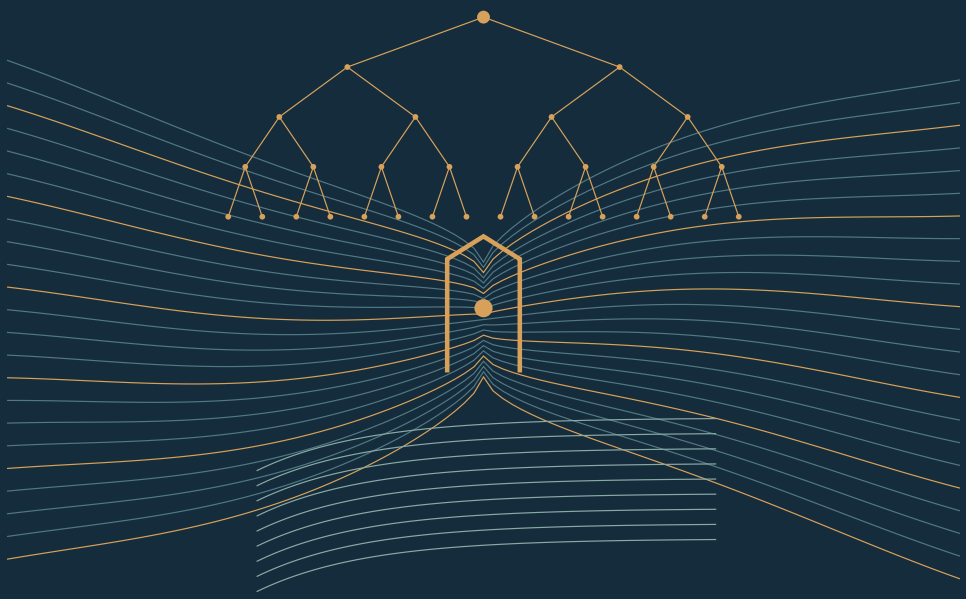
EL LIBRO DE

SIBUNA

Las matemáticas y la ingeniería de la defensa web



TRABAJO · CONFIANZA · ESTADO ACOTADO

Vikrant Rathore, con la colaboración de Ronak Rathore y Kanak Rathore

Versión 1.0.0-beta.3 · nov 13, 2026

Índice

Prefacio . . . 6

Cómo leer este libro . . . 8

- Tres itinerarios de lectura . . . 8
- Convenciones . . . 9

Parte I: El coste de una petición . . . 10

- La admisión es una decisión económica . . . 10
- Qué demuestra una prueba de trabajo . . . 11
- La búsqueda de hashes como variable aleatoria . . . 11
- Las sesiones amortizan el trabajo . . . 12
- El estado acotado es un segundo presupuesto . . . 12
- Los invariantes que mantendremos . . . 13

Parte II: Trabajos previos y el espacio de diseño . . . 14

- De dónde viene Sibuna . . . 14
- Tres ideas en un solo binario . . . 16
- Comparación de características . . . 17

Parte III: Fundamentos de la investigación criptográfica . . . 23

- Hashcash: las matemáticas del primer nivel . . . 23
- Prueba de trabajo secuencial: el segundo nivel . . . 24
- Autenticación simétrica: desafíos y sesiones . . . 28

Parte IV: El protocolo de Sibuna . . . 31

- El intercambio del desafío . . . 31
- El registro de desafío sin estado . . . 33
- Orden de verificación y uso único . . . 34
- La cookie de sesión . . . 35

Parte V: Ingeniería sin asignaciones en Rust . . . 38

- El bucle de conexión . . . 38
- Estado concurrente sin asignaciones . . . 50

Parte VI: Autómatas, tries y el cortafuegos semántico . . . 60

- Búsqueda de múltiples patrones en una pasada . . . 60
- El trie radix para IPv4 e IPv6 . . . 62
- El cortafuegos semántico . . . 63
- El motor declarativo de políticas . . . 64

Parte VII: El motor de prueba de trabajo del navegador . . . 67

- Una fuente, dos destinos . . . 67
- El protocolo del worker . . . 71
- Calibración . . . 72
- La página intermedia . . . 72

Parte VIII: Evaluación empírica . . . 74

- Metodología . . . 74

Latencias de primitivas . . . 79
Comparación de productos completos . . . 82
Latencia con la misma carga ofrecida . . . 84
Sibuna y Anubis . . . 86
Sobrecarga de la consola . . . 90
Lo que no se mide . . . 91

Parte IX: Operaciones y despliegue . . . 94
Modos de protección y configuración . . . 94
Vista previa de la consola . . . 98
Topologías de despliegue . . . 115
Política declarativa . . . 123
Almacenamiento persistente y clúster . . . 123
Empaquetado . . . 128

Parte X: Referencia de consulta y diagnóstico . . . 133
Endpoints . . . 133
Métricas . . . 134
Códigos de estado . . . 135
Catálogo de errores . . . 136
Esquema de políticas . . . 138
Tablas de almacenamiento . . . 139
Objetivos de compilación . . . 139

Parte XI: Tarjeta de referencia rápida . . . 142
Compilar . . . 142
Ejecutar . . . 142
Consola (versión preliminar opcional) . . . 142
Las opciones más importantes . . . 143
Endpoints bajo `/__sibuna/` . . . 144
Códigos de estado . . . 144
Cabeceras . . . 145
Esqueleto del archivo de políticas . . . 145
Almacenamiento (Edge) en SQL . . . 145
Diagnósticos . . . 146
Cifras que recordar . . . 146
Enrutamiento en el proxy de entrada . . . 147

Soluciones seleccionadas . . . 148
Parte I: Coste y probabilidad . . . 148
Parte II: Trabajos previos . . . 148
Parte III: Criptografía . . . 148
Parte IV: Protocolo . . . 149
Parte V: Estado y orden . . . 149
Parte VI: Algoritmos . . . 149
Parte VII: Motor del navegador . . . 150
Parte VIII: Evaluación . . . 150

Parte IX: Operaciones . . . 150
Parte X: Referencia . . . 151
Continuar la investigación . . . 151
Glosario . . . **152**
Bibliografía . . . **155**
Índice . . . **156**

El libro de Sibuna, Versión 1.0.0-beta.3
© 2026, Vikrant Rathore y los colaboradores de Sibuna

Las matemáticas y la ingeniería de la defensa web.

Prefacio

Una petición llega con un método, una ruta, algunas cabeceras y la pretensión de que el servidor le preste atención. Ninguno de esos bytes demuestra que merezca la pena atenderla. Sibuna sitúa una decisión entre la petición y el origen: admitirla, rechazarla o pedir a su remitente un trabajo verificable. Este libro sigue esa decisión desde su modelo matemático hasta los bytes en memoria.

El sistema tiene dos modos de protección. *Gate* ofrece admisión por prueba de trabajo y controles locales frente a inundaciones de peticiones. *Shield* añade inspección de aplicaciones y decisiones de política. Ambos pueden usar almacenamiento persistente para políticas, reputación e historial de incidentes, local a un nodo o compartido por un clúster; llamamos *Edge* a ese despliegue. Una sesión válida establece la admisión; no exime a la petición de la inspección. Compartir el estado duradero en un clúster lo distribuye; no convierte un contador local del límite de tasa en una cuota global.

La cuestión central de ingeniería no es si un hash es rápido. Es si el camino completo permanece acotado cuando un cliente no fiable elige la entrada. ¿Cuántos bytes puede inspeccionar un analizador? ¿Cuánta memoria puede consumir un desafío? ¿Cuántas conexiones puede mantener abiertas un cliente? ¿Qué escrituras pueden reintentarse? ¿Qué sigue disponible cuando desaparece un nodo o la base de datos? Cada respuesta debe nombrar tanto un invariante como su límite.

Las matemáticas sirven al mismo fin. Un coste esperado no es una garantía de latencia. Un argumento de muestreo para un compromiso fijo no es una demostración frente a cualquier probador adaptativo. Una cola absorbe una ráfaga; no puede compensar a un consumidor permanentemente más lento. Deducimos pequeños modelos, resolvemos ejemplos a mano y después preguntamos dónde se aparta de ellos la implementación.

El libro está escrito para usarse de tres maneras. Leídas en orden, las partes I a VII forman un curso: cada capítulo empieza con lo que deberíamos poder hacer al terminarlo, resuelve un ejemplo a mano y termina con un ejercicio y un invariante que explicar con nuestras propias palabras; al final hay soluciones seleccionadas. Las partes VIII a X son la referencia: mediciones con su procedencia, cada opción y topología, cada endpoint, estado, error y tabla. La parte XI es una tarjeta de dos páginas para el terminal. Un glosario y una bibliografía cierran el libro.

La parte II sitúa a Sibuna junto a Anubis, SafeLine y el WAF de Cloudflare, característica por característica, y la parte VIII presenta las mediciones propias de Sibuna, desde primitivas individuales hasta ejecuciones del producto completo, y compara Sibuna con Anubis como productos completos bajo el mismo generador de carga. Ambas están escritas para que el lector pueda comprobar las afirmaciones: la comparación nombra sus fuentes y cada medición nombra el archivo de resultados confirmado en el repositorio del que se copia.

El lector debe conocer programación básica, logaritmos y probabilidad condicional. Rust se presenta a través de la propiedad (ownership) y la disposición de los datos, no mediante un repaso del lenguaje. Los capítulos pueden leerse en orden: el modelo de costes motiva el protocolo, el protocolo determina el estado y el estado determina el diseño de concurrencia y

almacenamiento. Los capítulos posteriores convierten esos invariantes en mediciones y procedimientos de operación.

Este es un libro de implementación, no una afirmación de que los desafíos computacionales eliminen el tráfico automatizado. Los clientes pueden comprar cómputo, las direcciones pueden compartirse y la sintaxis de las aplicaciones es más rica que un detector acotado. El resultado útil es un sistema cuyos costes y limitaciones pueden examinarse, probarse y cambiarse sin esconderlos tras un eslogan.

Edición para Sibuna 1.0.0-beta.3, noviembre de 2026

El código fuente, los ejercicios y las mediciones reproducibles están en el repositorio de Sibuna.

Cómo leer este libro

Empezamos con una sola pregunta: ¿cuál es la decisión segura más barata que el servidor puede tomar ahora? Mantenemos esa pregunta a nuestro lado a lo largo de los capítulos sobre el protocolo y la implementación. Una operación barata que admite la petición equivocada es un defecto; una operación correcta con un coste no acotado es otro tipo de defecto.

Pregunta	**Dónde se desarrolla la respuesta**
¿Qué se obtiene con un desafío?	Partes I a III: el coste, la probabilidad y los límites de la demostración.
¿Cómo se compara Sibuna?	Parte II: el linaje y la tabla de características; parte VIII: las mediciones.
¿Qué significa una sesión?	Parte IV: vínculos, orden de verificación, caducidad y reutilización.
¿Adónde va la memoria?	Partes V y VI: búferes, hilos, tablas, autómatas y publicación.
¿Cómo resuelve el navegador?	Parte VII: mensajes del worker, memoria y calibración.
¿Cómo sabemos que funciona?	Parte VIII: tres programas de medición y lo que excluye cada uno.
¿Cómo lo ejecutamos?	Parte IX: opciones, topologías, políticas, almacenamiento, clúster y empaquetado.
¿Qué era ese error?	Parte X: endpoints, estados, errores y esquema; parte XI: la tarjeta.

Tres itinerarios de lectura

- *Aprender el diseño.* Las partes I a IV en orden, después la traza resuelta de la parte VI y luego la parte VII. Haz los ejercicios; compáralos con las soluciones del final solo después. Vuelve a la parte V cuando una traza llegue a una tabla compartida o a un slice prestado.
- *Operar un despliegue.* La parte IX y después la parte XI, con la parte X abierta para consultas. Lee la "Comparación de características" de la parte II antes de elegir un modo de protección, y la "Comparación de productos completos" de la parte VIII antes de elegir los límites de workers y conexiones.
- *Cambiar el código.* Las partes V y VI, las referencias al código fuente de cada capítulo y los registros SID que citan. La parte VIII explica qué programa de medición detectará una regresión en lo que hayamos tocado; la parte II enumera las restricciones que un cambio debe respetar.

Convenciones

Un *ejemplo resuelto* muestra los estados intermedios, no solo la respuesta. Un *ejercicio* nos pide cambiar un supuesto. Las pistas sugieren un primer paso; las soluciones seleccionadas al final del libro permiten comprobar el razonamiento. Los diagramas distinguen el camino de la petición del trabajo en segundo plano. Una referencia al código fuente nombra la implementación que hay que examinar cuando la prosa y el código parecen no coincidir. Los recuadros titulados "Implementación" nombran una función y su archivo; los recuadros titulados "Explica el invariante" nos piden enseñar la idea con nuestras palabras.

En las ecuaciones, b es la dificultad de Hashcash en bits, n es la profundidad del trabajo secuencial, t es el número de aperturas, p es la probabilidad de éxito por intento, K es el número de intentos, T es el intervalo de emisión de un limitador de tasa, τ su tolerancia a ráfagas y Q es la capacidad de una cola. Un símbolo es local a su sección salvo que se indique lo contrario. Los nanosegundos y las peticiones por segundo de la parte VIII son mediciones de una ejecución identificada. Los números de los ejemplos resueltos son entradas elegidas, no afirmaciones de rendimiento. Los modelos estadísticos enuncian sus supuestos antes de extraer conclusiones.

PARTE I

El coste de una petición

Antes de elegir un algoritmo, identifica el recurso que debe proteger. Este capítulo construye pequeños modelos con los que se mide cada decisión de diseño posterior: el trabajo del origen, el trabajo del desafío, la amortización mediante sesiones y la memoria que un cliente no fiable puede hacer que el servidor retenga.

Objetivos Al terminar este capítulo, deberíamos poder escribir la desigualdad que decide si un control de admisión compensa su coste, enunciar lo que una prueba de trabajo demuestra y lo que no, deducir la esperanza y la cola de la búsqueda de hashes, explicar cómo las sesiones amortizan el trabajo y enumerar los cinco invariantes que mantiene el resto del libro.

La admisión es una decisión económica

Supongamos que un origen hace una consulta a la base de datos y genera una página por cada petición admitida. Sea c_o ese trabajo, c_g el trabajo del control de admisión por petición y r la tasa de llegada. Sin control de admisión, el origen debe sostener rc_o unidades de trabajo por segundo. Un control que admite una fracción a cambia esa demanda a rac_o, mientras él mismo gasta rc_g. El control solo es útil cuando el trabajo ahorrado al origen supera su propio coste y el coste que impone a los visitantes legítimos.

$$rc_g + rac_o < rc_o \quad \Leftrightarrow \quad c_g < (1-a)c_o.$$

Este es un modelo de capacidad, no una medición. Omite el ancho de banda, el estado de las conexiones, el almacenamiento y el retraso del cliente. Su valor está en que cada término omitido es visible. Un acierto de caché puede hacer pequeño c_o; una consulta costosa puede hacerlo grande. No existe una relación universal entre el coste del solicitante y el del servidor.

Ejemplo resuelto: cuándo compensa un control de admisión Elegimos $c_o = 100$ unidades de trabajo y $c_g = 1$. Si se rechaza la mitad de las peticiones, el sistema con control gasta $1 + 0.5 \times 100 = 51$ unidades por llegada en lugar de 100. Si solo se rechaza una de cada mil, gasta 100.9: en este modelo, el control cuesta más de lo que ahorra. Estos números elegidos no son tiempos medidos. Muestran por qué la carga de trabajo debe figurar en toda afirmación de rendimiento.

Qué demuestra una prueba de trabajo

Un desafío para el cliente demuestra que alguien encontró una entrada que satisface una regla de verificación pública. No demuestra humanidad, identidad ni buena intención. Un solicitante puede alquilar cómputo, reutilizar una sesión válida durante su vida útil o repartir el trabajo entre varias máquinas. El desafío cambia el coste de admisión; las políticas y la inspección siguen decidiendo lo que pueden hacer las peticiones admitidas.

Los límites basados en direcciones son complementarios. Restringen una dirección, pero una dirección compartida puede representar a muchas personas y un solicitante puede usar muchas direcciones. Ni un desafío ni una dirección son un oráculo de identidad. Por eso Sibuna mantiene la admisión, la inspección y la reputación como decisiones separadas.

La búsqueda de hashes como variable aleatoria

Para un resumen ideal de 256 bits, exigir b bits iniciales a cero da una probabilidad de éxito $p = 2^{-b}$ en cada intento independiente. Sea K el número de intentos hasta el primer éxito, incluido este. Entonces

$$P(K = k) = (1-p)^{k-1}p, \quad P(K > k) = (1-p)^k, \quad E[K] = 1/p = 2^b.$$

La esperanza se deduce de la suma de la cola:

$$E[K] = \sum_{k=0}^{\infty} P(K > k) = \sum_{k=0}^{\infty} (1-p)^k = 1/p.$$

Un paso de dificultad duplica los intentos esperados. No promete que cada desafío tarde el doble. Algunos tienen éxito en el primer intento; otros tardan mucho más que la media. Un cliente que mide una sola resolución ha medido sobre todo esta aleatoriedad.

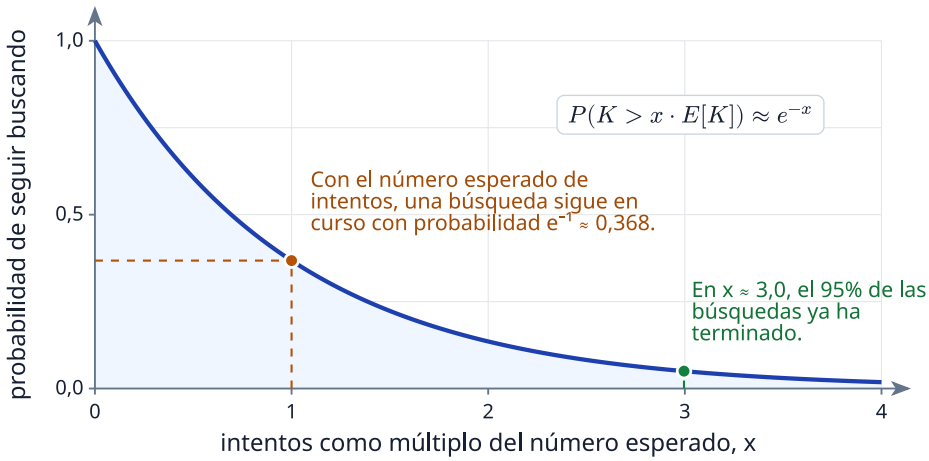


Fig. 1 Probabilidad de supervivencia de la búsqueda de hashes: la probabilidad de que una búsqueda siga en curso tras x veces el número esperado de intentos, en la aproximación para trabajo grande $P(K > x/p) \approx e^{-x}$.

Ejemplo resuelto: la esperanza no es un plazo Con $b = 16$, el trabajo esperado es de 65.536 hashes. La probabilidad de seguir buscando después de ese número de intentos es aproximadamente $e^{-1} = 0.368$. El número de intentos del percentil 95 es $\lceil \ln(0.05) / \ln(1 - 2^{-16}) \rceil \approx 196327$, casi el triple de la media. Una interfaz de usuario debe tolerar esa dispersión en lugar de anunciar un fallo en el tiempo de finalización esperado.

Las sesiones amortizan el trabajo

Supongamos que una sesión permite m peticiones antes de caducar, que un desafío cuesta c_p, que la verificación cuesta c_v y que una comprobación de sesión cuesta c_s. Si ignoramos los intentos fallidos, el coste amortizado del control por petición es $c_v/m + c_s$, mientras que el coste del desafío para el cliente es c_p/m. Alargar la vida de la sesión ayuda por igual a las personas y a los clientes automatizados. Elegirla es una decisión de política, no una optimización criptográfica.

Ejercicio 1.1. Un cliente obtiene 100 peticiones por sesión y resuelve un desafío que cuesta un millón de intentos. ¿Cuál es el trabajo amortizado por petición? ¿Qué cambia si comparte la sesión con un segundo proceso?
Pista: distingue el modelo contable de los vínculos reales del token.

El estado acotado es un segundo presupuesto

Sacar trabajo del camino de la petición no lo hace desaparecer. Supongamos que los incidentes llegan a una tasa λ, que el hilo de almacenamiento los guarda a una tasa media μ y que la cola contiene Q registros. Cuando $\lambda > \mu$, una aproximación de fluido da

$$t_{\text{fill}} \approx \frac{Q}{\lambda - \mu}.$$

Una cola más grande gana tiempo; no aumenta el rendimiento. Agrupar en lotes amortiza el coste de las transacciones, pero los registros retenidos siguen necesitando memoria y ser atendidos en algún momento. Bajo sobrecarga, Sibuna descarta los nuevos registros de incidentes y los cuenta. Sigue tomando decisiones sobre las peticiones a partir del estado en memoria. El operador debe vigilar la evidencia perdida además de la tasa de éxito HTTP.

Ejercicio 1.2. Una cola contiene 512 registros. La tasa de llegada es de 1.000/s y la de vaciado de 600/s. ¿Durante cuánto tiempo puede una cola inicialmente vacía absorber el exceso? ¿Por qué la respuesta es solo una aproximación?
Pista: usa la diferencia de tasas y después considera las ráfagas y las confirmaciones por lotes.

Los invariantes que mantendremos

1. Las peticiones no fiables no pueden crear estado no acotado en el servidor.
2. Un token de admisión válido no elude una denegación de la aplicación.
3. Un búfer sigue vivo mientras exista cualquier slice tomado prestado de él.
4. Reintentar una escritura duradera no puede multiplicar un incidente ni su efecto sobre la reputación.
5. Una instantánea de políticas publicada es inmutable hasta que su último lector la libera.

El resto del libro deduce los mecanismos que hacen verdaderas estas afirmaciones y las pruebas que revelarían una violación. Un camino rápido solo es útil mientras esas afirmaciones sigan siendo verdaderas.

Explica el invariante. Enuncia con palabras la desigualdad del control de admisión $c_g < (1 - a)c_o$ y después explica por qué una latencia de verificación medida no basta, por sí sola, para que un operador sepa si merece la pena ejecutar el control.

PARTE II

Trabajos previos y el espacio de diseño

Situamos Sibuna entre los sistemas de los que aprendió: los desafíos de cliente y el trabajo secuencial de la literatura criptográfica, los cortafuegos de aplicaciones por firmas y semánticos, las páginas intermedias de prueba de trabajo y las redes perimetrales alojadas. Después comparamos, característica por característica, las cuatro herramientas que un operador sopesará con más probabilidad, y registramos qué hechos se comprobaron.

De dónde viene Sibuna

Objetivos Al terminar este capítulo, deberíamos poder trazar el linaje de los desafíos de cliente desde 1992 hasta las construcciones de trabajo secuencial de 2018, nombrar las tres ideas de arquitectura que combina Sibuna, situar Sibuna, Anubis, SafeLine y el WAF de Cloudflare en un mismo espacio de diseño, y decir qué afirmaciones de la comparación se verificaron y cuáles solo se leyeron en la documentación.

Desafíos de cliente y trabajo secuencial

"Pricing via Processing" de Dwork y Naor (CRYPTO 1992) propuso que un servicio exigiera un cálculo moderadamente difícil y fácil de comprobar antes de trabajar para quien se lo pide. Hashcash de Back (1997, descrito en un artículo en 2002) convirtió el desafío en una inversión parcial de hash cuya dificultad es un solo entero, y los desafíos de cliente de Juels y Brainard (NDSS 1999) aplicaron la idea a las inundaciones de conexiones, donde el servidor no debe recordar nada de un desafío sin resolver. Los tres comparten la forma que conserva Sibuna: un enunciado elegido por el servidor, una solución de coste ajustable y una verificación varios órdenes de magnitud más barata que la resolución.

La búsqueda de hashes es trivialmente paralelizable, así que un solicitante con más núcleos termina antes. Mahmoody, Moran y Vadhan (CRYPTO 2013) definieron las pruebas de trabajo *secuencial*, en las que el probador debe realizar una larga cadena de pasos dependientes, tenga los núcleos que tenga, y Cohen y Pietrzak (EUROCRYPT 2018) propusieron la construcción sencilla basada en un grafo de hashes cuyo verificador solo necesita un trabajo $O(t \log N)$. Blocki, Lee y Zhou (ITC 2021) demostraron que la misma construcción es segura frente a probadores cuánticos en el modelo de oráculo aleatorio cuántico. La parte III deduce de esa línea de trabajo el segundo nivel de Sibuna; SID 0006 recoge los supuestos.

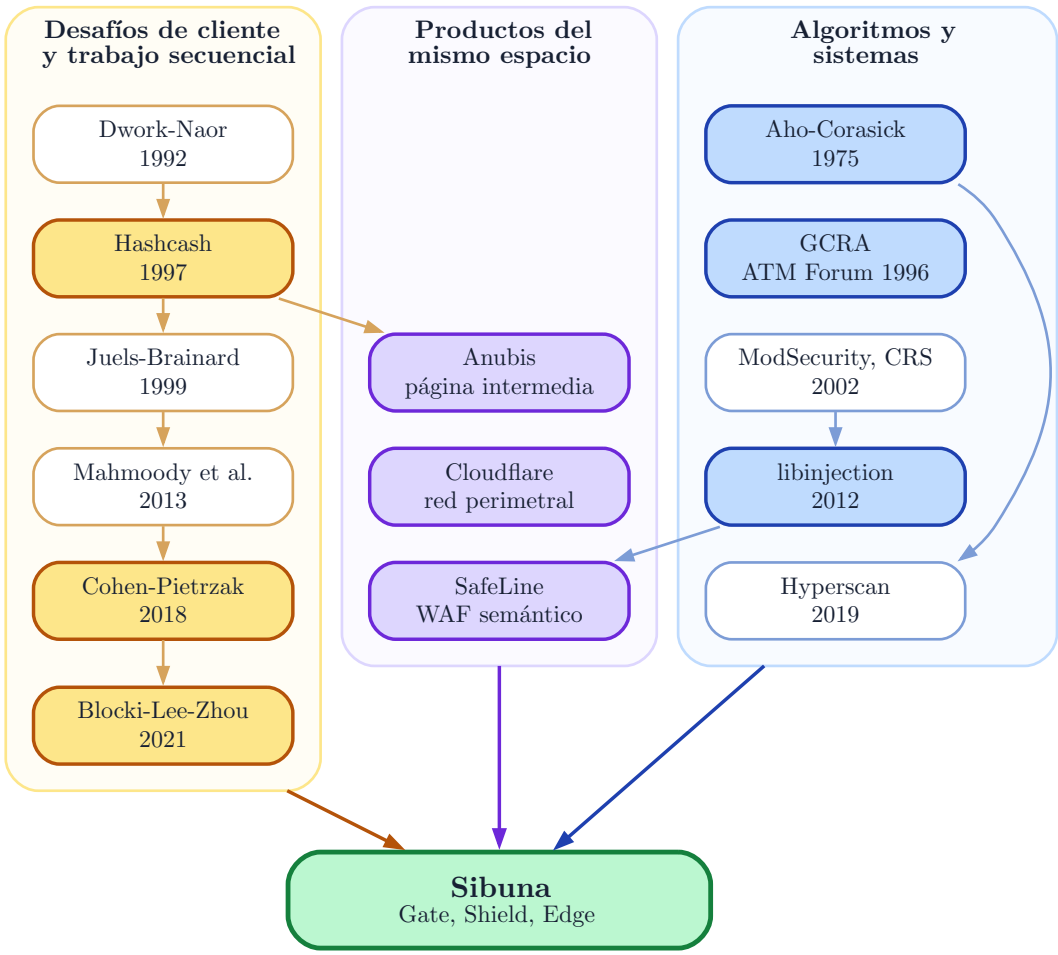


Fig. 2 Linaje de las ideas que combina Sibuna. La columna izquierda es la criptografía de los desafíos de cliente y del trabajo secuencial; la derecha, los algoritmos y sistemas en los que se apoya Sibuna, y la central, los productos del mismo espacio; el tiempo avanza hacia abajo. Las flechas finas indican descendencia directa, y las cajas rellenas son las ideas que Sibuna toma directamente. Un diseño de almacenamiento anterior se basaba en Multi-Paxos (Lamport 1998; SID 0005); ahora Sibuna guarda su estado en SQLite o PostgreSQL (SID 0011).

