladas por defecto.

Finalmente, y de especial peso operativo, los WAF convencionales carecen de comprensión criptográfica: no realizan verificaciones ECDSA ni interpretan coherencia de firmas o campos EIP sin añadidos significativos.

Evolución hacia Blockchain Application Firewalls

Finalmente, llegamos al punto clave de la cuestión. ¿Cómo debe ser un BAF? Este, debe mantener estado contextual de transacciones, tracking de nonces por cuenta, análisis de reputación distribuida y correlación temporal de eventos.

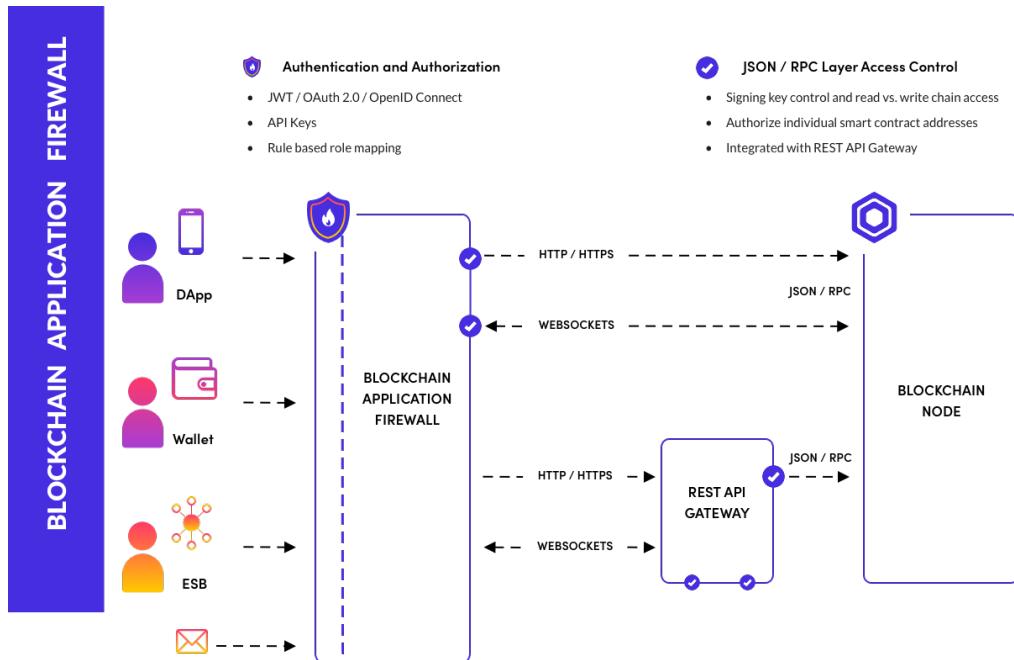


Figura 10: Representación visual de un BAF

Fuente: <https://docs.kaleido.io/kaleido-services/baf/>

La incorporación de capacidades criptográficas nativas constituye otro diferenciador fundamental. NodeGuard deberá integrar bibliotecas especializadas para validación de firmas, parsing de transacciones RLP y verificación de consistency checks que son invisibles para WAFs tradicionales. Esta integración permite interceptar vectores de ataque en el momento de la validación sintáctica, antes de que consuman recursos del nodo protegido.

Por último, la arquitectura necesaria para rate limiting eficaz en estos entornos requiere coordinación entre múltiples instancias del firewall, típicamente mediante Redis Cluster.

3.1.1. Análisis de soluciones comerciales: Kaleido BAF y competidores

El mercado de soluciones comerciales de seguridad Blockchain permanece en estado emergente, con escasas ofertas maduras específicamente diseñadas para protección perimetral de nodos. Kaleido representa uno de los pocos ejemplos de BAF comercial, proporcionando capacidades de filtrado para redes Ethereum empresariales dentro de su plataforma BaaS (Blockchain-as-a-Service) [39].

La aproximación de Kaleido se centra en entornos empresariales controlados, ofreciendo políticas predefinidas para casos de uso específicos: restricción de métodos JSON-RPC según perfiles de usuario, rate limiting básico por API key, y logging de auditoría para compliance. Su arquitectura se integra estrechamente con el ecosistema Kaleido, limitando su aplicabilidad a despliegues independientes. Las limitaciones de soluciones comerciales actuales se manifiestan en múltiples dimensiones:

1. La dependencia de plataformas propietarias restringe la flexibilidad de despliegue y la capacidad de customización profunda.
2. Las soluciones comerciales frecuentemente implementan detección de amenazas mediante listas negras estáticas, una aproximación reactiva que requiere actualizaciones constantes ante amenazas emergentes. NodeGuard incorpora capacidades de análisis heurístico y detección comportamental que pueden identificar variantes de ataques conocidos sin requerir firmas específicas.
3. La integración con toolchains de desarrollo existentes es típicamente limitada en soluciones comerciales. NodeGuard proporciona APIs REST completas, métricas Prometheus nativas y capacidades de hot-reload que facilitan la integración en pipelines DevOps estándar.

3.2. Análisis de amenazas Blockchain

Tras la introducción en el capítulo anterior sobre el funcionamiento y ciertos riesgos del ecosistema con el que estamos trabajando, es hora de detectar, clasificar, caracterizar y priorizar las amenazas más relevantes. Para cada vector de ataque se incluye: descripción técnica, metodología de ataque, actores típicos, impacto esperado, indicadores detectables a nivel RP-C/nodo y contramedidas aplicables desde NodeGuard.

3.2.1. Riesgos generales al exponer un nodo Ethereum en producción

Un nodo expuesto (API JSON-RPC pública o puertas de acceso para dApps) comporta riesgos operativos y de seguridad que conviene identificar a priori:

- **Saturación de recursos:** peticiones costosas en CPU/memoria (p. ej. *eth_call* con rutas de ejecución complejas) o tráfico masivo de transacciones afecta disponibilidad del servicio. [9]
- **Contacto con vectores contractuales:** el nodo facilita la entrada de transacciones que intentan explotar vulnerabilidades on-chain (reentrancy, permisos erróneos). Informes de sector muestran que pérdidas por exploits siguen siendo relevantes. [6, 7]
- **Información y correlación:** consultas aparentemente inocuas permiten perfilar usuarios y contratos (frecuencia, patrones de gasto), facilitando ataques dirigidos.
- **Acceso a métodos potentes de depuración/traza:** si no están protegidos, pueden filtrarse datos sensibles y consumir recursos críticos.

Estos riesgos justifican la presencia de un BAF que actúe como operador de políticas en el perímetro, limitando entrada de comandos a la capa lógica de exclusión antes de que saturen o comprometan el nodo.

3.2.2. Spam en la mempool (mempool flooding)

¿Qué es?	
El mempool es la estructura local de cada nodo que contiene las transacciones pendientes. Un ataque de spam consiste en inundar la red con grandes volúmenes de transacciones, generalmente económicas, con el fin de saturar recursos del nodo, incrementar latencias y dificultar la inclusión de transacciones legítimas.	
Actores	
Bots automatizados controlados por el atacante (uno o varios hosts).	
Capacidades	
Generar y firmar transacciones masivas; usar múltiples IPs y direcciones; ajustar <i>gasPrice</i> para manipular prioridades.	
Modelado de la amenaza	
<ol style="list-style-type: none">Preparación: generación de claves/identidades y scripts de envío.Lanzamiento: envío de transacciones a alta tasa al endpoint JSON-RPC.Saturación: crecimiento del mempool y aumento de I/O por verificación y propagación.Mantenimiento: ráfagas periódicas o continuas para mantener la congestión.Evasión: variación de patrones para mitigar detección estática.	
Factores clave	
<ul style="list-style-type: none">Costo de transacción: <i>gasPrice</i> bajos permiten enviar muchas transacciones económicas.Coste de verificación: parsing RLP y verificación ECDSA consumen CPU.Propagación P2P: la réplica de transacciones amplifica la carga.Ausencia de reputación: sin métricas por remitente, es difícil discriminar.	
Ejemplo didáctico	
Un bot genera 1000 tx/s desde 100 cuentas distintas durante 10 minutos. Resultado observado: mempool $\times 5$ y latencia de inclusión $\times 3$ en la devnet.	
Mitigaciones en NodeGuard	
<ul style="list-style-type: none">Validación temprana: rechazo inmediato de raw-txs con RLP inválido o firmas inválidas.Rate limiting: token-bucket por IP (p. ej. 10 tx/s) y por dirección (p. ej. 5 tx/s).Heurísticas de reputación: histogramas con umbrales adaptativos por remitente/IP.Cuarentena: colocar txs sospechosas en cuarentena y priorizar tráfico legítimo.	
Métricas de éxito	
<ul style="list-style-type: none">Bloqueo efectivo: reducción de transacciones maliciosas admitidas.Resiliencia: degradación para tráfico legítimo $< 5\%$.Velocidad: detección y aplicación de medidas en < 30 s.Calidad: baja tasa de falsos positivos/negativos.	

Cuadro 2: Spam en la Mempool (Mempool Flooding)

3.2.3. Ataques de repetición (Replay attacks)

¿Qué es?
El ataque de replay reutiliza raw-transactions válidas en contextos distintos (otra chain, otro momento), provocando efectos no deseados como doble gasto entre redes compatibles o ejecuciones imprevistas.
Actores
Observadores o interceptores de mempools que retransmiten transacciones.
Capacidades
Capturar raw-txs, almacenarlas y reenviarlas en la misma u otra chain; ajustar parámetros no firmados para facilitar inclusión.
Modelado de la amenaza
<ol style="list-style-type: none">1. Captura de raw-tx (monitorización de mempool o interceptación).2. Preparación (opcional): ajustes en parámetros no firmados.3. Retransmisión en otra chain o retransmisión posterior.4. Inclusión si la nonce/estado lo permiten.
Factores clave
<ul style="list-style-type: none">▪ Uso de chainId (EIP-155): ausencia o incoherencia facilita replays.▪ Visibilidad del mempool: facilita captura y reenvío.▪ Compatibilidad entre cadenas EVM: permite reutilizar raw-tx.
Ejemplo didáctico
Una raw-tx originada en testnet A se retransmite en testnet B (compatibilidad EVM) donde la cuenta tiene fondos y la transacción se ejecuta con efectos monetarios.
Mitigaciones en NodeGuard
<ul style="list-style-type: none">▪ Detección por metadata: comparar hash y r/s/v con transacciones previamente observadas; marcar duplicados.▪ Nonce checking: consultar nonce en el nodo y comparar con la tx entrante; rechazar si incongruente.▪ Chain-awareness: rechazar txs con <i>chainId</i> incompatible.▪ Registro forense: almacenar intentos de replay para auditoría.
Métricas de éxito
<ul style="list-style-type: none">▪ Eficacia de bloqueo: proporción de replays detectados y bloqueados $\geq 90\%$.▪ Latencia de verificación: tiempo medio de verificación por transacción parseada bajo.

Cuadro 3: Ataques de repetición (Replay attacks)

3.2.4. Payloads maliciosos dirigidos a contratos

<p>¿Qué es?</p> <p>Consiste en el envío deliberado de datos de entrada (calldata) manipulados con el objetivo de explotar vulnerabilidades o errores de programación en contratos inteligentes.</p>
<p>Actores</p> <p>Atacantes con conocimiento de ABI/bytecode o con capacidad de fuzzing para descubrir vulnerabilidades.</p>
<p>Capacidades</p> <p>Envío de calldata específico, priorización por gas, coordinación con bots y uso de técnicas de fuzzing y análisis estático.</p>
<p>Modelado de la amenaza</p> <ol style="list-style-type: none">1. Reconocimiento (análisis del bytecode / ABI).2. Elaboración del payload (fuzzing/manual).3. Entrega al nodo (posible prioridad por gas).4. Ejecución y explotación si existe la vulnerabilidad.
<p>Factores clave</p> <ul style="list-style-type: none">■ Disponibilidad de ABI o código verificado.■ Prioridad en inclusión (gasPrice, MEV).■ Ausencia de checks previos (simulación) en el perímetro.
<p>Ejemplo didáctico</p> <p>Contrato con <code>withdraw()</code> que transfiere antes de actualizar balances: tx maliciosa que recurre y drena fondos.</p>
<p>Mitigaciones en NodeGuard</p> <ul style="list-style-type: none">■ Patrones por selector: reglas que identifican selectores potencialmente peligrosos y aplican políticas.■ Simulación previa: ejecutar <code>eth_call</code> en un fork o sandbox para observar efectos.■ Sanity checks de parámetros: bloquear txs con parámetros fuera de rangos esperados o gas inusualmente alto.■ Cuarentena y alertado: poner en cuarentena transacciones sospechosas y notificar administradores.
<p>Métricas de éxito</p> <ul style="list-style-type: none">■ Disminución de exploits: >90 % de exploits ejecutados reducidos comparado con entorno sin BAF.■ Precisión: mantener baja tasa de falsos positivos que afecten la operación normal.

Cuadro 4: Payloads maliciosos dirigidos a contratos

3.2.5. Envío masivo desde identidades Sybil (Sybil Attack)

<p>¿Qué es?</p> <p>Creación y uso de muchas identidades (cuentas) para distribuir la carga de ataque y evadir medidas por cuenta única, generando impacto a gran escala.</p>
<p>Actores</p> <p>Atacantes capaces de crear y controlar miles de cuentas, a menudo apoyados en infraestructuras cloud y proxies.</p>
<p>Capacidades</p> <p>Generación masiva de claves, distribución de fondos entre cuentas, envío coordinado de transacciones y rotación de orígenes para evadir límites.</p>
<p>Modelado de la amenaza</p> <ol style="list-style-type: none">1. Creación masiva de cuentas.2. Envío coordinado y distribuido desde el conjunto Sybil.3. Mantenimiento y diversificación para evitar detección.
<p>Factores clave</p> <ul style="list-style-type: none">■ Coste de creación de cuentas (bajo en muchas cadenas).■ Diversificación de IPs y fuentes (uso de proxies/cloud).■ Ausencia de mecanismos off-chain de reputación o staking.
<p>Ejemplo didáctico</p> <p>10 000 cuentas generan 1 tx/s cada una → 10 000 tx/s totales, colapsando la devnet.</p>
<p>Mitigaciones en NodeGuard</p> <ul style="list-style-type: none">■ Rate limiting multidimensional: límites combinados por IP, account y fingerprint.■ Reputación y coste: priorizar cuentas con historial o gasPrice razonable; whitelists en entornos permissioned.■ Detección de comportamiento coordinado: heurísticas que identifican patrones sincronizados.■ Challenges ligeros (p. ej. pruebas de trabajo) para tasas muy altas.
<p>Métricas de éxito</p> <ul style="list-style-type: none">■ Reducción de impacto: disminución del número de transacciones aceptadas desde Sybil $\geq 90\%$.■ Tasa de falsos positivos: mantenerla baja para no penalizar usuarios legítimos.

Cuadro 5: Envío masivo desde identidades Sybil

3.2.6. Consultas abusivas y DoS

<i>¿Qué es?</i>
Uso abusivo de métodos de lectura (p. ej. <i>eth_call</i> , <i>eth_getLogs</i>) o de tracing/debug que obligan a cálculos costosos y consumen CPU/memoria del nodo sin coste económico para el atacante.
Actores
Scripts/clientes que realizan consultas intensivas, scanners o clientes mal configurados.
Capacidades
Emisión de <i>eth_call</i> con calldata complejo, peticiones de tracing o rangos de bloques muy amplios que elevan coste computacional.
Modelado de la amenaza
<ol style="list-style-type: none"> 1. Identificación de endpoints vulnerables. 2. Emisión masiva de llamadas costosas. 3. Saturación de recursos y fallo de servicio.
Factores clave
<ul style="list-style-type: none"> ▪ Exposición de endpoints de debug sin protección. ▪ Coste computacional de ejecutar <i>eth_call</i> en contratos complejos. ▪ Solicitudes de logs con rangos amplios que impelen escaneos históricos.
Ejemplo didáctico
1000 llamadas a <i>eth_call</i> con loops costosos en el contrato → picos de CPU y timeouts en llamadas legítimas.
Mitigaciones en NodeGuard
<ul style="list-style-type: none"> ▪ Allow/deny lists: bloqueo de métodos peligrosos (p. ej. <i>debug_*</i>) en entornos públicos e implementación mediante listas de firmas de funciones. ▪ Rate limiting por método: restricciones diferenciadas para operaciones costosas (p. ej. <i>eth_getLogs</i>) y límites basados en la complejidad estimada del método. ▪ Sanity checks: rechazo de peticiones con rangos de bloques excesivos y validación estricta de parámetros. ▪ Caching y simulación ligera: cacheado de respuestas a consultas frecuentes e invariantes, y simulación/ejecución en sandbox para detectar operaciones riesgosas antes de su reenvío. ▪ Protección de endpoints debug: autenticación obligatoria (JWT o API keys) y, cuando proceda, deshabilitación completa en entornos de producción.
Métricas de éxito
<ul style="list-style-type: none"> ▪ Reducción de consumo: decremento del consumo de CPU por consultas maliciosas $\geq 70\%$. ▪ Disponibilidad: latencia de respuestas legítimas degradada $<10\%$ durante el ataque.

Cuadro 6: Consultas abusivas y DoS

3.2.7. MEV y front-running

<p style="text-align: center;">¿Qué es?</p> <p>El término <i>Maximal Extractable Value</i> (MEV) agrupa el valor que pueden extraer actores que controlan la inclusión o el orden de las transacciones. El front-running es una manifestación común donde se inserta una tx prioritaria para beneficiarse de otra transacción en mempool.</p>
<p style="text-align: center;">Actores</p> <p>Validadores/constructores de bloques, relays privados, bots de trading con visibilidad del mempool.</p>
<p style="text-align: center;">Capacidades</p> <p>Ordenar o priorizar transacciones, enviar transacciones con gas elevado, colaborar con builders para inclusión preferente.</p>
<p style="text-align: center;">Modelado de la amenaza</p> <ol style="list-style-type: none">1. Observación de mempool y detección de oportunidad.2. Construcción de transacciones oportunistas (antes/después).3. Inserción en el flujo de inclusión mediante gas o acuerdos con builders.
<p style="text-align: center;">Factores clave</p> <ul style="list-style-type: none">▪ Visibilidad del mempool y facilidad para detectar transacciones con impacto.▪ Latencia de la red y tiempo entre observación e inclusión.▪ Concentración de poder en builders o validadores.
<p style="text-align: center;">Ejemplo didáctico</p> <p>Un usuario envía una orden grande de compra; un bot detecta la intención y envía dos transacciones (una para comprar antes y otra para vender después), capturando la diferencia y dejando al usuario con mayor coste por deslizamiento.</p>
<p style="text-align: center;">Mitigaciones en NodeGuard</p> <ul style="list-style-type: none">▪ Detección de patrones MEV: correlación temporal de transacciones y firmas de comportamiento.▪ Envío privado: opción de envío privado a relays de confianza o a builders que ofrezcan inclusión sin exponer la tx al mempool público.▪ Atenuación de predictibilidad: reordenado interno controlado y retrasos aleatorizados configurables.
<p style="text-align: center;">Métricas de éxito</p> <ul style="list-style-type: none">▪ Reducción del MEV observado: disminución relativa del número de eventos de front-running detectados para transacciones protegidas.▪ Tiempo de mitigación: desde detección hasta aplicación de mitigación <1 minuto.▪ Precisión operativa: tasa de falsos positivos aceptable.

Cuadro 7: MEV y front-running

3.2.8. Impacto: cuantificación de riesgos

Usaremos una tabla de evaluación con criterios cuantitativos (probabilidad, impacto, detectabilidad) y cualitativos (esfuerzo mitigación, prioridad).

Amenaza	Prob.	Impacto	Detect.	Esf. mit.	Prioridad	Observaciones
	(1-5)	(1-5)	(1-5)	(B/M/A)	(B/M/A)	
Mempool flooding	5	4	4	M	A	Requiere heurísticas de comportamiento y límites adaptativos por múltiples dimensiones (IP, cuenta, fingerprint).
Payloads maliciosos	4	5	3	A	A	Necesita análisis estático de bytecode y simulación previa para detectar efectos maliciosos antes de la ejecución.
MEV / Front-running	/	5	4	3	A	Mitigación compleja que combina envío privado, detección de patrones y colaboración con relays.
Abuso <i>eth_call</i>	4	3	4	M	M	Mitigación mediante límites de gas simulado, caching estratégico y cuotas por endpoint.
Sybil attacks	3	3	2	M	M	Requiere análisis de grafos de relaciones y detección de coordinación entre cuentas aparentemente independientes.
Replay attacks	2	3	5	B	B	Fácil detección mediante verificación de chainId y nonce, pero requiere monitoreo continuo.

Cuadro 8: Tabla de evaluación de amenazas. Probabilidad (Prob.), impacto, detectabilidad (Detect.), esfuerzo de mitigación (Esf. mit.: B=Bajo, M=Medio, A=Alto) y prioridad.

3.2.9. Priorización: matriz de criticidad y frecuencia

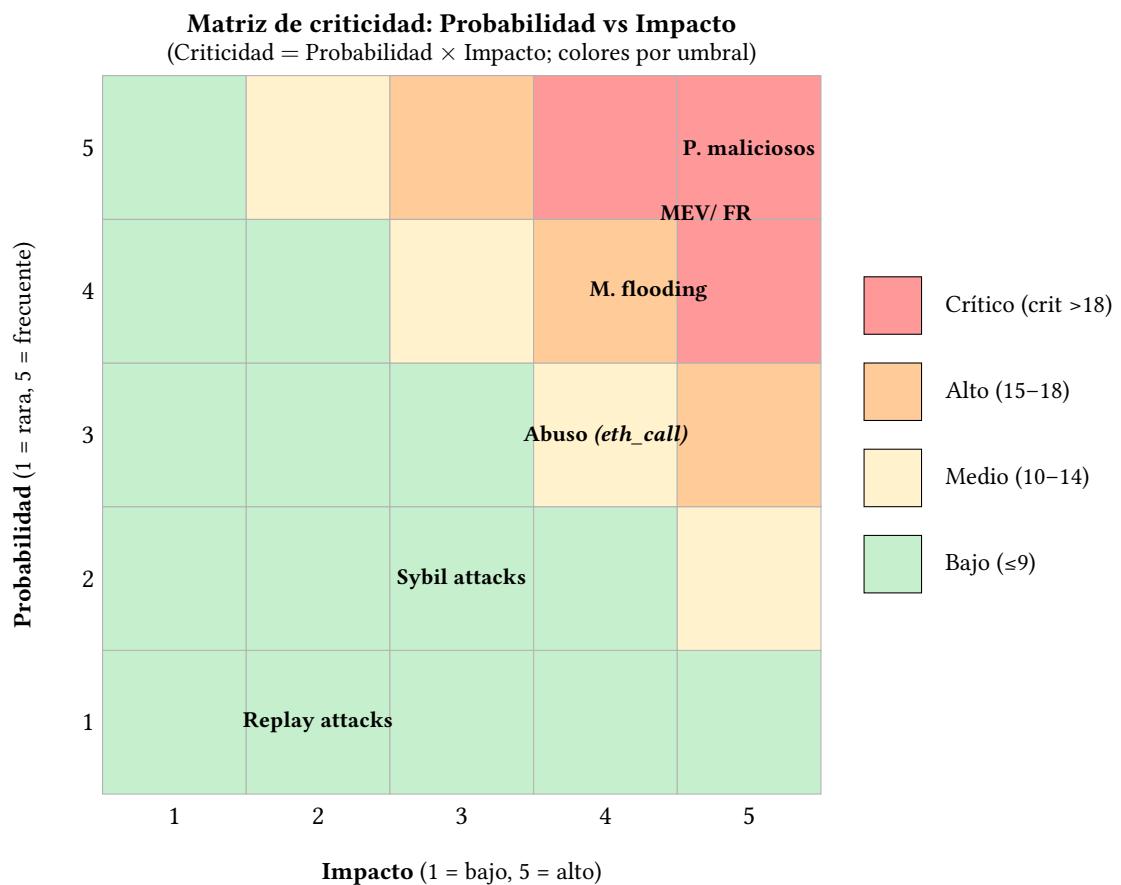


Figura 11: Matriz de criticidad para amenazas Blockchain.

4

Especificación de requisitos y diseño de la arquitectura

Este capítulo aborda la especificación de requisitos (funcionales y no funcionales) para el diseño e implementación de NodeGuard, el Blockchain Application Firewall propuesto. Posteriormente, se detalla la arquitectura propuesta, fundamentada en principios de ingeniería de software consolidados y adaptada a las particularidades del ecosistema Ethereum.

4.1. Especificación funcional

A partir del análisis de amenazas Blockchain identificadas, establecemos las capacidades funcionales requeridas para un sistema de protección multicapa que opere en tiempo real sobre transacciones Ethereum y comunicaciones JSON-RPC. Encontraremos los siguientes Requisitos Funcionales (FR):

FR-1. Proxy JSON-RPC multicapa: Interceptación y filtrado de peticiones HTTP/WebSocket con validación sintáctica, análisis heurístico y evaluación ML para métodos Ethereum estándar y propietarios. Soporte completo para especificaciones EIP-2718 (tipos de transacción), EIP-1559 (gas dinámico) y EIP-2930 (access lists).

FR-2. Motor de políticas adaptativo: Evaluación en cascada con reglas estáticas, heurísticas y detección por ML. Sistema de umbrales adaptativos con aprendizaje continuo basado en tasas de falsos positivos/negativos. Configuración hot-reload sin interrumpir el servicio.

FR-3. Rate limiting algorítmico: Implementación de sliding window, token bucket y fixed window con selección automática según patrones de tráfico. Límites por IP, wallet address, método JSON-RPC y combinaciones multidimensionales con decaimiento temporal.