Cortafuegos de aplicaciones

La primera generación de cortafuegos de aplicaciones web comparaba los bytes de la petición con expresiones regulares. ModSecurity (2002) y el OWASP Core Rule Set hicieron de ese enfoque una solución abierta y muy extendida, y también dieron a conocer sus costes: cientos de patrones con retroceso por petición, puntuaciones de anomalía ajustadas a mano y una larga cola de falsos positivos con entradas corrientes. libinjection (Hanson, 2012) sustituyó las listas de patrones por un tokenizador que pregunta si una cadena *se analiza sintácticamente* como SQL, lo que redujo tanto el coste como los falsos positivos. SafeLine, de Chaitin, lleva esa idea semántica a un producto completo, y Hyperscan (Wang et al., NSDI 2019) mostró hasta dónde puede llevarse la búsqueda vectorizada de literales cuando las entradas son largas. La

parte VI explica por qué, para los campos cortos que inspecciona, Sibuna eligió un autómata denso junto con pequeños tokenizadores estructurales en lugar de expresiones regulares o de un motor SIMD.

Páginas intermedias y redes perimetrales

La página intermedia de prueba de trabajo delante de un sitio web se hizo común en 2025, cuando el tráfico de los rastreadores que alimentan el entrenamiento de modelos de lenguaje hizo inasumible el coste de mantener orígenes corrientes. Anubis (Techaro) es la implementación abierta más conocida: un proxy inverso en Go que sirve un desafío en JavaScript, firma un token de sesión y, por lo demás, deja pasar el tráfico. El modo de protección Gate de Sibuna resuelve el mismo problema con un desafío sin estado, un token simétrico y un segundo nivel de trabajo; la comparación siguiente recoge en qué se diferencian ambos.

En el otro extremo de la escala, las redes perimetrales alojadas como Cloudflare sitúan la inspección, la limitación de tasa y la puntuación de bots en cientos de puntos de presencia con un plano de control global. El despliegue Edge de Sibuna toma prestada esa forma (un plano compartido de políticas y reputación que alimenta instantáneas locales e inmutables) sin dejar de ser autoalojado; por eso la capa de almacenamiento es una base de datos que gestiona el operador (un archivo SQLite en un nodo, una base de datos PostgreSQL compartida por un clúster), leída por un hilo de almacenamiento fuera del camino de la petición, y no una llamada a un servicio.

Tres ideas en un solo binario

Sibuna combina un desafío de trabajo en el navegador para la admisión, una inspección acotada de la carga útil de la aplicación y un plano de control compartido que publica instantáneas locales de políticas. Gate y Shield son los dos modos de protección del producto; el almacenamiento y el despliegue distribuido son opcionales en ambos.

Propiedad	Gate	Shield
Admisión	Verificación nativa de Hashcash o de trabajo secuencial	Igual
Sesión	BLAKE3 con clave; Ed25519 opcional	Igual
Controles locales	Reglas, CIDR, GCRA, bloqueos	Igual
Inspección	Desactivada	Firmas acotadas y tokenizadores estructurales
Estado	Políticas y reputación persistentes opcionales, compartidas en un clúster	También análisis forense de la inspección

El solucionador del navegador se compila a partir de las mismas fuentes de pruebas que el verificador nativo. La búsqueda densa de literales evita el retroceso y mantiene el trabajo de

escaneo lineal en la longitud de la entrada. Las instantáneas inmutables de políticas aíslan el tratamiento de las peticiones de SQL y de la disponibilidad de la base de datos. Ninguna de estas decisiones establece una optimalidad algorítmica universal: las cabeceras cortas, los cuerpos largos, la diversidad de patrones, el número de núcleos y el tamaño de la caché cambian los compromisos.

Restricciones de diseño adoptadas

1. Ninguna asignación en el heap en el camino de la petición; cada tabla tiene una capacidad fija y un comportamiento de saturación declarado.
2. Emitir un desafío no escribe nada; solo una solución pagada ocupa memoria.
3. Una sesión admite; nunca exime a una petición de la inspección ni de una denegación explícita.
4. Toda cifra que se presenta la mide un programa de medición confirmado en el repositorio o lleva la etiqueta de modelo.
5. Las decisiones criptográficas citan una publicación y una demostración, y declaran su postura frente a la computación cuántica.

Comparación de características

Las cuatro tablas siguientes comparan Sibuna con las tres herramientas que se mencionan con más frecuencia junto a él. Los datos sobre los otros productos se tomaron de su documentación pública y de los artefactos de sus versiones en septiembre de 2026: Anubis 1.27.0 (licencia MIT, Go), SafeLine community edition 9.x (código de gestión GPL-3.0 en torno a imágenes de detector cerradas, Docker Compose) y la documentación para desarrolladores del WAF de Cloudflare. Anubis también se ejecutó en el host de pruebas de rendimiento; SafeLine y Cloudflare no (la parte VIII explica por qué). Un guion significa que el producto no ofrece la característica; un nombre de plan significa que la característica existe, pero solo en ese plan. Cada medición de Sibuna nombra el archivo de resultados confirmado del que procede. La fila de la huella TLS se comprobó en octubre de 2026 con el registro de cambios de Anubis (1.21.0 y 1.26.0), las notas de versión de SafeLine (8.10.0 y 9.2.0) y la página de Cloudflare "JA3/JA4 fingerprint".

Tabla 3 Comparación de características, parte 1: despliegue

Característica	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
Se ejecuta como	Un binario; proxy inverso o forward-auth; archivo SQLite opcional o base de datos PostgreSQL compartida	Un binario Go; proxy inverso o forward-auth	Siete contenedores Docker (tengine, detector, mgt, luigi, fvm, chaos, PostgreSQL)	Red alojada; el DNS apunta a Cloudflare

Característica	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
Lenguaje, entorno de ejecución	Rust, sin recolector de basura; ningún crate externo cuando el almacenamiento está desactivado	Go, con recolector de basura	Proxy y detector en C/C++, gestión en Go, PostgreSQL	Software perimetral propietario
Licencia	LGPL-3.0-or-later	MIT	Código de gestión GPL-3.0; imágenes de detector cerradas	Servicio propietario
Consumo en el host	Binario de 35,7 MiB (`benchmarks/results/comparison-latest.json`); 5,9 MiB de RSS en reposo (`benchmarks/results/primitives-latest.json`); pico de RSS de 16,0 MiB como máximo bajo carga (`benchmarks/results/product-latest.md`)	Binario de 38,1 MiB; 21 MiB de RSS en reposo; entorno de pruebas Linux, consola desactivada	1 núcleo de CPU, 1 GB de RAM, 5 GB de disco como mínimo, Docker 20.10+	Ninguno en instalaciones propias

Tabla 4 Comparación de características, parte 2: admisión y sesiones

Característica	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
Admisión por prueba de trabajo	Hashcash (con precisión de bits) y trabajo secuencial de Cohen-Pietrzak, elegidos por regla; WebAssembly con alternativa en JavaScript	Hashcash SHA-256 en nibbles hexadecimales (`fast`, `slow`); desafíos sin trabajo `metarefresh` y `preact`	Desafío antibots en JavaScript y CAPTCHA; no condicionados al trabajo	Managed challenge, JS challenge, Turnstile; Bot Fight Mode emite un desafío de cálculo

Característica	**Sibuna**	**Anubis 1.27**	**SafeLine CE 9**	**Cloudflare WAF**
Estado del desafío en el servidor	Ninguno hasta que se resuelve; etiquetas resueltas en una tabla Robin Hood fija	Almacén: memoria, bbolt, Valkey o S3	Gestionado por la pila	Gestionado por Cloudflare
Token de sesión	Etiqueta BLAKE3 con clave, cookie de 75 caracteres vinculada al trabajo pagado; Ed25519 opcional	JSON Web Token firmado con Ed25519; HS512 opcional	Cookie emitida por la pila	Cookie `cf_clearance`
Postura poscuántica del token por defecto	Simétrico: solo se aplica la aceleración cuadrática de Grover	Ed25519: roto por el algoritmo de Shor	No documentada	No documentada
Lenguaje de políticas	Reglas JSON: ruta, User-Agent, cabeceras, CIDR IPv4/IPv6, huella JA4; ALLOW, DENY, CHALLENGE, WEIGH con umbrales; reglas dinámicas desde la base de datos	Bots en YAML/JSON: expresiones regulares sobre UA, ruta y cabeceras; CIDR; expresiones CEL; WEIGH con umbrales	Reglas personalizadas definidas en la consola, ACL, grupos de IP	Expresiones del lenguaje de reglas; *5 / 20 / 100 / 1000 reglas personalizadas según el plan*
Huella TLS (JA4)	Las reglas comparan una huella JA4 de forma exacta o por prefijo; la calcula un proxy de entrada de confianza, que la reenvía en una cabecera (`--ja4-header`); Sibuna no la calcula	No documentada; las políticas pueden hacer referencia a la huella HTTP JA4H, que Anubis calcula y añade como cabecera `X-Http-Fingerprint-JA4H`	Se calcula en su propia terminación TLS; aparece en los detalles de los ataques; las reglas personalizadas la comparan (igual, distinto); una regla integrada deniega las huellas de su inteli-	Huellas JA3 y JA4 en las reglas personalizadas del WAF y en la analítica *(Enterprise con Bot Management)*

Característica	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
			gencia sobre JA4 maliciosos	

Tabla 5 Comparación de características, parte 3: inspección y estado

Característica	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
Inspección de SQLi, XSS, RCE y recorrido de rutas	Shield: autómata etiquetado más tokenizadores estructurales, primeros 8 KB del cuerpo	-	Motor de análisis semántico sobre las categorías OWASP, además de SSRF, XXE y CRLF	Conjuntos de reglas gestionados *(Pro y superiores)*; OWASP Core Ruleset; puntuación de ataque *(Business y superiores)*
Limitación de tasa	GCRA por cliente, local de cada nodo	- (la carga del sistema se expone a las reglas CEL)	Por IP, ruta y sesión	*1 / 2 / 5 / 100 reglas según el plan*; claves solo por IP por debajo de Enterprise
Bloqueos y reputación	Bloqueos por honeypot; trie de reputación cargado desde la tabla de reputación compartida en el clúster	DNSBL; ASN y GeoIP mediante Thoth *(de pago, cerrado)*	Grupos de IP y listas negras; inteligencia de amenazas *(Pro)*	Listas de IP; listas gestionadas y puntuación de bots *(Enterprise)*
Análisis forense	SQLite o PostgreSQL: búsqueda de texto completo en incidentes, agrupación vectorial de campañas	Solo métricas de Prometheus	Registro de ataques en PostgreSQL con consola	Security Events; muestreado en el plan Free
Multinodo	Una base de datos PostgreSQL compartida por todos los nodos para políticas y reputación; semilla compartida;	Clave de firma compartida; almacén Valkey compartido	Una pila por host	Anycast global

Característica	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
	verificación de desafíos con afinidad de nodo			

Tabla 6 Comparación de características, parte 4: operaciones

Característica	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
Métricas	Prometheus en `/__sibuna/metrics`	Prometheus en un puerto separado	Paneles de la consola	Panel y analítica
Terminación TLS	- (proxy de entrada)	- (proxy de entrada)	**Sí**	**Sí**
Consola de gestión	Opcional; SID 0007 en estado Committed	-	**Sí**	**Sí**
Medido en el host de pruebas de rendimiento	**Sí** (parte VIII)	**Sí**	- (solo Docker)	- (alojado)

Tres diferencias pesan más que el resto a la hora de elegir.

- *Dónde recae el trabajo.* Anubis y Sibuna Gate hacen que el *cliente* pague antes de que el origen haga nada. SafeLine y Cloudflare inspeccionan primero y desafían de forma selectiva; su postura por defecto es admitir y filtrar. Sibuna Shield hace ambas cosas, en ese orden: la inspección puede denegar una petición que ya tiene una sesión válida.
- *Lo que recuerda el servidor.* Anubis guarda el estado de los desafíos en un almacén (memoria, bbolt, Valkey o S3) y firma un JSON Web Token con Ed25519; Sibuna emite un registro autoautenticado y solo recuerda las etiquetas resueltas en una tabla fija. SafeLine guarda los registros de ataques en PostgreSQL; Cloudflare lo guarda todo, con visibilidad muestreada en el plan Free.
- *Lo que tiene que ejecutar un operador.* Sibuna es un binario con un archivo SQLite opcional o una base de datos PostgreSQL compartida por un clúster; Anubis es un binario Go; SafeLine son siete contenedores tras un daemon de Docker en un host Linux con al menos un núcleo, un gigabyte de memoria y cinco gigabytes de disco; Cloudflare es un cambio de DNS y una suscripción.

Alcance actual del producto Sibuna deja la terminación TLS de entrada al proxy del despliegue, y no puntúa bots con un modelo entrenado ni publica firmas de pago. Compara la huella JA4 que calcula un proxy de entrada de confianza; no puede calcularla por sí

mismo (SID 0014, parte IX). Su consola opcional ofrece paneles autenticados, enriquecimiento por país, edición de políticas, investigación y gestión del clúster. La búsqueda de país y el trabajo de almacenamiento se ejecutan fuera de la clasificación de peticiones. SID 0007 está en estado Committed y la consola supera sus flujos de trabajo en el navegador (SID 0010). Su criterio formal de aislamiento del rendimiento, con ocho paneles activos bajo carga, aún no se ha medido; la parte VIII recoge lo que sí se ha medido.

Ejercicio 2.1. Explica por qué compilar una sola fuente del verificador a código nativo y a WebAssembly reduce la deriva del protocolo. ¿Qué propiedades siguen exigiendo pruebas independientes o una revisión criptográfica?

Ejercicio 2.2. Usando solo las tablas de comparación, escribe el despliegue más pequeño que dé a un sitio estático (a) un control de admisión por prueba de trabajo, (b) inspección de inyección SQL y (c) un bloqueo en todo el clúster. ¿Qué filas determinaron cada elección?
Pista: empieza por las filas "Se ejecuta como" y "Multinodo".

Explica el invariante. Describe cómo Gate, Shield y el almacenamiento compartido opcional se reparten la admisión, la inspección y la distribución de políticas. Después nombra una fila de las tablas de comparación que querríamos volver a verificar antes de citarla a un cliente, y di cómo la verificaríamos.

PARTE III

Fundamentos de la investigación criptográfica

Deducimos los modelos de coste de los dos niveles de prueba de trabajo de Sibuna, presentamos la prueba de trabajo secuencial (Proof of Sequential Work) de Cohen y Pietrzak con su argumento de seguridad y justificamos la autenticación con hash con clave para desafíos y sesiones. El SID 0006 recoge los supuestos y las referencias.

Hashcash: las matemáticas del primer nivel

Objetivos Al terminar este capítulo, deberíamos poder enunciar el trabajo esperado, la varianza y la probabilidad de cola de un desafío Hashcash medido en bits, explicar por qué el coste de verificación del servidor acota la ventaja del atacante y describir los dos límites que motivan un segundo nivel.

Definición y coste

El desafío del primer nivel de Sibuna para la cadena de desafío C y la dificultad de b bits es: encontrar $N \in \mathbb{N}$ tal que SHA-256 $(C \parallel ":" \parallel \mathrm{dec}\,(N))$ tenga b bits iniciales a cero. La representación decimal del nonce es deliberada: mantiene idéntico el formato de transmisión entre el solucionador WebAssembly, la alternativa en JavaScript y el verificador nativo, sin un paso de codificación binaria.

La unidad de dificultad es un *intento de hash* esperado, no una llamada a la función de compresión. Una entrada puede ocupar varios bloques de SHA-256. El precálculo del prefijo también puede cambiar el trabajo por nonce. La regla de transmisión sigue siendo el predicado sobre el resumen; las implementaciones deben coincidir en el enunciado y en la codificación del nonce, no en un coste de CPU estimado.

Para resúmenes ideales independientes, el número de intentos X es geométrico con $p = 2^{-b}$. Su mediana es $\lceil \ln(1/2)/\ln(1-p) \rceil$, aproximadamente $2^b \ln 2$ para p pequeño, y su varianza es $(1-p)/p^2$. El verificador comprueba un resumen candidato en lugar de realizar la búsqueda. El análisis sintáctico, la autenticación del desafío, las comprobaciones de huella, la inserción en el conjunto de desafíos gastados y la respuesta son trabajo adicional. El tiempo medido de una primitiva no permite deducir la capacidad de un servidor frente a inundaciones.

Dos límites de la búsqueda

El primero es la varianza: una búsqueda larga no indica necesariamente un worker averiado. El segundo es el paralelismo: los intentos de nonce independientes pueden ejecutarse en paralelo. Más núcleos reducen el tiempo transcurrido de la búsqueda aunque el número total de intentos

esperado siga siendo el mismo. Por tanto, el rendimiento del hardware importa para la política de admisión. Ni una tasa de hash declarada ni una única prueba de rendimiento pueden acotar la ventaja de cada solicitante.

Ejercicio 3.0. Con la cola geométrica, estima la probabilidad de necesitar más de 4.6×2^b intentos. ¿Por qué no cambia el número esperado de intentos al repartir el espacio de nonces entre varios workers?
Pista: cuenta los intentos totales, no los segundos transcurridos.

Prueba de trabajo secuencial: el segundo nivel

Objetivos Comprende el etiquetado de Cohen-Pietrzak, por qué obliga a un cálculo secuencial, qué comprueba el verificador, la cota condicional de muestreo $(1-\alpha)^t$ y sus limitaciones, el resultado poscuántico, la cota de memoria del probador y los datos de calibración en los que se basa la correspondencia de bits de trabajo.

Por qué trabajo secuencial

Una construcción de trabajo secuencial introduce dependencias entre cálculos: una etiqueta posterior necesita un valor producido antes. Esto aborda una magnitud distinta del trabajo total, es decir, la profundidad del cálculo. Los desafíos independientes pueden seguir ejecutándose en paralelo, y la latencia del hash en el hardware sigue importando.
Sibuna sigue la construcción basada en hashes descrita por Cohen y Pietrzak. Su grafo tiene $2^{n+1}-1$ etiquetas a profundidad n, pero un número fijo de etiquetas no es una latencia determinista en el navegador: las etiquetas pueden ocupar varios bloques de compresión, las aperturas pueden recalcular subárboles y la planificación añade variación. Las afirmaciones de seguridad de los análisis en el modelo de oráculo no deben convertirse en una afirmación sin matices sobre cualquier implementación concreta o adversario.

La construcción

Sea n la profundidad y consideremos el árbol binario completo cuyos nodos se nombran con cadenas binarias de longitud como máximo n (raíz ε, hijos $v0$ y $v1$, hojas de longitud n). Cada nodo v lleva una etiqueta de 32 bytes $\ell_v = H(\chi, v, \text{parents}\,(v))$, donde χ es el enunciado del desafío y H es SHA-256 con χ absorbido una sola vez (el mismo truco del prefijo que en el solucionador de Hashcash):

- un nodo interno calcula el hash de sus dos hijos: $\ell_v = H(\chi, v, \ell_{v0}, \ell_{v1})$;
- una hoja $u = u_1 \ldots u_n$ calcula el hash de los *hermanos izquierdos de sus antepasados* allí donde el camino desde la raíz giró a la derecha: $\ell_u = H(\chi, u, \{\ell_{u_1 \ldots u_{i-1} 0} : u_i = 1\})$.

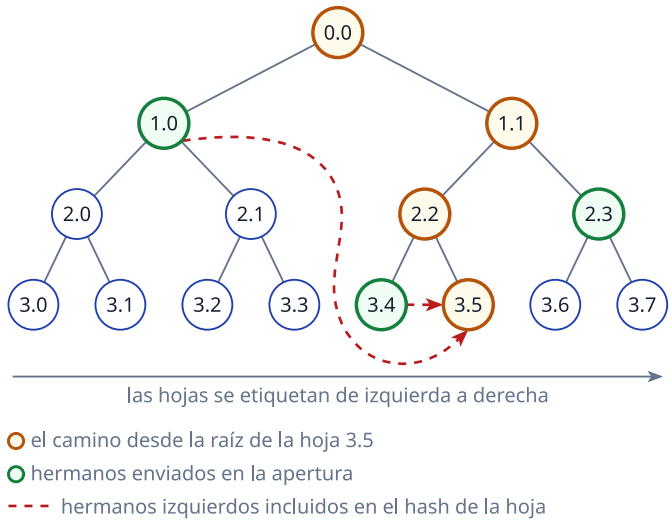


Fig. 3 El árbol de profundidad 3: una apertura de la hoja 3.5 envía la etiqueta de la hoja y las tres etiquetas hermanas; las aristas discontinuas son las dependencias de los hermanos izquierdos que hacen secuencial el etiquetado.

Las aristas de los hermanos izquierdos son lo que hace secuencial el cálculo: la hoja u no puede etiquetarse hasta que esté completo cada subárbol a su izquierda, así que se impone un recorrido en postorden en profundidad, y la etiqueta raíz $\varphi = \ell_\varepsilon$ depende de las $2^{n+1} - 1$ etiquetas en secuencia.

Aperturas. Tras calcular φ, el probador deriva t hojas mediante Fiat-Shamir, $\gamma_i = H(\chi \parallel \varphi \parallel i) \bmod 2^n$, y para cada una envía la etiqueta de la hoja y las n etiquetas hermanas a lo largo de su camino hasta la raíz. El verificador recalcula la hoja a partir de las hermanas que son sus padres, calcula los hashes hacia arriba a lo largo del camino y comprueba que llega a φ. El coste del verificador es de $O(tn)$ cálculos de etiquetas con un espacio de trabajo acotado. Un cálculo de etiqueta puede abarcar varios bloques de hash; contar etiquetas no es contar llamadas de compresión.

> **Implementación: `posw::verify_opening`** Recalcula una hoja abierta a partir de sus etiquetas hermanas y recorre el camino hasta la raíz. En `crates/pow/src/posw.rs` (crate `sibuna-pow`).

```
fn verify_opening(base: &Sha256, n: u8, phi: &Label, gamma: u32, opening:
&[Label]) -> bool {
    let Some((leaf, siblings)) = opening.split_first() else {
        return false;
    };
    // `siblings` va de la profundidad n a la 1; invertido, se empareja con 1..=n.
    let mut hasher = node_hasher(base, n, gamma);
    for (d, sibling) in (1..=n).zip(siblings.iter().rev()) {
        if turns_right(gamma, n, d) {
```

```
            hasher.update(sibling);
        }
    }
    if hasher.finalize() != *leaf {
        return false;
    }
    let mut current = *leaf;
    for (d, sibling) in (1..=n).rev().zip(siblings) {
        let mut parent = node_hasher(base, d - 1, gamma >> (n - d + 1));
        if turns_right(gamma, n, d) {
            parent.update(sibling);
            parent.update(&current);
        } else {
            parent.update(&current);
            parent.update(sibling);
        }
        current = parent.finalize();
    }
    current == *phi
}
```

Muestreo, solidez y sus límites

Consideremos un compromiso fijo con una fracción defectuosa α detectable de forma independiente. Si cada apertura muestrea de manera uniforme e independiente, la probabilidad de no encontrar ninguna posición defectuosa en t aperturas es $(1-\alpha)^t$. Este cálculo elemental explica por qué ayudan más aperturas. No es, por sí solo, una demostración de seguridad para un protocolo de trabajo secuencial.

Ejemplo resuelto: una cota condicional de muestreo Con una fracción defectuosa fija $\alpha = 0.1$ y $t = 16$ aperturas independientes, la probabilidad de no encontrar ninguna posición defectuosa es $0.9^{16} \approx 0.185$. Con 32 aperturas es de alrededor de 0,034. La premisa fija el compromiso antes de elegir las muestras. Si un probador puede probar muchas raíces por fuerza bruta, elegir qué compromiso revelar o correlacionar los fallos, este cálculo no tiene en cuenta esa estrategia.

La construcción de trabajo secuencial a la que nos referimos tiene un análisis de seguridad en un modelo de oráculo. La separación de dominios, la correspondencia de parámetros, el truncamiento y la implementación de Sibuna deben seguir comprobándose frente a los supuestos de la construcción. El muestreo de Fiat-Shamir hace que las aperturas sean reproducibles a partir de un compromiso; la reproducibilidad por sí sola no demuestra la resistencia a la búsqueda por fuerza bruta. No afirmamos de forma incondicional que hacer trampa nunca sea más barato.

Ejercicio 3.1. Si un atacante prueba g compromisos independientes, cada uno aceptado con probabilidad q, deduce la probabilidad de que al menos uno sea aceptado. ¿Qué costes faltan en esa expresión?
Pista: calcula primero la probabilidad de que fallen los g.