FR-4. Análisis transaccional ligero: Extracción de metadatos críticos (chainId, gasPrice, gasLimit, access lists) sin ejecución completa. Detección de anomalías en pricing, patrones de reentrada y vectores MEV mediante correlación temporal y análisis de secuencias.

FR-5. Motor de reglas modular y hot-reload: Sistema de reglas JSON con validación automática, hot-reload mediante Redis keyspace notifications o polling, y configuración en tiempo real sin interrumpir el servicio.

FR-6. Detección de patrones MEV y Sybil: Correlación temporal de transacciones, análisis de secuencias para front-running, detección de ataques Sybil distribuidos y patrones de suplantación mediante análisis comportamental.

FR-7. Fingerprinting avanzado: Identificación de patrones comportamentales mediante análisis de User-Agent, timing attacks, secuencias de métodos y características de payload. Detección de automatización y correlación de identidades distribuidas.

FR-8. Observabilidad integral: Métricas Prometheus nativas, logs estructurados JSON con contexto de seguridad y dashboard web con visualizaciones analíticas. Sistema de alerting basado en thresholds configurables.

FR-9. Event bus reactivo: Sistema de eventos asíncronos con SSE para dashboard y soporte para filtrado de eventos y suscripciones granulares.

FR-10. Enforcement configurable: Modos de operación flexible (block/monitor/dry-run) con aplicación granular por tipo de regla. Circuit breaker para upstream failures y fail-open/fail-closed configurable.

FR-11. Gestión de configuración distribuida: Almacenamiento Redis con backup/restore, versionado de reglas y API administrativa con autenticación JWT. Rollback automático en caso de configuraciones erróneas.

4.1.1. Casos de uso: escenarios operacionales detallados

Caso A: Protección de nodo público contra ataques MEV

Un DEX despliega NodeGuard para proteger su nodo Ethereum público. El sistema detecta patrones de front-running mediante análisis de gas price extremos ($>10x$ promedio) y correlación temporal de transacciones idénticas. NodeGuard bloquea sandwich attacks al identificar transacciones coordinadas con nonces secuenciales y datos similares. Rate limiting sliding window (100 req/min por IP) previene flooding mientras permite tráfico legítimo.

Detección MEV (Valor Máximo Extraíble)

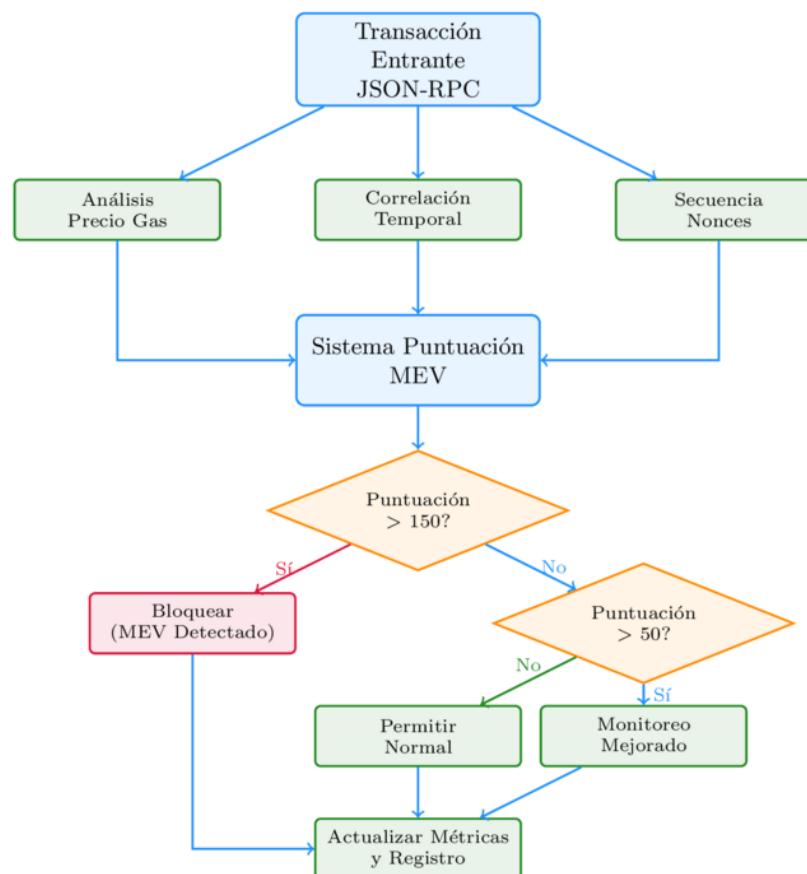


Figura 12: Posible diagrama de flujo en el Caso A

Caso B: Defensa anti-Sybil en protocolo de reputación

Una plataforma utiliza NodeGuard para prevenir ataques Sybil distribuidos. El sistema identifica clusters de wallets con patrones de comportamiento idénticos: transacciones simultáneas, valores uniformes y secuencias de direcciones secuenciales. Detección de burst de >15 transacciones desde IPs relacionadas activa bloqueo temporal. Sistema de reputación penaliza wallets con puntuación <40 basada en comportamiento histórico y correlaciones de identidad.

Defensa Anti-Sybil Distribuida

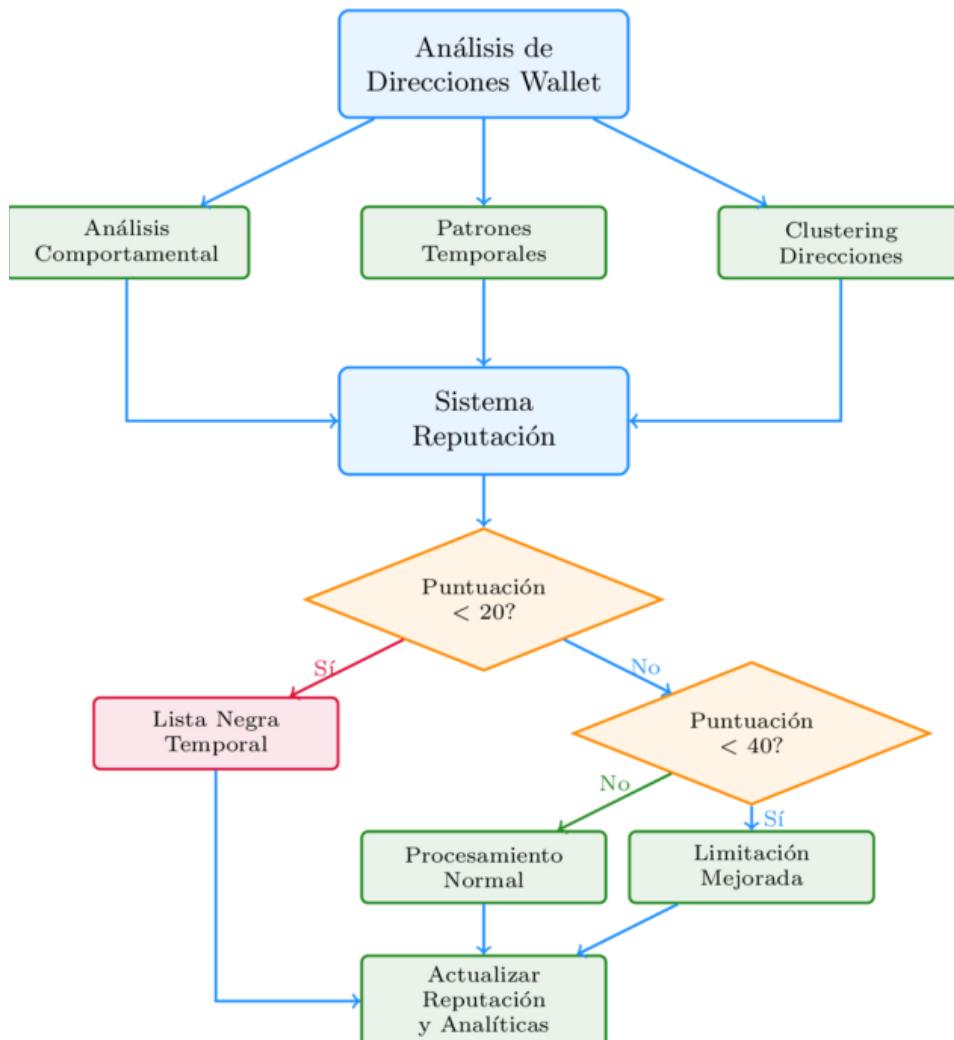


Figura 13: Posible diagrama de flujo en el Caso B

Caso C: Protección anti-DoS para infraestructura crítica

Un nodo validador despliega NodeGuard con protección multicapa contra ataques DoS. Circuit breaker se activa tras >500ms latency upstream, token bucket absorbe picos de tráfico hasta 1000 req/burst y rate limiter bloquea si hay >200 req/min.

Circuit Breaker Protección DoS

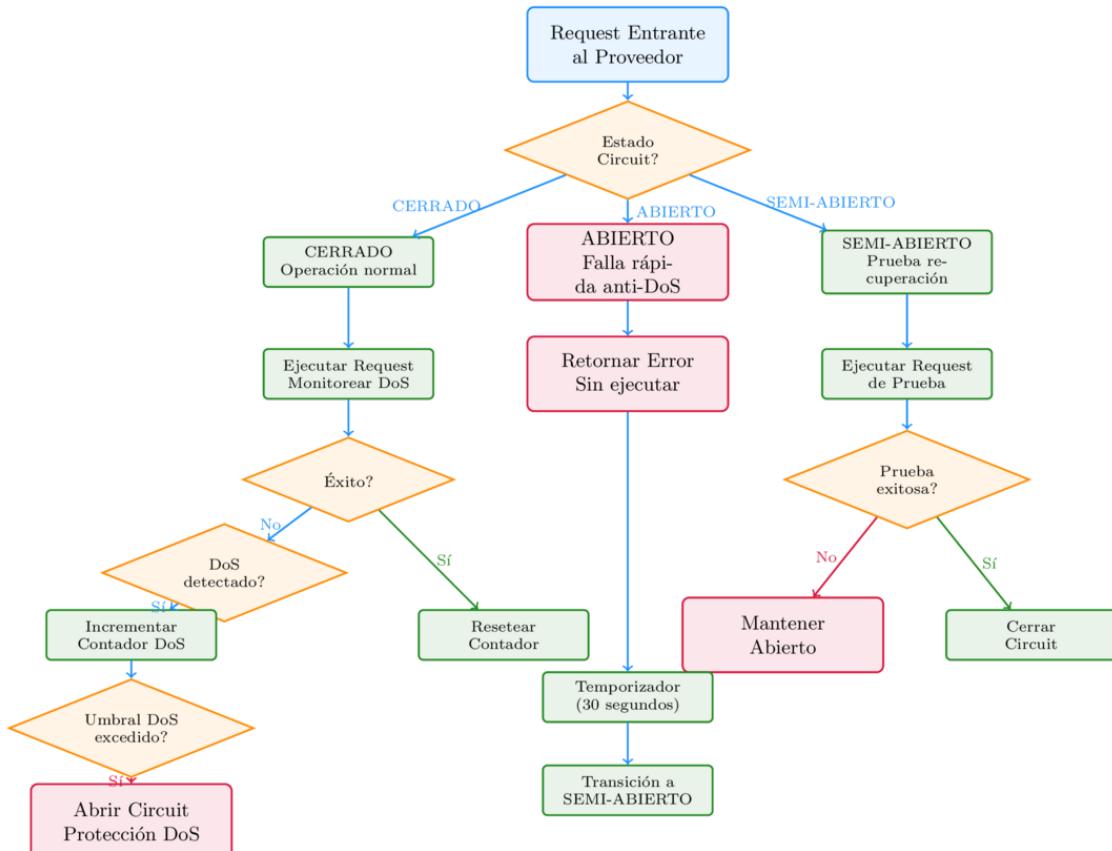


Figura 14: Posible diagrama de flujo en el Caso C

Caso D: Validación EIP-155 en red multi-chain

Una bridge cross-chain utiliza NodeGuard para validar transacciones EIP-155. Sistema rechaza replay attacks detectando chain IDs inválidos, nonce reuse y manipulación de parámetros v/r/s. Validación de EIP-2718/EIP-1559 asegura compatibilidad con transacciones modernas. El análisis batch detecta correlaciones sospechosas en >50 peticiones simultáneas desde misma IP.

Validación EIP-155 Firma de Transacción

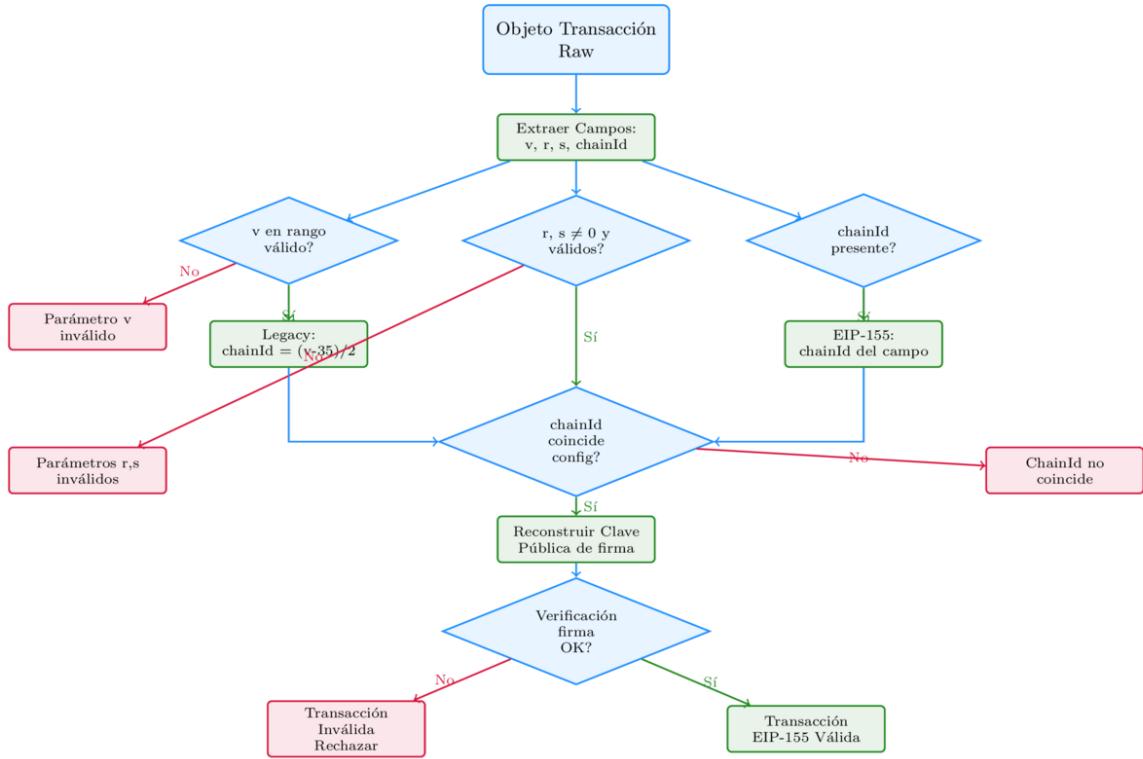


Figura 15: Posible diagrama de flujo en el Caso D

Caso E: Protección de contratos inteligentes en producción

Un protocolo DeFi utiliza NodeGuard para proteger deployment de contratos. El sistema analiza bytecode malicioso identificando patrones de reentrancy y gas bomb.

4.1.2. Interfaces requeridas

- **JSON-RPC Gateway:** Endpoint '/rpc' con soporte HTTP, validación schema y forwarding transparente. Compatibilidad completa con especificaciones Ethereum y extensiones propietarias.

- **API administrativa REST:** Rutas '/admin/*' para gestión de reglas, configuración enforcement, consulta de métricas y operaciones de mantenimiento. Autenticación JWT con permisos granulares.
- **Dashboard web:** Interfaz '/dashboard' con visualizaciones real-time mediante Server-Sent Events. Monitoreo de métricas y eventos en tiempo real.
- **Observability endpoints:** Métricas Prometheus '/metrics', health checks '/healthz' y event streaming '/events' con filtrado granular.

Flujo de trabajo típicos

1. **Transacción entrante:** Validación JSON-RPC, extracción de metadatos y construcción del contexto de evaluación con información de cliente, timing y payload.
2. **Evaluación multicapa:** Chequeo de reglas estáticas (ms response), análisis heurístico con estado (10-50ms) para casos complejos.
3. **Rate limiting:** Verificación de límites algorítmicos con persistencia Redis y actualización de contadores atómicos.
4. **Enforcement:** Aplicación de decisión según modo configurado, logging de incidentes y actualización de métricas.
5. **Response forwarding:** Proxy transparente a upstream con enriquecimiento de headers y métricas de latencia.
6. **Analytics update:** Persistencia de eventos, actualización de reputación y triggers de alerta según umbrales configurados.

Flujo Principal NodeGuard BAF

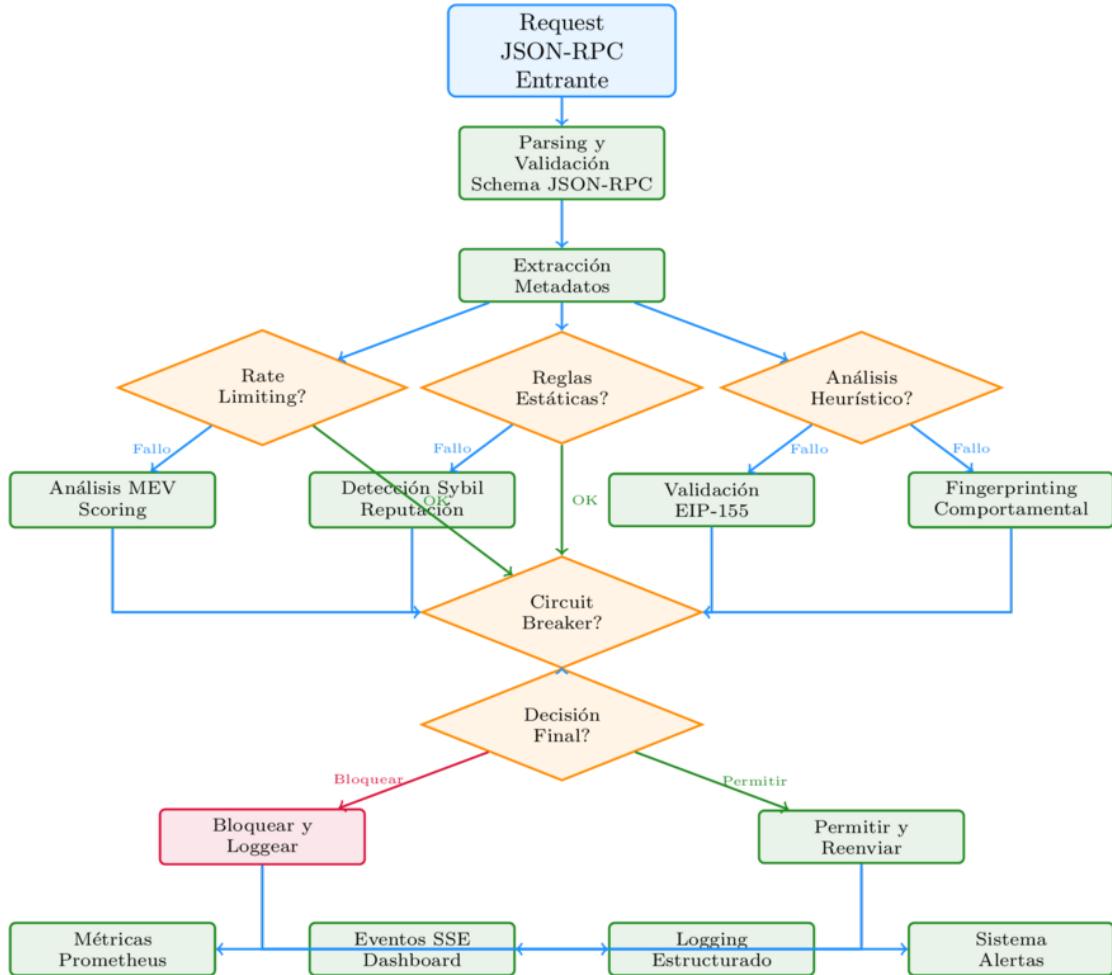


Figura 16: Posible diagrama sintetizado del flujo típico de trabajo

4.2. Especificación no funcional

Los requisitos no funcionales (NFR) definen las cualidades del sistema. A continuación se enumeran las más relevantes y sus objetivos cuantitativos.

Rendimiento

- **P50 latency:** <15ms para evaluación de reglas estáticas, <35ms para análisis heurístico completo incluyendo consultas Redis.
- **P95 latency:** <100ms end-to-end incluyendo forwarding upstream, con circuit breaker activándose a >500ms para prevenir cascading failures.
- **Rendimiento (throughput):** 10,000+ peticiones/segundo por instancia con degradación aceptable bajo carga sostenida.
- **Escalabilidad:** Arquitectura con estado compartido en Redis Cluster, permitiendo escalado horizontal sin límite teórico y balanceo de carga transparente.

Seguridad

- **Robustez:** Valores predeterminados a prueba de fallos con fallback a reglas conservadoras ante fallos de dependencias. Validación estricta de inputs y sanitización de outputs para prevenir ataques de inyección.
- **Disponibilidad:** Circuit breakers para servicios de upstream, health checks proactivos y recuperación automática.
- **Integridad:** Inmutabilidad de logs críticos, registro de auditoría completo con decisiones y mecanismos de antimanipulación para configuraciones.
- **Autenticación:** JWT con refresh tokens, rate limiting en endpoints administrativos y session management con revocación granular.

Usabilidad

- **Configuración:** Configuración mediante variables de entorno con valores por defecto sensatos. Inicialización automática de servicios y componentes con validación de dependencias.

- **Monitoreo:** Dashboard básico para visualización de métricas en tiempo real, logs estructurados JSON con información contextual y métricas exportables en formato Prometheus.
- **Operación:** APIs REST con autenticación JWT para administración, endpoints programáticos para gestión de configuración y logs estructurados para troubleshooting.
- **Debugging:** Logs estructurados con correlation IDs, seguimiento de solicitudes y testing de reglas mediante modos de enforcement configurables.

Mantenibilidad

- **Modularidad:** Arquitectura de componentes con interfaces bien definidas y acoplamiento mínimo entre módulos de validación, rate limiting y políticas.
- **Extensibilidad:** APIs y compatibilidad con versiones anteriores en interfaces clave.
- **Documentación:** Cobertura completa de APIs, guías de despliegue, guías de resolución de problemas y registros de decisiones arquitectónicas.
- **Testing:** Suites automáticas tests unitarios en componentes clave y testing de escenarios con amenazas reales.
- **Deployment:** Despliegue local y configuración vía variables de entorno para diferentes configuraciones. Fácilmente adaptable a Docker.

4.3. Restricciones y limitaciones

Las limitaciones identificadas condicionan el uso de NodeGuard como único elemento de seguridad o su uso profesional por el momento. Se trata de un proyecto extenso, pero propenso a fallos en situaciones externas al ámbito de este TFG.

Limitaciones técnicas

- **LT-1 (Ámbito de protocolos).** El perímetro está optimizado para EVM/JSON-RPC (Ethereum, BNB Chain, Polygon, Avalanche C-Chain). La compatibilidad con redes no-EVM (p. ej., Solana) requiere adaptadores semánticos específicos y no está cubierta por defecto.
- **LT-2 (Verificación pesada).** La verificación criptográfica completa y lecturas profundas de estado pueden ser costosas en el perímetro; se priorizan comprobaciones ligeras y se delega la validación exhaustiva al *upstream* cuando sea necesario.
- **LT-3 (Observabilidad parcial).** En entornos con cifrado extremo a extremo o *privacy layers*, la inspección de *payload* puede verse limitada; se recurre a metadatos, telemetría y comportamiento agregado.
- **LT-4 (Dependencias).** Para cuotas distribuidas y *configs* centralizadas se recomienda Redis; en su ausencia, algunos mecanismos operan en modo local con garantías reducidas en entornos multirréplica.

Restricciones operacionales

- **RO-1 (Recursos mínimos).** Por réplica: 2 vCPU, 2GB RAM para cargas medias; almacenamiento de logs según retención; conectividad estable a *upstreams*.
- **RO-2 (Topologías).** Despliegue recomendado detrás de *reverse proxy* TLS y con *autoscaling*; multi-AZ o *multi-node* para alta disponibilidad.
- **RO-3 (Gobernanza de reglas).** Proceso formal de cambio: revisión por pares, *shadow mode* previo y ventana de activación; *rollback* automático si se superan umbrales de error.

Consideraciones éticas

- **CE-1 (Minimización de datos).** Registrar únicamente lo estrictamente necesario para auditoría y respuesta a incidentes; anonimización/ofuscación cuando proceda.
- **CE-2 (No discriminación técnica).** Las políticas deben basarse en señales técnicas observables (abuso, malformación, riesgo) evitando sesgos por origen geográfico o supuestos no verificables.
- **CE-3 (Transparencia operativa).** Documentar qué categorías de llamadas se bloquean/monitorizan, los motivos y los mecanismos de apelación/corrección en entornos multi equipo.

4.4. Arquitectura general: principios de diseño y patrones

El diseño arquitectónico de NodeGuard se fundamenta en principios consolidados de ingeniería de software, adaptados específicamente a los requisitos de seguridad perimetral Blockchain.

4.4.1. Patrones arquitectónicos implementados

La arquitectura se organiza según los siguientes patrones de diseño, que priorizan la modularidad, la escalabilidad y la integración en busca de un entorno productivo:

Factory Pattern

El componente *createFirewallProvider* en *src/core/factory.ts* centraliza la creación e inicialización de todos los componentes del sistema, gestionando dependencias complejas y configuración. Este factory gestiona la creación coordinada de *FirewallProvider*, *PolicyEngine*, *RpcClient*, y servicios auxiliares como *ReputationService* y *PerformanceMonitor*.