Memoria del probador

Un probador ingenuo guarda todas las etiquetas ($2^{n+1} - 1$ en total). El probador de Sibuna mantiene una pila con las etiquetas de los hermanos izquierdos del camino actual ($n + 1$ etiquetas) durante la primera pasada y conserva solo los $m = \min(n, 10)$ niveles superiores ($2^{m+1} - 1$ etiquetas, como máximo 64 KB). Para abrir una hoja recalcula el subárbol del antepasado de la hoja a profundidad m, $2^{n-m+1} - 1$ hashes, inicializando la pila a partir de los niveles conservados. El coste total del probador es de $2^{n+1} - 1 + t\,(2^{n-m+1} - 1)$ hashes y todo el espacio de trabajo, incluida la prueba de salida, ocupa menos de 100 KB: una pestaña del navegador y una prueba nativa comparten el mismo tipo `Workspace` (`sibuna_pow::posw::Workspace`).

Los parámetros no son tiempos

Sibuna hace corresponder los bits de trabajo con la profundidad mediante $n = b - 3$, limitada al intervalo admitido $[4, 24]$. Es una convención de escala de trabajo, no una garantía de igual tiempo real entre algoritmos. Una hoja puede calcular el hash de varias etiquetas padre, mientras que Hashcash prueba un nonce repetidamente. El compilador, el navegador, el procesador y la elección de parámetros afectan a sus costes relativos. La prueba de rendimiento de primitivas confirmada en el repositorio mide la verificación; no establece la latencia de resolución en el navegador.

El tamaño de la prueba es de $32(1 + t(n + 1))$ bytes. A profundidad 13 con dieciséis aperturas, esto da $32(1 + 16 \times 14) = 7200$ bytes. Aumentar el número de aperturas incrementa tanto la prueba como el trabajo del verificador, aunque el coste dominante de construir el árbol siga siendo el mismo.

Ejercicio 3.2. Calcula el tamaño de la prueba a profundidad 17 con 16 aperturas. Después duplica el número de aperturas. ¿Qué término se duplica y cuál permanece fijo?

Elegir una construcción

La comparación relevante es un presupuesto de recursos, no la reputación de un algoritmo. Registra la memoria del probador, el trabajo del verificador en el peor caso, los bytes de la prueba y los supuestos necesarios para la propiedad afirmada. La búsqueda con uso intensivo de memoria (memory-hard) puede ser útil, pero un verificador que debe repetir un trabajo costoso da al atacante otro lugar donde gastar los recursos del servidor. Las construcciones de retardo verificables públicamente sirven a un modelo de confianza distinto del de un control de admisión con secreto compartido.

Sibuna conserva la verificación barata por hash y una opción de trabajo secuencial basada en hashes. Esta última requiere más bytes de prueba y un análisis de parámetros más cuidadoso. Ninguna de las dos elecciones elimina la necesidad de rechazar las pruebas mal formadas antes de entrar en una verificación costosa.

Explica el invariante. Explica la diferencia entre un árbol de autenticación, que demuestra que una etiqueta pertenece a un compromiso, y las aristas de dependencia, que restringen cómo se calculan las etiquetas.

Autenticación simétrica: desafíos y sesiones

Objetivos Explica por qué una función pseudoaleatoria con clave es la primitiva correcta cuando el emisor y el verificador pertenecen al mismo dominio de confianza, enuncia la cota de falsificación para una etiqueta de 128 bits, describe el registro de desafío sin estado y la cota del conjunto de desafíos gastados, y lee la derivación de claves.

El emisor es el verificador

Una firma digital permite verificar a *cualquiera* que tenga la clave pública, lo cual es la herramienta adecuada cuando los verificadores no deben poder emitir. El verificador de Sibuna *es* el emisor (o un clúster que comparte una semilla). Un código de autenticación de mensajes construido a partir de una función pseudoaleatoria proporciona autenticación dentro de ese dominio de confianza. No proporciona verificación pública: cualquiera que tenga la clave MAC también puede emitir tokens. El capítulo de pruebas de rendimiento compara las operaciones de token implementadas.

Lema 3 (cota de falsificación) Con BLAKE3 en modo con clave modelado como una PRF y una etiqueta de 16 bytes, un adversario que hace q consultas de verificación falsifica un token válido con probabilidad como máximo $q \cdot 2^{-128} + \varepsilon_{\mathrm{PRF}}$. A un millón de consultas por segundo, eso es 2^{-108} por segundo.

Una segunda consecuencia importa a largo plazo: Ed25519 se basa en logaritmos discretos en un grupo de curva elíptica, que el algoritmo de Shor rompe; un hash con clave se basa solo en la función hash, frente a la que los algoritmos cuánticos dan como mucho una aceleración cuadrática. Sibuna mantiene Ed25519 como opción (`--token-scheme ed25519`) para los despliegues que necesitan verificadores que no puedan emitir tokens.

La derivación de claves

Una semilla maestra de 32 bytes (de `--secret-file`, de `SIBUNA_SECRET` o un valor aleatorio nuevo) da lugar, mediante BLAKE3 con clave y una cadena de dominio, a cada clave de uso específico: las claves de token, de desafío, de Ed25519 y de huella, y la clave de requisito que sella los tickets de requisito (parte IV).

Implementación: `Keys::derive` Deriva las cinco claves de uso específico a partir de la semilla. En `crates/challenge/src/keys.rs` (crate `sibuna-challenge`). `Keys` y `Seed` ponen a cero sus bytes al liberarse, y su salida `Debug` oculta su contenido.

```
pub fn derive(seed: &Seed) -> Keys {
    Keys {
        token: subkey(seed, "sibuna/token/v1"),
        challenge: subkey(seed, "sibuna/challenge/v1"),
        ed25519_seed: subkey(seed, "sibuna/ed25519/v1"),
```

```
        fingerprint: subkey(seed, "sibuna/fingerprint/v1"),
        requirement: subkey(seed, "sibuna/requirement/v1"),
    }
}

fn subkey(seed: &Seed, purpose: &str) -> [u8; 32] {
    blake3::keyed_hash(seed.as_bytes(), purpose.as_bytes())
}
```

En un clúster, `Keys::bind_node` sustituye la clave de desafío por BLAKE3 con esa clave sobre `sibuna/challenge-node/v1` y el identificador del nodo, de modo que un desafío debe volver al nodo que lo emitió, mientras que los tokens de sesión y los tickets de requisito siguen siendo válidos en todos los miembros.
La separación de dominios evita la reutilización accidental entre usos; comprometer la semilla maestra compromete todas las claves derivadas. Un clúster solo tiene que ponerse de acuerdo en una semilla.

Desafíos sin estado

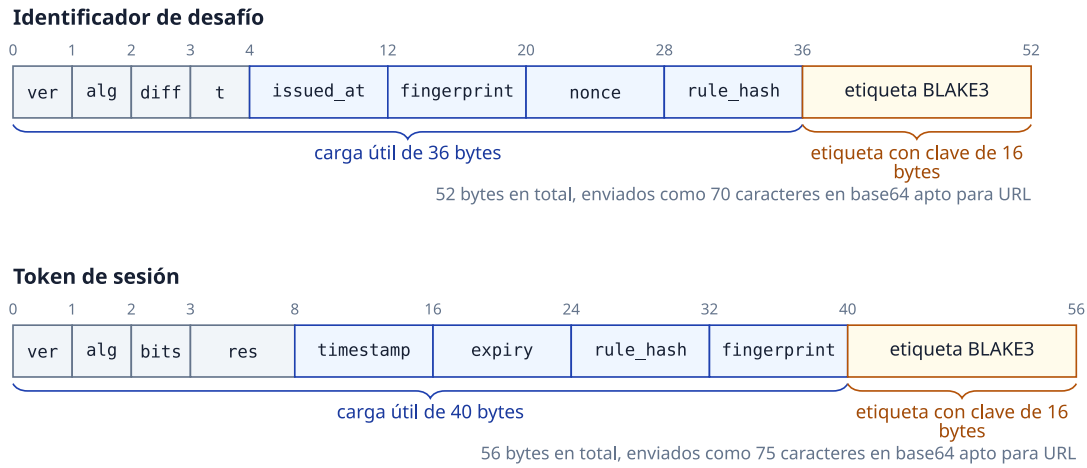


Fig. 4 Formatos de transmisión del identificador de desafío y del token de sesión. Los números sobre cada registro son desplazamientos en bytes.

Un identificador de desafío es un registro autoautenticado: versión, algoritmo, dificultad, número de aperturas, hora de emisión, la huella con clave del cliente, un nonce derivado de la PRF y el hash de la regla de política que exigió el desafío, seguidos de una etiqueta de 16 bytes bajo la clave de desafío. Emitir uno no escribe nada. La verificación decodifica el registro, comprueba la etiqueta en tiempo constante, comprueba la antigüedad frente al TTL del desafío, comprueba la huella frente al cliente que lo envía y solo entonces examina la prueba.

Lema 4 (el estado solo crece con trabajo pagado) Sea S el conjunto de etiquetas de desafío que recuerda el daemon. Una etiqueta entra en S solo después de aceptarse

una prueba válida para ella, y sale tras el TTL del desafío. Por tanto, $|S| \leq r \cdot \text{TTL}$ cuando r acota la tasa de soluciones distintas aceptadas en ese intervalo. Obtener un desafío gratuito no reserva memoria para ninguna etiqueta. La implementación también tiene una capacidad fija y puede rechazar una solución válida cuando está saturada. Es una cota sobre el estado, no una cota inferior sobre la estrategia de cálculo de cualquier adversario.

El conjunto de desafíos gastados está formado por 16 particiones de direccionamiento abierto Robin Hood sobre las etiquetas de 16 bytes (parte V). La huella es un hash con clave de la dirección del cliente y del User-Agent, con separación de longitudes para que dos entradas no colisionen por concatenación, y con clave para que no puedan calcularse sin conexión las huellas de otros clientes.

Ejercicio 3.3. La carga útil de un token lleva un nivel de trabajo, `timestamp`, `expiry`, `rule_hash` y `fingerprint`. Explica qué impide cada campo (qué fallaría si se eliminara) y por qué la etiqueta debe cubrir los cinco.

Explica el invariante. Un compañero propone cambiar todos los tokens a Ed25519 "porque es más fuerte". Explica el argumento del dominio de confianza, la diferencia de coste y la consideración poscuántica.

PARTE IV

El protocolo de Sibuna

Especificamos el protocolo de comunicación tal como está implementado: el intercambio con la página intermedia, el registro de desafío sin estado, los dos formatos de solución, el orden de verificación, la garantía de uso único y la cookie de sesión.

El intercambio del desafío

Objetivos Al terminar este capítulo, deberíamos poder seguir a un navegador desde su primera petición hasta la emisión de su sesión, nombrar cada endpoint y cada campo JSON que intervienen y explicar por qué el desafío lleva consigo la decisión de política de la ruta protegida.

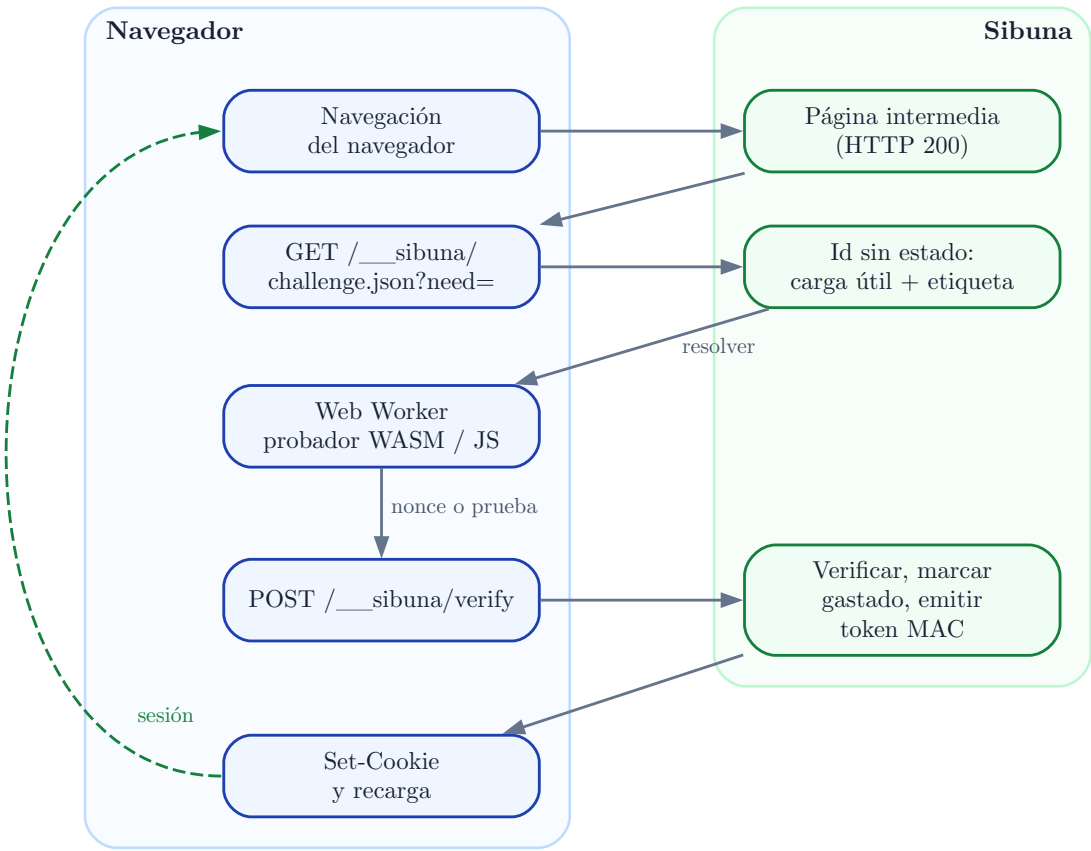


Fig. 5 El intercambio del desafío. Los pasos de la izquierda se ejecutan en el navegador, y los de la derecha, en Sibuna.

1. Una navegación sin sesión válida llega al motor de políticas y se clasifica como `CHALLENGE`. Como la petición acepta `text/html`, el daemon responde `200` con la página intermedia y la cabecera `X-Sibuna-Status: CHALLENGE`; un cliente de API que no acepta HTML recibe `401` con un cuerpo JSON cuya URL `challenge` ya puede pedirse. En ambos casos la respuesta lleva un *ticket de requisito*: el desafío que exige esta decisión (algoritmo, bits de trabajo, aperturas) y el hash de la regla, sellados para este cliente con una etiqueta BLAKE3 calculada con su propia clave derivada y válidos durante el tiempo de vida de un desafío.
2. La página intermedia pide `GET /__sibuna/challenge.json?path=<original URL>&need=<ticket>`. Un ticket válido decide el desafío: el trabajo lo fija la decisión que lo exigió, y no se recalcula a partir de la URL que comunica la página ni de las cabeceras de esta segunda petición, que difieren de las de la navegación (`Accept`, `Sec-Fetch-*`). Sin un ticket válido (un cliente de API que ignora la URL, una página que sigue abierta más allá del tiempo de vida) el servidor evalúa la URL comunicada, que la misma función que lee la línea de petición divide en ruta y consulta, con la ruta en la forma normal que usa la admisión (SID 0009); una URL de más de 8 KiB se rechaza con `414`, nunca se trunca, y una ruta sin forma normal, con `400`. El ticket es orientativo: la admisión sigue comprobando el nivel de trabajo de la sesión, así que un ticket reutilizado o falsificado solo puede costar trabajo a quien lo presenta.
3. La respuesta es el registro de desafío:

   ```
   {"id":"AQEND...70
   chars","algorithm":"posw","difficulty":13,"challenges":16,"expires_at":1757241234}
   ```

   Para `hashcash`, `difficulty` es el número de bits y `challenges` vale `0`.
4. Un Web Worker lo resuelve con el módulo WebAssembly, o con el probador JavaScript si WebAssembly no está disponible (parte VII), y envía o bien

   ```
   {"challenge_id":"...","nonce":"90766"}
   ```

   o bien

   ```
   {"challenge_id":"...","proof":"<base64url, 32(1 + t(n+1)) bytes>"}
   ```

   a `POST /__sibuna/verify`.
5. Si la verificación tiene éxito, el servidor responde `200 {"status":"ok"}` con una cabecera `Set-Cookie`, y la página recarga su ubicación original. Cada petición posterior lleva la cookie y se admite antes de evaluar las políticas.

Por qué la ruta viaja con el desafío La página intermedia se sirve en la URL protegida, así que el navegador sabe dónde está. Enviar la ruta permite al servidor elegir la dificultad y el algoritmo *de cada regla* (una ruta de pago puede exigir 20 bits de trabajo secuencial mientras que un blog exige 16 bits de Hashcash) y vincula el hash de la regla al token, desde donde `X-Sibuna-Rule-Hash` lo comunica al origen.

El registro de desafío sin estado

Objetivos Lee el codificador de desafíos, comprende el propósito de cada campo y observa cómo se deriva el nonce de cada desafío sin una llamada al sistema para obtener entropía en la ruta crítica.

Implementación: `Coordinator::create_challenge_with_spec` Construye la carga útil de 36 bytes, la etiqueta con la clave de desafíos y codifica en base64url los 52 bytes como un identificador de 70 caracteres. En `crates/challenge/src/coordinator.rs` (crate `sibuna-challenge`).

```
pub fn create_challenge_with_spec(
    &self,
    client_ip: &[u8],
    user_agent: &[u8],
    now: u64,
    spec: ChallengeSpec,
    rule_hash: u64,
) -> Challenge {
    self.adaptive.observe(now.saturating_mul(1000));
    let effective = ChallengeSpec {
        difficulty: spec.difficulty.saturating_add(self.adaptive.bump()),
        ..spec
    };
    let fingerprint = self.fingerprint(client_ip, user_agent);
    let difficulty = effective.parameter();
    let challenges = match effective.algorithm {
        Algorithm::Hashcash => 0,
        Algorithm::Posw => effective.posw_challenges,
    };

    let mut payload = [0u8; ID_PAYLOAD_LEN];
    payload[0] = ID_VERSION;
    payload[1] = effective.algorithm.to_byte();
    payload[2] = difficulty;
    payload[3] = challenges;
    payload[4..12].copy_from_slice(&now.to_le_bytes());
    payload[12..20].copy_from_slice(&fingerprint.to_le_bytes());
    payload[20..28].copy_from_slice(&self.next_nonce(now,
fingerprint).to_le_bytes());
    payload[28..36].copy_from_slice(&rule_hash.to_le_bytes());
    let mut raw = [0u8; ID_RAW_LEN];
    raw[..ID_PAYLOAD_LEN].copy_from_slice(&payload);
    raw[ID_PAYLOAD_LEN..].copy_from_slice(&self.challenge_tag(&payload));

    Challenge {
        id: encoded::encode(&raw),
        algorithm: effective.algorithm,
        difficulty,
        challenges,
```

```
        expires_at: now.saturating_add(u64::from(self.challenge_ttl)),
    }
}
```

El identificador es un `Encoded<70>`: el texto se guarda en un array de su longitud exacta, así que emitir un desafío nunca asigna memoria. `ChallengeSpec::parameter` convierte los bits de trabajo en la unidad propia de cada nivel: bits de Hashcash (como máximo 64) o profundidad de PoSW (`difficulty - 3`, limitada al intervalo de 4 a 24).
Tres detalles merecen atención:

- *Dificultad adaptativa.* El coordinador mantiene en tres atómicos una media móvil con ponderación exponencial de la tasa de emisión de desafíos. Por encima de una base de 50 desafíos por segundo, cada duplicación de la tasa añade un bit de trabajo, hasta un máximo de seis. Una inundación paga un coste exponencialmente mayor, mientras que el tráfico tranquilo mantiene el coste base.
- *El nonce es la salida de una PRF* con la clave de desafíos, calculada sobre un contador, el reloj y la huella. Es único (por el contador) e impredecible (por la clave), así que un adversario no puede precalcular soluciones para identificadores que no se le han emitido, y la ruta crítica nunca llama al sistema operativo para obtener entropía.
- *La dificultad viaja dentro de la etiqueta.* Un cliente no puede reducir la dificultad editando el registro: la etiqueta fallaría antes de examinar ninguna prueba.

Orden de verificación y uso único

Objetivos Comprende por qué las comprobaciones baratas se ejecutan antes que la prueba, por qué el desafío se marca como gastado solo después de que la prueba sea válida y cómo se resuelven dos envíos simultáneos de la misma solución.

Implementación: `Coordinator::verify_and_mint` Decodifica y autentica el identificador, comprueba la antigüedad y el vínculo con el cliente, verifica la prueba para el nivel registrado, registra la etiqueta como gastada y emite el token de sesión. En `crates/challenge/src/coordinator.rs` (crate `sibuna-challenge`).

```
pub fn verify_and_mint(
    &self,
    challenge_id: &[u8],
    solution: Solution<'_>,
    client_ip: &[u8],
    user_agent: &[u8],
    now: u64,
) -> Result<Verified, VerifyError> {
    let record = self.decode(challenge_id)?;
    let expires_at =
record.issued_at.saturating_add(u64::from(self.challenge_ttl));
```

```
    if now > expires_at || record.issued_at >
now.saturating_add(CLOCK_SKEW_SECONDS) {
        return Err(VerifyError::ChallengeExpired);
    }
    if record.fingerprint != self.fingerprint(client_ip, user_agent) {
        return Err(VerifyError::FingerprintMismatch);
    }
    check_solution(challenge_id, &record, solution)?;
    self.spent.mark_spent(&record.tag, expires_at, now).map_err(|error| match
error {
        SpendError::DoubleSpend => VerifyError::DoubleSpendAttempt,
        SpendError::Full => VerifyError::StoreFull,
    })?;
    let work = WorkLevel {
        algorithm: record.algorithm.to_byte(),
        bits: record.algorithm.work_bits(record.difficulty),
    };
    Ok(Verified {
        token: self.mint_token(now, record.rule_hash, record.fingerprint, work),
        algorithm: record.algorithm,
        difficulty: record.difficulty,
        challenges: record.challenges,
        ttl_seconds: self.token_ttl,
    })
}
```

El orden es deliberado. La etiqueta, la antigüedad y el vínculo son comprobaciones en tiempo constante que descartan la basura antes de calcular ningún hash. Una prueba errónea no debe consumir el desafío, así que el conjunto de desafíos gastados solo se escribe después de que `check_solution` tenga éxito. Dos hilos que verifican la misma solución válida compiten en `mark_spent`; el cerrojo de la partición los serializa, exactamente uno gana y el otro recibe `VerifyError::DoubleSpendAttempt`. La cláusula `issued_at > now + 60` (`CLOCK_SKEW_SECONDS` vale 60) rechaza los registros emitidos por un nodo cuyo reloj va adelantado más de un minuto, que de otro modo alargarían la vida de un desafío.

Los errores llegan al cliente como `400` con un diagnóstico al estilo de Elm (parte X): cada `VerifyError` se convierte en un `sibuna_core::errors::ErrorKind`, cuya explicación es el cuerpo de la respuesta. Las pruebas de extremo a extremo comprueban `DOUBLE SPEND`, `FINGERPRINT MISMATCH`, `WRONG SOLUTION TYPE` y `MALFORMED CHALLENGE` en el cuerpo de la respuesta.

La cookie de sesión

Objetivos Lee el token basado en un hash con clave, comprende los atributos de la cookie y explica el vínculo con el cliente.

Implementación: `token::mac::mint` y `token::mac::verify` Serializa la carga útil de 40 bytes, le añade los primeros 16 bytes de un BLAKE3 con clave calculado sobre ella y la verifica con una comparación en tiempo constante. En `crates/challenge/src/token/`

`mac.rs` (crate `sibuna-challenge`). `Coordinator::verify_cookie` la invoca con la huella del cliente.

```
pub fn verify(
    key: &[u8; 32],
    text: &[u8],
    now: u64,
    expected_fingerprint: Option<u64>,
) -> Result<Payload, TokenError> {
    if text.len() != TOKEN_LEN {
        return Err(TokenError::InvalidTokenLength);
    }
    let raw: [u8; RAW_LEN] =
        encoded::decode::<RAW_LEN,
TOKEN_LEN>(text).ok_or(TokenError::InvalidEncoding)?;
    let (payload, given) = raw.split_at(PAYLOAD_LEN);
    let mut bytes = [0u8; PAYLOAD_LEN];
    bytes.copy_from_slice(payload);
    // Tiempo constante: una salida anticipada diría al falsificador
    // cuántos bytes de la etiqueta acertó.
    if !ct::eq(&tag(key, &bytes), given) {
        return Err(TokenError::InvalidTokenSignature);
    }
    let payload = Payload::from_bytes(&bytes)?;
    payload.check(now, expected_fingerprint)?;
    Ok(payload)
}
```

La etiqueta se comprueba antes de analizar la carga útil: una carga útil de otra versión se rechaza con `TokenError::InvalidEncoding` solo cuando ya se sabe que es auténtica, y una falsificada nunca llega al analizador.
La cookie se emite así:

```
Set-Cookie: __sibuna_token=<75 chars>; Path=/; Max-Age=86400; HttpOnly;
    SameSite=Lax[; Secure]
```

`HttpOnly` la mantiene fuera del alcance de los scripts de la página, `SameSite=Lax` impide la reutilización desde otros sitios y permite la navegación de nivel superior, y `Secure` se añade con `--secure-cookie` cuando TLS termina delante del daemon. Como la carga útil lleva la huella con clave de la dirección y del User-Agent, una cookie copiada a otra máquina falla con `TokenError::TokenBoundAddressMismatch`; la prueba de extremo a extremo "the cookie is bound to the client identity" recorre exactamente ese camino.
La carga útil empieza con un byte de versión y el *nivel de trabajo* que pagó su titular: el mecanismo (Hashcash o PoSW) y los bits de trabajo realmente resueltos, incluido cualquier incremento adaptativo por carga. Una sesión evita un desafío solo cuando su nivel alcanza lo que exige la ruta, calculado a partir de la dificultad de la regla con la misma limitación que aplica el propio desafío. Los niveles están ordenados, no tienen nombre: una cookie obtenida con 24 bits cubre también cualquier ruta de 16 bits, mientras que una cookie de 16 bits presentada en una ruta de 24 bits pasa a la página intermedia más exigente, y la nueva cookie

sustituye a la antigua. `rule_hash` sigue siendo la identidad de auditoría de la regla que emitió el desafío; se comunica al origen y nunca se usa para la admisión. Una sesión nunca anula los hallazgos del WAF ni las denegaciones explícitas. La prueba de extremo a extremo "a session earned on a cheaper route does not admit a route that demands more work" comprueba ambos sentidos.

Direcciones reenviadas Detrás de un proxy de entrada, la dirección del cliente llega en `X-Forwarded-For`. Sibuna confía en esa cabecera solo con `--trust-forwarded` (implícita en el modo forward-auth). Un proxy inverso expuesto directamente a internet debe dejarla desactivada, o cualquier cliente podría elegir su propio vínculo y su propio cubo de límite de tasa.