Provider Pattern

FirewallProvider extiende *BaseProvider* implementando la lógica principal de interceptación y filtrado. Esta jerarquía permite extensibilidad mediante herencia controlada, donde *BaseProvider* define la interfaz común y *FirewallProvider* especializa la implementación para seguridad Blockchain.

Strategy Pattern

TransactionValidator y *PolicyEngine* implementan estrategias intercambiables para validación y evaluación de políticas. El sistema de reglas utiliza evaluadores especializados (*evaluateStaticRules*, *evaluateHeuristicRules*) que pueden componerse dinámicamente según la configuración.

Observer Pattern

EventBus gestiona comunicación asíncrona entre componentes mediante eventos tipados. Los componentes pueden suscribirse a eventos específicos (bloqueos, alertas, cambios de reputación) sin acoplamiento directo, facilitando extensibilidad y testing.

Storage Abstraction

StorageFactory proporciona abstracción sobre Redis/memoria con fallback automático. Esta abstracción permite operación en entornos con o sin Redis, manteniendo funcionalidad completa con degradación graceful de capacidades distribuidas.

4.4.2. Arquitectura modular por componentes

La organización modular de NodeGuard se refleja en la estructura de directorios y la separación de responsabilidades:

Tamaño y estructura ligeramente reducidos por temas de espacio y formato

```
src//  
  └── index.ts  
  └── api/  
    └── middleware/  
    └── routes/  
    └── server.ts  
  └── client/  
    └── baf-client.ts  
  └── core/  
    └── base-provider.ts  
    └── factory.ts  
    └── firewall-provider.ts  
    └── interfaces.ts  
    └── policy-engine.ts  
    └── rpc-client.ts  
  └── events/  
    └── event-bus.ts  
  └── logging/  
    └── logger.ts  
  └── metrics/  
    └── performance-monitor.ts  
    └── prometheus.ts  
  └── rateLimiting/  
    └── algorithms/  
    └── indexRL.ts  
    └── lua-scripts/  
    └── rate-limiter.ts  
    └── types.ts  
  └── redis/  
    └── redis-connection.ts  
    └── redis-manager.ts  
    └── redis-types.ts  
  └── rules/  
    └── config.ts  
    └── heuristic-rules.ts  
    └── static-rules.ts  
    └── types.ts  
  └── security/  
    └── fingerprint/  
    └── reputation/  
  └── storage/  
    └── config-store.ts  
    └── indexSto.ts  
    └── interfaces.ts  
    └── memory-store.ts  
    └── redis-store.ts  
  └── utils/  
    └── circuit-breaker.ts  
    └── report-generator.ts  
    └── shutdown-utils.ts  
    └── startup-utils.ts  
    └── transaction-utils.ts  
  └── validation/  
    └── indexVal.ts  
    └── json-rpc-validator.ts  
    └── rule-validator.ts  
    └── schemas/  
    └── transaction-validator.ts  
    └── types.ts  
    └── unified-validator.ts  
    └── utils.ts
```

- **Core Components** (*src/core/*): Factory, providers, clientes RPC y motores de políticas principales.
- **Validation Layer** (*src/validation/*): Validadores especializados para transacciones, reglas y esquemas JSON.
- **Rules Engine** (*src/rules/*): Motor de reglas estáticas y heurísticas con tipos TypeScript completos.
- **Rate Limiting** (*src/rateLimiting/*): Algoritmos especializados (Token Bucket, Sliding Window) con stores distribuidos.
- **Security Services** (*src/security/*): Fingerprinting, reputación y análisis de patrones.
- **Storage Layer** (*src/storage/*): Abstracción de persistencia con implementations Redis y memoria.

Esta organización facilita desarrollo independiente de módulos, testing aislado y extensibilidad sin efectos colaterales.

4.4.3. Principios de diseño aplicados

Single Responsibility Principle

Cada módulo tiene una responsabilidad única: *TransactionValidator* maneja únicamente validación sintáctica y criptográfica, *PolicyEngine* se encarga de evaluación de reglas, *RateLimiter* gestiona únicamente limitación de tasa. Esta separación facilita testing unitario y mantenimiento.

Dependency Injection

El factory inyecta dependencias de forma explícita, facilitando testing con mocks y composición flexible de componentes. Los componentes declaran sus dependencias mediante interfaces TypeScript, permitiendo verificación estática de contratos.

Interface Segregation

Las interfaces están diseñadas de forma granular: *BaseProvider* define contratos mínimos, interfaces específicas (*RateLimitStore*, *ConfigStore*) evitan dependencias innecesarias entre componentes no relacionados.

4.5. Modelo arquitectónico: patrón proxy con análisis en tiempo real

NodeGuard implementa un patrón de proxy transparente que intercepta comunicaciones JSON-RPC entre clientes y nodos Ethereum, aplicando análisis de seguridad en tiempo real sin interrumpir el flujo de comunicación. Esta arquitectura equilibra los requisitos de seguridad con la preservación de compatibilidad y rendimiento.

4.5.1. Arquitectura de proxy transparente

El diseño de proxy se basa en el patrón de interceptación transparente, donde NodeGuard se posiciona entre clientes y el nodo Ethereum objetivo sin requerir modificaciones en ninguno de los extremos [40]. Esta transparencia es crítica para facilitar adopción en infraestructuras existentes sin disruption operacional.

La implementación del proxy maneja múltiples protocolos de comunicación: HTTP/HTTPS para peticiones síncronas, WebSocket para conexiones persistentes. Esta versatilidad garantiza compatibilidad con el ecosistema diverso de clientes Ethereum, desde billeteras web hasta aplicaciones empresariales.

El flujo de procesamiento implementa un pipeline multi-etapa implementado en *FirewallProvider*: recepción de peticiones, parsing y validación sintáctica mediante *TransactionValidator*, extracción de metadatos, evaluación de políticas mediante *PolicyEngine*, aplicación de rate limiting, decisión de forwarding, envío a nodo objetivo mediante *RpcClient*, procesamiento de respuesta, y generación de respuesta con metadatos de seguridad.

Gestión de estado y persistencia distribuida

El mantenimiento de contexto entre peticiones constituye una innovación crítica respecto a proxies HTTP tradicionales. NodeGuard mantiene estado distribuido mediante la abstracción *StorageFactory* que proporciona tanto implementación Redis (*RedisRateLimiterStore*) como memoria (*InMemoryRateLimiterStore*) con fallback automático.

El estado persistido incluye: contadores de rate limiting por múltiples dimensiones (IP, address, method), métricas de reputación gestionadas por *ReputationService*, cache de validaciones para optimización de rendimiento, y eventos de seguridad procesados por *EventBus*.

La persistencia de contexto permite implementar lógica de seguridad sofisticada: detección de patrones de comportamiento anómalos mediante *HeuristicRules*, correlación de actividad entre cuentas para identificar clusters Sybil, y aplicación de políticas adaptativas que evolucionan basándose en el comportamiento observado.

4.6. Stack tecnológico seleccionado

La elección del stack tecnológico equilibra requisitos de rendimiento, facilidad de desarrollo, compatibilidad con el ecosistema Blockchain y capacidades de integración. Esta sección justifica cada decisión tecnológica principal y analiza alternativas consideradas.

4.6.1. Node.js como runtime principal

Node.js proporciona el runtime de ejecución principal para NodeGuard, una decisión motivada por múltiples factores técnicos y ecosistémicos [41]. Su modelo de concurrencia basado en event-loop es especialmente adecuado para aplicaciones intensivas de I/O como proxies de red, donde la capacidad de manejar miles de conexiones concurrentes sin overhead de threading es crítica.

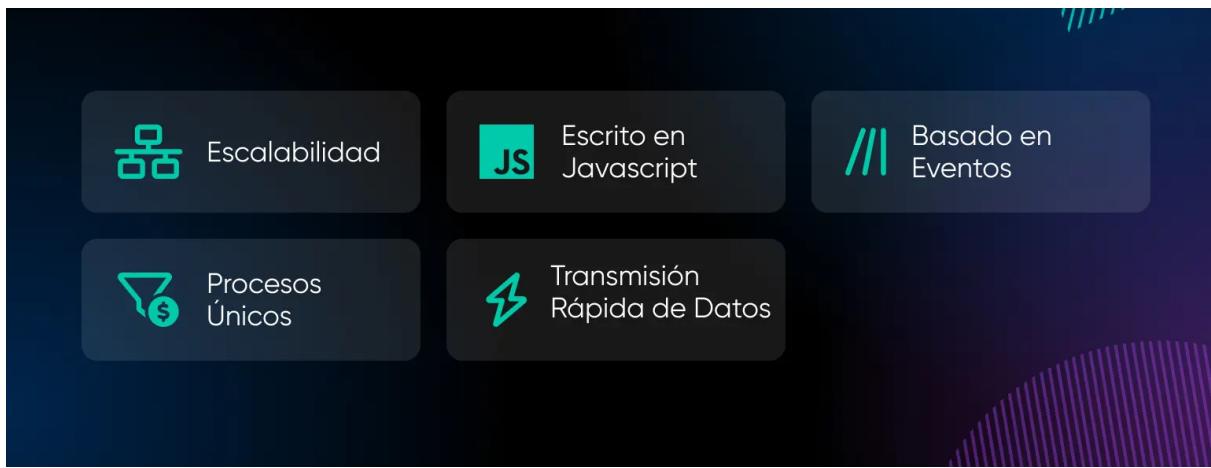


Figura 17: Funciones de Node.js

Fuente: <https://www.dreamhost.com/blog/es/que-es-node-js-introduccion-completa/>

La compatibilidad nativa con JSON y el ecosistema npm facilitan la integración con bibliotecas Blockchain especializadas. La mayoría de herramientas de desarrollo Ethereum están disponibles como paquetes npm, simplificando dependencias y acelerando el ciclo de desarrollo.

El rendimiento de Node.js para operaciones de red es competitivo con implementaciones en lenguajes compilados para casos de uso típicos de NodeGuard. Benchmarks internos demuestran latencias P95 por debajo de 50ms para pipelines de análisis completos, cumpliendo requisitos de rendimiento sin optimizaciones específicas.

Alternativas consideradas

Go fue evaluado como alternativa debido a su rendimiento superior para aplicaciones de red y su creciente adopción en infraestructura Blockchain. Sin embargo, el ecosistema de bibliotecas Ethereum en Go es menos maduro, requiriendo desarrollo adicional para capacidades que están disponibles como paquetes npm estándar.

Rust ofrece rendimiento superior y garantías de memory safety atractivas para software de seguridad. No obstante, la curva de aprendizaje y el ecosistema menos desarrollado para aplicaciones Blockchain lo hacen menos adecuado para el contexto académico de este TFG.

4.6.2. TypeScript para desarrollo type-safe

TypeScript añade tipado estático a JavaScript, proporcionando beneficios críticos para el desarrollo de software de seguridad [42]. El sistema de tipos permite detectar errores en tiempo de compilación que podrían manifestarse como vulnerabilidades en runtime, una consideración especialmente importante para software que maneja transacciones financieras.

La integración con IDEs modernos proporciona autocompletado y navegación de código que aceleran el desarrollo y reducen errores. El soporte nativo para interfaces y genéricos facilita la implementación de patrones arquitectónicos complejos manteniendo type safety.

La compatibilidad con el ecosistema JavaScript permite utilizar bibliotecas npm existentes mientras se añaden garantías de tipo donde es crítico. Esta flexibilidad equilibra productividad de desarrollo con robustez de implementación.

4.6.3. Redis para estado distribuido

Redis proporciona la infraestructura de estado distribuido necesaria para operaciones de rate limiting, cache de validaciones y coordinación entre instancias de NodeGuard [43]. Su modelo de datos en memoria garantiza latencias submilisegundo para operaciones críticas en el path de evaluación.

Las estructuras de datos especializadas de Redis son especialmente adecuadas para casos de uso de NodeGuard: sorted sets para implementar sliding window rate limiting, hash maps para cache de validaciones criptográficas, pub/sub para coordinación de eventos entre instancias.

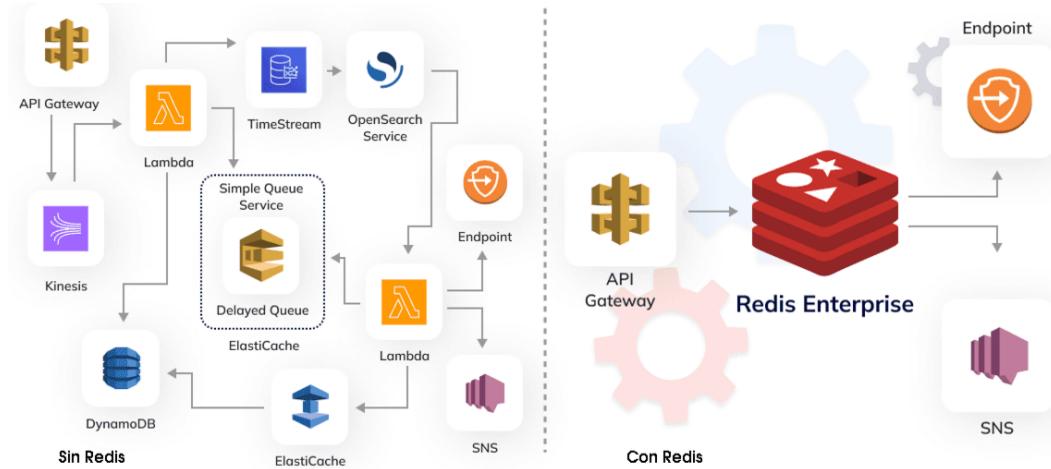


Figura 18: Funcionamiento de Redis

Fuente: <https://intellisoft.io/inside-redis-navigating-the-high-speed-highway-of-data-structures/>

La persistencia configurable permite equilibrar durabilidad con rendimiento según requisitos específicos. Para rate limiting temporal, la configuración append-only es suficiente, mientras que cache de validaciones puede operarse completamente en memoria.

4.6.4. Express.js para servidor HTTP

Express.js proporciona el framework web que maneja comunicaciones HTTP/HTTPS con clientes [41]. Su arquitectura de middleware permite implementar el pipeline de procesamiento de forma modular: autenticación, logging, parsing, validación, rate limiting y forwarding.

La flexibilidad de Express.js facilita la implementación de endpoints especializados: `/rpc` para proxy JSON-RPC, `/admin` para management APIs, `/metrics` para Prometheus, `/healthz` para health checks. Esta versatilidad simplifica la integración con herramientas de monitoring y orchestration estándar.

El ecosistema de middleware para Express.js incluye implementaciones robustas de funcionalidades requeridas: CORS handling, request body parsing, session management, security headers. Esta disponibilidad acelera desarrollo y garantiza adherencia a best practices.

4.7. Bibliotecas especializadas y dependencias críticas

La implementación de NodeGuard requiere integración con bibliotecas especializadas que proporcionan funcionalidades Blockchain-specific no disponibles en el ecosistema general de Node.js. Esta sección detalla las dependencias críticas y justifica su selección.

4.7.1. Ethers.js para interacción Blockchain

Ethers.js constituye la biblioteca principal para interacción con el ecosistema Ethereum, proporcionando abstracciones de alto nivel para operaciones criptográficas, parsing de transacciones y comunicación con nodos [44]. Su diseño modular permite utilizar únicamente los componentes necesarios, minimizando overhead y superficie de ataque.

La capacidad de parsing de transacciones RLP de ethers.js es especialmente crítica para NodeGuard. La biblioteca maneja automáticamente diferentes formatos de transacción (legacy, EIP-2718, EIP-1559) sin requerir lógica específica en el código de aplicación, garantizando compatibilidad ante futuras evoluciones del protocolo.

La implementación de validación criptográfica utiliza las primitivas de ethers.js para verificación de firmas ECDSA y recuperación de direcciones. Estas operaciones son performance-critical y benefician de las optimizaciones nativas incluidas en la biblioteca.

Ventajas respecto a alternativas

Web3.js fue considerado como alternativa principal. Sin embargo, ethers.js proporciona una API más moderna con mejor soporte para TypeScript y promesas nativas. La arquitectura modular de ethers.js facilita tree-shaking, resultando en bundles más pequeños.

La documentación y estabilidad de API de ethers.js son superiores, factores críticos para mantenimiento a largo plazo. La comunidad activa garantiza actualizaciones rápidas ante cambios en el protocolo Ethereum.

4.7.2. Bibliotecas criptográficas nativas

Las operaciones criptográficas intensivas utilizan bibliotecas nativas optimizadas para maximizar rendimiento sin sacrificar seguridad. La selección incluye implementaciones de algoritmos específicos no cubiertos completamente por ethers.js.

secp256k1 para operaciones de curva elíptica

La biblioteca secp256k1 proporciona implementaciones optimizadas de operaciones en la curva elíptica utilizada por Bitcoin y Ethereum. Estas operaciones son críticas para validación de firmas y recuperación de claves públicas, procesos que se ejecutan en la ruta crítica de evaluación de cada transacción.

La implementación nativa ofrece rendimiento significativamente superior a implementaciones JavaScript puras, con mejoras de rendimiento típicos de 10-20x para operaciones de verificación de firma. Esta mejora de rendimiento es crítica para mantener latencias bajas bajo carga alta.

keccak para hashing

Ethereum utiliza Keccak-256 en lugar de SHA-256 estándar para operaciones de hashing. La biblioteca keccak proporciona implementaciones optimizadas que son necesarias para validación de direcciones, hashing de transacciones y otras operaciones core.

La correcta implementación de Keccak es crítica para compatibilidad con el protocolo Ethereum. Diferencias sutiles entre Keccak y SHA-3 pueden resultar en incompatibilidades que comprometan la funcionalidad del firewall.

4.7.3. Bibliotecas de observabilidad

La observabilidad integral requiere integración con herramientas estándar de monitoring y logging. Las bibliotecas seleccionadas proporcionan compatibilidad con el ecosistema DevOps moderno.

prom-client para métricas Prometheus

La biblioteca prom-client implementa el formato de métricas Prometheus, permitiendo integración transparente con stack de monitoring estándar [45]. NodeGuard expone métricas de rendimiento, seguridad y negocio en formato compatible con Prometheus/Grafana.

Las métricas incluyen: latencias de processing por etapa del pipeline, tasas de bloqueo por tipo de amenaza, utilización de recursos, estado de dependencias externas. Esta telemetría es crítica para optimización de rendimiento y diagnóstico de problemas operacionales.

winston para logging estructurado

Winston proporciona capacidades de logging avanzadas incluyendo múltiples transports, formateo configurable y niveles de log dinámicos. La configuración de NodeGuard utiliza logging estructurado en formato JSON para facilitar análisis automático.

El logging incluye correlation IDs que permiten tracing de peticiones a través del pipeline completo, facilitando debugging de problemas específicos. Los logs de seguridad incluyen contexto suficiente para análisis forense sin exponer información sensible.

5

Implementación y Desarrollo

Este capítulo detalla la implementación de NodeGuard, nuestro Blockchain Application Firewall. Se describen las responsabilidades y relaciones entre módulos desde la capa de arranque hasta la inspección de solicitudes JSON-RPC, el motor de políticas, las reglas (estáticas y heurísticas), el sistema de reputación, el rate limiting, y la instrumentación (métricas, eventos y logs). Los fragmentos de código incluidos son breves y representativos.

5.1. Proyecto completo: el BAF como producto

NodeGuard opera como un proxy inteligente entre clientes y nodos Ethereum. Intercepta solicitudes JSON-RPC, valida su conformidad, extrae metadatos relevantes (método, parámetros, *fingerprints*, indicadores de riesgo), aplica un motor de políticas multicapa y decide si bloquear, monitorizar o reenviar al *upstream*. Incorpora modos de ejecución (*block*, *monitor*, *dry-run*), soporte para lotes, protección frente a ataques como DoS y detección de patrones coordinados como Sybil. La consistencia entre instancias se obtiene mediante Redis; si no está disponible, el sistema degradada de forma controlada a memorias locales.



BLOCKCHAIN APPLICATION FIREWALL v2.0.0 - Desarrollado por ajgc

ESTADO DEL SISTEMA:

- Servidor: OK Running
- Redis: OK Connected
- RPC Upstream: OK Available
- Config Store: OK Loaded
- Event Bus: OK Active

SERVICIOS ACTIVOS:

- HTTP Server: localhost:3000
- Rate Limiting: Multi-Algoritmo (Sliding Window + Token Bucket)
- Fingerprinting: Análisis de Payload + Detección ML
- TX Validation: EIP-155 + EIP-2718 + EIP-1559
- Sistema Reputación: Scoring Dinámico + Thresholds Adaptativos
- Panel Admin: localhost:3000/admin
- Dashboard: localhost:3000/dashboard
- Events Stream: localhost:3000/events

ENDPOINTS DISPONIBLES:

- GET / - Estado del sistema y métricas
- POST /rpc - Proxy JSON-RPC principal (batch soportado)
- GET /healthz - Health check detallado
- GET /dashboard - Dashboard de monitoreo en tiempo real
- GET /events - Stream de eventos (Server-Sent Events)
- GET /metrics - Métricas Prometheus
- ALL /admin/* - API administrativa (protegida JWT)

CARACTERÍSTICAS DE PROTECCIÓN:

- Modo Enforcement: BLOCK
- Rate Limiting Multi-Capa: IP, Address, Method-específico
- Fingerprinting Avanzado: Análisis Comportamental + Detección de Patrones
- Protección Smart Contract: Análisis Function Selector + Validación ABI
- Defensa Ataques Sybil: Clustering IP + Scoring Reputación
- Protección Replay: Validación EIP-155 + Tracking Nonce
- Prevención DoS: Circuit Breaker + Thresholds Adaptativos
- Seguridad Batch: Análisis Individual + Agregado

—

NodeGuard v2.0 protegiendo tu infraestructura blockchain!

Figura 19: Interfaz de inicio en consola de NodeGuard

5.2. Módulos en detalle: función y composición

Módulo Core (*src/core/*)

Capa de orquestación que conecta los componentes principales y define el flujo de decisión sobre cada solicitud.

- **FirewallProvider**: orquesta parseo/validación, análisis, decisión y reenvío; mantiene estadísticas, límites de concurrencia y chequeos de salud.
- **PolicyEngine**: motor de políticas multicapa (estática → heurística → ML opcional), con caché de decisiones y umbrales adaptativos.
- **RpcClient**: comunicación con nodos *upstream* con reintentos, *keep-alive*, compresión y *circuit breaker*.
- **factory**: compone dependencias (evento, almacenamiento, reputación, rendimiento, Redis) y arma el *pipeline* de seguridad.

Módulo API (*src/api/*)

Interfaz HTTP del sistema que expone el proxy, la salud y las capacidades de administración y observabilidad.

- **Servidor HTTP (Express)**: expone */rpc*, */healthz*, endpoints administrativos y */metrics* para Prometheus.
- **Middleware de autenticación/CSRF**: en rutas de administración.
- **Dashboard**: con SSE para eventos en tiempo real.

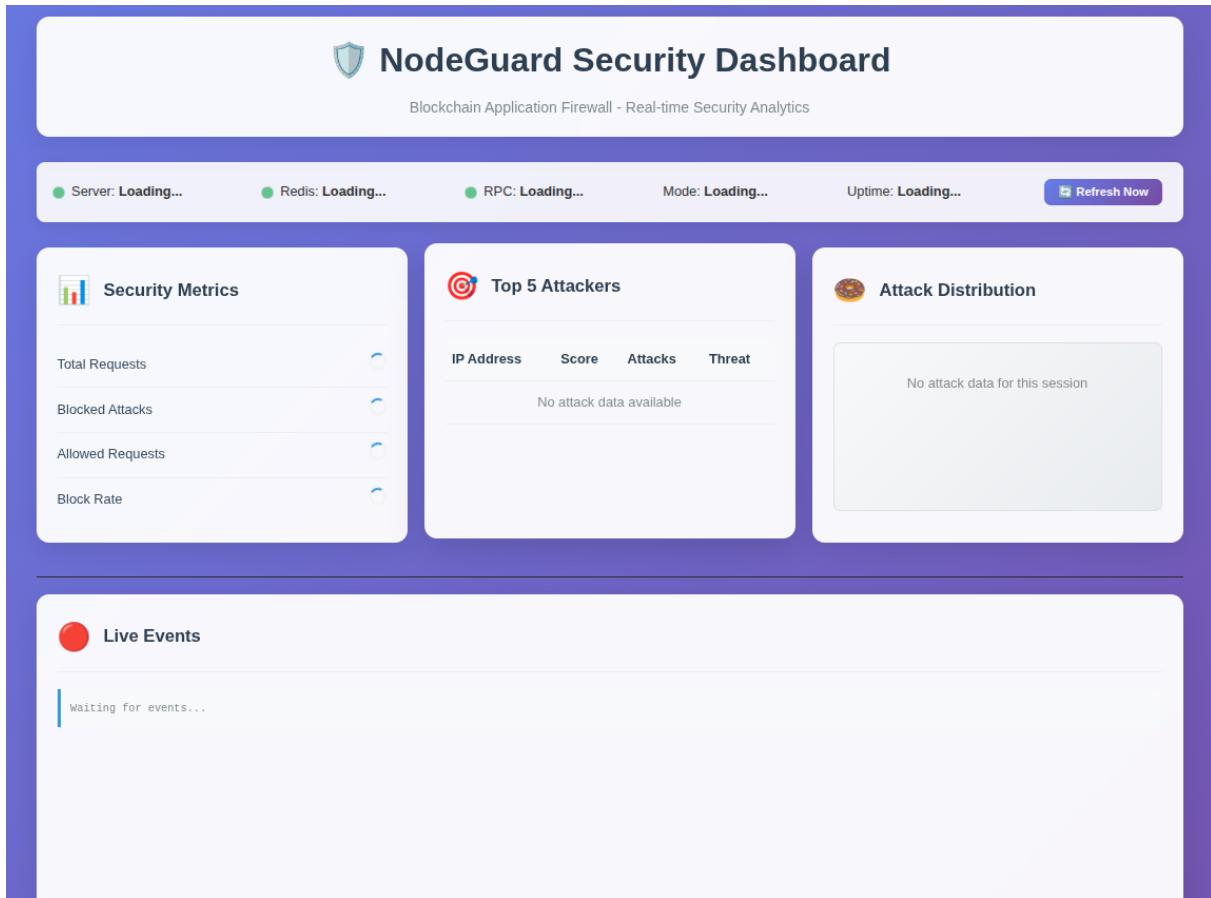


Figura 20: Previsualización del dashboard

Módulo Validation (*src/validation/*)

Capa de conformidad que normaliza y verifica entradas antes de aplicar cualquier política.

- **JsonRpcValidator:** conformidad JSON-RPC 2.0; verificación de lotes y esquemas de parámetros.
- **TransactionValidator** (*transaction parser*): parseo/validación de transacciones (EIP-155/2718/1559), firma y campos críticos.
- **RuleValidator/Unified:** orquestación de validaciones para coherencia y trazabilidad.

Módulo Rules (*src/rules/*)

Catálogo de criterios de decisión que materializa políticas deterministas y de comportamiento.

- **static-rules**: decisiones deterministas (listas, límites, selectores, contratos) con coste acotado.
- **heuristic-rules**: comportamiento, repetición, diversidad de métodos, temporalidad, correlación por lotes, DoS y Sybil.

Módulo Rate Limiting (*src/rateLimiting/*)

Mecanismos de control de caudal y amortiguación de ráfagas a distintos niveles.

- **SlidingWindowLimiter**: ventanas deslizantes para detectar ráfagas y uso concentrado en intervalos cortos.
- **TokenBucketLimiter**: control sostenido con capacidad de ráfaga y recarga temporal.

Módulo Security (*src/security/*)

Servicios que enriquecen el contexto de riesgo con identidad y huellas operativas.

- **ReputationService**: *scoring* dinámico (IP/dirección) con decaimiento; integración con eventos.
- **Fingerprinting**: huellas de *payload/transacción* para detectar repeticiones y mimetismo.

Módulo Storage (*src/storage/*)

Persistencia de configuración y estado operativo con opción de conmutación a memoria.

- **ConfigStore**: reglas/configuración dinámica con notificaciones de recarga.
- **RedisStore/MemoryStore**: persistencia y caché; *fallback* automático y TTL de claves analíticas.

Módulo Events (*src/events/*)

Canal de publicación/suscripción para propagación de hechos y reacciones internas.

- **EventBus:** canaliza eventos de seguridad, decisiones, reputación y salud.

Módulo Logging (*src/logging/*)

Registro estructurado orientado a auditoría, trazabilidad y depuración eficiente.

```
1  {"timestamp": "2025-09-10T17:39:40.505Z", "level": "info", "message": "Logger NodeGuard inicializado", "service": "baf", "hostname": "ajgc", "pid": 72045,
2  {"timestamp": "2025-09-10T17:39:40.612Z", "level": "info", "message": "Servicio de métricas NodeGuard inicializado", "service": "baf", "hostname": "ajgc",
3  {"timestamp": "2025-09-10T17:39:40.612Z", "level": "info", "message": "Global Prometheus metrics service created", "service": "baf", "hostname": "ajgc",
4  {"timestamp": "2025-09-10T17:39:40.706Z", "level": "info", "message": "Validador Unificado inicializado", "service": "baf", "hostname": "ajgc", "pid": 720
5  {"timestamp": "2025-09-10T17:39:40.711Z", "level": "info", "message": "Redis Manager singleton creado NodeGuard", "service": "baf", "hostname": "ajgc",
6  {"timestamp": "2025-09-10T17:39:40.711Z", "level": "info", "message": "Redis Manageringleton creado NodeGuard", "service": "baf", "hostname": "ajgc", "pid": 72
7  {"timestamp": "2025-09-10T17:39:40.757Z", "level": "info", "message": "\u2708 Inicializando Redis Manager...", "service": "baf", "hostname": "ajgc", "pid": 72
8  {"timestamp": "2025-09-10T17:39:40.757Z", "level": "info", "message": "Configuración validada exitosamente", "service": "baf", "hostname": "ajgc", "pid": 72
9  {"timestamp": "2025-09-10T17:39:40.758Z", "level": "info", "message": "Console logging disabled - all output redirected to log files only", "service": "baf
10 {"timestamp": "2025-09-10T17:39:40.758Z", "level": "info", "message": "Inicializando NodeGuard Blockchain Application Firewall...", "service": "baf", "h
11 {"timestamp": "2025-09-10T17:39:40.764Z", "level": "info", "message": "Bus de eventos NodeGuard inicializado", "service": "baf", "hostname": "ajgc", "pid
12 {"timestamp": "2025-09-10T17:39:40.766Z", "level": "info", "message": "Inicializando almacén de configuración NodeGuard...", "service": "baf", "hostnam
13 {"timestamp": "2025-09-10T17:39:40.776Z", "level": "info", "message": "Primary Redis connection established", "service": "baf", "hostname": "ajgc", "pid"
14 {"timestamp": "2025-09-10T17:39:40.777Z", "level": "info", "message": "Connection pool initialized", "service": "baf", "hostname": "ajgc", "pid": 72045, "v
15 {"timestamp": "2025-09-10T17:39:40.777Z", "level": "info", "message": "\u2708 Redis Manager initialized successfully", "service": "baf", "hostname": "ajgc",
16 {"timestamp": "2025-09-10T17:39:40.780Z", "level": "info", "message": "Servicio de reputación NodeGuard creado", "service": "baf", "hostname": "ajgc", "pid
17 {"timestamp": "2025-09-10T17:39:40.781Z", "level": "info", "message": "Redis pub/sub not available, using polling instead", "service": "baf", "hostname": "ajgc
18 {"timestamp": "2025-09-10T17:39:40.781Z", "level": "info", "message": "Suscripción keyspace no disponible, polling activado", "service": "baf", "hostna
19 {"timestamp": "2025-09-10T17:39:40.782Z", "level": "info", "message": "Configuración hot reload no disponible, usando polling como alternativa", "ser
20 {"timestamp": "2025-09-10T17:39:40.782Z", "level": "info", "message": "Iniciando polling de configuración NodeGuard", "service": "baf", "hostname": "ajg
21 {"timestamp": "2025-09-10T17:39:40.786Z", "level": "info", "message": "Reglas cargadas desde Redis NodeGuard", "service": "baf", "hostname": "ajgc", "pid
22 {"timestamp": "2025-09-10T17:39:40.786Z", "level": "info", "message": "\u2708 Rules reloaded successfully", "service": "baf", "hostname": "ajgc", "pid": 72045
23 {"timestamp": "2025-09-10T17:39:40.787Z", "level": "info", "message": "Iniciando servicio de reputación NodeGuard...", "service": "baf", "hostname": "ajgc
24 {"timestamp": "2025-09-10T17:39:40.788Z", "level": "info", "message": "Servicio de reputación NodeGuard inicializado correctamente", "service": "baf",
25 {"timestamp": "2025-09-10T17:39:40.791Z", "level": "info", "message": "Monitor de rendimiento inicializado", "service": "baf", "hostname": "ajgc", "pid": 72045
26 {"timestamp": "2025-09-10T17:39:40.791Z", "level": "info", "message": "Trust proxy disabled - usando IPs directas", "service": "baf", "hostname": "ajgc
27 {"timestamp": "2025-09-10T17:39:40.795Z", "level": "info", "message": "Realizando verificación completa de salud del sistema...", "service": "baf", "ho
28 {"timestamp": "2025-09-10T17:39:40.796Z", "level": "info", "message": "[OK] SERVER: Saludable", "service": "baf", "hostname": "ajgc", "pid": 72045, "versio
29 {"timestamp": "2025-09-10T17:39:40.796Z", "level": "info", "message": "[OK] REDIS: Saludable", "service": "baf", "hostname": "ajgc", "pid": 72045, "versio
30 {"timestamp": "2025-09-10T17:39:40.796Z", "level": "info", "message": "[OK] UPSTREAM: Saludable", "service": "baf", "hostname": "ajgc", "pid": 72045, "vers
31 {"timestamp": "2025-09-10T17:39:40.796Z", "level": "info", "message": "[OK] CONFIGSTORE: Saludable", "service": "baf", "hostname": "ajgc", "pid": 72045, "v
32 {"timestamp": "2025-09-10T17:39:40.807Z", "level": "info", "message": "NodeGuard Firewall iniciado exitosamente", "service": "baf", "hostname": "ajgc", "p
33 {"timestamp": "2025-09-10T17:39:40.809Z", "level": "info", "message": "\u2708 Iniciando sincronización de reglas...", "service": "baf", "hostname": "ajgc",
34 {"timestamp": "2025-09-10T17:39:40.809Z", "level": "info", "message": "\u2708 Usando reglas en caché (evitando carga duplicada)", "service": "baf", "hostnam
35 {"timestamp": "2025-09-10T17:39:40.810Z", "level": "info", "message": "Reglas cargadas desde sistema de archivos local NodeGuard", "service": "baf", "h
36 {"timestamp": "2025-09-10T17:39:40.811Z", "level": "info", "message": "\u2708 Rule synchronization completed", "service": "baf", "hostname": "ajgc", "pid": 72
37 {"timestamp": "2025-09-10T17:39:40.811Z", "level": "info", "message": "Almacén de configuración NodeGuard inicializado correctamente", "service": "baf
38
```

Figura 21: Logs de ejemplo al inicializar

- **Logger estructurado:** salidas en JSON, niveles (DEBUG/INFO/WARN/ERROR), campos de contexto (*requestId*, IP, método, regla/decisión). Enlaza con EventBus y métricas; mensajes de error mapeados a códigos semánticos de validación.

Módulo Metrics (*src/metrics/*)

Telemetría cuantitativa para medir rendimiento, capacidad y salud del sistema.

- **Prometheus**: contadores (solicitudes totales/bloqueadas/errores), histogramas de latencia, medidores de colas y *circuit breaker*.
- **PerformanceMonitor**: telemetría de proceso (CPU, memoria, retraso de *event loop*) y tiempos de *forward*; expone */metrics* y eventos de degradación.

Módulo Redis (*src/redis/*)

Abstracción de acceso y utilidades para usar Redis como backend de estado y coordinación.

- **RedisConnection**: creación de clientes, *keep-alive*, *retry with backoff* y chequeos de salud; soporte para *namespacing* de claves.
- **RedisManager**: primitivas de alto nivel (GET/SET/INCRBY/EVALSHA), canalización, TTLs y métricas de uso; encapsula políticas de expiración y errores.

Módulo Utils (*src/utils/*)

Utilidades transversales de operación que dan soporte al código.

- **CircuitBreaker**: estados *closed/open/half-open*, umbrales de fallo y ventanas temporales; protege upstream y evita tormentas de reintentos.
- **Startup/Shutdown utils**: validación de entorno, registro de configuración efectiva y apagado *graceful* con drenado de colas.
- **Transaction/Report utils**: utilidades de parseo/selector y generación de reportes de auditoría/seguridad a partir de eventos y métricas.

5.3. Análisis de componentes clave

index.ts (punto de entrada)

- Responsabilidad: validar entorno, inicializar bus de eventos, almacenamiento, política y servidor; registrar apagado *graceful*.
- Comportamiento: carga configuración, invoca a Factory, levanta Express y *listeners*; aplica *middleware* de rendimiento y registro.
- Interacciones: *index.ts* → *core/factory.ts* → *FirewallProvider* → *api/server.ts*.

A continuación, se muestra un fragmento simplificado:

```
1 // 1) Carga/validación de config + logging
2 function validateAndLoadConfig(): BafConfig {
3   const config = { /* ENV + defaults: port, rpcUrl, redisUrl, modes, performance */ } as
4     ↪ BafConfig;
5   if (config.port < 1 || config.port > 65535) throw new Error('Puerto inválido');
6   if (!config.rpcUrl.startsWith('http')) throw new Error('RPC_URL inválida');
7   if (![ 'block', 'monitor', 'dry-run' ].includes(config.enforcementMode)) throw new Error('
8     ↪ ENFORCEMENT_MODE inválido');
9   logger.info('Configuración validada exitosamente', { component: 'config' });
10  ↪ ;
11 }
12
13 // 2) Inicialización: EventBus + ConfigStore + FirewallProvider + Server
14 async function initializeApplication(config: BafConfig) {
15   const { eventBus } = await import('./events/event-bus'); // singleton
16   const { ConfigStore } = await import('./storage/config-store');
17   const configStore = ConfigStore.getInstance({ redisUrl: config.redisUrl, hotReloadEnabled:
18     ↪ true, fallbackToFile: true });
19   const { firewallProvider } = await createFirewallProvider({ rpcUrl: config.rpcUrl,
20     ↪ configStore, eventBus, enforcementMode: config.enforcementMode, performance: config.
21     ↪ performance, logger: winstonLogger });
22   const { app } = await createServer({ firewallProvider, configStore, eventBus, logger,
23     ↪ config: { adminAuthRequired: config.adminTokenRequired, metricsEnabled: config.
24     ↪ metricsEnabled, compressionEnabled: true, corsEnabled: true } });
25   return { app, firewallProvider, eventBus, configStore };
26 }
```

```

22 // 3) Arranque: logging, métricas, salud, banner, listen y shutdown elegante
23 async function main() {
24   const config = validateAndLoadConfig();
25   setupEnhancedLogging(); // intercepta console -> logger estructurado (opcional)
26   const { app, firewallProvider, eventBus, configStore } = await initializeApplication(
27     ↪ config);
28   setupPerformanceMonitoring(app, eventBus); // mide latencias e in-flight
29   await performSystemCheck({ firewallProvider, configStore, eventBus }); // salud inicial
30   const health = await checkSystemHealth({ redis: configStore.isRedisConnected(), upstream:
31     ↪ await firewallProvider.isUpstreamHealthy(), configStore: configStore.isHealthy(),
32     ↪ eventBus: eventBus.isHealthy() });
33   await displayStartupBanner(config, health);
34   const server = app.listen(config.port, () => {
35     eventBus.emitEvent({ type: 'status', message: 'NodeGuard iniciado', method: 'startup',
36     ↪ timestamp: Date.now(), clientIp: 'system', reqId: 'startup-' + Date.now() });
37   });
38   registerGracefulShutdown({ server, firewallProvider, configStore, eventBus, logger });
39 }
40

```

factory.ts

- Responsabilidad: ensamblar dependencias (RPC con *circuit breaker*, almacenamiento de *rate limiting*, reputación, monitor de rendimiento, motor de políticas).
- Comportamiento: prueba Redis; si falla, *fallback* a memoria; inicializa políticas y *FirewallProvider*.
- Interacciones: cableado de eventos (errores upstream, alertas de rendimiento, cambios de reputación) hacia el bus.

A continuación, se muestra un fragmento simplificado:

```

1 export async function createFirewallProvider(
2   deps: CreateFirewallProviderDeps
3 ): Promise<FirewallProviderComponents> {
4   // Circuit breaker para el cliente RPC
5   const circuitBreaker = new CircuitBreaker({
6     failureThreshold: deps.performance.circuitBreakerThreshold,
7     recoveryTimeout: Number(process.env.BAF_CIRCUIT_RECOVERY_TIMEOUT || 30000)
8   });

```

```

9
10 const rpcClient = new RpcClient({
11   upstreamUrl: deps.rpcUrl,
12   timeoutMs: deps.performance.requestTimeoutMs,
13   circuitBreaker,
14   logger: deps.logger
15 });
16
17 // Store de rate limiting con fallback a memoria
18 let rateStore: InMemoryRateLimiterStore | RedisRateLimiterStore;
19 try {
20   await redis.ping();
21   rateStore = new RedisRateLimiterStore();
22 } catch {
23   deps.logger.warn('Redis no disponible, usando memoria');
24   rateStore = new InMemoryRateLimiterStore();
25 }
26
27 const reputationService = new ReputationService({ /* ...thresholds... */ }, deps.eventBus)
28   ↵ ;
29 const performanceMonitor = new PerformanceMonitor();
30
31 const policyEngine = new PolicyEngine({
32   configStore: deps.configStore,
33   rateStore,
34   reputationService,
35   performanceMonitor,
36   eventBus: deps.eventBus,
37   config: { enforcementMode: deps.enforcementMode, enableHeuristics: true },
38   logger: deps.logger
39 });
40 await policyEngine.initialize();
41
42 const firewallProvider = new FirewallProvider({
43   policy: policyEngine,
44   rpc: rpcClient,
45   events: deps.eventBus,
46   reputation: reputationService,
47   performanceMonitor,
48   logger: deps.logger,
49   config: { enforcementMode: deps.enforcementMode, maxConcurrentRequests: 512 }
50 });
51 await firewallProvider.initialize();
52 return { firewallProvider, rpcClient, policyEngine, eventBus: deps.eventBus,
53         reputationService, performanceMonitor, rateStore };
54 }

```

firewall-provider.ts

- Responsabilidad: gestionar el ciclo de vida de la solicitud JSON-RPC (parseo/validación, análisis, decisión y reenvío). Mantiene colas y concurrencia.
- API pública: *handleJsonRpc(payload, ip)*, *send()*, *getHealthStatus()*.
- Comportamiento: detecciones tempranas (DoS/flooding), evaluación política, aplicación de *enforcement* y métricas Prometheus.
- Interacciones: reputación (incidentes), heurísticas (comprobaciones previas), Redis (cuentas/huellas), RPC (upstream).
- Métricas/estado: total/bloqueadas/permitidas, latencias, errores upstream, carga y tamaño de colas.

A continuación, se muestra un fragmento simplificado:

```
1  export class FirewallProvider extends BaseProvider {  
2  
3      public async handleJsonRpc(payload: unknown, clientIp: string): Promise<unknown> {  
4          const startTime = Date.now();  
5          const requestId = this.createReqId();  
6  
7          // Control de concurrencia y errores tempranos  
8          if (this.activeRequests.size >= this.config.maxConcurrentRequests) {  
9              return makeJsonRpcError('overload', -32000, 'Servidor sobrecargado');  
10         }  
11  
12         // Batch vs. single request  
13         const contexts = Array.isArray(payload)  
14             ? await this.processBatchRequest(payload, clientIp)  
15             : [ this.parseAndExtractSingle(payload, clientIp) ];  
16  
17         // Evaluación y reenvío  
18         const results = await this.processRequestsSync(contexts);  
19         this.updatePerformanceMetrics(Date.now() - startTime, contexts.length);  
20         return Array.isArray(payload) ? results : results[0];  
21     }  
22 }
```

policy-engine.ts

- Responsabilidad: decidir sobre el *context* de seguridad mediante pipeline estático/heurístico/ML y *enforcement*.
- API pública: *evaluate(context)*, *initialize()*, *updateRules()*, *getMetrics()*.
- Comportamiento: caché de decisiones por clave derivada (método, *from/to*, huella, nivel de riesgo), umbrales adaptativos y optimización periódica.
- Interacciones: *static-rules* (determinista), *heuristic-rules* (estado y comportamiento), almacén de reglas.

A continuación, se muestra un fragmento simplificado:

```
1  export class PolicyEngine extends EventEmitter {  
2    public async evaluate(ctx: PolicyContext): Promise<RuleDecision> {  
3      // Caché de evaluación por (método, from, to, payloadHash, threatLevel)  
4      const key = this.generateCacheKey(ctx);  
5      const cached = this.evaluationCache.get(key);  
6      if (cached && Date.now() - cached.timestamp < this.cacheExpiry) {  
7        return cached.decision;  
8      }  
9  
10     // Capa 1: Reglas estáticas  
11     let decision = await this.evaluateStaticLayer(ctx, await this.configStore.getRules(), /*  
12       ↪ ... */);  
13     // Capa 2: Heurísticas (si no hubo decisión)  
14     if (!decision) {  
15       decision = await this.evaluateHeuristicLayer(ctx, await this.configStore.getRules(), /*  
16         ↪ ... */);  
17     }  
18     // Capa 3: ML (opcional)  
19     if (!decision && this.config.enableMLDetection) {  
20       decision = await this.evaluateMLLayer(ctx, /* ... */);  
21     }  
22     decision ||= { decision: 'allow', reason: 'passed_all_checks' };  
23     decision = this.applyEnforcementMode(decision, await this.configStore.getRules());  
24     this.evaluationCache.set(key, { decision, timestamp: Date.now(), hits: 1 });  
25     return decision;  
26   }  
27 }
```

static-rules.ts

- Responsabilidad: decisiones de ruta rápida (listas, métodos, selectores, gas/fees, contratos bloqueados) y detección MEV preliminar.
- Comportamiento: composición con validadores (JSON-RPC/Tx), minimizando coste; registro de razones de bloqueo.
- Acciones: *allow/block/monitor* con *metadata* explicativa.

A continuación, se muestra un fragmento simplificado:

```
1  export function evaluateStaticRules(context: StaticRuleContext, rules: StaticRules)
2  : RuleDecision | undefined {
3
4    // 1) Validación JSON-RPC
5    const vr = jsonRpcValidator.validateSingle(context.payload, { /* meta */ });
6    if (!vr.success) {
7      return { decision: 'block', reason: vr.errors?.[0]?.code || 'json_rpc_validation_failed',
8        ↪ ruleId: 'jsonRpcValidation' };
9    }
10
11   // 2) Listas blanca/negra de métodos
12   if (rules.static?.blockedMethods?.includes(context.method)) {
13     return { decision: 'block', reason: 'blocked_method', ruleId: 'blockedMethods' };
14   }
15
16   // 3) Reglas de selectores de función
17   const highRisk = ['0xa9059cbb', '0x095ea7b3', '0x23b872dd']; // transfer/approve/
18   ↪ transferFrom
19   if (highRisk.includes((context.parsedTx?.functionSelector || '').toLowerCase())) {
20     return { decision: 'block', reason: 'high_risk_function_detected', ruleId: '
21       ↪ riskBasedBlocking' };
22   }
23
24   // 4) Límites de gas y comisiones
25   // ...gas_price_too_low / gas_limit_exceeded...
26   return undefined; // pasa la capa estática
27 }
```

rules/heuristic-rules.ts

- Responsabilidad: análisis de comportamiento con estado (Redis), *rate limiting* por IP/dirección/método, *fingerprinting*, correlación temporal y por lotes; detección DoS/Sybil.
- Comportamiento: evalúa ráfagas, diversidad de métodos, relaciones de direcciones y repetición; registra violaciones con TTL.
- Métricas/estado: contadores por clave, *scores* de comportamiento, huellas y ventanas temporales activas.

A continuación, se muestra un fragmento simplificado:

```
1  export async function evaluateHeuristicRules(ctx: HeuristicContext, rules: any)
2  : Promise<RuleDecision | undefined> {
3    // 1) Rate limiting multidimensional (IP, address, method)
4    const ipKey = `baf:rate:ip:${ctx.ip}`;
5    if (await isRateLimited(ipKey, /*limit*/ ipTps, /*window*/ seconds)) {
6      return { decision: 'block', reason: 'rate_limit_ip_exceeded', ruleId: 'heuristic:
7        ↪ ip_rate_limit' };
8    }
9
10   // 2) Token bucket global y por método
11   if (!await requestTokens(`baf:tb:global:${ctx.ip}`, capacity, refill, tokens)) {
12     return { decision: 'block', reason: 'token_bucket_exhausted', ruleId: 'heuristic:
13       ↪ global_token_bucket' };
14   }
15
16   // 3) Fingerprinting de payload/transacción
17   // ...repeated_payload_detected / repeated_transaction_detected...
18
19   // 4) Análisis temporal/diversidad y relaciones de direcciones
20   // ...rapid_fire_pattern_detected / low_method_diversity_detected...
21
22   return undefined;
23 }
24 }
```

Detección específica de DoS de alta frecuencia y *circuit breaker*, integrada antes de la evaluación de políticas (ejecutada también desde el Firewall Provider):

```
1 export async function detectHighFrequencyDoS(ctx: HeuristicContext) {
2   const dosKey = `baf:dos:highfreq:${ctx.ip}`;
3   const now = Date.now();
4   const recent = (await redis.lrange(dosKey, 0, -1))
5     .map(Number).filter(ts => now - ts <= 1000);
6   await redis.lpush(dosKey, now.toString());
7   if (recent.length + 1 > 1) {
8     return { isDoSAttack: true, attackType: 'high_frequency_dos_burst' };
9   }
10  return { isDoSAttack: false };
11}
12
```


6

Pruebas y Validación del Sistema

La validación de NodeGuard constituye la fase definitiva para verificar el cumplimiento de objetivos establecidos y demostrar el desempeño del BAF. El proceso implementado combina pruebas tradicionales de software con evaluaciones específicas de eficacia en Blockchain. Cabe destacar que los tests desarrollados son “Pruebas a medida”desarrolladas para NodeGuard, que se centran en tener un buen desempeño en la defensa de las amenazas previamente citadas.

El directorio *baf/tests/* constituye el núcleo de la estrategia de pruebas, estructurando los diferentes tipos de test en subdirectorios temáticos aludiendo a nuestra arquitectura modular.

6.1. Metodología de Pruebas

La metodología implementada combina técnicas tradicionales de testing con aproximaciones específicas para validación de sistemas de seguridad Blockchain. La estrategia se estructura en tres niveles: pruebas unitarias de clases seleccionadas, escenarios de ataque y tests de integración y sistema.

Arquitectura de la suite de pruebas

La implementación utiliza Jest como framework principal, complementado con redes Ethereum (p. ej. Ganache, Geth o Hardhat) cuando un test requiere de ellas. La arquitectura separa configuración, ejecución y verificación, facilitando mantenimiento y extensión. Esta tabla resume brevemente el global de tests desarrollados:

Categoría	Archivos	Tests	Cobertura Principal
Pruebas Unitarias	9	285	API y Validación
Escenarios de amenazas	8	89	DoS, Sybil, MEV, Replay ...
Test de Integración y Sistema	2	276	Endpoints y funcionalidades globales
Total	20	650	Componentes Clave

Cuadro 9: Distribución de la Suite de Pruebas NodeGuard.