Ejercicio 4.1. Esboza la secuencia de dos navegadores tras una misma dirección NAT con User-Agents idénticos. ¿Qué comprobaciones comparten y cuáles no? ¿Qué necesitaría un atacante en esa NAT para reutilizar la cookie del otro navegador?

Explica el invariante. Explica a un operador por qué el daemon no "recuerda" los desafíos pendientes y por qué eso es una propiedad de seguridad y no un atajo.

PARTE V

Ingeniería sin asignaciones en Rust

Seguimos una conexión desde la aceptación hasta la respuesta a través de un único búfer fijo, y examinamos las estructuras de datos concurrentes que permiten a muchos hilos de conexión compartir estado sin asignaciones en el heap en el camino de la petición. Donde la seguridad de memoria necesitaría de otro modo una maquinaria propia (contadores de lectores, arenas, tablas de descriptores, cadenas de desmontaje), mostramos cómo la aporta, en su lugar, el sistema de propiedad (ownership) de Rust.

El bucle de conexión

Objetivos Al terminar este capítulo, deberíamos poder explicar cómo un único búfer de 64 KiB sirve a toda una conexión keep-alive, cómo se localizan el encabezado y el cuerpo sin copiarlos, por qué cada conexión recibe su propio hilo acotado, cómo averigua el proxy la delimitación de las respuestas del origen para que una conexión de cliente a través del proxy pueda seguir abierta, y cuáles de estas garantías demuestra por nosotros el comprobador de préstamos.

Un hilo por conexión, con un límite

Los workers de aceptación (`--workers`, uno por CPU por defecto, 64 como máximo) comparten el socket de escucha. Cada conexión aceptada se entrega a un hilo propio con una pila de un megabyte, y el bucle de aceptación vuelve directamente a `accept`. Un hilo que termina su conexión queda en espera y toma la siguiente, porque crear y terminar un hilo para cada conexión frenaba a los demás hilos (parte VIII, SID 0002). La alternativa, servir cada conexión hasta el final en el hilo de aceptación, fue el primer diseño: daba buenas mediciones con un solo cliente y malas con sesenta y cuatro, porque sesenta de ellos esperaban a que un worker agotara su cuota de 256 peticiones, y la latencia de cola crecía con cada cliente en espera. Con un hilo por conexión, el p99 con 64 conexiones se mantiene por debajo de un milisegundo en todas las cargas de trabajo del producto completo, de 305 us a 857 us (`benchmarks/results/product-latest.md`; parte VIII).

El número de hilos de conexión está acotado por `--max-connections` (1.024 por defecto, 8.192 como máximo). Por encima del límite, el bucle de aceptación responde `503 Service Unavailable` en el nuevo socket, lo cierra sin crear ningún hilo y cuenta el evento en `sibuna_overloaded_total`.

Implementación: `accept_loop` Acepta conexiones, comprueba cada una frente al límite de conexiones y la lanza en un hilo propio tomado del conjunto de tareas. En `cmd/sibuna/src/server.rs` (crate `sibuna`).

```
fn accept_loop(server: &Arc<Server>, listener: &TcpListener) {
    while !server.is_stopping() {
        let stream = match listener.accept() {
            Ok((stream, _)) => stream,
            // ... sin descriptores ni memoria: espera 1 ms en vez de girar
        };
        let limit = server.max_connections.load(Ordering::Relaxed);
        if server.connections.fetch_add(1, Ordering::Relaxed) >= limit {
            server.connections.fetch_sub(1, Ordering::Relaxed);
            server.metrics.overloaded.bump();
            let text = "Service Unavailable: connection limit reached";
            reject(server, &stream, text);
            continue;
        }
        set_no_delay(&stream);
        let task_server = Arc::clone(server);
        let launched = server.tasks.launch(move || {
            let _counted = Counted(&task_server);
            crate::connection::serve(&task_server, &stream);
        });
        // ... un lanzamiento fallido devuelve la plaza y cuenta una sobrecarga
    }
}
```

`Counted` es un guardián cuyo `Drop` devuelve la plaza de la conexión dentro del límite termine como termine el hilo, por retorno o por pánico. El camino del `503` vacía lo que el cliente ya ha enviado antes de cerrar su lado del socket: cerrar un socket con bytes sin leer provoca un reset, y el reset descartaría el `503`.

Los hilos proceden de un `TaskPool` (`cmd/sibuna/src/tasks.rs`), una tabla fija de posiciones cuyos hilos admiten join. Un hilo terminado conserva su pila hasta que se hace join sobre él, así que el join de una posición se hace cuando vuelve a ocuparse, y una tarea nueva toma la posición terminada *más baja* en lugar de avanzar un cursor. Un cursor solo volvería a una posición terminada tras dar una vuelta completa a la tabla, lo que dejaría residentes miles de hilos terminados, y la memoria crecería con las conexiones servidas en lugar de con las conexiones abiertas. Un segundo guardián, `Finished`, marca la posición como libre cuando la tarea termina, así que una conexión que entra en pánico también libera la suya.

La memoria se acota del mismo modo. Un hilo de conexión guarda en su pila su búfer de lectura de 64 KiB y su búfer de escritura de 16 KiB, y en el heap un espacio de trabajo del proxy de 40 KiB, asignado una sola vez al empezar la conexión. El peor caso es, por tanto, un número conocido y no depende de cuántos sockets pueda abrir un cliente. El recolector de conexiones inactivas (más adelante) cierra las conexiones que dejan de enviar, así que un cliente lento no puede retener un hilo más allá de `--idle-timeout`.

Selección del asignador

Rust 1.100 estabiliza el núcleo de la API `Allocator` y los parámetros de asignador de `Vec` y `Box` (notas de la versión). Podemos elegir un asignador para una colección sin sustituir el asignador del proceso. Esta API no nos ahorra asignaciones en el recorrido de las peticiones: las peticiones en conexiones ya preparadas no asignan memoria, como comprueban `cmd/sibuna/tests/steady_state.rs` y `crates/net/tests/steady_state.rs`. El proxy asigna sus tres búferes una vez por conexión y los reutiliza. El limitador y el conjunto de pruebas usadas también asignan sus tablas una sola vez; las asignaciones inicializadas a cero permiten que las páginas sin tocar sigan sin ocupar memoria residente, como comprueba `crates/store/tests/resident.rs`. Antes de cambiar el asignador mediríamos la construcción de instantáneas de políticas y el análisis JSON de la consola. Sus colecciones temporales asignan memoria fuera del recorrido de las peticiones, pero el asignador de una colección no controla automáticamente las asignaciones de sus elementos. Nuestro árbol JSON también conserva cadenas con escapes mediante `Cow<str>`, cuya forma propia es `String`; `String` no tiene un parámetro de asignador en Rust 1.100. `Arc::new_in` sigue siendo inestable en esta versión (documentación de la API). Los motores publicados se comparten mediante `Arc` hasta que su último lector los libera, por lo que una reconstrucción no puede restablecer la memoria que aún usa un motor anterior. Una arena tendría que conservar esos tiempos de vida y los límites de entrada del analizador.
Conservamos el asignador global `WipeOnFree` del módulo WebAssembly de la consola en `cmd/console-ui/src/abi.rs`. Borra los bloques liberados, incluido el bloque anterior tras una reasignación. Elegir `System` directamente para una colección omitiría ese borrado. El módulo WebAssembly de prueba de trabajo no tiene asignador de montón, y Argon2 ya usa un espacio de trabajo reutilizado que pertenece al llamador. Por tanto, conservamos los asignadores actuales y la compatibilidad con Rust 1.94 hasta que una carga medida justifique un cambio. Un candidato debe mejorar la latencia o la memoria residente y conservar el borrado, el procesamiento acotado de la entrada y el recorrido de peticiones sin asignaciones.

Un búfer por conexión

Cada asignación dinámica en el camino de una petición es una palanca de denegación de servicio: fragmentación a lo largo de los días, contención del cerrojo del asignador entre hilos y amplificación por atacantes que fabrican peticiones para maximizar las asignaciones. La regla de Sibuna es absoluta: la clasificación, la verificación y la entrega al proxy se ejecutan con cero asignaciones en el heap. El manejador de la conexión posee un búfer de lectura de 64 KiB y un búfer de escritura de 16 KiB durante toda la vida del socket; todo lo demás son slices de ellos.

> **Implementación: `connection::serve`** Registra la conexión en el recolector de conexiones inactivas, prepara una sola vez los búferes, el espacio de trabajo del proxy y la caché del motor, y sirve hasta 4.096 peticiones. En `cmd/sibuna/src/connection.rs` (crate `sibuna`).

```
pub fn serve(server: &Server, stream: &TcpStream) {
    let Some(lease) = server.idle.register(stream) else { return };
    // ...
    let mut read_buffer = [0u8; MAX_REQUEST_BYTES]; // 64 KiB
```

```
    let mut write_buffer = [0u8; WRITE_BUFFER]; // 16 KiB
    let mut reader = Reader::new(stream, &mut read_buffer[..]);
    let mut peer = PeerText::default();
    peer.format(stream.peer_addr().ok());
    let mut link = Link {
        server,
        stream,
        writer: Writer::new(stream, &mut write_buffer[..]),
        workspace: Workspace::new(),
        cache: EngineCache::new(),
        activity: lease.activity(),
        last: false,
    };
    for served in 0..MAX_REQUESTS_PER_CONNECTION {
        link.last = served + 1 == MAX_REQUESTS_PER_CONNECTION;
        if !serve_one(&mut reader, &mut link, peer.as_bytes()) {
            break;
        }
        lease.touch();
    }
}
```

`read_head` llena el búfer hasta que aparece la línea en blanco, buscando solo en los bytes llegados desde la última búsqueda, y rechaza con `431` los encabezados de más de 16 KiB. `parse_request` (crate `sibuna-http`) produce un `Request<'a>` cuyos método, ruta, consulta, cabeceras y cookies son slices del búfer. El cuerpo declarado se llena hasta donde cabe y se toma como slice a continuación del encabezado. Después, `reader.consume(head_len + body_len)` hace avanzar el lector, de modo que la siguiente petición keep-alive empieza limpia; el lector solo mueve los bytes no leídos al principio cuando necesita espacio. Un cuerpo chunked se decodifica in situ detrás del encabezado (SID 0009): la delimitación de cada paso se elimina del búfer, de modo que el prefijo decodificado sigue siendo contiguo para la inspección. Las rutas internas están delimitadas por longitud y mantienen abierta la conexión. Las peticiones a través del proxy también la mantienen abierta, siempre que la respuesta del origen esté delimitada y el cuerpo se haya almacenado entero en el búfer; las secciones siguientes muestran cómo decide el proxy.

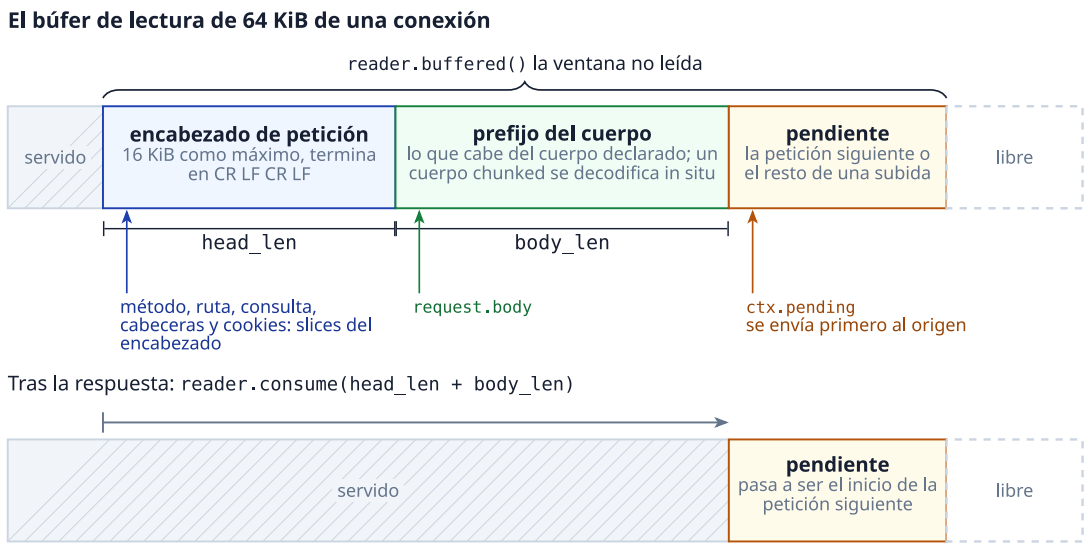


Fig. 6 El búfer de lectura de la conexión mientras se sirve una petición. Cada campo de la petición es un slice de él.

```rust
fn serve_one<W: Write>(
    reader: &mut Reader<'_, &TcpStream>,
    link: &mut Link<'_, W>,
    peer: &[u8],
) -> bool {
    let head_len = match read_head(reader) {
        Head::Complete(length) => length,
        Head::TooLarge => { /* 431 "Request head exceeds 16 KB" */ return false; }
        Head::Closed | Head::Failed => return false,
    };
    let framing = {
        let bytes = reader.buffered();
        let head = bytes.get(..head_len).unwrap_or_default();
        let request = match parse_request(head) {
            Ok(request) => request,
            Err(error) => return reject_head(link, error),
        };
        let declared = request.content_length().unwrap_or(0);
        // ... Expect: 100-continue
        if declared == 0 && !request.chunked {
            let upload = Upload::declared(&request);
            let pending = bytes.get(head_len..).unwrap_or_default();
            let keep = dispatch(link, request, pending, upload, 0, peer);
            reader.consume(head_len);
            return keep;
        }
        (declared, request.chunked)
    };
    serve_with_body(reader, link, peer, head_len, framing)
}
```

Lo que demuestra el comprobador de préstamos Los slices de la petición solo son válidos hasta el siguiente llenado del búfer. Sibuna expresa esa regla como un tipo, no como un comentario: la petición es un `Request<'a>`, y `'a` es un préstamo compartido del búfer del lector. Llenar o consumir el búfer requiere `&mut reader`, que el compilador rechaza mientras siga en uso un `Request` tomado prestado de él. En `serve_one` la petición se mueve a `dispatch`, y solo cuando esa llamada vuelve puede ejecutarse `reader.consume`; si se escriben en el orden inverso, el código no compila.
La misma regla explica dos decisiones que a primera vista parecen extrañas. Primero, `serve_with_body` analiza el encabezado una segunda vez. El primer análisis debe terminar su préstamo antes de almacenar el cuerpo en el búfer, porque almacenar un cuerpo chunked mueve bytes dentro del búfer; el segundo análisis toma prestado el búfer ya estable. El análisis es determinista y tarda unos 133 ns para una petición de navegador con una cookie (`zero_copy_request_and_cookie`, 132,5 ns, en `benchmarks/results/primitives-latest.md`), así que el segundo análisis cuesta poco y da el mismo veredicto. Segundo, el proxy toma prestados la petición analizada y los bytes pendientes y no debe volver a llenar el búfer de la conexión, así que lee los bytes adicionales del cliente en un búfer de subida propio. Como no puede devolver lo que ha leído, la conexión del cliente solo se conserva cuando el cuerpo se almacenó entero en el búfer.

La reescritura del encabezado en el proxy

El encabezado no se reenvía literalmente. `write_request_head` (`crates/net/src/request.rs`, crate `sibuna-net`) vuelve a emitir la línea de petición y cada cabecera hacia el escritor del origen, y descarta los campos salto a salto (`Connection`, `Keep-Alive`, `Proxy-Connection`, `Transfer-Encoding`, `TE`, `Trailer`, `Upgrade`, los campos de autorización del proxy), cualquier campo de reenvío o de auditoría entrante (`X-Forwarded-For`, `X-Real-IP`, `Forwarded`, `X-Forwarded-Proto`, `X-Forwarded-Host`, `X-Forwarded-Port`, `X-Sibuna-Status`, `X-Sibuna-Rule`), `Expect`, `Content-Length` y cada campo que nombre la cabecera `Connection` de la petición. Después añade el conjunto de auditoría. La delimitación del cuerpo se genera, nunca se copia:

```
for header in request.headers() {
    if is_hop_by_hop(header.name) || nominated(request, header.name) {
        continue;
    }
    out.write_all(header.name)?;
    out.write_all(b": ")?;
    out.write_all(header.value)?;
    out.write_all(b"\r\n")?;
}
match upload {
    Upload::None => {}
    Upload::Length(length) => {
        out.write_all(b"Content-Length: ")?;
        write_decimal(out, *length)?;
        out.write_all(b"\r\n")?;
    }
    Upload::Chunked(_) => out.write_all(b"Transfer-Encoding: chunked\r\n")?,
```

```
}
```

Un `Content-Length` se vuelve a emitir. Un cuerpo chunked que terminó dentro del búfer de la conexión se decodifica in situ y se envía con su longitud. Uno más largo se anuncia como `Transfer-Encoding: chunked` y se vuelve a dividir en chunks lectura a lectura (SID 0009): cada lectura llega al búfer de subida de 16 KiB del proxy, se decodifica allí in situ y sale como un chunk canónico, de modo que el origen nunca ve los tamaños de chunk, las extensiones ni los trailers del cliente. El conjunto de auditoría:

```
Connection: keep-alive
X-Forwarded-For: <client>
X-Forwarded-Proto: <scheme>
X-Real-IP: <client>
X-Sibuna-Status: PASS
X-Sibuna-Rule: session | robots-txt | ip/cidr-trie | ...
```

Los cuerpos mayores que el búfer se retransmiten en trozos de 16 KiB del cliente al escritor del origen (`relay_body`) antes de devolver la respuesta en streaming. La prueba unitaria `proxy_head_rewrite_drops_hop_by_hop_headers_and_injects_audit_fields` de `crates/net/src/request.rs` comprueba que la reescritura descarta un `X-Forwarded-For` falsificado e inyecta los campos de auditoría.

Los tres búferes del proxy, el búfer de lectura del origen (un encabezado de respuesta, 16 KiB), el búfer de escritura del origen (8 KiB) y el búfer de subida (16 KiB), residen en un `Workspace` que la conexión asigna una sola vez y reutiliza en cada petición. El espacio de trabajo hace que su vida útil sea la de la conexión, no la del intercambio, y el lector y el escritor del origen los toman prestados para un intercambio cada vez.

Retransmisión de la respuesta del origen

El primer proxy transmitía los bytes del origen hasta que este cerraba y después cerraba la conexión del cliente: una delimitación correcta sin análisis, al precio de una conexión TCP nueva por cada petición a través del proxy. Bajo un generador de carga, ese precio se veía como decenas de miles de sockets en `TIME_WAIT` y, en loopback, como puertos efímeros agotados. Ahora el proxy lee el encabezado del origen (16 KiB como máximo) y clasifica el cuerpo según las reglas de RFC 9112:

> **Implementación: `parse_response_head`** Devuelve el estado, una de cuatro delimitaciones (sin cuerpo para HEAD, 1xx, 204 y 304; un `Content-Length`; codificación de transferencia chunked; o delimitado por cierre) y si el origen mantiene la conexión. En `crates/net/src/response.rs` (crate `sibuna-net`).

```
pub enum Framing { None, Length(u64), Chunked, UntilClose }

pub fn parse_response_head(
    head: &[u8],
    head_request: bool,
```

```
) -> Option<ResponseHead> {
    let status = status_code(head)?;
    let mut fields = FramingFields {
        keep_alive: head.get(7) == Some(&b'1'),
        ..Default::default()
    };
    for line in crlf_lines(head).skip(1) {
        if line.is_empty() {
            break;
        }
        fields.read_line(line)?; // Transfer-Encoding, Content-Length, Connection
    }
    let mut framing = fields.length.map_or(Framing::UntilClose, Framing::Length);
    if head_request || status / 100 == 1 || status == 204 || status == 304 {
        framing = Framing::None;
    } else if fields.encoded {
        framing =
            if fields.chunked { Framing::Chunked } else { Framing::UntilClose };
    }
    let mut keep_alive = fields.keep_alive;
    if framing == Framing::UntilClose || fields.closing {
        keep_alive = false;
    }
    if fields.nominates_framing {
        return None;
    }
    Some(ResponseHead { status, framing, keep_alive })
}
```

El analizador rechaza lo que el proxy no debe retransmitir: una línea de estado incorrecta, un nombre de campo que no es un token, un byte de control en un valor, longitudes contradictorias, una codificación `chunked` repetida o que no es la última, y un campo `Connection` que nombra `Content-Length` o `Transfer-Encoding`. Su trabajo es lineal en el encabezado y nunca entra en pánico.

El encabezado se vuelve a emitir hacia el cliente con las cabeceras `Connection` del origen sustituidas por la decisión propia de Sibuna, y el cuerpo se retransmite con exactitud: `copy_exact` para una longitud, chunk a chunk (línea de tamaño, datos, trailers) para la codificación chunked, `copy_remaining` para el caso heredado. Solo el caso heredado cierra la conexión del cliente. Toda retransmisión se basa en un único paso de copia: reenviar lo que la fuente ya tenga y, solo cuando no tenga nada, vaciar el destino y después leer una vez. Así, un flujo de escrituras pequeñas se entrega a medida que se produce, mientras que una transferencia masiva sigue moviendo un búfer completo por lectura y escritura.

```
let keep = client_keep_alive && parsed.framing != Framing::UntilClose;
write_client_head(client, head, keep).map_err(client_head_error)?;
origin.consume(head_length);
match parsed.framing {
    Framing::None => {}
    Framing::Length(length) => copy_exact(origin, client, length, progress)?,
    Framing::Chunked => relay_chunked(origin, client, progress)?,
    Framing::UntilClose => copy_remaining(origin, client, progress)?,
```

```
}
client.flush().map_err(|_| ProxyError::ClientFailed)?;
Ok(Relayed { client_keep: keep, origin_reusable: parsed.keep_alive })
```

Analizar un encabezado de origen de ocho campos tarda 322 ns, y un `GET` keep-alive con una respuesta de 1 KiB a través del proxy por loopback tarda 29,8 us (medido en octubre de 2026; `crates/net/src/lib.rs` documenta la medición).

El conjunto de conexiones al origen

Analizar la delimitación también le dice al proxy si el socket del *origen* puede volver a usarse: una respuesta HTTP/1.1 sin `Connection: close` (o una HTTP/1.0 con `keep-alive`) cuyo cuerpo se consumió entero deja el socket en una frontera limpia entre peticiones. Esos sockets van a un conjunto fijo de 256 conexiones inactivas al origen, y la siguiente petición a través del proxy toma una en lugar de abrir una conexión. Sin el conjunto, cada petición a través del proxy abre una conexión nueva al origen, y cada conexión cerrada retiene su puerto local en `TIME_WAIT`; bajo carga, un proxy en loopback se queda pronto sin puertos efímeros y su rendimiento se desploma.

Un conjunto bajo un único cerrojo haría que todas las peticiones a través del proxy compitieran por ese cerrojo. Por eso el conjunto (`crates/net/src/pool.rs`) está dividido en franjas: 16 franjas de 16 sockets, cada una un `Mutex<Vec<Arc<TcpStream>>>` en su propia línea de caché. A cada hilo se le asigna una franja propia por turno rotatorio en su primer uso; deposita en ella y toma de ella en primer lugar, y mira en las demás franjas, sin esperar a un cerrojo ocupado, antes de abrir una conexión nueva.

```
pub fn take(&self) -> Option<Arc<TcpStream>> {
    if self.is_stopping() {
        return None;
    }
    let home = home_stripe();
    if let Some(socket) =
        self.stripe(home).and_then(|stripe| lock(&stripe.sockets).pop())
    {
        return Some(socket);
    }
    (1..STRIPES).find_map(|offset| {
        let stripe = self.stripe((home + offset) % STRIPES)?;
        try_lock(&stripe.sockets)?.pop()
    })
}
```

Puede que el origen haya cerrado un socket del conjunto mientras estaba inactivo. El proxy lo detecta de una de dos maneras, una escritura fallida o un fin de flujo antes del primer byte de la respuesta, y en ambos casos todavía no ha llegado nada al cliente, así que cierra el socket y reintenta una vez con una conexión nueva, nunca con otra del conjunto: tras un periodo de inactividad todos los sockets del conjunto pueden estar caducados, y la primera versión de este reintento, que tomaba un segundo socket del conjunto, respondía `502` a la primera petición después de cada periodo de calma. El reintento solo se permite para un método seguro (`GET`, `HEAD`, `OPTIONS`) cuyo cuerpo se almacenó entero en el búfer. La falta de respuesta no demuestra

que el origen no haya aplicado ya una modificación, y un cuerpo retransmitido por trozos no puede volver a enviarse; en ambos casos el cliente recibe `502`.

```
let safe = matches!(request.method, Method::Get | Method::Head | Method::Options);
let buffered = exchange.upload.is_buffered(request.body.len());
let retryable = safe && buffered;
for attempt in 0..2 {
    // Un reintento siempre abre una conexión nueva: tras una pausa, todo
    // socket del conjunto puede estar caducado, y un segundo socket
    // caducado haría fallar la petición.
    let pooled = if attempt == 0 { pool.take() } else { None };
    // ... conecta si el conjunto no dio nada, adjunta a Activity, intercambia
    match outcome {
        Ok(relayed) => {
            if relayed.origin_reusable {
                pool.give(origin);
            }
            return Ok(relayed.client_keep && buffered);
        }
        Err(error) => {
            drop(origin);
            let stale = was_pooled
                && retryable
                && matches!(
                    error,
                    ProxyError::UpstreamClosed | ProxyError::UpstreamWriteFailed
                );
            // ... cualquier otro error se notifica de inmediato
        }
    }
}
```

Las pruebas unitarias retransmiten cadenas de bytes fijas a través de las mismas funciones y comprueban que la salida es idéntica byte a byte salvo la cabecera de conexión (`crates/net/src/relay/tests.rs`). Una prueba de extremo a extremo del daemon envía dos peticiones a través del proxy por un mismo socket de cliente y comprueba, mediante una cabecera de secuencia que añade el origen simulado, que la segunda reutilizó la conexión al origen del conjunto. Una prueba de vectores en `crates/net/tests/` reproduce escenarios de proxy grabados (una petición de cliente en bruto y una respuesta de origen predefinida) y compara cada byte que reciben el origen y el cliente.

Por qué el socket del origen es un `Arc` Un descriptor de socket es un entero pequeño que el kernel reutiliza en cuanto se cierra. La parada debe interrumpir un intercambio en curso, pero nunca debe tocar un descriptor que entretanto se haya reutilizado para otra conexión. Una tabla de descriptores activos bajo un cerrojo podría evitarlo, al precio de un cerrojo en cada intercambio. Sibuna no necesita esa tabla: el socket del origen es un `Arc<TcpStream>`. El conjunto guarda una referencia, el intercambio guarda otra, y la `Activity` de la conexión guarda un clon mientras dura el intercambio. El descriptor se cierra cuando se suelta la última referencia, así que no puede cerrarse, ni reutilizarse, mientras alguien pueda todavía acceder a él.

Plazos de inactividad

Cada conexión de cliente registra su socket en una `IdleTable` (`crates/net/src/idle.rs`) y recibe un `Lease`. La `Activity` del lease guarda el momento en que se movieron bytes por última vez en la conexión. El daemon la actualiza tras servir cada petición; el proxy la actualiza en cada lectura del origen y por cada 16 KiB de subida, y adjunta el socket del origen del intercambio en curso. El hilo recolector se despierta cada cuarto del plazo más corto (entre 200 ms y un segundo) y cierra cada conexión que lleve en silencio más tiempo que su plazo, junto con el socket del origen adjunto, de modo que el hilo bloqueado en cualquiera de los dos lados ve el fin del flujo y libera su posición.

Por tanto, los plazos miden el silencio, nunca la duración de una respuesta: una descarga grande que sigue avanzando nunca se corta. Los bytes de subida vienen del lado no fiable y solo renuevan la marca de tiempo cada 16 KiB, lo que exige al cliente al menos 16 KiB por periodo de inactividad: la misma defensa contra cuerpos lentos que dan los plazos de petición con tasa mínima de transferencia.

La tabla tiene un pequeño cerrojo por posición y ningún cerrojo para toda la tabla. Guarda un `try_clone` de cada socket de cliente y un `Arc` de cada socket de origen adjunto, por el motivo explicado arriba. Un `Lease` da de baja su conexión en `Drop`, y una posición recolectada sigue perteneciendo a su lease hasta entonces, así que una conexión antigua nunca puede liberar una posición que haya tomado una nueva.

Conexiones con cambio de protocolo

Tras un `101` validado, una conexión que cambia a WebSocket se retransmite en ambos sentidos. Cada sentido se ejecuta en un hilo propio con un único búfer fijo: el propio hilo de la conexión lleva los bytes del origen al cliente, y un hilo con ámbito (scoped) con una pila de 128 KiB lleva los bytes del cliente al origen. Un solo hilo podría servir ambos sentidos con `poll(2)`, pero la biblioteca estándar de Rust no lo expone; dos hilos con plazos de socket mantienen la retransmisión libre de `unsafe`. Una conexión que ha cambiado de protocolo ocupa, por tanto, dos hilos, así que el número de hilos de conexión puede llegar al doble de `--max-connections` cuando todas las conexiones han cambiado de protocolo.

Ambos sentidos comparten un mismo plazo de inactividad (`--websocket-idle-timeout`, 300 segundos por defecto): los bytes que se mueven en cualquier sentido renuevan una única marca de tiempo. Cada espera en un socket lleva un plazo igual al tiempo restante; una espera que vence vuelve a leer la marca compartida y espera de nuevo el resto o termina la retransmisión. Cuando un sentido llega al fin del flujo, se cierra el lado de envío del otro socket (un cierre parcial); cuando un sentido falla, se cierran ambos sockets para que el otro sentido también termine. Antes de pasar por el proxy un cambio de protocolo, la conexión libera su motor de políticas en caché, para que una conexión que puede vivir horas no mantenga vivo un motor antiguo.

Parada

`Running::stop`, e igualmente `Drop` para `Running`, detiene el daemon en un orden fijo. Activa el indicador de parada y despierta a cada worker de aceptación conectándose al listener, hace join de los workers, cierra cada socket de cliente registrado y su socket de origen adjunto

(`IdleTable::shutdown`), detiene el conjunto de conexiones al origen, hace join de cada hilo de conexión, hace join del recolector y cierra los sockets del conjunto.
SIGINT y SIGTERM llegan a esa secuencia con Rust corriente. Antes de lanzar ningún hilo, el daemon bloquea ambas señales (todos los hilos posteriores heredan la máscara) y dedica un hilo a `sigwait`, que registra la orden de parada y despierta de inmediato al hilo principal. No se ejecuta código en un manejador de señales, y ningún hilo consulta un indicador en bucle. El módulo de señales (`cmd/sibuna/src/signals.rs`) contiene el único código `unsafe` del daemon: las declaraciones de las pocas funciones de la biblioteca de C que `std` no envuelve.

Liberación mediante `Drop` Las llamadas explícitas de limpieza deben escribirse en orden inverso a la preparación, y cada retorno anticipado debe leerse teniéndolas en cuenta. En Sibuna, cada recurso se libera a sí mismo en `Drop`: `Daemon` y `Running` detienen el servidor, un `Lease` da de baja su conexión, `Counted` devuelve una plaza dentro del límite de conexiones y `Finished` libera una posición de tarea. El compilador inserta cada liberación en todas las salidas del ámbito, incluido un pánico que se propaga a través de él, así que ningún camino puede olvidar una.

Lo que medimos

La tabla muestra Shield en modo proxy inverso en un Ryzen 9 5950X: el daemon fijado a cuatro CPU, `wrk` con dos hilos y 64 conexiones en otras ocho CPU, ejecuciones de cinco segundos, la mediana de tres. Las filas se copian de `benchmarks/results/product-latest.md`; la parte VIII contiene el registro completo.

Tabla 7 Shield en modo proxy inverso, según `benchmarks/results/product-latest.md`

Carga de trabajo	Estado	req/s	p50 / p99 (us)	CPU por petición (us)
admitida con sesión, a través del proxy	200	138.965	438 / 857	28,57
desafiada (página intermedia)	200	195.188	162 / 328	13,73
estática permitida, a través del proxy	200	145.673	415 / 746	27,27
denegada por el WAF (con sesión)	403	209.507	151 / 306	12,17

`wrk` mantiene 64 peticiones en curso, así que en los caminos limitados por CPU a través del proxy las peticiones hacen cola para las cuatro CPU, y el p99 en ellos es la cola del planificador, no el coste de una petición. Un generador en lazo abierto, que envía a una tasa fija y mide cada petición desde su envío previsto, mide en cambio la latencia con la misma carga ofrecida. Para las peticiones admitidas en modo proxy inverso, el p99 es de 65 us a 10.000 peticiones por

segundo y de 100 us a 100.000; con 150.000 ofrecidas, las cuatro CPU se saturan, se alcanzan 134.344 peticiones por segundo y la cola crece hasta segundos (`benchmarks/results/latency-latest.md`). Una conexión keep-alive ya caliente sirve cada decisión sin una asignación en el heap: `cmd/sibuna/tests/steady_state.rs` cuenta las asignaciones del daemon, y `crates/net/tests/steady_state.rs` no encuentra ninguna en 500 peticiones keep-alive a través del proxy.

Ejercicio 5.1. Un cliente envía una subida de 200 KB. Traza qué bytes están en el búfer de 64 KiB, cuáles retransmite `relay_body` a través del búfer de subida y por qué el proxy debe escribir el encabezado hacia el origen *antes* de retransmitir.

Estado concurrente sin asignaciones

Objetivos Analiza los cerrojos por partición, el conjunto Robin Hood de desafíos gastados, el limitador GCRA, la tabla versionada de bloqueos, el anillo MPSC de incidentes y la publicación read-copy-update del motor de políticas con `Arc` y una caché comprobada por generación.

Cada almacén de `sibuna-store` asigna su capacidad fija una sola vez, al construirse, y después nunca asigna memoria ni se bloquea en un cerrojo global. Ninguno necesita un barrido en segundo plano: el estado caducado se recupera in situ.

Cerrojos y particiones

Las secciones críticas de las tablas de Sibuna tienen decenas de instrucciones, mucho menos que un viaje de ida y vuelta de un futex del kernel. Las tablas se dividen en 16 particiones según el hash de la clave, de modo que dos operaciones compiten con probabilidad 1/16, y cada partición ocupa su propia línea de caché, así que los cerrojos vecinos no sufren falso compartimiento (false sharing).

Las tablas usan `std::sync::Mutex`: una operación compare-and-swap si no hay contención, una breve espera activa mientras el titular se ejecuta y después una espera en futex. La alternativa es un spin lock atómico de dos estados con la pista de espera activa (spin-loop hint) en el bucle de espera. SID 0010 nos pidió elegir entre ambos a partir de mediciones. Un spin lock ahorra alrededor de un nanosegundo por operación sin contención, pero pierde con contención:

Tabla 8 Rendimiento de los cerrojos de partición en un Ryzen 9 5950X (octubre de 2026)

Carga de trabajo	`Mutex`	Spin lock
conjunto de desafíos gastados, 16 hilos	10,8 M/s	8,6 M/s
conjunto de desafíos gastados, 64 hilos (más hilos que CPU)	11,2 M/s	6,9 M/s

Carga de trabajo	`Mutex`	Spin lock
GCRA sobre seis direcciones calientes, 16 hilos	44,9 M/s	19,3 M/s
GCRA sobre seis direcciones calientes, 64 hilos	37,4 M/s	6,5 M/s

Un servidor perimetral ejecuta más hilos de conexión que CPU, y un spin lock cuyo titular es desalojado hace que cada hilo en espera agote su cuanto de tiempo, así que mantenemos el mutex. En un cerrojo envenenado se entra en lugar de propagar el envenenamiento: ninguna operación entra en pánico mientras tiene tomada una partición, y la partición sigue siendo estructuralmente válida aunque alguna lo hiciera.

El conjunto Robin Hood de desafíos gastados

Solo las pruebas aceptadas entran en el conjunto de desafíos gastados (parte III). Tiene 16 particiones, cada una un array de direccionamiento abierto de 4.096 posiciones de `Option<Entry>`, donde una entrada es `{ tag: [u8; 16], expires_at: u64, dist: u8 }`. La inserción Robin Hood (Celis, 1986) desplaza a cualquier ocupante más cercano a su posición inicial que la clave entrante, lo que mantiene muy agrupadas las longitudes de sondeo; las búsquedas se detienen en cuanto encuentran una entrada más cerca de su posición inicial de lo que ellas están de la suya.

Implementación: `Table::insert` Inserta una etiqueta gastada con su caducidad y recupera las posiciones caducadas que encuentra a su paso. En `crates/store/src/spent.rs` (crate `sibuna-store`).

```
fn insert(
    &mut self,
    tag: &Tag,
    expires_at: u64,
    now: u64,
) -> Result<(), SpendError> {
    if !self.can_insert(tag, now) {
        return Err(SpendError::Full);
    }
    let mut carry = Entry { tag: *tag, expires_at, dist: 0 };
    let mut idx = Self::home(tag);
    while carry.dist < MAX_PROBE {
        let slot = &mut self.entries[idx];
        let Some(e) = slot else {
            *slot = Some(carry);
            return Ok(());
        };
        // Sobrescribir un ocupante caducado solo es seguro si la distancia
        // nueva no es menor; así, las claves que sondean más allá de esta
        // posición siguen superando la prueba de terminación anticipada.
        if e.is_expired(now) && carry.dist >= e.dist {
            *e = carry;
```

```
            return Ok(());
        }
        if carry.dist > e.dist {
            std::mem::swap(&mut carry, e);
            // Una entrada caducada desplazada simplemente se descarta.
            if carry.is_expired(now) {
                return Ok(());
            }
        }
        idx = (idx + 1) % SHARD_CAPACITY;
        carry.dist += 1;
    }
    Err(SpendError::Full)
}
```

La regla de las posiciones caducadas es la parte sutil: una entrada caducada solo puede sobrescribirse in situ con una clave al menos igual de lejos de su posición inicial; de lo contrario, una clave posterior que sondee más allá de esa posición se detendría antes de tiempo y se perdería. Las entradas caducadas que resultan desplazadas simplemente se descartan. No hace falta ningún hilo de barrido. La inserción simula primero el camino de desplazamiento sin cambiar nada (`can_insert`) y comprueba que el desplazamiento acotado puede tener éxito sin perder una etiqueta vigente.

Invariante Una inserción rechazada nunca expulsa una etiqueta vigente. Por tanto, una etiqueta gastada y no caducada siempre se encuentra, y el rechazo (`SpendError::Full`) puede denegar el servicio, pero nunca permitir una reutilización.

Una posición es un `Option<Entry>`. Una posición vacía no tiene etiqueta que leer, así que no hay un indicador `occupied` que un lector pueda olvidar comprobar, y `match` no compila hasta que se tratan ambos casos. El tipo de error es igual de exacto: el conjunto de desafíos gastados puede fallar de dos maneras (`DoubleSpend` y `Full`), y `SpendError` nombra exactamente esas dos.

GCRA: limitar la tasa con un entero

El Generic Cell Rate Algorithm (ATM Forum, 1996) es la forma de planificación virtual del cubo con fugas. Para cada cliente guarda un único *tiempo teórico de llegada* (TAT). Con el intervalo de emisión $T = \max(1, \lceil W/N \rceil)$ para un límite positivo N por ventana W, y la tolerancia de ráfaga $\tau = (N-1)T$:

$$\text{la llegada en } t \text{ es conforme} \iff \mathrm{TAT} \le t + \tau, \quad \text{y entonces } \mathrm{TAT} \leftarrow \max(\mathrm{TAT}, t) + T.$$

Teorema (cota del cubo de tokens) Tras la primera llegada conforme en t_1, la k-ésima llegada conforme cumple $t_k - t_1 \ge (k-1)T - \tau$. Por tanto, cualquier intervalo de longitud L contiene como máximo $N + \lfloor L/T \rfloor$ llegadas conformes: una ráfaga de N y después una por cada T. No hay ningún artefacto de frontera de ventana fija más allá de la ráfaga definida, ni historial por petición.

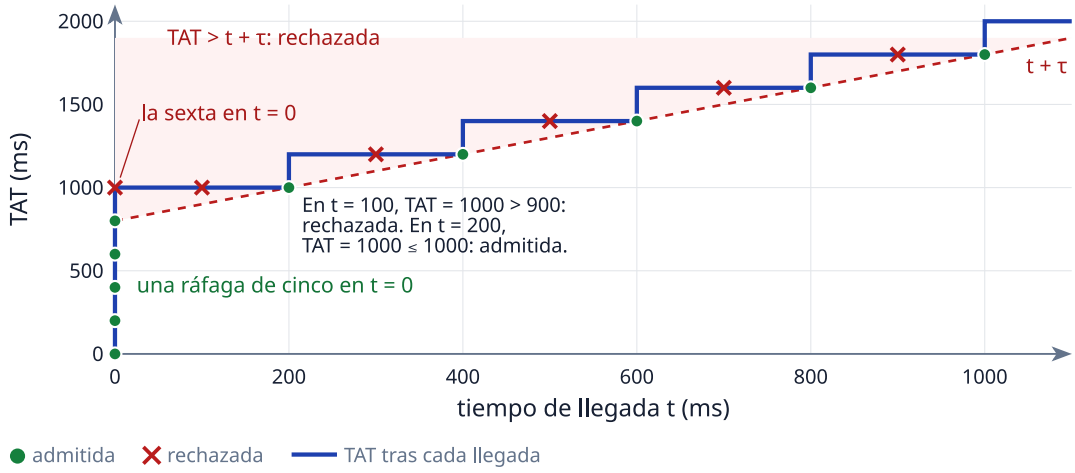


Fig. 7 Admisión GCRA con un límite de 5 por segundo: una ráfaga de cinco y después una cada 200 ms. Una llegada se admite cuando el TAT que encuentra está sobre la recta $t + \tau$ o por debajo de ella.

Ejemplo resuelto: cinco por segundo a mano Tomemos $N = 5$ y $W = 1000$ ms, de modo que $T = 200$ ms y $\tau = 800$ ms, y supongamos que un cliente inactivo envía seis peticiones en $t = 0$. La primera encuentra una celda vacía: $\mathrm{TAT} = \max(0, 0) = 0 \leq 800$, así que es conforme y $\mathrm{TAT} \leftarrow 200$. Las cuatro siguientes son conformes del mismo modo y dejan $\mathrm{TAT} = 1000$. La sexta encuentra $1000 > 0 + 800$ y se rechaza con `Retry-After` $\mathrm{TAT} - \tau - t = 200$ ms. En $t = 199$, una petición todavía encuentra $1000 > 999$ y se rechaza; en $t = 200$ encuentra $1000 \leq 1000$, es conforme y deja $\mathrm{TAT} = 1200$. La prueba unitaria `gcra_admits_exactly_the_burst_then_drains_at_the_emission_rate` de `crates/store/src/limiter.rs` comprueba exactamente estas cifras.

Implementación: `arrive` Aplica una llegada a las celdas de una partición cuyo cerrojo ya está tomado y devuelve la decisión, el retardo de reintento y el presupuesto restante. En `crates/store/src/limiter.rs` (crate `sibuna-store`).

```
fn arrive(
    cells: &mut [Cell; SLOTS_PER_SHARD],
    key: u64,
    now_ms: u64,
    limits: Limits,
) -> Decision {
    let interval = limits.emission_interval();
    let tau = limits.burst_tolerance();
    let Some(cell) = locate(cells, key, now_ms, tau) else {
        return Decision {
            limited: true,
            capacity_exhausted: true,
```

```
            retry_after_ms: interval,
            remaining: 0,
        };
    };
    let tat = cell.tat_ms.max(now_ms);
    let horizon = now_ms.saturating_add(tau);
    if tat > horizon {
        return Decision {
            limited: true,
            capacity_exhausted: false,
            retry_after_ms: tat - tau - now_ms,
            remaining: 0,
        };
    }
    cell.tat_ms = tat.saturating_add(interval);
    // ... remaining = min(rate, floor((t + tau - TAT) / T) + 1)
}
```

El `check` de la partición toma el cerrojo y llama a `arrive` con las celdas protegidas por él; el guardián libera el cerrojo cuando `check` vuelve. Las celdas ocupan 16 bytes (una clave y un TAT) en 16 particiones de 512 posiciones con una ventana de sondeo de 16 posiciones, unos 128 KiB en total. Una celda cuyo TAT es anterior a $t - \tau$ se ha vaciado y se recupera en el acto. La cabecera `Retry-After` se calcula con la misma aritmética. Las ventanas de sondeo saturadas rechazan clientes nuevos en lugar de expulsar estado de cuota activo, y la decisión lo indica (`capacity_exhausted`), para que quien llama no convierta un rechazo por saturación en un bloqueo. Una tasa configurada a cero rechaza las peticiones. Un único limitador sirve a varias reglas: la clave de la celda es un Wyhash de la dirección del cliente con la semilla del ámbito de la regla, de modo que el mismo cliente tiene presupuestos independientes en ámbitos distintos.

La tabla versionada de bloqueos

Los bloqueos usan 4.096 posiciones con campos atómicos de versión, clave y caducidad, con direccionamiento abierto mediante un hash con clave de la dirección bloqueada y una ventana de sondeo de 8 posiciones. Un escritor, serializado con los demás escritores mediante un mutex, rodea cada actualización con incrementos de versión; los lectores no toman ningún cerrojo, solo aceptan una versión par estable y reintentan si hay cambios concurrentes.

Implementación: `BanTable::ban` y `BanTable::read_slot` Los incrementos de versión que rodean la escritura y la instantánea coherente del lector. En `crates/store/src/ban_table.rs` (crate `sibuna-store`).

```
pub fn ban(&self, banned: &[u8], until: u64, now: u64) {
    let key = slot_key(banned);
    let _writers = self.lock_writers();
    let victim = &self.slots[self.victim(key, now)];
    // Una clave y una caducidad forman un solo valor lógico: la sustitución
    // ocurre bajo una versión impar, para que los lectores no puedan combinar
```

```
    // una clave antigua con la caducidad de la clave siguiente.
    victim.version.fetch_add(1, Ordering::SeqCst);
    victim.until.store(until, Ordering::SeqCst);
    victim.key.store(key, Ordering::SeqCst);
    victim.version.fetch_add(1, Ordering::SeqCst);
}

fn read_slot(&self, slot: &Slot) -> (u64, u64) {
    loop {
        let before = slot.version.load(Ordering::SeqCst);
        if before & 1 != 0 {
            std::hint::spin_loop();
            continue;
        }
        let key = slot.key.load(Ordering::SeqCst);
        let until = slot.until.load(Ordering::SeqCst);
        if before == slot.version.load(Ordering::SeqCst) {
            return (key, until);
        }
    }
}
```

La consistencia secuencial impide mezclar una identidad antigua con la caducidad de su sustituta: todo acceso pertenece a un único orden total, lo que mantiene sencillo el razonamiento. Los lectores no toman ningún mutex, pero pueden esperar a un escritor y no son wait-free. Estas operaciones atómicas tienen un coste real, medido indirectamente en el programa de medición HTTP en lugar de suponerlo nulo. Una ventana de sondeo llena sobrescribe una posición que haya caducado o, si no la hay, la que caduque antes. `clear` levanta todos los bloqueos pero deja las claves en su sitio, porque una posición vaciada podría ocultar una clave en colisión situada más adelante en una cadena de sondeo.

El anillo MPSC de incidentes

Cuando el modo de protección Shield deniega una petición o salta el honeypot, el incidente se copia en un registro fijo de unos 1,3 KB (dirección del cliente, User-Agent, método, ruta, categoría y carga útil, cada uno recortado a su capacidad) y se añade a un anillo acotado de múltiples productores y un solo consumidor (el diseño de Vyukov con números de secuencia, 512 celdas, `BoundedQueue<IncidentRecord, 512>`). Los productores son los hilos de conexión; el consumidor es el hilo de almacenamiento. Un anillo lleno descarta el registro más reciente en lugar de bloquear una respuesta: una política de pérdida explícita, no una garantía de durabilidad. El contador de descartes hace observable esa pérdida.

```
/// Copia `incident` en el anillo; si está lleno, lo descarta y lo cuenta.
pub(crate) fn record(&self, incident: &Incident<'_>) {
    if self.queue.push(IncidentRecord::from_incident(incident)).is_err() {
        self.metrics.incidents_dropped.bump();
    }
}
```

Tras mover un registro a un lote pendiente, el hilo de almacenamiento lo conserva, junto con las filas preparadas, hasta que se confirma la transacción; no lo descarta en silencio ante un error de la base de datos, y un reintento escribe exactamente el mismo lote.
Cada celda lleva un número de secuencia que indica a los productores si la celda está libre y al consumidor si está llena, así que ningún lado toma un cerrojo. Un anillo así no puede escribirse en Rust seguro: los productores y el consumidor tocan a la vez celdas distintas de un mismo array, y solo el protocolo de secuencia, no el sistema de tipos, demuestra qué hilo puede tocar qué celda. Por eso el módulo `crates/store/src/ring.rs` guarda sus valores en celdas `UnsafeCell<MaybeUninit<T>>`, y cada bloque `unsafe` nombra el paso del protocolo que lo justifica:

```
if diff == 0 {
    // Relaxed: el CAS solo decide qué productor obtiene la posición `pos`.
    match self.tail.0.compare_exchange_weak(
        pos, pos.wrapping_add(1), Ordering::Relaxed, Ordering::Relaxed,
    ) {
        Ok(_) => {
            // SAFETY: el CAS dio a este hilo la posición `pos`, y
            // `sequence == pos` indica que la celda está vacía; ningún otro
            // productor puede reclamar `pos` y el consumidor no la lee antes
            // del store Release de abajo.
            unsafe { (*slot.value.get()).write(value) };
            // Release: publica el valor escrito para el consumidor.
            slot.sequence.store(pos.wrapping_add(1), Ordering::Release);
            return Ok(());
        }
        Err(current) => pos = current,
    }
} else if diff < 0 {
    return Err(value);
}
```

Dos convenciones del anillo son tipos, no comentarios. Con el anillo lleno, `push` devuelve `Err(value)` y entrega de vuelta el registro para que quien llama decida si lo cuenta o lo descarta; un simple `false` lo perdería. El anillo tiene un único consumidor: `pop` toma `&mut self`, y el código compartido debe reclamar primero el `Consumer` único, de modo que un segundo consumidor no compila o se rechaza en tiempo de ejecución.

Publicación read-copy-update del motor

El motor de políticas se reconstruye fuera de la ruta crítica cuando cambian las políticas. Las peticiones nunca deben observar un motor a medio construir, y un motor nunca debe liberarse mientras una petición lo esté leyendo. Sibuna publica cada motor como un `Arc<Engine>` en un `EngineSlot`, y cada conexión lo lee a través de una `EngineCache`:

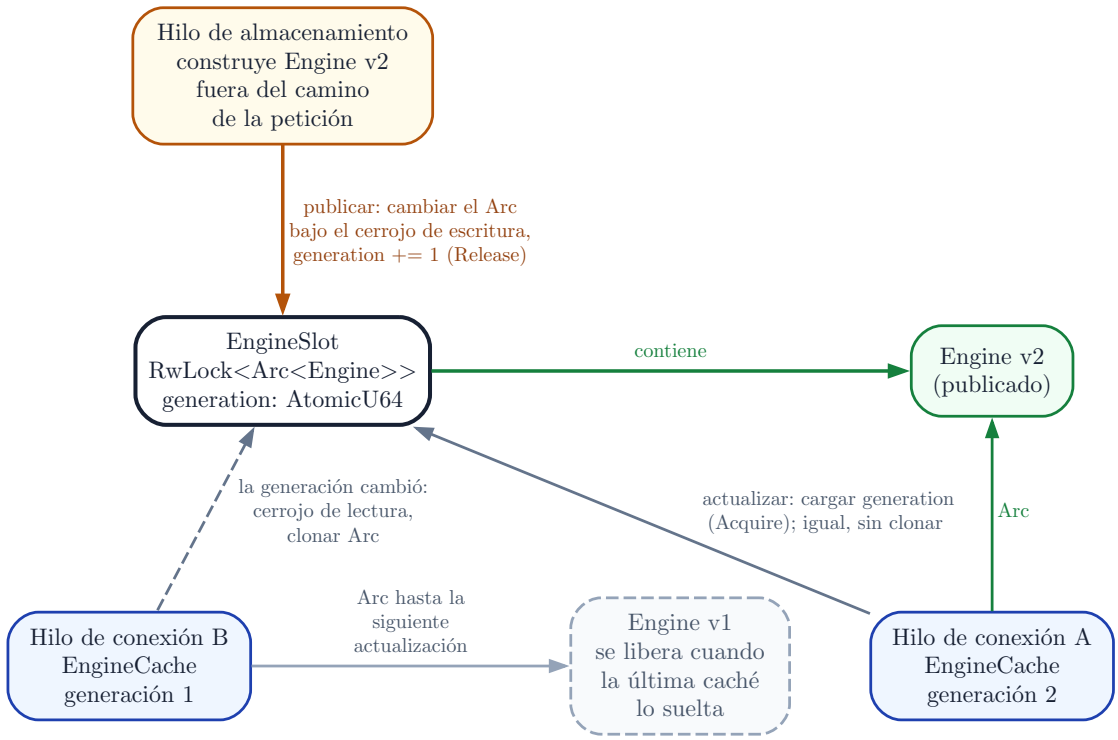


Fig. 8 Publicación del motor. El hilo de almacenamiento publica un motor completo y avanza la generación; cada conexión comprueba la generación con una carga acquire y clona el Arc publicado solo después de una publicación. Un motor antiguo se libera cuando lo suelta la última caché que lo retiene.