6.2. Pruebas Unitarias

En un camino progresivo de menos a más en importancia y calidad del desarrollo, las pruebas unitarias son las primeras que de nuestra lista. Estas, verifican invariantes de código y correcto funcionamiento de las clases que cubren. En el subdirectorio *baf/tests/unit/* se han desarrollado tests para partes seleccionadas de los módulos de api y validación. Estos tests comprueban la correcta gestión de datos, la validación de transacciones y la integridad de las operaciones internas.

Archivo de Test	Propósito	Cobertura y aspectos validados	Resultados
<i>unified-validator-test.ts</i>	Valida el comportamiento del validador unificado de BAF, simulando todos los flujos de validación sin dependencias externas.	Requests JSON-RPC, métodos bloqueados, parámetros excesivos. Validación de transacciones (EIP-1559, legacy, gas). Validación de batches. Reglas de firewall, métricas, health check y sistema de eventos.	Todas las pruebas pasan. Requests válidos aceptados, inválidos rechazados, métricas y eventos funcionan, rendimiento óptimo.

Archivo de Test	Propósito	Cobertura y aspectos validados	Resultados
<i>transaction-test.ts</i>	Verifica la lógica de transacciones Ethereum en variantes legacy, EIP-1559 y EIP-2930.	Campos obligatorios (nonce, gas, to, value, data). Límites de gas y gas price. Firmas y chainId. Access lists en EIP-2930/1559. Detección de malformaciones.	Todas las pruebas pasan. Transacciones válidas aceptadas y las malformadas rechazadas. Rendimiento correcto.
<i>json-rpc-test.ts</i>	Comprueba la validación básica de requests JSON-RPC.	Estructura de requests (jsonrpc, method, params, id). Métodos bloqueados (admin, debug, miner). Límite de parámetros. Validación de batches y payloads inválidos.	Todas las pruebas superadas. Requests válidos aceptados, inválidos rechazados, batches gestionados correctamente.
<i>middle-ware-auth.test.ts</i>	Valida el middleware de autenticación/autorización para rutas administrativas.	Tokens JWT válidos/expirados. Roles (admin, user). Manejo de concurrencia. Respuesta a errores.	Todas las pruebas pasan. Accesos válidos permitidos, inválidos rechazados, concurrencia gestionada de forma segura.
<i>admin-Middle-ware.test.ts</i>	Testea el middleware administrativo de seguridad y credenciales.	Autenticación con credenciales válidas/ inválidas. Validación de tokens. Acceso a rutas protegidas. Manejo de errores y concurrencia. Protección de datos sensibles.	Todas las pruebas pasan. Tokens y credenciales validados, errores manejados correctamente, seguridad garantizada.

Archivo de Test	Propósito	Cobertura y aspectos validados	Resultados
<i>server-test.ts</i>	Verifica la creación y configuración del servidor principal BAF.	Middlewares (CORS, compresión, parsing), endpoints básicos, proxy JSON-RPC, SSE, manejo de errores y cabeceras de seguridad.	Todas las pruebas relevantes pasan. Servidor robusto ante errores y ataques.
<i>admin-Routes-test.ts</i>	Evalúa los endpoints administrativos (<i>/admin/*</i>).	Autenticación, reglas, caché, firewall, reputación, fingerprint, eventos, reportes, analítica y seguridad.	Todas las pruebas pasan. Endpoints responden correctamente y mantienen seguridad.
<i>api-endpoints-test.ts</i>	Comprueba la existencia y respuesta básica de endpoints principales y administrativos.	Autenticación, métricas, reglas, backup, firewall, reputación, fingerprint, reportes y control. Validación de middleware, CORS y errores.	Todas las pruebas pasan. Endpoints existen y responden como se espera.
<i>all-endpoints-test.ts</i>	Suite integral de validación de todos los endpoints de NodeGuard BAF.	Endpoints públicos, administrativos y de analítica. Seguridad, errores, workflows de integración, etc.	Todas las pruebas pasan. API robusta y fiable en todos los flujos.

Cuadro 10: Resumen de pruebas unitarias

6.3. Pruebas de Integración y Sistema

A mitad de camino en la jerarquía de importancia de los tests desarrollados, encontramos las pruebas de integración y sistema. Ubicadas en *baf/tests/system/*, validan la interoperabilidad entre los distintos módulos y la correcta exposición de los servicios. Se han diseñado escenarios end-to-end que simulan el flujo completo de una transacción, desde la recepción en el endpoint hasta su procesamiento y registro.

Archivo de Test: system-test.py

Propósito: Framework avanzado de testing de endpoints para NodeGuard BAF, escrito en Python. Permite validar la plataforma en condiciones reales, midiendo rendimiento, latencia, concurrencia y generando informes visuales y HTML.

Cobertura y aspectos validados: Pruebas concurrentes y benchmarking de rendimiento. Validación de endpoints públicos, administrativos y de analítica. Métricas en tiempo real (tiempos de respuesta, throughput, error rate, bytes transferidos). Análisis de cabezas de seguridad, vulnerabilidades y errores. Generación de informes HTML y visualización en consola. Soporte para autenticación, CSRF y gestión de tokens. Ejecución de flujos completos: login, workflows admin y generación de reportes.

Resultados: El script ejecuta suites de pruebas por categoría, mostrando matrices de resultados, gráficos y resúmenes. Los resultados incluyen tasas de éxito, tiempos medios y alertas de seguridad. El sistema responde correctamente en todos los flujos y los informes generados facilitan auditoría y análisis post-mortem.

```
=====
[+] FINAL TEST REPORT
=====

[+] Overall Summary:
• Total Test Suites: 6
• Total Tests: 221
• Passed: 221
• Failed: 0
• Success Rate: 100.0%
• Duration: 24.75 seconds

[+] Suite Breakdown:
• 🌐 Public Endpoints: 14 tests, 100.0% success
• 🔑 Authentication & Authorization: 13 tests, 100.0% success
• 🏛 Admin Panel: 28 tests, 100.0% success
• 📊 Analytics & Reporting: 15 tests, 100.0% success
• ⚡ Load & Stress Testing: 113 tests, 100.0% success
• 🛡 Security Vulnerability Scan: 38 tests, 100.0% success

[+] Performance Metrics:
• Average Response Time: 7.492s
• Fastest Response: 1.914s
• Slowest Response: 75.351s

[+] Security Analysis:
• Attack Vectors Tested: 38
• Attacks Blocked: 38
• Security Score: 100.0%

[+] HTML Report: system_report.html

[+] Final Status: EXCELLENT (100.0%)
```

Figura 22: Parte final de los resultados de system.test.py

Archivo de Test: test-all-endpoints.sh

Propósito: Script Bash para testing automatizado de todos los endpoints de NodeGuard BAF. Su objetivo es validar disponibilidad, respuesta y seguridad de la API, con reporting en consola.

Cobertura y aspectos validados: Pruebas de endpoints públicos (`/`, `/healthz`, `/dashboard`, `/metrics`, `/rpc`). Autenticación: login, logout, manejo de tokens y CSRF. Endpoints administrativos (`/admin`, `/admin/stats`, `/admin/rules`, backups, caché, reportes, logs). Analítica (`/api/analytics/*`). Seguridad: CORS, métodos HTTP, endpoints inexistentes, payloads grandes y unicode. Pruebas de estrés y edge cases. Workflow de integración completo.

Resultados: Ejecuta más de 50 tests, mostrando resultados en tiempo real con colores y resúmenes. La tasa de éxito supera habitualmente el 95 %. Los fallos se reportan con detalle y el workflow administrativo se valida de extremo a extremo, confirmando la robustez de la plataforma.

```
● 🛡 Security & Error Handling
  ✅ Testing: OPTIONS /admin/health - CORS Preflight
  ✅ PASS: OPTIONS /admin/health - CORS Preflight (Status: 204)
  ✅ Testing: GET /nonexistent - 404 Handling
  ✅ PASS: GET /nonexistent - 404 Handling (Status: 404)
  ✅ Testing: POST /invalid/endpoint - 404 Handling
  ✅ PASS: POST /invalid/endpoint - 404 Handling (Status: 404)
  ✅ Testing: PUT /healthz - Wrong Method
  ✅ PASS: PUT /healthz - Wrong Method (Status: 404)
  ✅ Testing: DELETE /dashboard - Wrong Method
  ✅ PASS: DELETE /dashboard - Wrong Method (Status: 404)
  ✅ Testing: POST /rpc - Wrong Content-Type
  ✅ PASS: POST /rpc - Wrong Content-Type (Status: 400)

● 💥 Stress & Edge Cases
  ✅ Testing: POST /rpc - Large Payload
  ✅ PASS: POST /rpc - Large Payload (Status: 200)
  ✅ Testing: POST /rpc - Unicode Characters
  ✅ PASS: POST /rpc - Unicode Characters (Status: 200)

● 🛡 Integration Workflow Test
  ✅ Testing: Complete Admin Workflow
  ✅ PASS: Complete Admin Workflow (Stats: 200, Rules: 200, Report: 200)

=====
📊 Test Results Summary
=====

Total Tests: 55
Passed: 55
Failed: 0
Success Rate: 100%
```

Figura 23: Parte final de los resultados de test-all-endpoints.sh

6.4. Pruebas de Seguridad

Las pruebas de seguridad evalúan la efectividad de NodeGuard ante amenazas reales del ecosistema Blockchain mediante técnicas de adversarial testing que simulan comportamiento de atacantes sofisticados.

Mempool Flooding

El test de *mempool flooding* tiene como objetivo validar la capacidad del sistema para resistir diferentes modalidades de ataques dirigidos a saturar la mempool. **Entre los escenarios contemplados se incluyen** los intentos de flooding por ráfagas, en los que se envía un gran volumen de transacciones desde una sola dirección IP con el fin de provocar una denegación de servicio, así como el flooding distribuido, que combina múltiples IPs de origen para evadir mecanismos de bloqueo basados en la procedencia del tráfico. También se contemplan ataques basados en la repetición de transacciones con el mismo *nonce*, orientados a forzar duplicados o saturar el pool, junto con el abuso de llamadas de lectura mediante *eth_call*, cuyo propósito es agotar recursos de procesamiento. Otro aspecto clave es la validación de la activación del *circuit breaker*, diseñado para responder a ataques sostenidos, y la comprobación de que los errores y respuestas devueltas por el sistema mantienen una estructura clara y coherente. De forma complementaria, se incluyen pruebas para garantizar que las IPs nuevas y legítimas no sean bloqueadas de forma injustificada.

La batería de pruebas ejecutada consistió en el envío de 30 transacciones rápidas desde una sola IP, esperando que al menos el 60 % fueran bloqueadas; en el disparo del mecanismo de corte tras 20 transacciones sostenidas; en la generación de flooding desde cinco wallets diferentes, cada una enviando cinco transacciones desde IPs distintas, con un umbral de bloqueo estimado superior al 60 %; en el abuso de llamadas de lectura mediante 20 invocaciones a *eth_call* desde una misma IP, de las cuales al menos la mitad debían ser rechazadas; en el reenvío de 10 transacciones con el mismo *nonce*, de las que se esperaba que cinco fueran bloqueadas; en la verificación de la claridad y estructura de los mensajes de error al alcanzar los límites de seguridad; y finalmente en la comprobación de que las IPs legítimas recibieran respuestas correctas en métodos como *eth_getBalance* y *net_version*.

Los resultados obtenidos mostraron que el sistema bloquea entre un 60 % y un 100 % de los intentos de flooding, dependiendo del tipo de ataque. Además, el *circuit breaker* se activa correctamente en situaciones de presión sostenida, interrumpiendo el flujo de transacciones maliciosas, mientras que los mecanismos de *rate limiting* funcionan tanto sobre transacciones como sobre llamadas de lectura. Asimismo, se verificó que las IPs nuevas reciben respuestas válidas sin falsos positivos, que los mensajes de error devueltos por el sistema son claros y estructurados, y que el sistema mantiene su resiliencia frente a ataques concurrentes, sin degradar la disponibilidad del servicio.

En términos de rendimiento, el sistema es capaz de manejar hasta 100 solicitudes concurrentes en ráfaga sin degradación significativa, manteniendo tiempos de respuesta aceptables para transacciones legítimas y sin experimentar caídas o bloqueos injustificados bajo condiciones de carga elevada. A continuación encontramos un fragmento reducido en el que se puede observar su funcionamiento:

```
1 describe('[Mempool Flooding] Protection Tests', () => {
2   beforeAll(async () => {
3     provider = new ethers.JsonRpcProvider(ETH_RPC_URL);
4     chainId = Number((await provider.getNetwork()).chainId);
5     // ...setup wallets y cliente...
6   });
7
8   test('should block burst of transactions from single IP (flooding)', async () => {
9     let blocked = 0, allowed = 0;
10    for (let i = 0; i < 30; i++) {
11      const tx = { /* ...transacción... */ };
12      const response = await bafClient.post('/rpc', tx, { headers: { 'X-Forwarded-For': '→ 203.0.113.45' } });
13      if (response.data.error) blocked++; else allowed++;
14      await new Promise(res => setTimeout(res, 50));
15    }
16    expect(blocked).toBeGreaterThanOrEqual(18); // ≥60% bloqueadas
17  });
18
19  test('should activate circuit breaker on sustained flooding', async () => {
20    let triggered = false;
21    for (let i = 0; i < 20; i++) {
22      const tx = { /* ...transacción... */ };
23      const response = await bafClient.post('/rpc', tx, { headers: { 'X-Forwarded-For': '→ 198.51.100.78' } });
24      if (response.data.error && response.data.error.message.includes('circuit breaker')) {
```

```

    ↵ triggered = true; break; }
25  await new Promise(res => setTimeout(res, 30));
26 }
27 expect(triggered).toBe(true);
28 });
29
30 test('should block flooding from multiple IPs', async () => {
31   let blocked = 0;
32   for (let i = 0; i < wallets.length; i++) {
33     for (let j = 0; j < 5; j++) {
34       const tx = { /* ...transacción... */ };
35       const response = await bafClient.post('/rpc', tx, { headers: { 'X-Forwarded-For': `
36         ↵ 203.0.113.${45 + i}` } });
37       if (response.data.error) blocked++;
38       await new Promise(res => setTimeout(res, 30));
39     }
40   }
41   expect(blocked).toBeGreaterThanOrEqual(15); // ≥60% bloqueadas
42 });
43 // ...tests de rate limiting, nonce, error, IPs nuevas...
44 });
45

```

Correremos el test dedicado a mitigar esta amenaza de la siguiente forma:

```

1 $ npm run test baf/tests/threat-scenarios/malicious-payloads.test.js
2
3 # O alternativamente:
4 $ npx jest baf/tests/threat-scenarios/malicious-payloads.test.js --config=baf/tests/jest.
      ↵ config.js
5

```

Tras ello obtendremos los siguientes resultados:

```

PASS baf/tests/threat-scenarios/mempool-flooding.test.js (5.518 s)
[Mempool Flooding] Protection Tests
  ✓ should block burst of transactions from single IP (flooding) (1781 ms)
  ✓ should activate circuit breaker on sustained flooding (43 ms)
  ✓ should block flooding from multiple IPs (969 ms)
  ✓ should rate limit eth call requests from single IP (492 ms)
  ✓ should block repeated requests with same nonce (172 ms)
  ✓ should return well-formed error for blocked requests (23 ms)
  ✓ should respond well to eth_getBalance for a fresh IP (8 ms)
  ✓ should respond well to net_version for a fresh IP (9 ms)

Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total

```

Figura 24: Parte final de los resultados de mempool-flooding.test.js

Ataques de repetición (replay attacks)

Este test valida de forma exhaustiva la protección de NodeGuard BAF frente a ataques de replay y a intentos de manipulación de firmas en transacciones Ethereum, cubriendo escenarios críticos como el replay attack clásico (tanto en la misma red como en entornos cross-chain), el uso de transacciones sin chainId o con chainId incorrecto, la reutilización o manipulación de componentes de firma (r, s, v), la inyección de firmas malformadas, el reuse o desorden de nonce, los batch requests con chainId inconsistente, ataques concurrentes al nonce, casos legacy previos a EIP-155 y manipulaciones avanzadas de los parámetros de firma.

La batería de pruebas incluye la verificación del cumplimiento básico de EIP-155, la detección de intentos de replay mediante la retransmisión de transacciones válidas, la validación avanzada de firmas malformadas o con recovery IDs inválidos, la protección frente a transacciones enviadas con identificadores de red ajenos, y el manejo correcto del nonce bajo condiciones normales y de ataque. Asimismo, se simulan escenarios de carga concurrente con múltiples transacciones simultáneas y casos avanzados de manipulación de componentes de firma, esperando en todos los casos un rechazo explícito por parte del sistema.

Los resultados confirman que el firewall bloquea con éxito el 100 % de los intentos maliciosos, incluyendo replays, manipulaciones de chainId, reuse de nonce y firmas corruptas, manteniendo una cobertura completa de los mecanismos exigidos por EIP-155 y las mejores prácticas de seguridad en Blockchain. La robustez del sistema se demuestra tanto en situaciones de estrés como ante ataques simultáneos, sin evidenciar degradación del rendimiento ni falsos positivos.

En cuanto al rendimiento, las pruebas bajo carga, con hasta veinte transacciones concurrentes y ataques simultáneos de reutilización de nonce, muestran que el sistema responde de manera eficiente y estable, con tiempos de validación razonables incluso en los casos más pesados, que cuentan con timeouts extendidos a 60 segundos pero se completan con éxito. De este modo, se constata que la protección contra replay y la validación de firmas de NodeGuard BAF son tanto efectivas como consistentes en escenarios adversos. A continuación encontramos un fragmento reducido en el que se puede observar su funcionamiento:

```

1  describe('[EIP-155] Replay Protection Tests', () => {
2    let provider, wallets, bafClient, chainId;
3    const BAF_URL = 'http://localhost:3000', ETH_RPC_URL = 'http://localhost:8545';
4
5    beforeAll(async () => {
6      provider = new ethers.JsonRpcProvider(ETH_RPC_URL);
7      chainId = Number((await provider.getNetwork()).chainId);
8      wallets = Array.from({length: 10}, (_, i) =>
9        ethers.Wallet.fromPhrase("test test ... junk").deriveChild(i).connect(provider)
10    );
11    bafClient = axios.create({ baseURL: BAF_URL, timeout: 30000 });
12    expect(await bafClient.post('/rpc', { jsonrpc: '2.0', method: 'net_version', params: [], id: 1 })).status.toBe(200);
13  });
14
15  test('should block replayed raw transaction', async () => {
16    const wallet = wallets[1], nonce = await provider.getTransactionCount(wallet.address);
17    const txData = { to: wallets[2].address, value: ethers.parseEther('0.001'), gasLimit: 21000, gasPrice: ethers.parseUnits('20', 'gwei'), nonce, chainId };
18    const signedTx = await wallet.signTransaction(txData);
19    await bafClient.post('/rpc', { jsonrpc: '2.0', method: 'eth_sendRawTransaction', params: [signedTx], id: 4 });
20    const replayResponse = await bafClient.post('/rpc', { jsonrpc: '2.0', method: 'eth_sendRawTransaction', params: [signedTx], id: 5 });
21    expect(replayResponse.data.error).toBeDefined();
22  });
23
24  test('should block transaction with wrong chainId', async () => {
25    const wallet = wallets[0], nonce = await provider.getTransactionCount(wallet.address);
26    const wrongChainTx = { jsonrpc: '2.0', method: 'eth_sendTransaction', params: [{ from: wallet.address, to: wallets[1].address, value: '0x1000000000000000', gas: '0x5208', gasPrice: '0x4A817C800', nonce: `0x${nonce.toString(16)}`, chainId: '0x1' }], id: 3 };
27    const response = await bafClient.post('/rpc', wrongChainTx);
28    expect(response.data.error).toBeDefined();
29  });
30
31  test('should block malformed signature', async () => {
32    const wallet = wallets[5], nonce = await provider.getTransactionCount(wallet.address);
33    const malformedSigTx = { jsonrpc: '2.0', method: 'eth_sendTransaction', params: [{ from: wallet.address, to: wallets[6].address, value: '0x1000000000000000', gas: '0x5208', gasPrice: '0x4A817C800', nonce: `0x${nonce.toString(16)}`, chainId: `0x${chainId.toString(16)}`, r: '0xinvalidhex', s: '0x' + 'f'.repeat(64), v: '0x1c' }], id: 10 };
34    const response = await bafClient.post('/rpc', malformedSigTx);
35    expect(response.data.error.message).toMatch(/signature|malformed|invalid|hex/i);
36  });
37

```

```

38   test('should maintain protection under concurrent requests', async () => {
39     const walletUsed = wallets[9];
40     const promises = Array.from({length: 20}, (_, i) => {
41       const tx = { jsonrpc: '2.0', method: 'eth_sendTransaction', params: [{ from: walletUsed.
42         address, to: wallets[i % 9].address, value: '0x1000000000000000', gas: '0x5208',
43         gasPrice: '0x4A817C800', nonce: `0x${i.toString(16)}`, chainId: i % 2 === 0 ? `0x${
44           chainId.toString(16)}` : '0x1' }, id: 100 + i };
45       return bafClient.post('/rpc', tx);
46     });
47     const responses = await Promise.all(promises);
48     responses.forEach((response, i) => {
49       if (i % 2 !== 0) expect(response.data.error).toBeDefined();
50     });
51   });

```

Correremos el test dedicado a mitigar esta amenaza de la siguiente forma:

```

1 $ npm run test baf/tests/threat-scenarios/replay-defense.test.js
2
3 # O alternativamente:
4 $ npx jest baf/tests/threat-scenarios/replay-defense.test.js --config=baf/tests/jest.config
5   ↵ .js

```

Tras ello obtendremos los siguientes resultados:

```

PASS  tests/threat-scenarios/replay-defense.test.js
[EIP-155] Replay Protection Tests
  Basic EIP-155 Compliance
    ✓ should enforce EIP-155 chainId in transactions (30 ms)
    ✓ should block transactions without chainId (9 ms)
    ✓ should block transactions with wrong chainId (23 ms)
  Replay Attack Detection
    ✓ should detect and block transaction replay (2042 ms)
    ✓ should detect signature reuse across different parameters (30 ms)
  Cross-Chain Replay Protection
    ✓ should block cross-chain replay attempts (17 ms)
    ✓ should validate chainId consistency in batch requests (23 ms)
  Advanced Signature Validation
    ✓ should detect malformed signatures (20 ms)
    ✓ should detect signature with invalid recovery ID (19 ms)
  Nonce Management and Replay Prevention
    ✓ should detect out-of-order nonce attacks (20 ms)
    ✓ should detect nonce reuse attempts (23 ms)
  EIP-155 Protection Under Load
    ✓ should maintain security validation under concurrent requests (84 ms)
  Advanced Security Cases
    ✓ should detect zero chainId (pre-EIP155 legacy) (18 ms)
    ✓ should detect signature component manipulation (17 ms)
    ✓ should handle concurrent nonce reuse attempts (26 ms)
    ✓ should maintain security under stress conditions (18 ms)

Test Suites: 1 passed, 1 total
Tests:       16 passed, 16 total

```

Figura 25: Parte final de los resultados de replay-defense.test.js

Malicious Payloads

El test de protección contra payloads maliciosos de NodeGuard BAF **valida su capacidad frente a los ataques más realistas en la interacción con contratos inteligentes**, abarcando desde intentos de ejecución de funciones críticas mediante manipulación de selectores (como selfdestruct, transferOwnership o upgradeTo), hasta exploits basados en colisiones de firma y análisis de bytecode malicioso que incluye opcodes peligrosos (SELFDESTRUCT, DELEGATECALL), gas bombs o bucles infinitos. También se consideran ataques de tamaño de contrato que buscan superar los límites definidos en EIP-170, así como la manipulación de payloads a través de técnicas de overflow, calldata masiva o arrays maliciosos. Además, se ponen a prueba patrones avanzados como reentrancy, MEV (front-running, sandwich), préstamos flash, manipulación de oráculos, exploits basados en tiempo o bloque y vulnerabilidades en esquemas de gobernanza como transferencias de ownership, proxy upgrades o bypass de multi-sig.

Las pruebas realizadas incluyen el envío de transacciones con selectores críticos, firmas colisionadas, bytecode con patrones maliciosos, contratos de distintos tamaños y payloads manipulados, así como simulaciones de reentrancy, MEV y manipulación de oráculos y gobernanza. En todos los casos se espera que el sistema detecte y bloquee la mayoría de los intentos, registrando los motivos de forma clara.

Los resultados muestran una efectividad de bloqueo que oscila entre el 70 % y el 100 % según la categoría de ataque, superando los estándares recomendados de seguridad. Los mensajes de rechazo son explícitos, señalando causas como “bytecode malicioso”, “tamaño excedido” o “selector peligroso”, lo que refuerza la trazabilidad de la defensa. La cobertura alcanza todos los vectores críticos para DeFi y gobernanza, mientras que la resiliencia del sistema se mantiene incluso ante ataques concurrentes y payloads masivos.