Implementación: `EngineSlot::publish` y `EngineCache::refresh` La publicación de un motor completo y la vista que tiene de él una conexión. En `cmd/sibuna/src/slot.rs` (crate `sibuna`).

```
pub struct EngineSlot {
    current: RwLock<Arc<Engine>>,
    generation: AtomicU64,
}

pub fn publish(&self, engine: Engine) {
    let fresh = Arc::new(engine);
    let previous = {
        let mut current =
            self.current.write().unwrap_or_else(PoisonError::into_inner);
        std::mem::replace(&mut *current, fresh)
    };
    self.generation.fetch_add(1, Ordering::Release);
    // El motor antiguo se libera fuera del cerrojo (si era su último titular).
    drop(previous);
}
```

```rust
pub struct EngineCache {
    generation: u64,
    engine: Option<Arc<Engine>>,
}

pub fn refresh(&mut self, slot: &EngineSlot) -> &Engine {
    let generation = slot.generation();
    if self.generation != generation {
        // Una publicación entre la carga de arriba y esta clonación etiqueta un
        // motor más nuevo con una generación más antigua; la siguiente
        // actualización simplemente vuelve a cargar.
        self.generation = generation;
        self.engine = None;
    }
    self.engine.get_or_insert_with(|| slot.load())
}
```

La generación se almacena con orden release una vez que el motor está en su sitio, y se carga con orden acquire: un lector que ve una generación nueva también ve el motor almacenado antes que ella. La carrera mencionada en el comentario es benigna. Si una publicación ocurre entre la carga de la generación por un lector y su clonación, el lector guarda en caché el motor más nuevo con el número de generación anterior, y su siguiente actualización detecta la discrepancia y vuelve a cargar. Un lector puede ir un motor por detrás durante una petición; nunca puede ver un motor a medio construir, porque en el espacio solo se almacenan motores completos.

Invariante Todo motor que lee una petición está completo, y ningún motor se libera mientras lo retenga alguna petición o caché de conexión. Lo primero se cumple porque un motor se construye por completo antes de que `publish` lo almacene; lo segundo, porque cada lector tiene un `Arc` y el motor solo se libera con su último `Arc`.

¿Por qué una caché y no simplemente un cerrojo? SID 0010 nos pidió medir el coste de lectura con 16 hilos y elegir entre `RwLock<Arc<Engine>>` y una alternativa sin cerrojos (lock-free) en Rust seguro. Leer a través del cerrojo toca la palabra del cerrojo y el contador de referencias, dos líneas de caché en las que escriben todos los núcleos. La generación, en cambio, se escribe una vez por publicación y por lo demás solo se lee, así que cada núcleo conserva su copia:

Tabla 9 Coste de una lectura del motor, Ryzen 9 5950X, 16 hilos leyendo en bucle (octubre de 2026)

Lector	1 hilo	16 hilos
solo el guardián de lectura de `RwLock`	4,2 ns	752 ns
lectura de `RwLock` y clonación de `Arc`	7,6 ns	833 ns
comprobación de generación con un `Arc` en caché	0,8 ns	1,8 ns

Con contención, la lectura en caché es más de 400 veces más barata que el cerrojo, así que el camino de la petición usa la caché, y el `RwLock` solo en el camino lento de una actualización. La medición es la prueba ignorada `read_cost_under_contention` de `cmd/sibuna/src/slot.rs`. Cada uso en el camino de la petición actualiza la caché, de modo que una petición ve el motor vigente en ese paso. El propio motor es pequeño: 209.052 bytes, unos 204 KiB, de tablas compactas en el heap (`engine_bytes` en `benchmarks/results/primitives-latest.json`), con capacidad para 128 reglas, nombres de regla de 128 bytes y 8.192 nodos del trie.

Por qué `Arc` y no contadores de lectores La alternativa de memoria fija mantiene dos espacios del motor, cada uno con un contador de lectores y su propia arena de asignación. Una petición fija un espacio incrementando su contador y volviendo a comprobar el puntero del espacio, y ambas operaciones atómicas deben ser secuencialmente consistentes para cerrar la carrera en la que un escritor hace el intercambio entre la carga del lector y su incremento. El constructor intercambia el puntero y después espera activamente a que se vacíen los lectores del espacio antiguo, antes de reiniciar la arena de ese espacio y reconstruir en ella. Un lector que retiene un espacio durante una reconstrucción bloquea, por tanto, al constructor, y un origen lento detendría la siguiente publicación a menos que cada petición liberara su espacio antes de empezar la E/S del proxy.
El recuento de referencias de `Arc` no necesita nada de esto. El hilo de almacenamiento construye un `Engine` nuevo a partir de cada instantánea y lo publica; no se reutiliza nada, así que no hay espacio de reserva, ni reinicio de arena, ni espera activa: `publish` nunca espera a los lectores. Un motor antiguo simplemente vive hasta que lo suelta su último titular, y un origen lento solo puede retrasar ese momento, nunca la siguiente publicación. El veredicto copia el nombre de la regla coincidente en un búfer de 128 bytes para terminar un préstamo: la decisión toma prestado el motor a través de la caché de la conexión, y el camino de la respuesta necesita todo el contexto de la conexión de forma mutable. Una conexión que ha cambiado de protocolo, y que puede vivir horas, libera su motor en caché antes de pasar por el proxy.

Ejercicio 5.2. Usando el teorema de GCRA, calcula el número máximo de peticiones que pueden admitirse a un solo cliente en los primeros 3 segundos tras un periodo de inactividad, con `--rate-limit 100 --rate-window 10`. Compáralo con un contador de ventana fija de 10 segundos y con un registro de ventana deslizante real.

Explica el invariante. Explica, con un ejemplo de tres posiciones, por qué una entrada Robin Hood caducada solo puede sobrescribirse con una clave cuya distancia sea al menos igual de grande.

PARTE VI

Autómatas, tries y el cortafuegos semántico

Examinamos los algoritmos que permiten clasificar en menos de un microsegundo: el autómata Aho-Corasick con etiquetas, el trie radix IPv4/IPv6, los analizadores por clases de bytes del WAF semántico y el motor declarativo de políticas con puntuación WEIGH.

Búsqueda de múltiples patrones en una pasada

Objetivos Al terminar este capítulo, deberíamos poder deducir la garantía de tiempo lineal del autómata Aho-Corasick, explicar la disposición en tabla densa y la conversión de mayúsculas mediante clases de bytes, y argumentar por qué no se adoptó un motor de literales SIMD.

Aho-Corasick como DFA denso

Sibuna tiene dos conjuntos de patrones literales: unas cuarenta firmas de bots y unas cien firmas de ataques. Comparar un campo con N patrones uno a uno cuesta $O(NM)$; el autómata de Aho y Corasick (1975) lo recorre una sola vez en $O(M)$ tras construir un trie con enlaces de fallo. Sibuna integra los enlaces de fallo en la tabla de transición al construirla, de modo que el recorrido hace una carga de tabla por byte de entrada, sin bifurcaciones sobre el conjunto de patrones. El autómata está en el crate `sibuna-patterns`:

```
/// Avanza desde `state` (un desplazamiento de fila) con `byte`.
fn step(&self, state: usize, byte: u8) -> usize {
    self.table[state + usize::from(self.classes[usize::from(byte)])] as usize
}

pub fn find_first(&self, haystack: &[u8]) -> Option<Match> {
    let mut state = 0;
    for (index, &byte) in haystack.iter().enumerate() {
        state = self.step(state, byte);
        if state >= self.first_match {
            let id = self.first_output_in(self.match_state(state),
self.all_tags)?;
            return Some(self.report(id, index + 1));
        }
    }
    None
}
```

Cada estado se guarda como el desplazamiento de su fila en una única tabla plana, así que un paso es una suma y una carga. Los estados se numeran de modo que todo estado de coincidencia va después de todos los demás; detectar una coincidencia es entonces una sola comparación con `first_match`, sin una segunda tabla que cargar en el bucle interno.

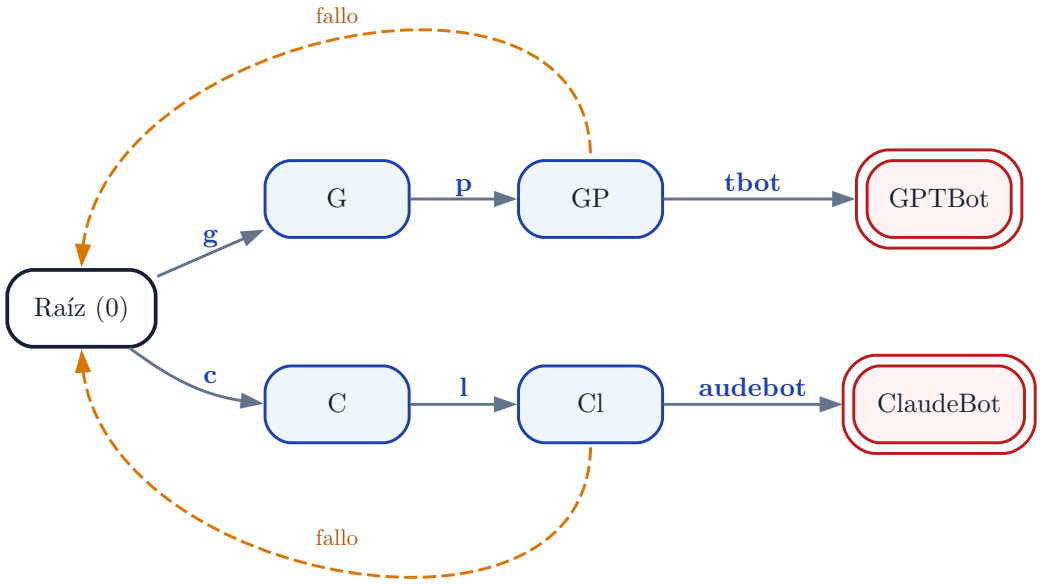


Fig. 9 Estados del trie y aristas de fallo para dos firmas

El presupuesto de estados del autómata es una opción de tiempo de ejecución. El motor de políticas reserva como máximo una raíz más la suma de las longitudes de todas las firmas, una cota segura del número de estados del trie. Así evita las tablas fijas históricamente sobredimensionadas y conserva una transición densa para cada clase de bytes. Cada patrón lleva una pequeña etiqueta de categoría (de 0 a 63, de modo que un conjunto de etiquetas cabe en una palabra de 64 bits). La conversión de mayúsculas usa una tabla de clases de bytes de 256 entradas, construida junto con el autómata: la mayúscula y la minúscula de una letra comparten clase, y los bytes que no aparecen en ningún patrón comparten una sola clase, de modo que una fila de la tabla tiene tantas columnas como bytes distintos haya en los patrones más uno, en lugar de 256. La parte VIII recoge los tamaños exactos del motor y la comparación con la búsqueda secuencial de subcadenas. Esta última usa los mismos patrones y la misma semántica de cualquier coincidencia.

Expresiones regulares, motores SIMD y la elección

Para la etapa de firmas se sopesaron tres familias:

- *Las expresiones regulares con retroceso* (al estilo de los conjuntos de reglas con retroceso) ejecutan cientos de patrones por petición y asignan memoria en cada coincidencia; son la opción más lenta y la más propensa a falsos positivos.
- *Los motores de literales SIMD* (Hyperscan, Vectorscan) prefiltran bloques de 16 a 32 bytes con máscaras de nibbles basadas en barajado (el algoritmo "Teddy") y verifican los candidatos con autómatas. Destacan con entradas largas y conjuntos grandes de literales.

El precio es una dependencia C++ de varios megabytes, una base de patrones compilada, la falta de un destino WebAssembly y, para entradas de unos cien bytes, ninguna ventaja: el campo entero se recorre en menos tiempo que un fallo de caché.
- *Aho-Corasick denso más pequeños analizadores estructurales*, en la línea de libinjection, ofrece un comportamiento lineal exacto en una tabla fija y no necesita nada fuera del ejecutable.

Los campos de Sibuna son cortos, así que la tercera familia gana tanto en memoria como en velocidad; la parte VIII registra unos 1,56 ns por byte en un cuerpo de 8 KiB (el recorrido semántico tarda 12.784,3 ns, `8kb_body_semantic_scan` en `benchmarks/results/primitives-latest.md`). Esa es la única carga de trabajo en la que un prefiltro SIMD compensaría, y el SID 0006 la mantiene como cuestión abierta.

El trie radix para IPv4 e IPv6

> **Objetivos** Sigue una búsqueda de prefijo más largo en el trie de 128 bits, explica el atajo de raíz IPv4 y observa cómo el trie sirve también de almacén de reputación alimentado por el clúster.

Un único trie binario sobre claves de 128 bits sirve a ambas familias: los prefijos IPv6 se insertan tal cual y los prefijos IPv4 se mapean a `::ffff:0:0/96`, de modo que un `/24` se convierte en un camino de profundidad 120. Los nodos viven en un único `Vec` plano indexado por `u32`, con el índice cero como raíz y como "sin hijo" a la vez, así que una búsqueda hace como máximo 128 cargas dependientes, sin perseguir punteros. El trie es el `PrefixTrie` del crate `sibuna-cidr`.
Recorrer los 96 niveles del prefijo mapeado en cada búsqueda IPv4 costaría 96 cargas dependientes antes de llegar al primer bit de la dirección IPv4. Por eso el trie guarda el nodo de profundidad 96 al inicializarse y empieza allí las búsquedas IPv4:

```
let (mut node, mut depth, mut best) = if is_ipv4_mapped(key) {
    let start = self.ipv4_root;
    let value = self.nodes[start as usize].value.as_ref();
    (start, IPV4_DEPTH, value.or(self.nodes[0].value.as_ref()))
} else {
    (0, 0, self.nodes[0].value.as_ref())
};
```

Clasificar una dirección IPv4 cuesta 32,0 ns y una dirección IPv6, 79,3 ns (`ipv4_cidr_classification` e `ipv6_cidr_classification` en `benchmarks/results/primitives-latest.md`). El motor consulta el trie dos veces: los veredictos terminales `allow`/`deny` antes de la tabla de reglas, de modo que un bloqueo en todo el clúster prevalece sobre cualquier regla, y los veredictos `challenge` después de ella.

El cortafuegos semántico

> **Objetivos** Comprende la pasada previa de clases de bytes, el autómata de firmas con etiquetas, el analizador SQL con detección de tautologías, las comprobaciones de estructura HTML y shell, la canonicalización y la disciplina de falsos positivos: una petición real de navegador debe pasar intacta.

Una lección sobre falsos positivos

La primera versión del WAF comparaba una lista plana de subcadenas con todas las cabeceras. La lista contenía `/*`, y todo navegador envía `Accept: text/html,...,*/*;q=0.8`. Todos los navegadores reales se clasificaban como inyección SQL y recibían `403`. La lección dio forma al diseño: un detector solo puede activarse ante *estructura*, y el conjunto de pruebas pasa ahora por el motor una petición completa de Chrome, incluido un cuerpo de comentario que contiene `select`, `--` y un apóstrofo, y comprueba que sale limpia.

Clases de bytes en una pasada

Cada campo inspeccionado se recorre una vez para recoger qué clases de bytes aparecen: comillas, `=`, `<`, separadores de shell, `%`, NUL, señales de canonicalización y espacios dobles. Cada detector consulta las clases en lugar de volver a recorrer el campo, y la segunda pasada canonicalizada solo se ejecuta si se vio una señal. El recorrido consulta una tabla de 256 entradas por byte, en el módulo WAF de `sibuna-policy`. Cuatro acumuladores mantienen cortas las cadenas de OR, y los dos pares de bytes (`/*` y dos espacios contiguos) solo se buscan cuando ningún byte aislado ha exigido ya la canonicalización:

```
pub(crate) fn scan(text: &[u8]) -> Classes {
    let (chunks, remainder) = text.as_chunks::<8>();
    // Cuatro acumuladores mantienen cortas las cadenas de OR para emitir las
cargas seguidas.
    let mut lanes = [0u16; 4];
    for chunk in chunks {
        lanes[0] |= TABLE[usize::from(chunk[0])] | TABLE[usize::from(chunk[4])];
        lanes[1] |= TABLE[usize::from(chunk[1])] | TABLE[usize::from(chunk[5])];
        lanes[2] |= TABLE[usize::from(chunk[2])] | TABLE[usize::from(chunk[6])];
        lanes[3] |= TABLE[usize::from(chunk[3])] | TABLE[usize::from(chunk[7])];
    }
    let mut bits = lanes[0] | lanes[1] | lanes[2] | lanes[3];
    for &byte in remainder {
        bits |= TABLE[usize::from(byte)];
    }
    // Los pares solo importan mientras ningún byte aislado haya exigido la
canonicalización.
    if bits & CANONICAL == 0
        && ((bits & SPACE != 0 && has_pair(text, b' ', b' '))
            || (bits & (SLASH | STAR) == SLASH | STAR && has_pair(text, b'/',
b'*')))
    {
        bits |= CANONICAL;
    }
```

```
    Classes { bits }
}
```

Detectores

- *Firmas.* Todas las firmas fuertes de todas las categorías están en un único autómata con etiquetas; una coincidencia es un `deny` terminal llamado `waf:sqli`, `waf:xss`, `waf:path-traversal` o `waf:rce`.
- *Inyección SQL.* Un recorrido de palabras en una pasada busca cada palabra alfanumérica (de siete bytes como máximo) en la tabla de palabras clave y, ante `or` / `and`, comprueba si hay una tautología `literal = literal`; `id=1 or 1=1` es terminal. En otro caso, es obligatorio un byte de comilla y la evidencia debe sumar al menos 4 puntos entre: comilla (1), hasta dos palabras clave (2), un marcador de comentario (1) y un `=` (1). `admin'--` es terminal por sí solo. "it's a group order from the shop" suma 3 y pasa.
- *Cross-site scripting.* Exige `<`; se activa cuando una etiqueta que puede alojar un script aparece junto con un atributo `on<event>=` en posición de atributo. `onboarding=1` no cuenta.
- *Inyección de comandos.* Exige un separador; se activa cuando `;`, `|`, `&&`, `$(`, un acento grave o un salto de línea van seguidos de un nombre de comando conocido. `status ok; done | next` pasa.
- *Recorrido de rutas.* Firmas (`../`, `..%2f`, `/etc/passwd`, `id_rsa`, ...) además de NUL y `%00`.

Se omiten las cabeceras estructurales cuya gramática contiene legítimamente comillas y asteriscos (`Accept*`, `Content-Type`, `Sec-*`, `If-*`, ...); se inspeccionan la ruta, la consulta, el User-Agent, las demás cabeceras y los primeros 8 KB del cuerpo, y los campos con escapes porcentuales, signos más, inicios de comentario o espacios colapsables se canonicalizan en un búfer de pila de 8 KB y se inspeccionan de nuevo, de modo que `1%27/**/UnIoN/**/SeLeCt` y `%252e%252e/` se reducen a las firmas sin codificar.

Coste

El coste de la inspección depende de los campos y bytes que recibe el motor, no solo del número de peticiones. El capítulo de pruebas de rendimiento presenta por separado los casos de Gate, Shield y cuerpos de 8 KB. La cota de 8 KB es un límite de inspección, no una afirmación de que el resto de una subida sea seguro. Un proxy de entrada que use forward-auth debe pasar la información relevante de la petición; este proceso no puede inspeccionar los bytes omitidos.

El motor declarativo de políticas

> **Objetivos** Lee el modelo de reglas, el orden de evaluación, la puntuación WEIGH con umbrales y la gramática JSON de políticas.

Una regla es una conjunción de criterios opcionales (patrón de ruta, patrón de User-Agent, hasta cuatro cabeceras, hasta ocho bloques CIDR IPv4/IPv6, una huella o un prefijo JA4) con una acción `ALLOW`, `DENY`, `CHALLENGE` o `WEIGH`, una dificultad opcional por regla en bits de trabajo y un algoritmo, y un `weight` con signo. Orden de evaluación:

1. WAF semántico (solo Shield): una infracción es un `deny` terminal.
2. Trie de reputación: los prefijos `deny` o `allow` son terminales.
3. Reglas en orden: la primera coincidencia terminal decide; las coincidencias `WEIGH` se acumulan.
4. Puntuación: los totales negativos dan `allow`; los totales ≥ 40 dan `deny`; los totales ≥ 10 dan `challenge` con un bit de trabajo extra por cada 5 puntos por encima del umbral, hasta un máximo de seis.
5. Rutas estáticas exentas, después los veredictos `challenge` del trie y después el autómata de bots.
6. Acción por defecto: `challenge`, salvo que el archivo de políticas diga otra cosa.

Las reglas integradas permiten `/.well-known/*`, `/favicon.ico`, `/robots.txt` y `/__sibuna/*`, deniegan los clientes `CF-Worker` y `Amazonbot`, y desafían todo User-Agent `Mozilla`: falsificar una cadena de navegador no cuesta nada, así que admitirla convertiría el control de admisión en un adorno.

```
{
  "default_action": "CHALLENGE",
  "waf": true,
  "thresholds": { "challenge_at": 10, "deny_at": 40, "bits_step": 5 },
  "ip_rules": { "10.0.0.0/8": "ALLOW", "2001:db8::/32": "DENY" },
  "rules": [
    { "name": "internal-api", "path": "/api/*", "headers": { "X-Api-Key": ".*" },
    "action": "ALLOW" },
    { "name": "checkout", "path": "/checkout/*", "action": "CHALLENGE",
      "challenge": { "difficulty": 20, "algorithm": "posw" } },
    { "name": "headless", "user_agent": "Headless", "action": "WEIGH", "weight":
30 },
    { "name": "no-language", "headers": { "Accept-Language": "" },
    "action": "WEIGH", "weight": -5 }
  ]
}
```

Traza resuelta: un ataque codificado con una sesión válida

Consideremos `GET /search?q=%2527%25200R%25201%253D1` desde un cliente con una sesión válida. Primero, el analizador HTTP separa la ruta y la consulta sin copiar sus bytes. Se aplican los límites locales y después Shield inspecciona la petición. El signo de porcentaje marca el campo para canonicalizarlo. Una pasada de decodificación deja al descubierto `%27%200R%201%3D1`; una segunda, `' OR 1=1`.

El analizador ve ahora una comilla, la palabra clave `OR` y operandos literales iguales. Una denegación es un resultado terminal de la política. La sesión válida no la convierte en un permiso: la comprobación de sesión solo resuelve una decisión de desafío. El gancho de almacenamiento copia un registro de incidente acotado; el SQL se ejecuta más tarde. El camino de la respuesta no necesita retener la instantánea del motor mientras una conexión con el origen está inactiva.

Ejercicio 6.1. Sustituye la carga útil por `O'Reilly` y después por `order=1`. ¿Por qué ninguna de las dos debería bastar por sí sola para establecer una inyección SQL? Da un ejemplo de falso positivo que podría producir un detector basado solo en subcadenas.

Ejercicio 6.2. Escribe las entradas de la tabla de clases de bytes necesarias para añadir un detector de inyección LDAP (patrones como `)(|(`) y explica qué bits de clase existentes puede reutilizar.

Explica el invariante. Explica a un revisor por qué `Accept: */*` era un falso positivo, qué propiedad estructural exige el nuevo detector SQL y cómo se reparten el trabajo el autómata y el analizador.

PARTE VII

El motor de prueba de trabajo del navegador

Examinamos el módulo WebAssembly que compila el propio crate verificador del servidor, el protocolo del Web Worker, los probadores JavaScript de respaldo y la página intermedia.

Una fuente, dos destinos

Objetivos Al terminar este capítulo, deberíamos poder explicar cómo se construye el módulo del navegador a partir del propio crate `sibuna-pow` del servidor, nombrar las funciones exportadas y describir la memoria que necesita en una pestaña el probador de trabajo secuencial.

Integración en la compilación

El solucionador del navegador se compila a partir de las mismas fuentes con las que verifica el servidor. `cmd/sibuna-pow-wasm` es un crate de biblioteca `no_std` de tipo `cdylib`, compilado para `wasm32-unknown-unknown`, cuyas únicas dependencias son los crates del workspace `sibuna-pow` y `sibuna-crypto`:

```
# cmd/sibuna-pow-wasm/Cargo.toml
[lib]
crate-type = ["cdylib", "rlib"]

[dependencies]
sibuna-crypto = { workspace = true }
sibuna-pow = { workspace = true }

# Cargo.toml (raíz del workspace): el módulo se compila para ocupar poco
[profile.release.package.sibuna-pow-wasm]
opt-level = "z"
debug = false
strip = true
```

El daemon integra el módulo. Su script de compilación (`cmd/sibuna/build.rs`) ejecuta un `cargo rustc -p sibuna-pow-wasm --target wasm32-unknown-unknown --profile wasm` anidado en el directorio de destino del workspace (`target/wasm32-unknown-unknown/wasm/`). La compilación anidada solo bloquea los directorios del perfil `wasm`, así que nunca espera a la compilación externa. El enlazado final recibe `--compress-relocations`. El script comprueba que el resultado empieza por los bytes mágicos de WebAssembly y lo copia al directorio de salida de la compilación. `SIBUNA_POW_WASM=<path>` integra en su lugar un módulo precompilado, para compilaciones sin el destino WebAssembly. Después, el daemon incluye los bytes en su binario:

```
/// El solucionador del navegador, compilado por `build.rs` desde
/// `cmd/sibuna-pow-wasm`.
pub const WASM_SOLVER: &[u8] =
    include_bytes!(concat!(env!("OUT_DIR"), "/sibuna-pow.wasm"));
/// El Web Worker del solucionador.
pub const WORKER_JS: &str = include_str!("../../../web/worker.js");
```

No hay una segunda implementación que pueda divergir. El daemon sirve el módulo en `/__sibuna/wasm/sibuna-pow.wasm` con una cabecera de caché de una hora (`Cache-Control: public, max-age=3600`). Compilado con el perfil anterior, el módulo ocupa 14.744 bytes con ambos niveles (`wasm_bytes` en `benchmarks/results/primitives-latest.json`).

Exportaciones

El módulo exporta seis funciones, y el script del worker las llama por su nombre:

Exportación	Función
`sibuna_get_buffer_ptr()` / `sibuna_get_buffer_len()`	Búfer de entrada de 256 bytes en el que el worker escribe la cadena del desafío
`sibuna_solve_step(ptr, len, bits, start, max_steps)`	Hashcash: prueba `max_steps` nonces a partir de `start`; devuelve el nonce o el centinela con todos los bits a uno
`sibuna_solve_sha256(ptr, len, bits)`	Hashcash en un solo lote desde el nonce 0 (`sibuna_solve_step` con `u32::MAX` pasos)
`sibuna_posw_solve(len, depth, challenges)`	Trabajo secuencial: ejecuta el probador sobre el búfer y devuelve la longitud de la prueba, o 0 si los argumentos no son válidos
`sibuna_posw_proof_ptr()`	Inicio de los bytes de la prueba en el espacio de trabajo del probador

El espacio de trabajo del probador (los niveles superiores retenidos, la pila de nodos hermanos y la prueba) es una región estática de unos 92 KiB; no se enlaza ningún asignador y el módulo no tiene importaciones. `Workspace::new` es una `const fn`, así que la variable estática solo ocupa memoria puesta a cero, no datos inicializados dentro del módulo.

> **Implementación: `sibuna_solve_step`** La exportación de Hashcash: toma prestado el desafío de la memoria lineal y ejecuta un lote de la búsqueda de nonces compartida. En `cmd/sibuna-pow-wasm/src/lib.rs` (crate `sibuna-pow-wasm`).

```
#[unsafe(no_mangle)]
pub unsafe extern "C" fn sibuna_solve_step(
    challenge_ptr: *const u8,
    challenge_len: usize,
    difficulty_bits: u32,
```

```
    start_nonce: u64,
    max_steps: u32,
) -> u64 {
    let challenge: &[u8] = if challenge_len == 0 {
        &[]
    } else {
        // SAFETY: quien llama garantiza `challenge_len` bytes legibles en
        // `challenge_ptr` que no cambian mientras dura la llamada.
        unsafe { core::slice::from_raw_parts(challenge_ptr, challenge_len) }
    };
    Search::new(challenge)
        .find(difficulty_bits, start_nonce, u64::from(max_steps))
        .unwrap_or(EXHAUSTED)
}
```

Las exportaciones son la frontera FFI (de funciones externas) del crate y el único lugar donde hay código `unsafe`. El búfer de entrada y el espacio de trabajo del probador son variables estáticas dentro de un `ExportCell`, un envoltorio de `UnsafeCell` declarado `Sync` con un motivo explícito: una instancia WebAssembly sin memoria compartida ejecuta un solo hilo, y cada exportación crea sus referencias, las usa y las suelta antes de volver, así que nunca hay más de una referencia viva a una celda. En el host, donde las pruebas llaman directamente a las exportaciones, un mutex que solo existe en las pruebas las serializa. Un pánico provoca una trampa (la instrucción `unreachable`) en lugar de formatear un mensaje, lo que mantiene el módulo libre de código de formateo y de importaciones; los pánicos son inalcanzables por construcción, porque las funciones de `sibuna-pow` son totales y nunca entran en pánico.

Hash previo del prefijo

El bucle interno de Hashcash absorbe el desafío y los dos puntos una sola vez y clona el estado SHA-256 para cada nonce, de modo que cada candidato cuesta una compresión en lugar de dos. La búsqueda es `sibuna_pow::hashcash::Search`, compartida por el módulo del navegador y el solucionador nativo de referencia:

```
pub fn new(challenge: &[u8]) -> Self {
    let mut prefix = Sha256::new();
    prefix.update(challenge);
    prefix.update(b":");
    Self { prefix }
}

pub fn digest(&self, nonce: u64) -> Digest {
    let mut buf = [0u8; NONCE_DIGITS_MAX];
    let mut hasher = self.prefix.clone(); // copia del prefijo absorbido
    hasher.update(format_decimal(nonce, &mut buf));
    hasher.finalize()
}

pub fn find(&self, bits: u32, start: u64, steps: u64) -> Option<u64> {
    let mut nonce = start;
    for _ in 0..steps {
        if meets_bits(&self.digest(nonce), bits) {
```

```
            return Some(nonce);
        }
        nonce = nonce.wrapping_add(1);
    }
    None
}
```

El estado del hasher es un valor simple de unos 110 bytes, así que clonarlo es barato. El probador de trabajo secuencial usa el mismo truco, con el enunciado χ absorbido una vez en `base_hasher` y clonado para cada nodo, y ahora el verificador también: deriva cada hoja desafiada de un estado que ya ha absorbido el enunciado una vez, lo que ahorra t compresiones en la verificación con los mismos resúmenes.

En el servidor, la función de compresión usa las extensiones SHA de x86-64 o las instrucciones SHA-2 de ARMv8, detectadas en tiempo de ejecución. En el navegador no existen esas instrucciones, y el solucionador pasa casi todo su tiempo en la función de compresión portable (`crates/crypto/src/sha256/portable.rs`). Esa función está escrita para WebAssembly: las 64 rondas están desenrolladas, la expansión del mensaje ocupa una ventana de 16 palabras en la que cada ronda a partir de la 16 sobrescribe la palabra que ya no necesita, y las ocho variables de trabajo se renombran de una ronda a otra en lugar de desplazarse:

```
fn compress_block(state: &mut State, block: &Block) {
    let mut w = [0u32; 16];
    for (word, bytes) in w.iter_mut().zip(block.as_chunks::<4>().0) {
        *word = u32::from_be_bytes(*bytes);
    }
    let [mut a, mut b, mut c, mut d, mut e, mut f, mut g, mut h] = *state;
    eight_rounds!(w, 0, a, b, c, d, e, f, g, h);
    eight_rounds!(w, 8, a, b, c, d, e, f, g, h);
    // ... rondas 16 a 55
    eight_rounds!(w, 56, a, b, c, d, e, f, g, h);
    for (word, value) in state.iter_mut().zip([a, b, c, d, e, f, g, h]) {
        *word = word.wrapping_add(value);
    }
}

/// Devuelve W(i) de la expansión; desde la ronda 16 la calcula in situ.
#[inline(always)]
fn schedule(w: &mut [u32; 16], i: usize) -> u32 {
    if i >= 16 {
        let w15 = w[(i + 1) % 16];
        let w2 = w[(i + 14) % 16];
        let s0 = w15.rotate_right(7) ^ w15.rotate_right(18) ^ (w15 >> 3);
        let s1 = w2.rotate_right(17) ^ w2.rotate_right(19) ^ (w2 >> 10);
        w[i % 16] = w[i % 16]
            .wrapping_add(s0)
            .wrapping_add(w[(i + 9) % 16])
            .wrapping_add(s1);
    }
    w[i % 16]
}
```

Cada `eight_rounds!` es una vuelta completa de los nombres de las variables: la macro los pasa rotados una posición por ronda, de modo que una ronda escribe solo dos de ellos y nunca mueve los otros seis. Con un índice de ronda literal, cada posición de la expansión es una constante, y el bloque entero se convierte en código lineal cuyos valores permanecen en variables locales de WebAssembly. Esta disposición es aproximadamente un 20% más rápida en V8 que una expansión de 64 palabras con un bucle sin desenrollar, y aproximadamente un 14% más rápida en ejecución nativa (medido sobre la implementación en Rust, octubre de 2026).

El protocolo del worker

> **Objetivos** Lee el protocolo de mensajes, comprende la jerarquía de respaldo y observa el error del centinela que causó la frontera de enteros con signo entre WebAssembly y JavaScript.

La página envía un mensaje a `web/worker.js` y recibe mensajes de progreso, de respaldo y de resultado:

```
// página -> worker
{ challenge: spec.id, algorithm: spec.algorithm, difficulty: spec.difficulty,
    challenges: spec.challenges }

// worker -> página
{ type: 'progress', iterations: 40000 }
{ type: 'fallback', message: 'Wasm unavailable; running JS prover' }
{ type: 'solved', challenge: spec.id, solution: { nonce: '90766' } }  // hashcash
{ type: 'solved', challenge: spec.id, solution: { proof: '<base64url>' } } // posw
{ type: 'error', message: '...' }
```

El worker prueba primero WebAssembly y recurre a JavaScript puro cuando `WebAssembly` no existe, está bloqueado por una política o no consigue instanciarse. Los probadores JavaScript reproducen etiqueta por etiqueta el probador Rust de `sibuna-pow` y producen una salida *idéntica byte a byte*. Las pruebas fijan nonces y pruebas concretos para ambos niveles, y SID 0010 exige que el solucionador funcione en un navegador sin interfaz gráfica tanto con WebAssembly como con el respaldo JavaScript. Los probadores JavaScript son mucho más lentos, pero garantizan que cualquier navegador pueda superar el desafío.

> **El centinela con signo** WebAssembly devuelve a JavaScript un entero de 64 bits como un `BigInt` *con signo*. La exportación de Hashcash señala "lote agotado" con todos los bits a uno (`EXHAUSTED`, `u64::MAX`), que JavaScript ve como `-1n`, no como `18446744073709551615n`. El worker original comparaba con el valor sin signo, así que el bucle terminaba tras el primer lote con un nonce falso. La corrección es una sola llamada, `BigInt.asUintN(64, result)`, y el worker la conserva:
>
> ```
> // Los resultados i64 de WebAssembly llegan como BigInt con signo, así
> // que el centinela u64 "exhausted" (todos los bits a uno) se lee como -1n.
> ```

```
const found = BigInt.asUintN(64,
    ex.sibuna_solve_step(ptr, encoded.length, bits, nonce, maxSteps));
if (found !== 18446744073709551615n) return found.toString();
```

La lección es general: todo u64 que cruza la frontera debe normalizarse. Rust no puede imponerlo en el lado de JavaScript; el tipo de la exportación dice `u64`, y la conversión ocurre en el motor.

Calibración

Objetivos Relaciona los ajustes de dificultad con el tiempo real en un motor de navegador y en ejecución nativa.

No elijas una dificultad a partir del tiempo medio de resolución de un portátil. Hashcash tiene una cola geométrica; los relojes del navegador incluyen planificación, estado térmico, arranque y compilación. Una calibración de despliegue debe registrar la versión del navegador, el dispositivo, el algoritmo, los parámetros y una distribución de resoluciones repetidas. Mantén la descarga y compilación del módulo separadas del cálculo de hashes en régimen estable.

Plan de calibración resuelto Mantén b fijo, calienta el módulo y resuelve al menos 100 desafíos emitidos de forma independiente en cada clase de dispositivo objetivo. Registra las duraciones mediana y de percentiles altos, los fallos y los tamaños de prueba. Repite con $b + 1$. El número esperado de intentos se duplica; una muestra individual no tiene por qué hacerlo. Elige un ajuste que respete el presupuesto de tiempo de los dispositivos lentos y después prueba el flujo completo de descarga, resolución y verificación.

Las mediciones de primitivas nativas del repositorio no demuestran ningún tiempo de resolución que pueda prometerse en un teléfono o un navegador. Un módulo WASM puede ejecutar la misma aritmética con costes distintos de arranque, ejecución y memoria: en ejecución nativa, el servidor calcula hashes con SHA-NI, mientras que el navegador ejecuta la función de compresión portable descrita arriba. Los mensajes del worker forman la frontera en la que esos costes pueden medirse sin añadir un temporizador al manejador de peticiones del servidor.

La página intermedia

La página (`web/challenge.html`, integrada en el daemon) es un único archivo HTML con una tarjeta, una barra de progreso indeterminada y una línea de estado. Pide el desafío para su propia ubicación (`?path=`), lanza el worker, envía la solución con `fetch` y recarga si tiene éxito; las llamadas `fetch` se reintentan con espera exponencial con variación aleatoria, y una respuesta de verificación `4xx` se muestra con su título de diagnóstico. La `Content-Security-Policy` de la página admite exactamente un script, el bloque del solucionador, por su hash

SHA-256, que el daemon calcula una vez a partir de la página integrada. Un enlace honeypot fuera de pantalla (`/__sibuna/honeypot`) es invisible para las personas y para los navegadores que ejecutan scripts; los rastreadores ciegos que siguen cada `href` se bloquean a sí mismos.

Ejercicio 7.1. Supongamos que el probador WebAssembly resuelve un desafío con la dificultad por defecto en 20 ms en un portátil, y que el probador secuencial JavaScript es 50 veces más lento. Estima el tiempo de resolución con el respaldo en ese portátil y en un teléfono cuatro veces más lento, y propón una política para los clientes que informan del respaldo (pista: el worker envía un mensaje `fallback` que la página puede reenviar).

Explica el invariante. Explica por qué compilar el módulo del navegador a partir de las fuentes del verificador del servidor es una garantía más fuerte que probar dos implementaciones una contra otra.

PARTE VIII

Evaluación empírica

Presentamos las mediciones de Sibuna, registradas por programas de medición incluidos en el repositorio: latencias de primitivas, productos completos bajo un generador de carga externo con contabilidad de CPU y memoria, latencia con la misma carga ofrecida desde un generador en lazo abierto, y una comparación con Anubis, un control de admisión por prueba de trabajo ampliamente desplegado. Una ejecución de la consola con ocho paneles en vivo completa el registro. Cada número de esta parte procede de un archivo de resultados incluido en el repositorio, que cada tabla nombra junto con su hora de generación, o de un registro de diseño. Una prueba funcional superada y una medición de rendimiento responden a preguntas distintas, y una medición en un host no es una clasificación en otro.

Metodología

Objetivos Al terminar este capítulo, deberíamos poder reproducir cada número de esta parte, decir qué incluye y qué excluye cada programa de medición, explicar qué significan las columnas de dispersión y distinguir una cifra medida de una registrada o publicada.

Los programas de medición son comandos de una sola herramienta en Rust, `tools/bench` (el binario `sibuna-bench`), que se ejecuta mediante `cargo xtask bench`. Otra herramienta registra la ejecución de la consola.

Tabla 11 Programas de medición y sus archivos de resultados, todos bajo `benchmarks/results/`

Programa	Qué mide	Archivo de resultados
`cargo xtask bench primitives`	Dieciséis cargas de trabajo de primitivas, más dos filas para la ruta del limitador de tasa con cliente en hash, en una CPU: siete lotes, mediana por operación con la dispersión mín-máx	`primitives-latest.json`, `primitives-latest.md`
`cargo xtask bench product`	Daemons completos bajo `wrk`: rendimiento, percentiles de latencia, tiempo de CPU por petición y memoria residente máxima de Gate y Shield, en los modos forward-auth y proxy inverso	`product-latest.json`, `product-latest.md`
`cargo xtask bench latency`	Percentiles de latencia de las peticiones admitidas a tasas ofrecidas fijas, desde un generador en lazo abierto	`latency-latest.json`, `latency-latest.md`

Programa	Qué mide	Archivo de resultados
`cargo xtask bench product --anubis`	Las cargas de trabajo de producto completo con Anubis como producto adicional, detrás de `caddy respond`	`comparison-latest.json`, `comparison-latest.md`
`cargo xtask bench admission --anubis`	Operaciones de admisión por segundo de Sibuna y Anubis en modo forward-auth, desde dos clientes keep-alive	`admission-latest.json`, `admission-latest.md`
`sibuna-browser dashboards` (`tools/browser`)	Una ejecución funcional de la consola: ocho paneles en vivo en Chromium sin interfaz gráfica mientras un generador envía tráfico	`console-8-dashboards-2026-10-15.md`

Los archivos `.md` son resúmenes. Los archivos `.json` contienen cada repetición, la máquina, la revisión y la ruta, el tamaño y el SHA-256 de cada binario medido. Git conserva el historial, así que el directorio guarda solo los archivos más recientes. Algunos archivos los registró una versión anterior del programa de medición y contienen además filas de una segunda compilación del daemon medida en la misma ejecución; esta parte presenta las filas de Sibuna.

Primitivas

Cada carga de trabajo se ejecuta en siete lotes cronometrados tras un calentamiento sin cronometrar de como máximo 1.000 operaciones. El estado mutable (el conjunto de desafíos gastados, el limitador de tasa) se reconstruye antes de cada lote, fuera del cronómetro. Los cronómetros rodean lotes completos, nunca operaciones sueltas: leer el reloj en cada operación costaría tanto como el trabajo de menos de 100 ns que mide. Por eso no damos percentiles por operación, y la dispersión se mide entre lotes. Damos el lote mediano por operación, con el lote más rápido y el más lento a su lado.

El conjunto de pruebas se ejecuta en un proceso hijo (`sibuna-bench primitives-run`) fijado a una CPU. En un host compartido, un hilo hermano SMT ocupado o un reloj más bajo ralentiza una ejecución completa, así que la herramienta ejecuta el conjunto tres veces (`--rounds`) y conserva, por fila, la ronda con la mediana más baja. Dentro de una ronda, las estadísticas son las descritas arriba.

Productos completos

Para cada modo (forward-auth, proxy inverso) y cada modo de protección (Gate, Shield), el programa de medición arranca el daemon en memoria con `--algorithm hashcash --difficulty 8 --rate-limit 100000000 --idle-timeout 5` y cuatro workers, uno por CPU de su conjunto. Obtiene una sesión resolviendo el desafío Hashcash del daemon con `sibuna-pow`, como lo haría un navegador. Después ejecuta cuatro cargas de trabajo con `wrk` sobre conexiones keep-alive: 2 hilos, 64 conexiones, 5 segundos, 3 repeticiones.

- `admitted`: `GET /private` con una cookie de sesión válida. En modo proxy inverso la petición llega al origen y su respuesta se retransmite; en modo forward-auth el daemon responde `200` por sí mismo.

- `challenged`: la misma petición sin cookie, desde un User-Agent de navegador. El daemon sirve su desafío: `200` con HTML en modo proxy inverso, `401` en modo forward-auth.
- `allowed_static`: `GET /robots.txt` desde `curl/8.0`, que ambos modos de protección admiten sin desafío.
- `attack`: una sesión válida más `q=' OR 1=1--`. Solo Shield la rechaza (`403`). Gate la admite y, en modo proxy inverso, la retransmite al origen.

El origen es un sustituto (`sibuna-bench origin`): solo biblioteca estándar, un hilo por conexión, keep-alive y una página de 41 bytes. El tiempo de CPU por petición es el tiempo de usuario más el de sistema del daemon, tomado de `/proc/<pid>/stat` durante la ejecución (ticks de 10 ms), dividido entre las peticiones que contó `wrk`; la CPU del origen no se incluye. La memoria máxima es `VmHWM` de `/proc/<pid>/status`, reiniciada mediante `/proc/<pid>/clear_refs` antes de cada ejecución, para que cada ejecución informe de su propio máximo. Cada cifra es la mediana de las repeticiones, con el mínimo y el máximo guardados a su lado en el archivo JSON.

`wrk` es un generador en lazo cerrado: mantiene 64 peticiones en curso. Por tanto, un daemon más rápido recibe más peticiones, y los percentiles de `wrk` omiten las peticiones que no envió mientras esperaba una respuesta lenta (omisión coordinada). Su p99 muestra cómo se comporta el daemon bajo su propia saturación, no su latencia con una carga dada. El programa de medición de latencia responde a esa pregunta.

Latencia con la misma carga

El generador en lazo abierto (`sibuna-bench generate`) ejecuta un hilo bloqueante por conexión. Cada una de las 64 conexiones envía `GET /private` con una sesión según un calendario fijo: la petición k de una conexión vence en $t_0 + o + kT$, donde o es el desfase de la conexión y T su intervalo, con independencia de las respuestas anteriores. Cuando una respuesta llega después de que venciera la siguiente petición, esa petición sale de inmediato y su latencia cuenta desde su hora de vencimiento, como en `wrk2`. Cuando la conexión estaba inactiva a la hora de vencimiento, la latencia cuenta desde el envío real, de modo que el exceso de espera del propio generador no se carga al daemon.

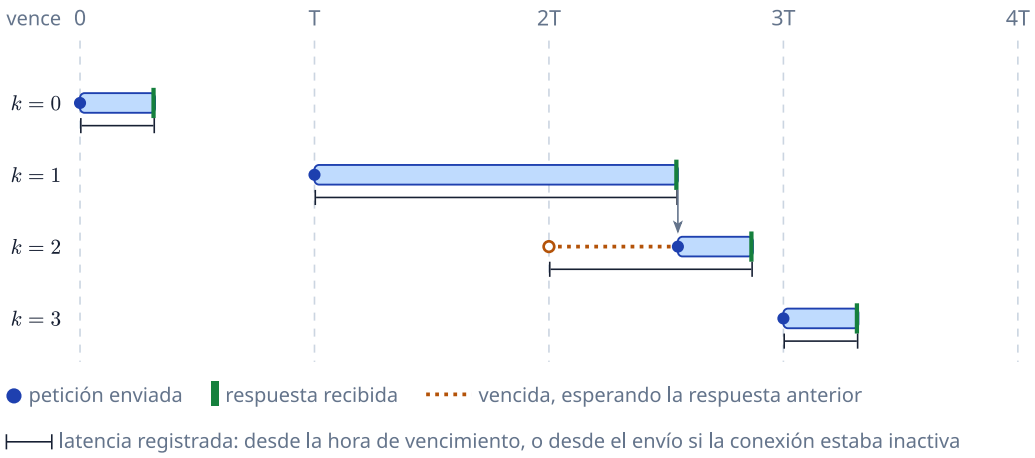


Fig. 10 Una conexión del generador en lazo abierto. La petición 1 se responde después de que venciera la petición 2, así que la petición 2 se envía de inmediato y su latencia cuenta desde 2T; la petición 3 encuentra la conexión inactiva y cuenta desde su envío.

El programa de medición ejecuta el modo de protección Shield y la carga admitida a cinco tasas ofrecidas: 10.000, 25.000, 50.000, 100.000 y 150.000 peticiones por segundo, cada una con 2 segundos de calentamiento, 10 segundos medidos y 3 ejecuciones. El programa puede medir dos compilaciones del daemon en una ejecución: entonces arranca ambas en el mismo conjunto de CPU y alterna entre ellas en cada tasa y repetición, sin que la inactiva use CPU, para que la deriva lenta de la máquina afecte a ambas por igual. La ejecución registrada lo hizo así. Los percentiles son de rango más cercano sobre todas las peticiones medidas de una ejecución; cada cifra es la mediana de las tres ejecuciones, con el mín-máx del p99 a su lado.

Conjuntos de CPU

Los conjuntos de CPU importan más que cualquier otro ajuste. Los daemons se ejecutan en las CPU 0 a 3 (`--cpus`; `--workers` es el tamaño de ese conjunto). `wrk`, el generador de latencia y el origen se ejecutan en las CPU 8 a 15 (`--load-cpus`). El conjunto de primitivas se ejecuta en la CPU 4 (`--cpu`). Cada proceso se fija con `taskset -c` antes de arrancar, de modo que cada hilo que crea hereda el conjunto. El programa de medición de admisión usa dos CPU, 0 y 1, para el producto, y las CPU 8 a 15 para sus clientes.

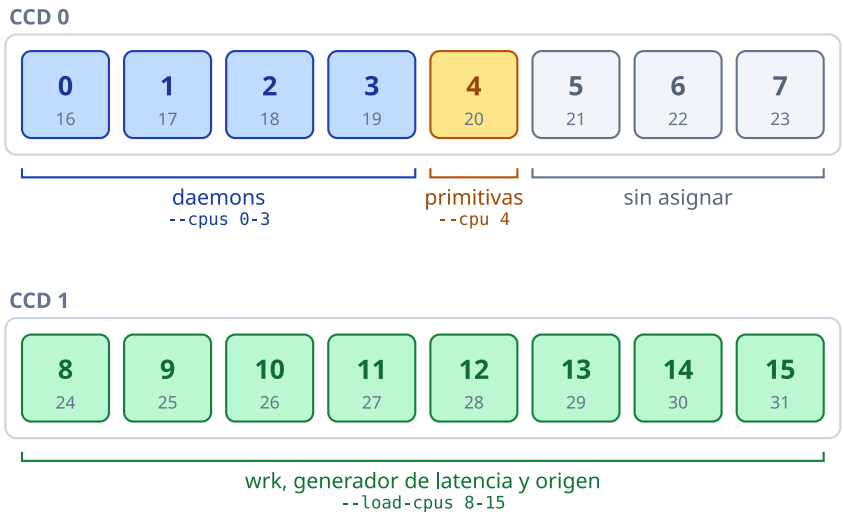


Fig. 11 Conjuntos de CPU en el AMD Ryzen 9 5950X. Las CPU 0 a 7 son los ocho núcleos del primer CCD, 8 a 15 los del segundo, y 16 a 31 sus hermanos SMT. El daemon y el generador de carga nunca comparten un núcleo.

En otras máquinas elegimos conjuntos que no compartan núcleos (`lscpu -e` muestra la topología).

Máquina y condiciones

Los archivos se registraron en `paxos-zig`: un AMD Ryzen 9 5950X 16-Core Processor (32 CPU lógicas), Linux 7.0.0-28-generic x86_64, el gobernador `powersave` con boost activado, rustc 1.99.0. Los archivos de primitivas, producto y latencia se registraron el 15 de octubre de 2026, y la comparación con Anubis el 16 de octubre de 2026. El daemon es `cargo build --release -p sibuna`, la compilación por defecto con almacenamiento SQLite y la consola incluida (SID 0010); los archivos de producto y latencia nombran un binario de 49.113.856 bytes. Cada archivo registra su revisión (`efce14d` para las primitivas, `249d913` para las ejecuciones de producto y latencia) con `dirty: true`; el árbol de trabajo de esta última ya contenía los hilos de conexión reutilizados que registra el SID 0002, y el SID 0002 cita `latency-latest.md` para su efecto. Los archivos de comparación y admisión nombran la revisión `54e4ec5` con `dirty: true`, porque el árbol de trabajo contenía las adiciones al programa de medición en `tools/bench`; su daemon es una compilación release de esa revisión (37.450.232 bytes). Sus cargas medias de un minuto fueron 3,53 antes de la ejecución de comparación y 0,56 al terminar la ejecución de admisión.

Los archivos registran cargas medias de un minuto de 2,46 en la ejecución de primitivas, 1,30 antes y 52,31 después de la ejecución de latencia, y 52,31 antes y 33,39 después de la ejecución de producto. Las cifras altas se deben a los 64 hilos del generador de latencia, a los que siguió directamente la ejecución de producto.