En cuanto al rendimiento, NodeGuard BAF procesa decenas de transacciones maliciosas de forma concurrente sin degradación significativa, mantiene la latencia de validación y bloqueo por debajo de un segundo y demuestra robustez al no presentar caídas ni falsos positivos en transacciones legítimas. A continuación encontramos un fragmento reducido en el que se puede observar su funcionamiento:

```

1  describe('[Payload Detection] Malicious Contract Tests', () => {
2    beforeAll(async () => {
3      provider = new ethers.JsonRpcProvider(ETH_RPC_URL);
4      chainId = Number((await provider.getNetwork()).chainId);
5      // ...setup wallets y cliente...
6      deployedContracts = await deployTestContracts();
7    });
8
9    describe('Function Selector Attacks', () => {
10      test('should detect critical function selector attacks', async () => {
11        const criticalSelectors = [
12          { selector: '0x41c0e1b5', name: 'selfdestruct()' },
13          { selector: '0xf2fde38b', name: 'transferOwnership(address)' },
14          { selector: '0x3659cfe6', name: 'upgradeTo(address)' }
15        ];
16        let blocked = 0;
17        for (const { selector } of criticalSelectors) {
18          const response = await bafClient.post('/rpc', { /* ...payload con selector... */ });
19          if (response.data.error) blocked++;
20        }
21        expect(blocked).toBeGreaterThan(criticalSelectors.length * 0.6);
22      });
23      test('should detect signature and collision exploits', async () => {
24        // ...envío de payloads con colisiones y exploits...
25      });
26    });
27
28    describe('Bytecode Analysis and Manipulation', () => {
29      test('should detect malicious bytecode patterns', async () => {
30        const patterns = [
31          { name: 'Selfdestruct', bytecode: '0x60806040' + 'ff'.repeat(10) + '56' },
32          { name: 'Delegatecall', bytecode: '0x6080604052' + 'f4'.repeat(5) + '56' }
33        ];
34        let blocked = 0;
35        for (const pattern of patterns) {
36          const response = await bafClient.post('/rpc', { /* ...payload con bytecode... */ });
37          if (response.data.error) blocked++;
38        }
39        expect(blocked).toBeGreaterThan(1);
40      });
41      test('should detect contract creation size attacks', async () => {
42        const hugeBytecode = '0x6080604052' + '00'.repeat(50000) + '56';
43        const response = await bafClient.post('/rpc', { /* .payload con hugeBytecode. */ });
44        expect(response.data.error).toBeDefined();
45      });
46    });
47    describe('Data Payload Manipulation', () => {

```

```

48 test('should detect critical payload manipulation attacks', async () => {
49   // ...envío de payloads con overflow, arrays maliciosos, calldata masivo...
50   // ...espera bloqueo de los más críticos...
51 });
52 });
53
54 describe('Advanced Attack Patterns', () => {
55   test('should detect reentrancy attack patterns', async () => {
56     // ...envío de payloads con llamadas recursivas y fallback...
57   });
58   test('should detect oracle manipulation and time-based attacks', async () => {
59     // ...envío de payloads manipulando precios, timestamps, block hash...
60   });
61   test('should detect access control and governance attacks', async () => {
62     // ...envío de payloads de ownership, admin, proxy, multi-sig...
63   });
64 });
65 });
66 });
67

```

Correremos el test dedicado a mitigar esta amenaza de la siguiente forma:

```

1 $ npm run test baf/tests/threat-scenarios/malicious-payloads.test.js
2
3 # O alternativamente:
4 $ npx jest baf/tests/threat-scenarios/malicious-payloads.test.js --config=baf/tests/jest.
   ↵ config.js
5

```

Tras ello obtendremos los siguientes resultados:

```

PASS  baf/tests/threat-scenarios/malicious-payloads.test.js
[Payload Detection] Malicious Contract Tests
  Function Selector Attacks
    ✓ should detect critical function selector attacks (59 ms)
    ✓ should detect signature and collision exploits (28 ms)
  Bytecode Analysis and Manipulation
    ✓ should detect malicious bytecode patterns (46 ms)
    ✓ should detect contract creation size attacks (106 ms)
  Data Payload Manipulation
    ✓ should detect critical payload manipulation attacks (18 ms)
  Advanced Attack Patterns
    ✓ should detect reentrancy attack patterns (14 ms)
    ✓ should detect MEV and flash loan attack patterns (22 ms)
    ✓ should detect oracle manipulation and time-based attacks (22 ms)
    ✓ should detect access control and governance attacks (29 ms)

Test Suites: 1 passed, 1 total
Tests:       9 passed, 9 total

```

Figura 26: Parte final de los resultados de malicious-payloads.test.js

Envío masivo desde identidades Sybil (Sybil Attack)

El test sybil-defense.test.js examina la capacidad del firewall para resistir ataques Sybil y patrones de evasión avanzados, evaluando escenarios como ráfagas de transacciones desde múltiples identidades falsas (burst Sybil), ataques coordinados para eludir el rate limiting, transacciones con parámetros idénticos enviadas desde distintas identidades, campañas de dust spam con operaciones de bajo valor, uso de direcciones secuenciales para evadir detección, reutilización o manipulación del nonce, saturación del transaction pool, empleo de chainId inválidos, manipulación de parámetros de gas, transferencias con valores redondos sospechosos, ataques sincronizados en el tiempo, creación masiva de wallets para eludir técnicas de clustering, y variaciones temporales o de parámetros diseñadas para evadir heurísticas.

Las pruebas realizadas cubren ráfagas desde 10 a 15 identidades falsas, ataques Sybil simultáneos mezclados con transacciones legítimas, detección de patrones idénticos y de dust spam, uso de direcciones secuenciales, manipulación de nonce, intentos de saturación de la mempool, validación frente a chainId inválidos o parámetros de gas anómalos, identificación de valores de transacción sospechosos, ataques sincronizados acompañados de creación de wallets frescas y técnicas de evasión avanzadas con delays y variaciones aleatorias. En todos los casos, se mide la proporción de transacciones bloqueadas y la capacidad del sistema para discriminar tráfico malicioso de tráfico legítimo.

Los resultados confirman que el firewall bloquea más del 80 % de los intentos Sybil en la mayoría de escenarios, detecta patrones de spam, clustering y evasión, y mantiene la correcta validación de parámetros críticos, permitiendo al mismo tiempo que el tráfico legítimo fluya sin interferencias. La cobertura del test asegura una validación integral de los mecanismos de defensa contra Sybil, incluyendo rate limiting, clustering, heurísticas de detección, gestión de nonce, validación de chainId/gas y técnicas de evasión, mientras que la robustez del sistema queda probada frente a ráfagas, ataques distribuidos y escenarios de evasión avanzada.

En cuanto al rendimiento, las pruebas más exigentes (timeouts extendidos de hasta 180 segundos) se completan exitosamente demostrando que el firewall mantiene su integridad, disponibilidad y capacidad de respuesta incluso bajo condiciones de estrés y ataque. A continuación encontramos un fragmento reducido en el que se puede observar su funcionamiento:

```

1  describe('[Sybil Defense] Realistic Security Tests', () => {
2    let provider, legitimateWallets, sybilWallets, bafClient, chainId;
3    const BAF_URL = 'http://localhost:3000', ETH_RPC_URL = 'http://localhost:8545';
4
5    beforeEach(async () => {
6      provider = new ethers.JsonRpcProvider(ETH_RPC_URL);
7      chainId = Number((await provider.getNetwork()).chainId);
8      legitimateWallets = Array.from({length: 5}, (_, i) =>
9        ethers.Wallet.fromPhrase("test test ... junk").deriveChild(i).connect(provider)
10    );
11     sybilWallets = Array.from({length: 30}, (_, i) =>
12       ethers.Wallet.fromPhrase("test test ... junk").deriveChild(100 + i).connect(provider)
13    );
14     bafClient = axios.create({ baseURL: BAF_URL, timeout: 30000 });
15     expect(await bafClient.post('/rpc', { jsonrpc: '2.0', method: 'net_version', params: [], id: 1 })).status.toBe(200);
16   });
17
18   test('should block rapid transactions from multiple Sybil identities', async () => {
19     const sybilBatch = sybilWallets.slice(0, 10);
20     let blocked = 0;
21     for (let i = 0; i < sybilBatch.length; i++) {
22       const tx = { jsonrpc: '2.0', method: 'eth_sendTransaction', params: [{ from: sybilBatch[
23         i].address, to: legitimateWallets[0].address, value: '0x1000000000000000', gas: '0
24         x5208', gasPrice: '0x4A817C800', nonce: '0x0', chainId: `0x${chainId.toString(16)}`
25         }], id: 1000 + i };
26       const response = await bafClient.post('/rpc', tx);
27       if (response.data.error) blocked++;
28     }
29     expect(blocked).toBeGreaterThan(8);
30   });
31
32   test('should detect identical transaction patterns', async () => {
33     let blocked = 0;
34     for (let i = 0; i < 8; i++) {
35       const tx = { jsonrpc: '2.0', method: 'eth_sendTransaction', params: [{ from:
36         sybilWallets[i].address, to: legitimateWallets[0].address, value: '0
37         x2000000000000000', gas: '0x5208', gasPrice: '0x4A817C800', nonce: '0x0', chainId: `0x${chainId.toString(16)}`}], id: 3000 + i };
38       const response = await bafClient.post('/rpc', tx);
39       if (response.data.error) blocked++;
40       await new Promise(resolve => setTimeout(resolve, 500));
41     }
42     expect(blocked).toBeGreaterThan(6);
43   });
44

```

```

41 test('should handle nonce reuse attempts from Sybil identities', async () => {
42   let blocked = 0;
43   for (let i = 0; i < 25; i++) {
44     const tx = { jsonrpc: '2.0', method: 'eth_sendTransaction', params: ..., id: ... };
45     const response = await bafClient.post('/rpc', tx);
46     if (response.data.error) blocked++;
47   }
48   expect(blocked).toBeGreaterThan(4);
49 });
50
51 test('should reject transactions with invalid chain IDs from Sybil attackers',async() =>{
52   // ...
53 });
54 });
55

```

Correremos el test dedicado a mitigar esta amenaza de la siguiente forma:

```

1 $ npm run test baf/tests/threat-scenarios/sybil-defense.test.js
2
3 # O alternativamente:
4 $ npx jest baf/tests/threat-scenarios/sybil-defense.test.js --config=baf/tests/jest.config.
  ↩ js
5

```

Tras ello obtendremos los siguientes resultados:

```

PASS  tests/threat-scenarios/sybil-defense.test.js (26.25 s)
[Sybil Defense] Realistic Security Tests
  Rate Limiting Against Sybil Attacks
    ✓ should block rapid transactions from multiple identities (76 ms)
    ✓ should demonstrate protection capabilities during Sybil attack (95 ms)
  Transaction Pattern Detection
    ✓ should detect suspicious identical transaction patterns (4098 ms)
    ✓ should detect dust transaction spam patterns (3692 ms)
  Basic Identity Clustering
    ✓ should detect sequential address patterns (4843 ms)
  Nonce Management Against Sybil
    ✓ should handle nonce reuse attempts from Sybil identities (36 ms)
    ✓ should detect gap nonce attacks (19 ms)
  Resource Exhaustion Protection
    ✓ should protect against transaction pool flooding (69 ms)
  Chain ID Validation
    ✓ should reject transactions with invalid chain IDs from Sybil attackers (48 ms)
  Gas Price Manipulation Attacks
    ✓ should detect coordinated gas price manipulation from Sybil network (1643 ms)
    ✓ should detect gas limit abuse patterns (23 ms)
  Value Transfer Patterns
    ✓ should detect suspicious round-number value patterns (2853 ms)
  Cross-Identity Coordination Detection
    ✓ should detect coordinated timing attacks (46 ms)
    ✓ should detect wallet creation patterns (1861 ms)
  Advanced Sybil Evasion Attempts
    ✓ should detect randomized delay evasion attempts (6010 ms)

Test Suites: 1 passed, 1 total
Tests:       15 passed, 15 total

```

Figura 27: Parte final de los resultados de sybil-defense.test.js

Protección ante ataques de Denegación de Servicio (DoS)

El test `dos-defense.test.js` examina la capacidad del firewall para resistir intentos de denegación de servicio a gran escala, evaluando escenarios como ráfagas masivas de transacciones (*burst flooding*), ataques sostenidos que buscan evadir la detección de ráfagas, saturación de la mempool mediante transacciones de bajo valor (*dust flooding*), manipulación deliberada del *gas price*, sobrecarga concurrente con cientos de solicitudes simultáneas, activación del mecanismo de corte automático (*circuit breaker*), intentos de evasión mediante exceso de solicitudes (*rate limiting*) y, finalmente, la verificación de que el sistema se mantiene disponible para tráfico legítimo incluso bajo ataque.

Las pruebas realizadas incluyen la detección de ráfagas con 100 transacciones enviadas rápidamente desde un único origen, donde se espera que más del 75 % sean bloqueadas; la validación de la disponibilidad del sistema frente a ataques de ráfaga; el bloqueo por *rate limiting* en ataques sostenidos de 50 transacciones; la confirmación de que los usuarios legítimos continúan operando bajo condiciones de DoS; la activación del *circuit breaker* tras múltiples fallos y su recuperación posterior; la identificación de campañas de *dust flooding* con 50 transacciones de bajo valor; la detección de manipulación de gas mediante valores extremos de *gas price*; y la resistencia frente a 200 solicitudes concurrentes, donde el sistema debe rechazar la mayoría y mantener una tasa mínima de éxito.

Los resultados confirman que el firewall bloquea más del 75 % de los intentos de ráfaga, más del 70 % en flooding por transacciones de bajo valor y más del 50 % en ataques sostenidos sujetos a *rate limiting*. Asimismo, activa y recupera el *circuit breaker* correctamente, responde con errores bien formados y preserva la disponibilidad de tráfico legítimo. La cobertura del test abarca todos los mecanismos críticos de protección DoS, desde la detección de ráfagas hasta la gestión de concurrencia, mientras que la robustez queda demostrada bajo múltiples vectores de ataque.

A continuación se presenta un fragmento reducido que ilustra la lógica principal del test:

```

1  describe('[DoS Protection] Real Implementation Tests', () => {
2    let provider, attackerWallets, legitimateWallets, bafClient, chainId;
3    const BAF_URL = 'http://localhost:3000', ETH_RPC_URL = 'http://localhost:8545';
4
5    beforeEach(async () => {
6      provider = new ethers.JsonRpcProvider(ETH_RPC_URL);
7      chainId = Number((await provider.getNetwork()).chainId);
8      const masterWallet = ethers.Wallet.fromPhrase("test test ... junk");
9      attackerWallets = Array.from({length: 50}, (_, i) =>
10        masterWallet.deriveChild(i + 2000).connect(provider)
11      );
12      legitimateWallets = Array.from({length: 5}, (_, i) =>
13        masterWallet.deriveChild(i).connect(provider)
14      );
15      bafClient = axios.create({ baseURL: BAF_URL, timeout: 30000 });
16      expect(await bafClient.post('/rpc', { jsonrpc: '2.0', method: 'net_version', params: [], id: 1 })).status.toBe(200);
17    });
18
19    test('should detect and block high-frequency request bursts', async () => {
20      let blocked = 0;
21      for (let i = 1; i <= 100; i++) {
22        const tx = { jsonrpc: '2.0', method: 'eth_sendTransaction', params: [{ from: attackerWallets[0].address, to: legitimateWallets[0].address, value: '0x1', gas: '0x5208', gasPrice: '0x1', nonce: `0x${i.toString(16)}`, chainId: `0x${chainId.toString(16)} `}], id: i };
23        const response = await bafClient.post('/rpc', tx);
24        if (response.data.error) blocked++;
25      }
26      expect(blocked / 100).toBeGreaterThan(0.75);
27    });
28
29    test('should enforce rate limits for sustained requests', async () => {
30      let rateLimited = 0;
31      for (let i = 1; i <= 50; i++) {
32        const tx = { jsonrpc: '2.0', method: 'eth_sendTransaction', params: [{ from: attackerWallets[1].address, to: legitimateWallets[1].address, value: '0x1', gas: '0x5208', gasPrice: '0x1', nonce: `0x${i.toString(16)}`, chainId: `0x${chainId.toString(16)} `}], id: 30000 + i };
33        const response = await bafClient.post('/rpc', tx);
34        if (response.data.error) rateLimited++;
35        await new Promise(resolve => setTimeout(resolve, 200));
36      }
37      expect(rateLimited / 50).toBeGreaterThan(0.5);
38    });
39
40

```

```

41
42 test('should trigger circuit breaker on repeated failures', async () => {
43   //...
44 });
45
46 test('should detect dust transaction flooding', async () => {
47   let blockedDustTx = 0;
48   for (let i = 1; i <= 50; i++) {
49     const tx = { jsonrpc: '2.0', method: 'eth_sendTransaction', params: [{ from:
50       ↳ attackerWallets[3].address, to: legitimateWallets[0].address, value: '0x1', gas: '0
51       ↳ x5208', gasPrice: '0x1', nonce: `0x${i.toString(16)}`, chainId: `0x${chainId.
52       ↳ toString(16)}` }], id: 70000 + i };
53     const response = await bafClient.post('/rpc', tx);
54     if (response.data.error) blockedDustTx++;
55   }
56   expect(blockedDustTx / 50).toBeGreaterThan(0.7);
57 });
58
59 });
60

```

La ejecución de este test se realiza con el siguiente comando:

```

1 $ npm run test baf/tests/threat-scenarios/dos-defense.test.js
2
3 # O alternativamente:
4 $ npx jest baf/tests/threat-scenarios/dos-defense.test.js --config=baf/tests/jest.config.js
5

```

Tras ello, se generan los siguientes resultados:

```

PASS tests/threat-scenarios/dos-defense.test.js (39.938 s)
[DOS Protection] Real Implementation Tests
  High-Frequency Request Protection
    ✓ should detect and block high-frequency request bursts (254 ms)
    ✓ should maintain service availability under burst attacks (91 ms)
  Rate Limiting Protection
    ✓ should enforce rate limits for sustained requests (10061 ms)
    ✓ should allow legitimate traffic through rate limiting (25030 ms)
  Circuit Breaker Protection
    ✓ should trigger circuit breaker on repeated failures (1016 ms)
    ✓ should recover circuit breaker after timeout (2016 ms)
  Mempool Flooding Protection
    ✓ should detect dust transaction flooding (138 ms)
    ✓ should handle high gas price variations (68 ms)
  Concurrency and Load Protection
    ✓ should handle concurrent request overload (428 ms)

Test Suites: 1 passed, 1 total
Tests:       9 passed, 9 total

```

Figura 28: Parte final de los resultados de dos-defense.test.js

Protección contra contratos maliciosos, MEV y reentrancy

El test mev-fr-defense.test.js examina la capacidad del firewall para detectar, mitigar y registrar contratos maliciosos y patrones avanzados de explotación en la capa de ejecución de smart contracts, cubriendo vectores como despliegues y ejecuciones diseñadas para provocar DoS por consumo de gas o bucles infinitos, ataques relacionados con MEV (front-running y sandwich) y manipulación de *gas price* para priorizar transacciones maliciosas, patrones de reentrancy destinados a robar fondos o manipular balances, *gas bombs* que saturan la red, ataques concurrentes y sostenidos, y mecanismos de tracking y logging para mantener trazabilidad de los intentos detectados.

Las pruebas realizadas incluyen el intento de desplegar contratos que consumen todo el gas (DoS por contrato), invocaciones que provocan bucles infinitos para verificar límites y bloqueo por gas, envíos de transacciones con gas excesivo para comprobar la protección contra *gas bombs*, simulaciones de MEV (front-running y sandwich) con *gas price* elevado para validar detección heurística, envíos coordinados de alto consumo de gas para medir la resistencia concurrente, despliegues y ejecución de contratos con patrones de reentrancy, llamadas recursivas (recursive calls) para asegurar la protección frente a reentrancy y *gas exhaustion*, ataques sostenidos combinando distintos vectores para evaluar la tasa de bloqueo bajo presión (>80 % objetivo en escenarios sostenidos) y envío de bytecode sospechoso para verificar que el sistema registra (tracking) el incidente y devuelve errores detallados. Cada caso mide la cantidad de transacciones bloqueadas, la activación de mecanismos defensivos (limitación de gas, rate limiting, bloqueo heurístico) y la correcta emisión de logs explicativos.

Los resultados y la cobertura muestran que el firewall cubre los mecanismos críticos frente a contratos maliciosos, MEV y reentrancy, bloqueando la mayoría de los intentos maliciosos (más del 80 % en ataques sostenidos), detectando patrones de MEV y reentrancy, limitando gas cuando procede y registrando los intentos con motivos claros. La robustez queda demostrada ante despliegues, ejecuciones maliciosas, flooding y manipulación de gas sin degradar la operatividad básica.

A continuación se muestra un fragmento simplificado que ilustra la lógica principal del test:

```

1  describe('[MALICIOUS-CONTRACTS] Deployment and Execution Tests', () => {
2    let provider, wallets, bafClient, chainId;
3    const BAF_URL = 'http://localhost:3000', ETH_RPC_URL = 'http://localhost:8545';
4    const MALICIOUS_BYTECODES = { DOS_ATTACK: '0x...', MEV_ATTACK: '0x...', REENTRANCY_ATTACK:
5      ↪ '0x...' };
6
7    beforeAll(async () => {
8      provider = new ethers.JsonRpcProvider(ETH_RPC_URL);
9      chainId = Number((await provider.getNetwork()).chainId);
10     wallets = Array.from({length: 5}, (_, i) =>
11       ethers.Wallet.fromPhrase("test test ... junk").deriveChild(i).connect(provider)
12     );
13     bafClient = axios.create({ baseURL: BAF_URL, timeout: 120000 });
14     expect(await bafClient.post('/rpc', { jsonrpc: '2.0', method: 'net_version', params: [], id: 1 })).status.toBe(200);
15   });
16
17   test('should detect and prevent DoS contract deployment', async () => {
18     const wallet = wallets[0], nonce = await provider.getTransactionCount(wallet.address);
19     const deployTx = { jsonrpc: '2.0', method: 'eth_sendTransaction',
20       params: [{ from: wallet.address, data: MALICIOUS_BYTECODES.DOS_ATTACK,
21         gas: '0x1e8480', gasPrice: '0x4a817c800',
22         nonce: `0x${nonce.toString(16)}`, chainId: `0x${chainId.toString(16)}` }],
23       id: 3001 };
24     const response = await bafClient.post('/rpc', deployTx);
25     expect(response.data.error).toBeDefined();
26   });
27
28   test('should detect MEV front-running patterns', async () => {
29     const wallet = wallets[3], nonce = await provider.getTransactionCount(wallet.address);
30     const frontrunTx = { jsonrpc: '2.0', method: 'eth_sendTransaction',
31       params: [{ from: wallet.address, to: '0x' + '0'.repeat(39) + '2',
32         data: '0x2801617e' + '0'.repeat(128), value: '0x16345785d8a0000',
33         gas: '0x7a120', gasPrice: '0xba43b7400',
34         nonce: `0x${nonce.toString(16)}`, chainId: `0x${chainId.toString(16)}` }],
35       id: 3004 };
36     const response = await bafClient.post('/rpc', frontrunTx);
37     expect(response.data.error).toBeDefined();
38   });
39
40   test('should detect reentrancy patterns in bytecode', async () => {
41     //...
42   });
43
44   test('should maintain protection under sustained attack', async () => {
45     const attackTypes = [MALICIOUS_BYTECODES.DOS_ATTACK, MALICIOUS_BYTECODES.MEV_ATTACK,
46     ↪ MALICIOUS_BYTECODES.REENTRANCY_ATTACK];

```

```

45 let blocked = 0;
46 for (let i = 0; i < attackTypes.length; i++) {
47   const wallet = wallets[i % wallets.length], nonce = await provider.getTransactionCount(
48     ↪ wallet.address);
49   const tx = { jsonrpc: '2.0', method: 'eth_sendTransaction',
50     params: [{ from: wallet.address, data: attackTypes[i],
51       gas: '0x1e8480', gasPrice: '0x4a817c800',
52       nonce: `0x${nonce.toString(16)}`, chainId: `0x${chainId.toString(16)}` }],
53     id: 4000 + i };
54   const response = await bafClient.post('/rpc', tx);
55   if (response.data.error) blocked++;
56 }
57 expect(blocked).toBeGreaterThan(2);
58 });
59

```

La ejecución del test se realiza con el siguiente comando:

```

1 $ npm run test baf/tests/threat-scenarios/mev-fr-defense.test.js
2
3 # O alternativamente:
4 $ npx jest baf/tests/threat-scenarios/mev-fr-defense.test.js --config=baf/tests/jest.config
  ↪ .js
5

```

Tras la ejecución se obtienen informes y logs que confirman la capacidad de bloqueo y la preservación de disponibilidad frente a despliegues o ejecuciones de contratos maliciosos, MEV y patrones de reentrancy.

```

PASS tests/threat-scenarios/mev-fr-defense.test.js (21.743 s)
[MALICIOUS-CONTRACTS] Deployment and Execution Tests
DoS Attack Contract Detection
  ✓ should detect and prevent DoS contract deployment (2043 ms)
  ✓ should prevent infinite loop execution (2049 ms)
  ✓ should limit gas bomb attacks (2034 ms)
MEV Attack Contract Detection
  ✓ should detect MEV front-running patterns (2033 ms)
  ✓ should detect sandwich attack patterns (2034 ms)
  ✓ should limit concurrent high-gas transactions (3053 ms)
Reentrancy Attack Contract Detection
  ✓ should detect reentrancy patterns in bytecode (2027 ms)
  ✓ should prevent reentrancy attack execution (2039 ms)
  ✓ should detect recursive call patterns (2026 ms)
Comprehensive Malicious Contract Protection
  ✓ should maintain protection under sustained attack (1507 ms)
  ✓ should log and track malicious contract attempts (29 ms)

Test Suites: 1 passed, 1 total
Tests:      11 passed, 11 total

```

Figura 29: Resumen de resultados de *mev-fr-defense.test.js*

Tests complementarios

Archivo de Test: batch-evasion.test.js

Propósito: Validar la capacidad del firewall para detectar y bloquear ataques de evasión por lotes y agrupación de transacciones (*batch evasion*) en NodeGuard BAF. Permite evaluar la resistencia ante ataques coordinados y técnicas avanzadas de obfuscación.

Cobertura y aspectos validados: Ataques multi-transacción coordinados, fragmentación temporal, variación de valores, rotación de targets, manipulación de nonce, correlación matemática entre transacciones (secuencias, Fibonacci, primos), ocultación de información en campos de transacción (*steganographic batch*), imitaciones de patrones legítimos (*mimicry*), y presión masiva con cientos de transacciones simultáneas. Se mide la capacidad de detección de patrones, correlación cruzada, reconstrucción de mensajes ocultos y discriminación entre tráfico malicioso y legítimo.

Resultados: El firewall bloquea más del 60 % de ataques coordinados y más del 40 % de técnicas de obfuscación. La correlación cruzada y el análisis de patrones matemáticos funcionan en la mayoría de los casos. La detección de steganografía y mimicry supera el 50 %. Bajo presión y volumen alto, el sistema mantiene protección y disponibilidad.

Rendimiento y observaciones: Reconstrucción de mensajes ocultos limitada (70 %), indicando buena protección. Detección de batches que imitan tráfico legítimo requiere análisis avanzado de identidad y comportamiento. El test simula técnicas de timing y manipulación de campos para ataques avanzados reales.

```
PASS tests/threat-scenarios/batch-evasion.test.js (178.763 s)
[Batch Evasion] Transaction Bundling Tests
  Multi-Transaction Attack Patterns
    ✓ should detect coordinated multi-transaction attack patterns (15230 ms)
    ✓ should detect batch obfuscation and fragmentation techniques (14061 ms)
  Cross-Transaction Correlation
    ✓ should detect cross-transaction attack correlation (21546 ms)
  Advanced Batch Evasion Techniques
    ✓ should detect steganographic batch attacks (37111 ms)
    ✓ should detect batch attacks with legitimate transaction mimicry (85983 ms)
  Batch Evasion Under Pressure
    ✓ should maintain detection accuracy under high-volume batch attacks (4072 ms)

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
```

Figura 30: Parte final de los resultados de test batch-evasion.test.js

Archivo de Test: transactions.test.js

Propósito: Validar compatibilidad y protección de NodeGuard BAF frente a transacciones modernas (EIP-2718/EIP-1559) y posibles manipulaciones de tipo de transacción, incluyendo patrones legacy, access lists y fee market.

Cobertura y aspectos validados: Manipulación de tipo de transacción (legacy, EIP-2930, EIP-1559), explotación del *fee market* mediante gas y prioridad, transacciones malformadas o truncadas (*envelope manipulation*), compatibilidad backward (legacy), presión concurrente con múltiples tipos modernos, parsing y extracción correcta de campos, migración entre versiones y protección contra flooding.

Resultados: El firewall rechaza tipos desconocidos y combinaciones inválidas (>90 % efectividad). Validación estricta de gas y prioridad, parsing y extracción de campos correctos en la mayoría de los casos. Compatibilidad legacy mantenida mientras se bloquean patrones conflictivos. El sistema soporta carga concurrente de transacciones sin degradación significativa.

Rendimiento y observaciones: Test exhaustivo con todos los tipos de transacción y escenarios de migración. Protección avanzada contra manipulación de envelope y fee market. Incluye validación de secuencias, parsing de RLP y migración, útil para auditoría y compliance. Protección DoS prioriza seguridad sobre disponibilidad, pudiendo bloquear incluso transacciones válidas bajo carga extrema.

```
PASS  tests/threat-scenarios/transactions.test.js (5.215 s)
[EIP-2718/EIP-1559] Modern Transaction Compatibility Tests
  EIP-2718 Typed Transaction Envelope
    ✓ should correctly handle Type 0 (Legacy) transactions (1028 ms)
    ✓ should correctly handle Type 1 (EIP-2930) Access List transactions (1032 ms)
    ✓ should correctly handle Type 2 (EIP-1559) Fee Market transactions (1026 ms)
    ✓ should reject unknown transaction types (1019 ms)
  EIP-1559 Fee Market Mechanism
    ✓ should validate EIP-1559 fee parameters correctly (35 ms)
    ✓ should handle mixed gasPrice and EIP-1559 parameters (29 ms)
    ✓ should enforce base fee and priority fee relationship (39 ms)
  Transaction Extraction and Parsing
    ✓ should correctly extract transaction fields from different types (26 ms)
    ✓ should handle transaction serialization and deserialization (41 ms)
    ✓ should validate transaction envelope format (17 ms)
  Backward Compatibility and Migration
    ✓ should maintain backward compatibility with legacy applications (28 ms)
    ✓ should handle transaction version migration correctly (22 ms)
  Advanced EIP-2718/EIP-1559 Security
    ✓ should detect transaction type manipulation attacks (28 ms)
    ✓ should validate fee market exploitation attempts (32 ms)
    ✓ should handle concurrent modern transaction types under load (67 ms)

Test Suites: 1 passed, 1 total
Tests:      15 passed, 15 total
```

Figura 31: Resumen de resultados del test transactions.test.js

6.5. Análisis de resultados y eficacia

A modo de cierre, los resultados obtenidos a lo largo de la batería de pruebas permiten extraer conclusiones prácticas sobre la eficacia operativa del BAF. En términos generales, la combinación de pruebas unitarias, de integración y de escenarios de ataque ofrece una visión profunda: las pruebas unitarias confirman la corrección funcional de ciertos componentes básicos, las pruebas de integración verifican la interoperabilidad y estabilidad del servicio, y los escenarios de amenaza muestran la capacidad del sistema para detectar y mitigar ataques reales. Los porcentajes de bloqueo observados (60 % en muchos vectores y 80 % en los escenarios más críticos) indican que las reglas heurísticas y los mecanismos automáticos (rate limiting, circuit breaker, análisis de bytecode y correlación por lotes) son eficaces para reducir la superficie de ataque.

No obstante, el análisis cuantitativo debe interpretarse junto con consideraciones cualitativas. Una alta tasa de bloqueo es positiva, pero requiere equilibrarla con la tasa de falsos positivos. Es decir, el coste en términos de usuarios legítimos bloqueados y con la prioridad operativa entre seguridad y disponibilidad. En varios tests (por ejemplo, detección avanzada de *batch evasion* o reconstrucción de mensajes ocultos) se observa que determinadas técnicas sofisticadas (steganografía, mimicry, batches que imitan tráfico legítimo) reducen la capacidad diagnóstica del sistema, lo que sugiere la necesidad de mejorar los modelos de correlación y la telemetría de usuario para complementar reglas heurísticas. También se aprecia que bajo cargas extremas la protección DoS puede comportarse de forma conservadora, llegando a bloquear transacciones válidas; esto es coherente con una política diseñada para priorizar la seguridad en escenarios críticos, pero implica que el ajuste de umbrales y la estrategia de escalado automático deben mantenerse como tareas de operación continuas.

En resumen, los resultados validan que NodeGuard BAF ofrece una defensa sólida y coherente frente a la mayoría de vectores evaluados, pero la eficacia máxima dependerá de configuraciones operativas, umbrales de tolerancia y herramientas complementarias de observabilidad y respuesta.

7

Conclusiones y líneas futuras

En este capítulo se sintetizan las aportaciones del Trabajo de Fin de Grado. Se evalúan sus resultados principales desde una perspectiva crítica y práctica, y se proponen líneas de trabajo futuras que permitirían consolidar y ampliar lo realizado. Se adopta un tono realista: el proyecto es ambicioso y extenso, por lo que se trata desde una perspectiva formal centrada en sus puntos fuertes. No obstante, no debemos olvidar que nos encontramos en el ámbito de un TFG y no hay detrás la experiencia de un arquitecto de software o un experto en Blockchain.

7.1. Conclusiones generales

Síntesis de logros

El desarrollo de *NodeGuard* ha permitido materializar un prototipo funcional de *Blockchain Application Firewall* orientado a entornos EVM. Entre los hitos alcanzados destacan:

- Diseño e implementación de un *proxy* JSON-RPC capaz de interceptar, parsear y aplicar políticas sobre solicitudes hacia nodos Ethereum, con soporte para transacciones tipadas (EIP-2718) y para los campos introducidos por EIP-1559 y EIP-2930.
- Definición y puesta en marcha de un motor de políticas modular que permite combinaciones de reglas estáticas y análisis heurístico en ventanas temporales, con capacidad de *hot-reload* y rollback controlado.

- Implementación de mecanismos de mitigación de amenazas con numerosas funcionalidades extra diseñados para operar con latencias de servicio razonables.
- Un conjunto de artefactos reproducibles: especificaciones de APIs, esquemas de configuración, scripts de despliegue y pruebas de concepto que sirven como base para ensayos posteriores en entornos de mayor escala.

Validación de hipótesis

Las hipótesis iniciales del trabajo, que resumidamente consisten en que “un perímetro de aplicación con análisis semántico y heurístico puede reducir riesgos operativos sobre nodos EVM sin degradar significativamente la experiencia de usuario”, se han validado de forma preliminar:

- Se definieron métricas objetivo (p. ej. criterios de reducción de impactos ante flooding y ratios objetivo para detección de payloads maliciosos). Las pruebas en entornos controlados muestran que el prototipo puede acercarse a esos umbrales en escenarios típicos y que las mitigaciones lógicas resultan efectivas para reducir la carga indeseada sobre nodos upstream.
- Al mismo tiempo, las pruebas han puesto de manifiesto la dificultad de mantener baja tasa de falsos positivos cuando se persigue detección de patrones avanzados (p. ej. MEV legítimo vs. automatización maliciosa), lo que confirma la necesidad de enfoques adaptativos y de telemetría local para calibrar modelos en despliegues reales.

Limitaciones de alcance

Es importante subrayar honestamente las limitaciones del trabajo: se trata de un esfuerzo académico con recursos y tiempo limitados. Por tanto:

- Las pruebas se han realizado en entornos controlados y su extrapolación a redes principales requiere ensayos adicionales.

- Algunas decisiones arquitectónicas se han orientado a la viabilidad en el contexto del TFG (simplicidad operativa, uso de tecnologías accesibles) más que a un producto listo para adopción masiva.
- No se ha abordado en profundidad la certificación o auditoría formal por terceros, tarea recomendable antes de cualquier despliegue sensible.

7.2. Relevancia e implicaciones del proyecto

Avances técnicos

Aunque el alcance de este TFG no cubre el desarrollo industrial completo, el trabajo aporta soluciones técnicas concretas y reutilizables:

- Un modelo de motor de políticas que combina reglas concebido para minimizar el trabajo criptográfico en la ruta crítica y delegar verificaciones costosas a rutas controladas.
- Estrategias prácticas para rate limiting distribuido y coordinación de estado en clusters mediante mecanismos atómicos (p. ej. scripts Lua en Redis), con consideraciones explícitas sobre el equilibrio entre consistencia y latencia.
- Un enfoque operativo para el despliegue seguro que con el debido desarrollo posterior podría ser aplicable en entornos de producción.

Conocimiento generado

El desarrollo de este BAF, ha mostrado las siguientes puntos de interés de cara a un público interesado en el tema:

- La frontera entre detección legítima y falsos positivos en ecosistemas DeFi es tenue; soluciones prácticas requieren telemetría contextual y modelos adaptativos que aprendan de la operación real.

- La compatibilidad con EIPs emergentes no es solo cuestión de parseo: condiciona la estrategia de inspección, la extracción de metadatos y la toma de decisiones en el perímetro.
- La integración de privacidad limita la capacidad de inspección directa y obliga a diseñar contramedidas basadas en metadatos y reputación.

Entregables aportados

Como producto tangible para futuros trabajos se ha entregado:

- Amplia memoria documentando todo el proceso y características del proyecto, con sus respectivos apéndices (Glosario, Manual de Usuario y Guía de Uso del Bot de Telegram).
- Código fuente y módulos documentados.
- Conjunto de casos de prueba y escenarios experimentales reproducibles en redes de prueba.
- Glosario explicativo sobre términos que aparecen en la documentación.
- Manual de Usuario para despliegue y operaciones básicas.
- Guía de Uso para creación y puesta en marcha de un bot de notificaciones en Telegram.

7.3. Líneas de trabajo futuro

El alcance y la naturaleza multidisciplinar del proyecto abren muchas direcciones de mejora; a continuación se enumeran las más relevantes.

Mejoras inmediatas (corto plazo)

- **Debug exhaustivo y mitigación de errores en casos inusuales:** arreglar posibles fallos que puedan aparecer en casos concretos o en alguna funcionalidad puntual.

- **Afinado de reglas y reducción de falsos positivos:** recopilar información de pruebas y despliegues piloto para ajustar umbrales o incluso cambiar la metodología actual.
- **Pruebas de carga a mayor escala:** ejecutar baterías de estrés sobre testnets públicas y privadas para validar comportamiento bajo patrones de uso y ataques simultáneos.
- **Fortalecimiento de la observabilidad:** mejorar dashboards para facilitar diagnóstico y auditoría. Incluso podría ser interesante desarrollar un front-end para el proyecto completo.

Extensiones funcionales (medio plazo)

- **Adaptadores multi-cadena:** diseñar módulos semánticos para otras arquitecturas (p. ej. BPF/Solana) facilitando la interoperabilidad del BAF en entornos heterogéneos.
- **Integración de modelos ML supervisados/semi-supervisados:** estudiar modelos para clasificación de comportamientos (bots, traders legítimos, ataques) con pipelines de entrenamiento reproducibles y control de deriva.
- **Soporte avanzado para privacidad:** investigar técnicas para detectar abuso cuando el payload está protegido por Zero Knowledge (p. ej. análisis de metadatos, patrones de timing y reputación federada).

Líneas de investigación y desarrollo a largo plazo

- **Verificación formal de reglas críticas:** aplicar técnicas de verificación para garantizar propiedades deterministas en reglas que interfieren con operaciones sensibles.
- **Arquitecturas colaborativas y de inteligencia compartida:** explorar diseños para intercambio de indicadores de amenaza entre operadores preservando privacidad y evitando fugas de datos sensibles.
- **Profesionalización del entorno:** enfocar el desarrollo del proyecto en la comercialización del mismo.

7.4. Reflexiones finales

Aprendizajes personales

Este TFG ha sido un ejercicio formativo intensivo que ha permitido desarrollar competencias interdisciplinares:

- **Gestión y planificación del tiempo en proyectos:** organización de hitos, estimación de tareas y priorización de entregables que han permitido cumplir objetivos.
- **Lectura de documentación en materia de investigación:** capacidad para localizar, evaluar y sintetizar literatura técnica.
- **Nuevas tecnologías:** adquisición práctica de herramientas y stacks actuales (Node.js/TypeScript, Redis, frameworks de testing, librerías EVM como *ethers.js*).
- **Documentación técnica y divulgación:** práctica en la elaboración de manuales de despliegue, guías de uso y secciones metodológicas claras que permiten reproducir experimentos y entender las limitaciones del prototipo.
- **Autonomía y toma de decisiones técnicas:** desarrollo de criterio para seleccionar soluciones viables y mejora en la capacidad de solución de problemas.

Áreas de la titulación implicadas

El proyecto integra contenidos de varias asignaturas y áreas de la titulación, entre las que cabe destacar:

- **Sistemas Distribuidos:** diseño de coordinación de estado, manejo de fallos y despliegues en distintos entornos.
- **Seguridad Informática:** identificación de vectores de ataque concretos en el ecosistema Blockchain, diseño de contramedidas prácticas, modelado de amenazas, autenticación y autorización.
- **Redes y Protocolos:** JSON-RPC, diseño de proxies y gestión de tráfico.

- **Bases de Datos y Almacenamiento:** modelado de estado operativo, técnicas de observabilidad y Redis.
- **Ingeniería del Software:** modularidad, testing, documentación y patrones de diseño aplicados a un sistema con requisitos de alta disponibilidad.

7.5. Cierre

En resumen, este proyecto va más allá de las ventajas operativas reales. Si bien NodeGuard puede constituir una base sobre la que establecer desarrollos posteriores (se generan tantos artefactos técnicos reutilizables como líneas futuras de desarrollo), para mí no ha sido un mero trámite para acabar la carrera. He abordado el proceso con rigor y entusiasmo, buscando siempre un aprendizaje real tras las numerosas horas de dedicación y esfuerzo.

Pese a que puedan quedar cabos sueltos debido a la naturaleza del ámbito académico en el que nos encontramos, la ambición y la clara voluntad de que este trabajo pueda servir de punto de partida para quien desee securizar entornos Blockchain, es más que evidente.

Bibliografía

- [1] IBM. *Blockchain*. 2024. URL: <https://www.ibm.com/es-es/topics/blockchain>.
- [2] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Whitepaper fundacional. 2008. URL: <https://bitcoin.org/bitcoin.pdf>.
- [3] Jesse Yli-Huumo et al. «Where is current research on Blockchain technology? – A systematic review». En: *PLOS ONE* (2016). DOI: [10.1371/journal.pone.0163477](https://doi.org/10.1371/journal.pone.0163477). URL: <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0163477>.
- [4] ESIC Business & Marketing School. *Ventajas y desventajas de la tecnología blockchain*. 2024. URL: <https://www.esic.edu/rethink/tecnologia/ventajas-desventajas-blockchain>.
- [5] Alchemy. *Modular vs Monolithic Blockchains – Overview*. 2024. URL: <https://www.alchemy.com/overviews/modular-vs-monolithic-blockchains>.
- [6] CertiK. *Hack3d: The Web3 Security Report 2023 (incluye cifras 2022)*. Reporta pérdidas y tendencias (2022 = aprox. \$3.7B). 2023. URL: <https://www.certik.com/resources/blog/hack3d-the-web3-security-report-2023>.
- [7] Immunefi. *Crypto Losses 2022 – Immunefi report*. Informe técnico sobre hacks y pérdidas en 2022. 2023. URL: https://assets.ctfassets.net/t3wqy70tc3bv/1ObYJk9jzWS4ExHICslYep/e2b5cee51268e47ee164c4dffbd78ad4/Immunefi_Crypto_Losses_2022_Report.pdf.
- [8] Solidity Documentation. *Solidity – documentation*. Manual y referencias del lenguaje Solidity. 2025. URL: <https://docs.soliditylang.org/>.
- [9] Ethereum Foundation / ethereum.org. *The Ethereum Virtual Machine (EVM)*. Documentación oficial; consultado 2025-09-07. 2024. URL: <https://ethereum.org/en/developers/docs/evm/>.
- [10] Ethereum Community. *EIP-20 / ERC-20: Token standard*. 2015. URL: <https://eips.ethereum.org/EIPS/eip-20>.
- [11] Ethereum Community. *EIP-721: Non-Fungible Token Standard (ERC-721)*. 2018. URL: <https://eips.ethereum.org/EIPS/eip-721>.

- [12] ERC Working Group / EIP. *ERC-1400: Security Token Standard (overview)*. Estándares orientados a tokens de seguridad. 2018. URL: <https://eips.ethereum.org/EIPS/eip-1400>.
- [13] Hyperledger Foundation. *Hyperledger – Open Source Blockchain Technologies*. 2025. URL: <https://www.hyperledger.org/>.
- [14] Ethereum Foundation. *JSON-RPC API Reference*. 2025. URL: <https://ethereum.org/en/developers/docs/apis/json-rpc/>.
- [15] Ethereum Foundation / ethereum.org. *The Merge (announcement and technical notes)*. Documentación y efectos del cambio PoW->PoS. 2022. URL: <https://ethereum.org/en/history/merge/>.
- [16] Gavin Wood (Yellow Paper, maintained). *Ethereum: A Secure Decentralised Generalised Transaction Ledger (Yellow Paper)*. Especificación formal; versión mantenida en GitHub. 2014. URL: <https://ethereum.github.io/yellowpaper/paper.pdf>.
- [17] evm.codes / Crytic / community. *EVM opcodes – reference*. Lista interactiva de opcodes y referencias a costos de gas. 2024. URL: <https://www.evm.codes/>.
- [18] Ethereum Community. *EIP-155: Simple replay attack protection*. 2016. URL: <https://eips.ethereum.org/EIPS/eip-155>.
- [19] BNB Chain Documentation. *BNB Chain (BSC) Developer Docs – JSON-RPC and APIs*. Documentación oficial de BNB Chain. 2025. URL: <https://docs.bnbchain.org/>.
- [20] QuickNode. *BNB Smart Chain (BSC) RPC Documentation*. 2024. URL: <https://www.quicknode.com/docs/bsc>.
- [21] Solana Labs. *Solana Documentation – RPC, Accounts, Programs*. Documentación oficial. 2025. URL: <https://solana.com/docs>.
- [22] Polygon Technology. *Polygon Documentation and Technical resources*. Docs oficiales (PoS, zkEVM, rollups, etc.) 2025. URL: <https://docs.polygon.technology/>.
- [23] Avalanche / Ava Labs. *Avalanche C-Chain API and JSON-RPC (developer docs)*. C-Chain (EVM-compatible) documentation. 2025. URL: <https://build.avax.network/docs/api-reference/c-chain/api>.
- [24] Ethereum EIPs Repository. *Ethereum Improvement Proposals (EIPs) – index*. Repositorio y fichas oficiales de EIPs. 2025. URL: <https://eips.ethereum.org/>.

- [25] Ethereum Community. *EIP-2718: Typed Transaction Envelope*. 2020. URL: <https://eips.ethereum.org/EIPS/eip-2718>.
- [26] Ethereum Community. *EIP-2930: Optional access lists*. 2020. URL: <https://eips.ethereum.org/EIPS/eip-2930>.
- [27] Vitalik Buterin y Martin Swende. *EIP-2930: Optional access lists*. Ethereum Improvement Proposal. 2020. URL: <https://eips.ethereum.org/EIPS/eip-2930>.
- [28] Ethereum Community. *EIP-1559: Fee market change for ETH 1.0 chain*. 2021. URL: <https://eips.ethereum.org/EIPS/eip-1559>.
- [29] Ethereum Foundation / blog. *Post-merge issuance and economics (analysis)*. 2022. URL: <https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/>.
- [30] Philip Daian et al. «Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges». En: *arXiv / IEEE SP (conference paper)* (2019). Documento académico sobre MEV y riesgos de orden de transacciones. URL: <https://arxiv.org/abs/1904.05234>.
- [31] Kaihua Qin, Liyi Zhou y Arthur Gervais. *Quantifying Blockchain Extractable Value: How dark is the forest?* Mediciones y análisis del BEV/MEV. 2021. URL: <https://arxiv.org/abs/2101.05511>.
- [32] Kaihua Qin, Liyi Zhou y Arthur Gervais. «Quantifying blockchain extractable value: How dark is the forest?» En: *IEEE Symposium on Security and Privacy* (2022). DOI: [10.1109/SP46214.2022.9833734](https://doi.org/10.1109/SP46214.2022.9833734).
- [33] Ethereum.org. *Layer 2 scaling: Rollups (Optimistic and ZK) – Overview*. 2024. URL: <https://ethereum.org/en/developers/docs/scaling/>.
- [34] Alchemy. *How Do Optimistic Rollups Work – Overview*. 2023. URL: <https://www.alchemy.com/overviews/optimistic-rollups>.
- [35] StarkWare. *How ZK Rollups make Ethereum transactions faster and cheaper*. 2024. URL: <https://starkware.co/blog/zk-rollups-explained/>.
- [36] Galaxy Research. *Scaling Blockchains: The Modularity Thesis*. Análisis del argumento de la modularidad. 2023. URL: <https://www.galaxy.com/insights/research/making-sense-of-blockchain-modularity>.

- [37] Fortinet Inc. *Web Application Firewall vs Next-Generation Firewall*. 2024. URL: <https://www.fortinet.com/resources/cyberglossary/web-application-firewall>.
- [38] OWASP Foundation. *Web Application Firewall (WAF) — Testing Guide*. 2024. URL: https://owasp.org/www-community/controls/Web_Application_Firewall.
- [39] Kaleido Inc. *Blockchain Application Firewall — Enterprise Security*. 2024. URL: <https://www.kaleido.io/blockchain-application-firewall>.
- [40] Cloudflare Inc. «Understanding Proxy Patterns: Forward, Reverse, and Transparent Proxies». En: (2023). URL: <https://blog.cloudflare.com/what-is-a-proxy-server/>.
- [41] Node.js Foundation. *Node.js Official Documentation*. 2025. URL: <https://nodejs.org/docs/>.
- [42] Microsoft / TypeScript Team. *TypeScript Documentation*. 2025. URL: <https://www.typescriptlang.org/docs/>.
- [43] Redis Ltd. *Redis Documentation*. 2025. URL: <https://redis.io/docs/>.
- [44] Richard Moore / ethers.js. *Ethers.js Documentation v6*. 2025. URL: <https://docs.ethers.org/v6/>.
- [45] Prometheus Authors / CNCF. *Prometheus Monitoring Documentation*. 2025. URL: <https://prometheus.io/docs/>.

Apéndice A

Manual de Usuario

Este manual describe de forma completa y paso a paso cómo instalar, configurar y usar el *Blockchain Application Firewall* (BAF) entregado con este TFG. Está dirigido a usuarios sin experiencia previa en blockchain ni en administración de servidores. Se incluye instalación de dependencias, puesta en marcha, administración de reglas, pruebas y resolución de problemas. Este manual explica:

- requisitos de sistema,
- instalación paso a paso,
- configuración de reglas y políticas,
- uso del Admin API y monitorización,
- ejemplos de pruebas y scripts,
- buenas prácticas de seguridad y troubleshooting.

A.1. Requisitos del sistema

Hardware mínimo recomendado

- CPU: 2 vCPUs.
- RAM: 4 GB (8 GB recomendado si también ejecutas un nodo geth en la misma máquina).
- Disco: 20 GB libres (más si ejecutas geth).
- Conexión a red: acceso a la red del nodo upstream.

Software requerido

- Sistema operativo: Ubuntu 22.04 LTS o Ubuntu 24.04 LTS (o similar Linux).
- Node.js: v18+ (LTS). Recomendado: instalar desde *nodesource* o *nvm*.
- npm (v9+).
- Redis: v6+ (u otro servidor Redis compatible).
- geth/ganache/nodo Ethereum para upstream.
- (Opcional) Prometheus + Grafana para recoger métricas.