> **Alcance de las mediciones** Solo se emiten mediciones locales. Cada ejecución usa un solo host por loopback: sin TLS, sin WAN y sin navegadores reales. Los daemons se ejecutan en memoria, sin almacenamiento. Los cronómetros rodean lotes; los reinicios de estado y el calentamiento no se cronometran. El programa de medición no cuenta asignaciones (los archivos dicen "not instrumented; API/source audit only"); en su lugar, las pruebas de estado estacionario de la parte V comprueban que el camino de la petición no hace asignaciones. Ningún programa de medición se enlaza con el daemon ni añade un gancho al camino de la petición: los programas de producto y de latencia ejecutan el binario release sin modificar, identificado por su SHA-256. Una clasificación en este host no es una clasificación en otro.

Cómo reproducirlas

Necesitamos Linux, `taskset` (util-linux) y `wrk` en `PATH`.

```
cargo xtask bench primitives
cargo xtask bench product
cargo xtask bench latency
cargo xtask bench help
```

`cargo xtask bench` compila primero el daemon con `cargo build --release -p sibuna` y después ejecuta `sibuna-bench` en modo release. La herramienta encuentra el daemon junto a su propio binario; `--rust <path>` lo sustituye. `cargo xtask bench help` enumera las opciones, entre ellas `--cpus`, `--load-cpus`, `--cpu`, `--rounds`, `--repetitions`, `--seconds`, `--rates` y `--warmup`. La sección sobre Anubis, más abajo, da los comandos que registraron esa comparación.

Latencias de primitivas

La tabla da cada fila de la ejecución de primitivas. El archivo JSON nombra cada fila por subsistema y carga de trabajo (`pow_verify` y `hashcash_16_bits`, por ejemplo).

Tabla 12 Tiempos de primitivas en nanosegundos por operación en un núcleo de Ryzen 9 5950X (`benchmarks/results/primitives-latest.md`, generado 2026-10-15T22:34:59Z)

Carga de trabajo	ns/op	mín-máx
Verificación Hashcash (16 bits)	70,2	69,9-71,0
Verificación PoSW (profundidad 13, t = 16)	22.272,4	22.257,0-22.316,4
Firmas de bots (40, Aho-Corasick)	55,7	55,3-57,2
Firmas de bots (40, búsqueda secuencial)	763,0	762,2-763,5
Búsqueda CIDR IPv4	32,0	31,8-32,2
Búsqueda CIDR IPv6	79,3	78,5-79,6
Token de sesión (BLAKE3 con clave)	155,8	155,3-156,4
Token de sesión (Ed25519)	40.587,5	40.544,6-40.631,1
Conjunto de desafíos gastados (Robin Hood), gastar y consultar	21,0	20,8-21,3
Limitador de tasa (GCRA), bytes del cliente	12,8	12,7-12,8
Limitador de tasa, cuatro ámbitos de regla, bytes del cliente	13,1	13,0-13,2
Limitador de tasa (GCRA), cliente en hash	6,2	6,1-6,3
Limitador de tasa, cuatro ámbitos de regla, cliente en hash	6,3	6,2-6,4
Análisis HTTP y cookie	132,5	132,3-133,2
Clasificación, perfil Shield	790,8	787,2-793,1
Clasificación, perfil Gate	75,9	75,6-76,1
Exploración WAF del cuerpo (8 KiB)	12.784,3	12.771,5-12.809,2
Exploración WAF del cuerpo (8 KiB), inyección SQL en auditoría	15.485,4	15.460,9-15.530,1

Las dispersiones son estrechas: ningún lote de ninguna fila se aleja más de un 2,8% de la mediana de su fila. La figura muestra las mismas filas en escala logarítmica, que permite leer una junto a otra filas de 6 ns y de 40 us.

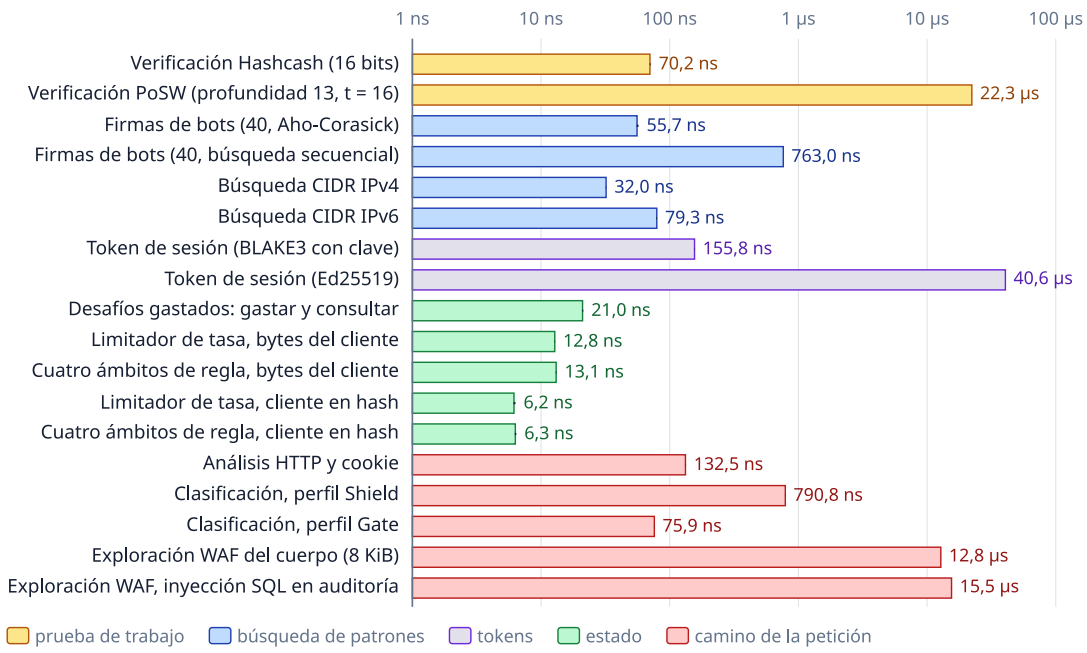


Fig. 12 Tiempos de las primitivas en escala logarítmica, tomados de `benchmarks/results/primitives-latest.json`.

Verificación de la prueba de trabajo

La verificación Hashcash calcula el hash de un desafío, un separador y un nonce decimal: 70,2 ns. El verificador de trabajo secuencial comprueba aperturas del grafo: 22.272,4 ns. Sus invocaciones de hash y sus recuentos de compresiones SHA-256 difieren de los de Hashcash porque las entradas tienen longitudes distintas. Ambas filas están limitadas por SHA-256, y `sibuna-crypto` usa las instrucciones SHA-NI cuando la CPU las tiene, detectadas en tiempo de ejecución (SID 0010).

Búsqueda de patrones

El autómata Aho-Corasick denso y la búsqueda secuencial de subcadenas usan patrones idénticos y la semántica de cualquier coincidencia. El autómata es 13,7 veces más rápido que la búsqueda secuencial con las mismas 40 firmas (55,7 frente a 763,0 ns). Sus tablas usan clases de bytes en lugar de 256 columnas por estado (SID 0006, parte F), lo que mantiene la tabla de firmas de bots en 42.244 bytes (véase la tabla de consumo de recursos más abajo). Los clasificadores CIDR tardan 32,0 ns (IPv4) y 79,3 ns (IPv6).

Tokens

El token MAC BLAKE3 se verifica en 155,8 ns. `sibuna-crypto` implementa BLAKE3 de forma portable, sin variantes SIMD, que compensan con entradas grandes; pero Sibuna aplica

BLAKE3 a entradas de menos de 100 bytes, donde el código portable es rápido. Los tokens compactos Ed25519 se verifican en 40.587,5 ns: el verificador usa dígitos NAF de anchura 5 y 8 y conserva la clave pública decodificada. Una verificación Ed25519 cuesta aproximadamente lo mismo que 260 verificaciones MAC; el MAC es el esquema de tokens por defecto de Sibuna.

Estado

La fila del conjunto de desafíos gastados inserta una etiqueta nueva y la vuelve a consultar: 21,0 ns. La tabla de esta fila tiene 16 particiones de 4.096 posiciones (65.536 etiquetas; el valor por defecto del daemon es 1.048.576), y se vacía en el sitio antes de cada lote, fuera del cronómetro. Por tanto, cada lote sondea memoria residente. Una versión anterior del conjunto de pruebas construía para cada lote una tabla nueva puesta a cero del tamaño por defecto del daemon; medía el tiempo que tarda el núcleo del sistema en mapear las páginas de la tabla en la primera escritura, un coste que un daemon en marcha paga una sola vez, y no es la fila que presentamos.

Las filas del limitador de tasa requieren cuidado. Las dos filas con bytes del cliente (`gcra_check` y `gcra_four_rule_scopes`) llaman a la API de bytes: cada comprobación toma la dirección del cliente como bytes y calcula primero un SipHash-1-3 con clave de ella, bajo un secreto obtenido de la aleatoriedad del sistema operativo. La clave secreta impide que nadie calcule direcciones que compartan la ventana de sondeo de una víctima (SID 0010 y la documentación de `sibuna-store`). Estas filas tardan 12,8 y 13,1 ns.

El daemon no usa la API de bytes en el camino de la petición. Calcula el hash del cliente una vez por petición (`ClientHasher`) y pasa el hash a cada tabla que consulta. Las dos filas con cliente en hash (`gcra_check_hashed` y `gcra_four_rule_scopes_hashed`) miden ese camino, con los hashes calculados antes del bucle: 6,2 y 6,3 ns por comprobación. La diferencia entre las medianas de las filas de bytes y en hash, 6,6 ns, es aproximadamente el precio del hash con clave sobre estas entradas, que el daemon paga una vez por petición y no una vez por tabla.

Analizador HTTP

La fila del analizador analiza una petición de navegador de 400 bytes y busca una cookie: 132,5 ns. La propia prueba de tiempos del crate coincide (`crates/http`: unos 135 a 180 ns en la misma máquina). El analizador no copia datos; cada campo es un slice del búfer de la conexión.

Motor de políticas

Una clasificación completa de una petición de navegador tarda 790,8 ns, y el perfil Gate 75,9 ns. Las filas dejan vacía la dirección del cliente ya analizada de la petición, así que el motor analiza el texto de la dirección del cliente dentro de la evaluación. El daemon analiza la dirección una vez por petición, antes de la evaluación, de modo que estas filas incluyen un coste pequeño que el daemon paga fuera del motor. Las tablas del motor en el heap suman 209.052 bytes (SID 0002).

Inspección WAF

La exploración semántica de un cuerpo de 8 KiB tarda 12.784,3 ns, es decir, $12784.3/8192 \approx 1.56$ ns por byte. La segunda fila explora el mismo cuerpo con la categoría de inyección SQL

en auditoría y las demás aplicadas: 15.485,4 ns. Cuando alguna categoría está en auditoría en lugar de aplicada, cada campo se inspecciona por categoría (SID 0004, "inspection modes"). El motor explora el campo con el autómata de firmas una sola vez, para el conjunto de categorías cuyas firmas aparecen, y después busca solo las categorías deseadas entre ellas; así, auditar una categoría cuesta aproximadamente una quinta parte más que aplicarlas todas, no una exploración por categoría.

Comparación de productos completos

> **Objetivos** Interpretar el rendimiento, la latencia, el tiempo de CPU y la memoria del daemon como proceso completo, comparar sus modos de protección y sus modos de despliegue, y decir en qué filas pone el límite el daemon y en cuáles el generador de carga.

Las tablas dan las medianas de tres ejecuciones de `wrk` de cinco segundos para cada modo de protección y carga de trabajo. El estado junto a cada carga de trabajo es la respuesta que el programa de medición esperaba y recibió.

Modo forward-auth

El proxy de entrada pregunta al daemon si debe admitir una petición; no hay origen ni retransmisión del cuerpo, así que esta es la medida más pura del coste de admisión.

Tabla 13 Modo forward-auth, Gate y Shield; latencia en microsegundos (`benchmarks/results/product-latest.md`, generado 2026-10-15T23:15:24Z)

Modo de protección	Carga de trabajo (estado)	req/s	p50	p90	p99	CPU us/req	RSS máx. KiB
Gate	`admitted` (200)	211.345	151	175	305	11,84	15.912
Gate	`challenged` (401)	198.315	160	180	326	13,15	16.176
Gate	`allowed_static` (200)	211.053	152	175	305	11,73	16.176
Gate	`attack` (200)	211.618	151	176	305	11,84	16.176
Shield	`admitted` (200)	208.101	152	178	308	12,87	15.928
Shield	`challenged` (401)	197.532	161	183	327	13,77	16.192
Shield	`allowed_static` (200)	211.636	151	175	305	11,78	16.192
Shield	`attack` (403)	210.549	152	175	307	12,32	16.192

En modo forward-auth el daemon no es el cuello de botella. Sus núcleos ocupados (tiempo de CPU dividido entre tiempo real, `cores_busy_median` en el archivo JSON) se mantienen entre

2,47 y 2,72 de las cuatro CPU del daemon, y todas las filas quedan entre 197.532 y 211.636 peticiones por segundo. Eso indica que el techo lo pone el generador de carga. Cerca de ese techo, una diferencia de un pequeño porcentaje entre filas cambia de una ejecución a otra: solo la fila `admitted` de Gate abarca de 205.151 a 211.513 peticiones por segundo en sus tres ejecuciones. La cifra más útil en este modo es el tiempo de CPU por petición: de 11,73 a 13,77 us, algo más para Shield que para Gate, y aún más para las filas `challenged`, que generan la respuesta de desafío.

Modo proxy inverso

El daemon se sitúa delante del origen sustituto y retransmite las peticiones admitidas y las respuestas. Por tanto, las filas retransmitidas incluyen el propio coste del origen y un salto loopback adicional.

Tabla 14 Modo proxy inverso, Gate y Shield; latencia en microsegundos (`benchmarks/results/product-latest.md`, generado 2026-10-15T23:15:24Z)

Modo de protección	Carga de trabajo (estado)	req/s	p50	p90	p99	CPU us/req	RSS máx. KiB
Gate	`admitted` (200)	142.498	427	548	768	27,68	15.896
Gate	`challenged` (200)	199.465	160	181	324	12,98	16.160
Gate	`allowed_static` (200)	147.303	411	541	759	26,88	16.324
Gate	`attack` (200)	143.900	422	555	745	27,67	16.348
Shield	`admitted` (200)	138.965	438	571	857	28,57	16.064
Shield	`challenged` (200)	195.188	162	185	328	13,73	16.328
Shield	`allowed_static` (200)	145.673	415	551	746	27,27	16.328
Shield	`attack` (403)	209.507	151	177	306	12,17	16.328

Aquí las filas retransmitidas (`admitted`, `allowed_static` y la fila `attack` de Gate, que Gate retransmite) están limitadas por la CPU: los núcleos ocupados son de 3,96 a 3,98 de cuatro. Sirven de 138.965 a 147.303 peticiones por segundo con 26,88 a 28,57 us de CPU por petición, aproximadamente el doble del coste de una respuesta que da el propio daemon. Las filas `challenged` y la fila `attack` de Shield las responde el propio daemon, y se parecen a las filas de forward-auth. En saturación, el p99 de `wrk` es la cola del planificador a la tasa propia del daemon, así que leemos en su lugar las colas del capítulo siguiente.

En tres filas de proxy inverso, `wrk` contó respuestas fuera de las clases 2xx y 3xx donde no se esperaba ninguna: 3 de 2.162.579 peticiones (`attack` de Gate), 1 de 2.082.640 (`admitted` de Shield) y 1 de 2.204.358 (`allowed_static` de Shield), sumadas sobre las tres ejecuciones. Los archivos no registran qué estado tenían. Al reproducir la ejecución con las métricas del daemon

aparecieron respuestas `503` contadas en `sibuna_overloaded_total`: `wrk` abre sus 64 conexiones desde una sola dirección, exactamente el valor por defecto de `--max-connections-per-ip`, y cuando una conexión se cerraba tras su petición número 4.096, `wrk` volvía a conectar antes de que la conexión antigua hubiera devuelto su plaza. Ahora el daemon devuelve la plaza antes de escribir la respuesta de cierre (SID 0002, "bug fix"); las tablas son anteriores a esa corrección.

Consumo de recursos

Tabla 15 Tamaños y memoria residente del daemon (`benchmarks/results/product-latest.json`, generado 2026-10-15T23:15:24Z, y `primitives-latest.json`, generado 2026-10-15T22:34:59Z)

Medida	Valor
Binario del daemon	49.113.856 bytes
Memoria residente en reposo, ejecuciones de producto	5.592 a 5.788 KiB
Memoria residente máxima durante una carga de trabajo	15.896 a 16.348 KiB
Memoria residente después de las cargas de trabajo	16.176 a 16.348 KiB
Memoria residente en reposo, ejecución de primitivas	6.052 KiB
Tablas del motor de políticas	209.052 bytes (heap)
Tabla de firmas de bots	42.244 bytes (heap)
Tabla WAF	164.908 bytes (heap)
Módulo del solucionador de navegador (WebAssembly)	14.744 bytes

Los rangos abarcan las cuatro ejecuciones de producto (dos modos, dos modos de protección) y, para el máximo, sus dieciséis cargas de trabajo. Los tamaños de las tablas son los tamaños en el heap que comunica el motor. El registro no desglosa el tamaño del binario.

Latencia con la misma carga ofrecida

> **Objetivos** Interpretar una distribución de latencia con una carga ofrecida fija, leer un percentil de cola junto con su dispersión y seguir un defecto de la cola desde su medición hasta su corrección.

Ambas tablas proceden de una ejecución de `cargo xtask bench latency`. La columna de errores cuenta los errores de conexión, de E/S y de encuadre del generador.

Tabla 16 Modo forward-auth, Shield, peticiones admitidas a tasas ofrecidas fijas; latencia en microsegundos (`benchmarks/results/latency-latest.md`, generado 2026-10-15T23:07:14Z)

req/s ofrecidas	req/s logradas	p50	p90	p99 (mín-máx)	p99.9	máx	Errores	CPU us/req
10.000	10.000	30	32	37 (36-39)	58	355	0	16,92
25.000	25.000	28	31	34 (32-34)	45	1.035	0	15,83
50.000	50.000	25	27	30 (30-30)	40	3.029	0	13,83
100.000	99.998	24	26	41 (40-42)	62	2.358	0	13,48
150.000	149.996	25	36	51 (51-52)	74	1.753	0	14,16

Tabla 17 Modo proxy inverso, Shield, peticiones admitidas a tasas ofrecidas fijas; latencia en microsegundos (`benchmarks/results/latency-latest.md`, generado 2026-10-15T23:07:14Z)

req/s ofrecidas	req/s logradas	p50	p90	p99 (mín-máx)	p99.9	máx	Errores	CPU us/req
10.000	10.000	56	61	65 (65-66)	76	555	0	32,25
25.000	25.000	51	54	58 (58-60)	72	1.105	0	28,63
50.000	50.000	47	50	60 (58-66)	74	3.066	0	27,45
100.000	99.996	55	75	100 (100-104)	150	2.239	0	28,42
150.000	134.344	583.934	1.544.160	1.906.273 (1.631.234-2.218.760)	1.967.134	1.989.964	4	29,22

En modo forward-auth, el p50 se mantiene entre 24 y 30 us a todas las tasas. El p99 se mantiene en 51 us o menos hasta 150.000 peticiones por segundo, y el p99.9 en 74 us o menos. El daemon gasta de 13,48 a 16,92 us de CPU por petición.

En modo proxy inverso, el p50 es de 47 a 56 us hasta 100.000 peticiones por segundo, y a 100.000 el p99 es de 100 us y el p99.9 de 150 us. A 150.000 peticiones por segundo el daemon se satura: completa 134.344 peticiones por segundo y la latencia crece hasta segundos, como debe mostrar una prueba en lazo abierto por encima de la capacidad. Esa tasa es cercana al rendimiento del proxy en lazo cerrado del capítulo anterior (138.965 peticiones por segundo para `admitted` de Shield). Los cuatro errores de esa fila se produjeron en la sobrecarga.

Las dispersiones del p99 son estrechas por debajo de la saturación; la más amplia es la del p99 de proxy inverso a 50.000 peticiones por segundo, de 58 a 66 us. La columna máx es una sola petición de hasta 1,5 millones en una ejecución. En un host compartido la interpretamos como el peor momento del host, no como una propiedad del daemon.

Qué cambió la medición

La carga externa ha cambiado el diseño de Sibuna. El daemon atiende cada conexión en un hilo propio (SID 0002, SID 0010), y sus primeras ejecuciones de latencia encontraron una cola que `wrk` no había mostrado. Con 64 conexiones, cada una cerrada por el daemon tras 4.096 peticiones, el percentil 99,9 a 100.000 peticiones admitidas por segundo era de 0,3 a 2 ms.

El diagnóstico, registrado en el SID 0002 y en la documentación de `cmd/sibuna/src/tasks.rs`, es la rotación de hilos. Crear y terminar un hilo mapea y desmapea su pila de señales y devuelve las páginas de su pila, y esos cambios del espacio de direcciones detenían los demás hilos del daemon. Un experimento lo zanjó: una compilación que nunca cerraba conexiones, de modo que no se creaba ningún hilo durante la ejecución, dio de 54 a 60 us a la misma tasa. La corrección reutiliza los hilos de conexión: un hilo que termina su conexión queda aparcado y toma la siguiente, así que un flujo constante de conexiones nuevas no crea hilos. Un hilo aparcado durante 10 segundos termina y lo recoge (join) el siguiente lanzamiento o el siguiente hilo que acabe una tarea, de modo que una ráfaga no deja sus pilas residentes. Un hilo cuya conexión entra en pánico termina en lugar de aparcarse. Como antes, se atienden a la vez como máximo `--max-connections` conexiones.
Las tablas anteriores proceden de la ejecución posterior a la corrección: un p99.9 de 62 us en modo forward-auth y de 150 us en modo proxy inverso a 100.000 peticiones por segundo. Las ejecuciones anteriores a la corrección no se conservan como archivos de resultados, por lo que sus cifras aparecen solo en el SID 0002 y en esta sección.

Ejemplo resuelto: por qué la pausa apareció en el p99.9 A 100.000 peticiones por segundo repartidas en 64 conexiones, cada conexión lleva $100000/64 = 1562.5$ peticiones por segundo. El daemon cierra una conexión keep-alive tras 4.096 peticiones (`MAX_REQUESTS_PER_CONNECTION` en `cmd/sibuna/src/connection.rs`), así que una conexión dura $4096/1562.5 \approx 2.62$ s, y el generador abre unas $64/2.62 \approx 24.4$ conexiones nuevas por segundo. Con un hilo por conexión, eso supone unos 24 arranques y 24 salidas de hilos cada segundo.
Una ejecución de 10 segundos a esta tasa mide 1.000.000 de peticiones. Su p99.9 lo superan 1.000 de ellas, 100 por segundo; su p99, 1.000 por segundo. Si cada rotación retrasa una fracción de milisegundo a unas $100/24.4 \approx 4$ peticiones en curso en otros hilos, las peticiones retrasadas bastan por sí solas para llenar el 0,1% superior, mientras que el 1% superior sigue formado por esperas en cola ordinarias. Una pausa rara y periódica mueve la cola lejana mucho antes de mover el p99. Por eso un defecto así necesita el p99.9 y un cronómetro en cada petición para hacerse visible.

Sibuna y Anubis

Objetivos Interpretar las cifras de Sibuna junto a las de un producto consolidado bajo la misma carga, y decir qué diferencias proceden de los productos y cuáles de la configuración.

Anubis, de Techaro, es un control de admisión por prueba de trabajo ampliamente desplegado que se sitúa delante de los sitios web, de forma muy parecida a Gate de Sibuna. Medimos el binario oficial de la versión v1.27.0 de Anubis (`anubis-1.27.0-linux-amd64`, 40.001.698 bytes) junto a Sibuna en el mismo host, en las mismas CPU y con las mismas cargas de trabajo. Los dos archivos de resultados se registraron con:

```
cargo xtask bench product --anubis <path/to/anubis>
cargo xtask bench admission --anubis <path/to/anubis>
```

Anubis se arrancó como lo arrancaban los scripts de Python anteriores de este proyecto: `--bind`, `--metrics-bind`, `--target=` (vacío en modo forward-auth, el origen en modo proxy inverso), `--difficulty 2`, `--slog-level ERROR`, `--cookie-secure=false` y una política que desafía a todo User-Agent `Mozilla` con DNSBL y el honeypot desactivados. `GOMAXPROCS` es igual al número de workers de Sibuna, y cada hilo de ambos productos se fija al mismo conjunto de CPU. El trabajo de Hashcash está igualado: 8 bits a cero para Sibuna, 2 dígitos hexadecimales a cero para Anubis, los mismos 256 hashes esperados. El programa de medición obtiene la sesión de cada producto como lo haría un navegador. Para Anubis, descarga la página de desafío, lee el desafío de `<script id="anubis_challenge">`, encuentra el nonce cuyo `SHA-256(randomData || nonce)` empieza por `00` y lo pasa a `pass-challenge` con la cookie de verificación de la página. En modo forward-auth, las cargas `admitted` y `challenged` consultan el endpoint de comprobación de Anubis (`/.within.website/x/cmd/anubis/api/check`).

Cargas de trabajo de producto completo

Las cuatro cargas de trabajo, la carga y las estadísticas son las del programa de medición de producto completo, con un cambio: el origen es `caddy respond`, fijado a las CPU de carga, como hacían los scripts anteriores. Antes de cada ejecución, el programa espera hasta que queden menos de 3.000 sockets en `TIME_WAIT`, porque las ejecuciones de Anubis en proxy inverso dejan decenas de miles.

Tabla 18 Modo forward-auth: medianas de tres ejecuciones de `wrk` de cinco segundos (`benchmarks/results/comparison-latest.md`, generado 2026-10-16T00:55:19Z).

Carga de trabajo	Cifra	Sibuna Gate	Sibuna Shield	Anubis 1.27.0
`admitted`	req/s (estado)	210.682 (200)	208.635 (200)	36.509 (200)
	p99 us	305	308	7.477
	CPU us/req	12,09	13,21	106,30
	RSS máx. KiB	16.228	16.316	32.624
`challenged`	req/s (estado)	198.251 (401)	193.087 (401)	113.170 (401)
	p99 us	326	342	3.781
	CPU us/req	13,49	15,02	32,96
	RSS máx. KiB	16.240	16.328	32.696
`allowed_static`	req/s (estado)	212.569 (200)	212.444 (200)	89.904 (200)
	p99 us	304	304	4.547
	CPU us/req	11,73	11,95	41,63
	RSS máx. KiB	16.240	16.328	33.396

Carga de trabajo	Cifra	Sibuna Gate	Sibuna Shield	Anubis 1.27.0
`attack`	req/s (estado)	212.086 (200)	211.347 (403)	36.853 (