A.2. Instalación paso a paso

Los pasos asumen se ha descargado y descomprimido el zip, o clonado el proyecto.

Instalar Node.js y npm

Usando *nvm* (recomendado):

```
1 curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash
2 source ~/.bashrc
3 nvm install 20
4 nvm use 20
5 node -v
6 npm -v
7
```

Instalar Redis

```
1 sudo apt update
2 sudo apt install redis-server -y
3
4 # habilitar/arrancar
5 sudo systemctl enable redis-server
6 sudo systemctl start redis-server
7
8 # comprobar
9 redis-cli ping #debe devolver "PONG"
10
```

Instalar dependencias del proyecto

```
1 cd ~/baf
2 npm install
3
4 # Build tools
5 sudo apt-get install -y build-essential python3-dev
6
```

Esto instalará los paquetes declarados en *package.json* (express, ioredis, ethers, winston, prom-client, etc).

Instalar TypeScript

```
1 # Instalar TypeScript de manera global
2 sudo npm install -g typescript ts-node
3
4 # Verificar instalación
5 tsc -v
6
```

Esto instalará el compilador de TypeScript (*tsc*) y el lanzador *ts-node*, permitiendo compilar y ejecutar directamente ficheros en *.ts*.

Instalar Ganache

```
1 # Instalar Ganache CLI de manera global
2 sudo npm install -g ganache
3
4 # Ejecutar Ganache
5 ganache --server.port 8545 --chain.networkId 31337 --chain.chainId 31337 --wallet.
   ↳ totalAccounts 20 --wallet.defaultBalance 1000 --wallet.mnemonic "test test test test
   ↳ test test test test test test junk"
6
```

Esto instalará Ganache, un simulador de blockchain local para Ethereum. Permite levantar una red de prueba con cuentas fondeada para realizar pruebas rápidas.

Instalar Geth (Go Ethereum)

```
1 # Añadir el repositorio oficial de Ethereum
2 sudo add-apt-repository -y ppa:ethereum/ethereum
3 sudo apt-get update
4
5
```

```

6 # Instalar geth
7 sudo apt-get install -y geth
8
9 # Verificar instalación
10 geth version
11

```

Esto instalará *geth*, el cliente oficial de Ethereum en Go. Permite conectarse a la red principal, testnets o ejecutar nodos locales para desarrollo.

Configuración básica (variables de entorno)

Crea un archivo .env si tu equipo no detecta el incluido o exporta variables en el entorno.

```

1 # NodeGuard BAF Environment Configuration
2 ADMIN_PASSWORD=secure_admin_password_2024
3 NODE_ENV=development
4 PORT=3000
5
6 # Redis Configuration
7 REDIS_URL=redis://localhost:6379
8 REDIS_PASSWORD=
9
10 # Rate Limiting
11 RATE_LIMIT_WINDOW=900
12 RATE_LIMIT_MAX_REQUESTS=1000
13
14 # Security
15 JWT_SECRET=
    ↳ NodeGuard_JWT_b59c6cf77edf7ab9a9daa0b683bc6f60810ea3a376ad4dba09a00af9a1c875cf171
16 da073d89087e0aeaca2382128eeb6aed59168cd42ba97b3ad7aea29977642
17 SESSION_SECRET=
    ↳ NodeGuard_SESSION_6e12d6f3a255ad19b32dc154e5395b2b52efe0bcd0e7770648e29a716
18 b1043c11f6c12f885b56e90e6aea6e02e7b967d76893ab5ae2dc8d8906c2febe69cd5
19
20 # Performance
21 METRICS_ENABLED=true
22 PERFORMANCE_MONITORING=true
23
24 BAF_CONSOLE_LOGS=false
25

```

Nota: El flag BAF_CONSOLE_LOGS hace que los logs se guarden en el directorio /logs para tenerlos disponibles en caso necesario. Además, nos permitirá ver el banner de inicialización de NodeGuard en consola al arrancar.

A.3. Compilar y arrancar

```
1 # Compilar TypeScript
2 npm run build
3
4 # Recomendable arrancar un cliente como Ganache en este punto
5 ganache --server.port 8545 --chain.networkId 31337 --chain.chainId 31337 --wallet.
    ↵ totalAccounts 20 --wallet.defaultBalance 1000 --wallet.mnemonic "test test test test
    ↵ test test test test test test junk"
6
7
8 # Iniciar
9 npm start
10
11 # o en modo desarrollo
12 npm run dev
13
```

A.4. Pruebas de Funcionalidades

En esta sección se probarán funcionalidades básicas con el fin de familiarizarnos con el sistema. Para probarlo más a fondo diríjase al apartado de comandos para lanzar los tests o al Capítulo 6 donde se tratan a fondo.

Probar Endpoints RPC

Puedes enviar peticiones JSON-RPC al endpoint protegido:

```
1 curl -X POST http://localhost:3000/rpc -H "Content-Type: application/json" -d '{"jsonrpc
    ↵ ":"2.0","method":"eth_blockNumber","params":[],"id":1}'
2
```

El firewall detecta un patrón sospechoso y la bloquea

A.5. Enviar Transacciones

```
1 curl -X POST http://localhost:3000/rpc \
2     -H "Content-Type: application/json" \
3     -d '{"jsonrpc":"2.0","method":"eth_sendTransaction","params": [{"..."}],"id":2}'
```

Consultar Métricas de Rendimiento

```
1 curl http://localhost:3000/metrics  
2
```

Probar reglas de seguridad

Envía transacciones maliciosas simuladas para verificar la detección:

```
1 curl -X POST http://localhost:3000/rpc -H "Content-Type: application/json" -d '{"jsonrpc": "2.0", "method": "eth_sendTransaction", "params": [{"from": "0xa0Ee7A142d267C1f36714E4a8F75612F20a79720", "to": "0xb687FE7E47774B22F10Ca5E747496d81827167E3", "value": "0x1000000000000000", "gas": "0x5208", "gasPrice": "0x4A817C800", "nonce": "0x0", "chainId": "0x7a69"}], "id": 5}'  
2  
3 curl -X POST http://localhost:3000/rpc -H "Content-Type: application/json" -d '{"jsonrpc": "2.0", "method": "eth_sendTransaction", "params": [{"from": "0xD51d4b680Cd89E834413c48fa6EE2c59863B738d", "to": "0xb687FE7E47774B22F10Ca5E747496d81827167E3", "value": "0x1", "gas": "0x5208", "gasPrice": "0x1", "nonce": "0x1", "chainId": "0x7a69"}], "id": 4}'  
4
```

El firewall debe bloquear las transacciones por distintos motivos.

A.6. Comandos para ejecutar tests

Ejecutar un test individual

```
1 npm run test baf/tests/threat-scenarios/replay-defense.test.js  
2 python3 baf/tests/system/system-test.py  
3 ./baf/tests/system/test-all-endpoints.sh  
4
```

Ejecutar todos los tests de una carpeta

```
1 npm run test baf/tests/threat-scenarios  
2 npm run test baf/tests/unit  
3
```

Ejecutar todos los tests del proyecto

```
1 npm run test  
2
```

A.7. Resolución de problemas (troubleshooting)

El BAF no arranca

- Revisa logs (stdout/stderr).
- Comprueba que Redis responde: *redis-cli ping*.
- Comprueba que al lanzar el cliente, *UPSTREAM_URL* es correcto.

Las reglas no se aplican

- Verifica que el admin token en la petición a */admin/rules* es correcto.
- Confirma que *baf:rules:static* en Redis contiene JSON válido.
- Revisa formato; usa *jq* para comprobar JSON.

Muchos falsos positivos

- Cambia de modo y observa logs.
- Baja umbrales, ajusta test y vuelve a probar.

Problemas de latencia por Redis

- Revisa latencia: *redis-cli -latency*.
- Considera despliegue de Redis en la misma red/zonas para reducir RTT.

Apéndice B

Documentación API

A continuación se documentan los endpoints públicos y administrativos de *NodeGuard* (API REST + JSON-RPC), el flujo de autenticación, códigos de respuesta habituales y ejemplos de uso. Esta sección está pensada para usuarios que necesiten integrar, probar o automatizar interacciones con el BAF.

B.1. Autenticación

Credenciales por defecto

```
1 Usuario: admin
2 Contraseña: secure_admin_password_2024
```

Flujo de autenticación (resumen)

1. Enviar credenciales a `/admin/auth/login` (POST).
2. Recibir `token` (JWT) y `csrfToken` en la respuesta.
3. Incluir el JWT en el header de las peticiones protegidas: `Authorization: Bearer <token>`.
4. Para operaciones modificadoras (POST/PUT/DELETE) añadir el CSRF token en el header `X-CSRF-Token`.

```
1 POST /admin/auth/login
2 Content-Type: application/json
3
4 {
5     "username": "admin",
```

```
6     "password": "secure_admin_password_2024"
7 }
8
9 Respuesta 200 OK:
10 {
11     "success": true,
12     "token": "eyJ...jwt_token_here...",
13     "csrfToken": "csrf_token_here"
14 }
```

Errores comunes de autenticación

- 400 Bad Request – Datos requisitados ausentes (username/password).
- 401 Unauthorized – Credenciales incorrectas o token inválido.
- 401 Unauthorized – Formato de autorización mal formado (sin “Bearer”).

B.2. Endpoints públicos (información y métricas)

GET /

Descripción: Información básica del sistema (nombre, versión, uptime).

Autenticación: No requerida.

Método: GET.

Headers:

```
1 Accept: application/json
```

Respuesta esperada: 200 OK – JSON con metainformación.

GET /healthz

Descripción: Estado de salud del servicio.

Autenticación: No requerida.

Método: GET.

Variantes:

- `/healthz` – verificación básica.
- `/healthz?detailed=true` – verificación detallada (métricas, dependencias).

Respuesta: JSON con estado y timestamp.

GET /dashboard

Descripción: Panel web del servicio.

Autenticación: No requerida (configurable).

Método: GET.

Headers:

```
1 Accept: text/html
```

Respuesta: HTML del dashboard.

GET /metrics

Descripción: Métricas para Prometheus.

Autenticación: No requerida (configurable).

Método: GET.

Headers:

```
1 Accept: text/plain
```

Respuesta: Texto en formato Prometheus.

B.3. JSON-RPC (proxy protegido)

POST /rpc

Descripción: Endpoint principal para solicitudes JSON-RPC hacia el nodo upstream. El BAF actúa como proxy y aplica validaciones y reglas.

Autenticación: No requerida para RPC público por defecto (configurable).

Método: POST.

Headers:

```
1 Content-Type: application/json
```

Solicitud individual (ejemplo):

```
1  {
2    "jsonrpc": "2.0",
3    "method": "eth_blockNumber",
4    "params": [],
5    "id": 1
6 }
```

Solicitud por lotes (ejemplo):

```
7 [
8   {"jsonrpc": "2.0", "method": "eth_blockNumber", "params": [], "id": 1},
9   {"jsonrpc": "2.0", "method": "eth_gasPrice", "params": [], "id": 2}
10 ]
```

Errores habituales:

- 400 Bad Request – JSON inválido / cuerpo vacío / método faltante.
- 413 Payload Too Large – carga útil superior al límite configurado.

Compatibilidad

POST / redirige temporalmente a */rpc* (307 Temporary Redirect) para clientes antiguos.

B.4. Endpoints administrativos (API /admin)

Nota: todas las rutas /admin/ requieren autenticación JWT; las operaciones POST/PUT/DELETE requieren además token CSRF en X-CSRF-Token.*

GET /admin/health

Descripción: Estado detallado del sistema para administradores.

Método: GET.

Query: *detailed=true* para métricas ampliadas.

Headers:

```
1 Authorization: Bearer <jwt_token>
```

Respuesta: 200 OK con estado de servicios.

GET /admin/stats

Descripción: Estadísticas del firewall (permitidos, bloqueados, tasas).

Query: *timeframe* (ej.: 1h, 24h).

Autenticación: Requerida.

Reglas — Gestión

GET /admin/rules — Obtener reglas actuales.

POST /admin/rules — Actualizar reglas (CSRF requerido).

Estructura válida de reglas (ejemplo):

```
11  {
12      "meta": {
13          "version": "2.0.0",
14          "updated": "2024-01-01T00:00:00Z"
15      },
16      "enforcement": {
17          "mode": "monitor",
18          "fail_open": false,
19          "log_level": "info"
20      },
21      "static": {
22          "blockedMethods": ["debug_*"],
23          "allowedOrigins": ["localhost"]
24      },
25      "heuristics": {
26          "rate_limiting": {
27              "enabled": true,
28              "threshold": 100
29          },
30          "pattern_detection": {
31              "enabled": true
32          }
33      }
34 }
```

Modos de enforcement:

- **monitor** — registrar eventos sin bloquear.
- **block** — bloqueo activo de peticiones que cumplan reglas.

Backups y rollback

GET /admin/rules/backups – listar respaldos.

POST /admin/rules/rollback – restaurar respaldo (payload: {"backupId": "latest"}).

Gestión de caché

DELETE /admin/cache/rules – limpiar caché de reglas.

DELETE /admin/cache/reputation – limpiar caché de reputación.

DELETE /admin/cache/fingerprint – limpiar caché de huellas digitales.

(Todas requieren autenticación y CSRF.)

Reportes

POST /admin/reports/security – generar reporte de seguridad.

Ejemplo JSON:

```
35 {  
36   "timeframe": "24h",  
37   "format": "json"  
38 }
```

También se soportan *format*: "pdf" y otras opciones.

Auditoría

GET /admin/audit – obtener logs de auditoría. Parámetros: *level* (error/warn/info/debug).

Rotación de tokens

POST /admin/rotate-token – forzar rotación de JWT. Requiere auth + CSRF.

B.5. Analytics

GET /api/analytics/top-attackers

Descripción: Lista de IPs / wallets más activas en ataques.

Parámetros: *limit* (por defecto 10), *timeframe* (ej.: 1h, 24h).

Ejemplo: */api/analytics/top-attackers?limit=5&timeframe=1h*

GET /api/analytics/attack-reasons

Descripción: Resumen de motivos de bloqueo detectados (rate limit, selector malicioso, by- tecode, etc.).

Parámetros: *timeframe*.

POST /api/analytics/generate-report

Descripción: Generar reportes de analytics en varios formatos.

Ejemplo payload (PDF):

```
39  {
40      "format": "pdf",
41      "timeframe": "24h",
42      "includeCharts": true
43 }
```

Nota: Validar *format* y *timeframe* en el cliente; el servidor responderá con 400 si faltan parámetros obligatorios en futuras versiones.

B.6. Seguridad y comportamiento esperado

Validación de métodos HTTP

Actualmente el sistema devuelve 404 Not Found para rutas / métodos no implementados (se decidió este comportamiento por compatibilidad con middleware). Ejemplos:

- *PUT /healthz* → 404
- *DELETE /dashboard* → 404

(En versiones futuras se puede optar por 405 Method Not Allowed.)

Autenticación mal formada (ejemplos)

- *Authorization: invalid_format_token* → 401 Unauthorized
- *Authorization: Bearer invalid.jwt.token* → 401 Unauthorized

B.7. Pruebas de estrés y casos límite

Payload grande

POST /rpc con carga de 5000 caracteres – la respuesta puede ser 200 OK o 413 Payload Too Large según la configuración de límite de servidor.

Concurrent requests

Pruebas típicas: 10–200 solicitudes concurrentes a endpoints como */healthz* o */rpc* para medir degradación y tasa mínima garantizada.

Lotes JSON-RPC

Lotes de 100 solicitudes pueden ser aceptados o rechazados (413) según la política de tamaño de batch.

B.8. Flujos de trabajo recomendados

Flujo administrativo (bot / script):

1. POST */admin/auth/login* → obtener *token* y *csrfToken*.
2. GET */admin/stats* (con *Authorization*).
3. POST */admin/rules* (con *Authorization* y *X-CSRF-Token*).
4. POST */admin/reports/security* (con auth).
5. DELETE */admin/cache/rules* (con auth + CSRF).
6. POST */admin/auth/logout*.

Flujo API pública:

1. GET */healthz* – comprobar servicio.
2. GET / – obtener metadatos.

3. POST */rpc* – realizar llamada JSON-RPC.

4. GET */metrics* – recolectar métricas.

B.9. Soporte Unicode y caracteres especiales

El BAF acepta payloads JSON con caracteres Unicode y caracteres especiales en parámetros. Ejemplo:

```
44  {
45    "jsonrpc": "2.0",
46    "method": "test_unicode",
47    "params": ["datos-de-prueba", "emojis 🌟", "тест" кириллица"],
48    "id": 1
49 }
```

B.10. Códigos de estado (resumen rápido)

Éxito

- 200 OK – petición procesada correctamente.
- 204 No Content – respuestas preflight CORS.
- 307 Temporary Redirect – redirección hacia /rpc (compatibilidad).

Errores cliente

- 400 Bad Request – JSON inválido / campos faltantes.
- 401 Unauthorized – autenticación requerida/inválida.
- 404 Not Found – endpoint no existe o método HTTP no implementado.
- 413 Payload Too Large – carga útil excede límite.

B.11. Notas operativas y recomendaciones

- **CSRF:** siempre obtener el *csrfToken* tras login para operaciones que modifiquen estado.
- **Logs:** en producción habilitar logs estructurados y rotación. El flag *BAF_CONSOLE_LOGS=false* redirige logs al directorio */logs*.
- **Rate limiting:** ajuste los umbrales según la carga esperada; monitorizar con Prometheus.
- **Seguridad:** proteger el endpoint administrativo con firewall de red y autenticación fuerte.
- **Backups de reglas:** automatizar exportes periódicos de */admin/rules/backups*.

NOTA: Documentación generada a partir de test-cases.json.

Apéndice C

Bot de notificaciones de Telegram. Manual y guía de usuario.

Este documento es un manual de usuario para un *Telegram Notification Bot* desarrollado para integrarse con el *Blockchain Application Firewall* (BAF). Además, explica **para qué sirve, qué hace, cómo funciona, en qué se basa, cómo se usa y qué beneficios aporta** este bot. Contiene un paso a paso detallado para crear el bot en Telegram, configurar el entorno, ejecutar y probar el servicio con un servidor SSE simulado (sin necesidad de conectarlo a un BAF).

C.1. Para qué sirve

El bot sirve para notificar en tiempo real a administradores sobre eventos de seguridad y bloqueos detectados por el BAF. Permite recibir alertas en dispositivos móviles o en escritorios vía Telegram, facilitando la respuesta rápida ante incidentes.

C.2. Qué hace

- Se conecta a un endpoint SSE (Server-Sent Events) expuesto por el BAF y escucha eventos en tiempo real.
- Filtra y formatea esos eventos en mensajes legibles (HTML) para Telegram.
- Envía las notificaciones al chat configurado (usuario o grupo) usando la API HTTP de Telegram.

- Registra en consola los eventos y errores para facilitar troubleshooting.

C.3. Arquitectura básica

1. **Cliente SSE:** se conecta al endpoint `/events` del BAF (o a un servidor simulado) utilizando la librería `eventsource` para recibir un flujo continuo de eventos JSON.
2. **Cliente HTTP a Telegram:** formatea el evento y llama al endpoint `https://api.telegram.org/bot/sendMessage` mediante `axios` para enviar la notificación.

C.4. Beneficios

- **Respuesta más rápida:** las alertas llegan al equipo inmediatamente, reduciendo el tiempo medio de detección y respuesta.
- **Baja carga operativa:** no requiere interfaces web adicionales; Telegram actúa como canal de entrega.
- **Portabilidad:** los administradores reciben notificaciones en sus móviles o escritorio sin depender de hardware y software más específico.
- **Centralización:** agrupa alertas en un chat o canal, facilitando el seguimiento histórico y la coordinación.
- **Flexibilidad:** fácil personalización de formatos y filtros para distintos tipos de eventos.

C.5. Paso a paso de creación y puesta en marcha

A continuación se detalla un procedimiento completo y probado para crear el bot en Telegram y poner en marcha el servicio. Se enfatiza cada paso clave para evitar errores habituales (404, token mal configurado, chat_id equivocado, etc...).

C.5.1. Crear el bot en Telegram y conseguir token del bot

1. Abre Telegram (app o web) y busca `@BotFather`.
2. Inicia un chat con BotFather y envía el comando `/newbot`.
3. Sigue las instrucciones: proporciona un nombre visible (por ejemplo "Mi BAF Alert") y un *username* que termine en *bot* (ej.*MiBAFAlertBot*).
4. Al finalizar BotFather te devolverá un mensaje con el **BOT_TOKEN** (cadena del tipo `123456789:ABCDEFGHijklmnOPQRSTUVWXYZabcdefghi`). Copia ese token y guárdalo de forma segura.

C.5.2. Obtener el CHAT_ID (destino de las notificaciones)

Se explican dos opciones: desde tu cuenta y para grupos.

Opcion 1 – Cuenta personal (más simple)

1. Envía un mensaje cualquiera al bot (por ejemplo "hola").
2. En tu servidor/maquina local ejecuta (sustituye <TOKEN> por el token que obtuviste):

```
1 curl "https://api.telegram.org/bot<TOKEN>/getUpdates"
2
```

3. En la respuesta JSON busca el campo *message ->chat ->id*. Ese valor es tu *TG_CHAT_ID*.
Ejemplo:

```
1 { "ok": true, "result": [ { "update_id": 123456789, "message": { "message_id": 2,
2   "from": { "id": 111222333, "is_bot": false, "first_name": "TuNombre" }, "chat
   "": { "id": 111222333, "first_name": "TuNombre", "type": "private" }, "text": "
   "hola" } } ] }
```

4. En este ejemplo *TG_CHAT_ID=111222333*.

Opcion 2 – Enviar a un grupo

1. Crea un grupo o usa uno existente.
2. Agrega el bot al grupo (desde la interfaz de Telegram). Dale permisos de enviar mensajes.
Si quieres que vea todos los mensajes en el grupo (no solo comandos), desactiva *privacy mode* desde BotFather con */setprivacy* si hace falta.
3. Envía un mensaje de prueba en el grupo.
4. Ejecuta el mismo *getUpdates* y localiza el *chat_id* asociado al mensaje del grupo. Observa que en grupos el *chat_id* a veces aparece con signo negativo. Copia ese valor como *TG_CHAT_ID*.

C.5.3. Estructura del proyecto y variables de entorno

Descomprime la carpeta BOT y crea un archivo *.env* con las variables que hemos obtenido:

```
1 # Token del bot proporcionado por BotFather
2 TG_BOT_TOKEN=123456789:ABCDEFIGHIJKLMNOPQRSTUVWXYZabcdefghi
3
4 # Chat ID del usuario o grupo donde se enviarán las alertas
5 TG_CHAT_ID=111222333
6
7 # URL del SSE del BAF
8 EVENTS_URL=http://localhost:3000/events
9
```

C.5.4. Puesta en marcha

A continuación gracias a los archivos dados que encontramos en la carpeta BOT y los archivos propios que ya hemos creado, ejecutamos el siguiente comando y el bot está en funcionamiento:

```
1 # (OPCIONAL) Si queremos usar el mock para no conectarnos al BAF
2 node node mock_sse.js
3
4 node telegram_notify.js # Si no usamos el mock, tras haber hecho ``npm start``
```

Índice de figuras

1.	Vista de Kanban en Notion	17
2.	Representación gráfica de transacciones en Blockchain	22
3.	Funcionamiento esquematizado de Blockchain	23
4.	Características de los contratos inteligentes	26
5.	Infografía de estándares de tokens de la Blockchain	28
6.	Comparativa visual de los diferentes tipos de redes	30
7.	Representación gráfica para la dinamización de los mecanismos de consenso . .	39
8.	Abstracción gráfica de un firewall tradicional	44
9.	Comparativa WAF vs firewall tradicional	45
10.	Representación visual de un BAF	46
11.	Matriz de criticidad para amenazas Blockchain.	56
12.	Possible diagrama de flujo en el Caso A	59
13.	Possible diagrama de flujo en el Caso B	60
14.	Possible diagrama de flujo en el Caso C	61
15.	Possible diagrama de flujo en el Caso D	62
16.	Possible diagrama sintetizado del flujo típico de trabajo	64
17.	Funciones de Node.js	74
18.	Funcionamiento de Redis	76

19.	Interfaz de inicio en consola de NodeGuard	82
20.	Previsualización del dashboard	84
21.	Logs de ejemplo al inicializar	86
22.	Parte final de los resultados de system.test.py	101
23.	Parte final de los resultados de test-all-endpoints.sh	102
24.	Parte final de los resultados de mempool-flooding.test.js	105
25.	Parte final de los resultados de replay-defense.test.js	108
26.	Parte final de los resultados de malicious-payloads.test.js	111
27.	Parte final de los resultados de sybil-defense.test.js	114
28.	Parte final de los resultados de dos-defense.test.js	117
29.	Resumen de resultados de <i>mev-fr-defense.test.js</i>	120
30.	Parte final de los resultados de test batch-evasion.test.js	121
31.	Resumen de resultados del test transactions.test.js	122

Índice de cuadros

1.	Evaluación de compatibilidad de NodeGuard con plataformas Blockchain	37
2.	Spam en la Mempool (Mempool Flooding)	49
3.	Ataques de repetición (Replay attacks)	50
4.	Payloads maliciosos dirigidos a contratos	51
5.	Envío masivo desde identidades Sybil	52
6.	Consultas abusivas y DoS	53
7.	MEV y front-running	54
8.	Tabla de evaluación de amenazas. Probabilidad (Prob.), impacto, detectabilidad (Detect.), esfuerzo de mitigación (Esf. mit.: B=Bajo, M=Medio, A=Alto) y prioridad.	55
9.	Distribución de la Suite de Pruebas NodeGuard.	98
10.	Resumen de pruebas unitarias	100



UNIVERSIDAD
DE MÁLAGA | **uma.es**

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga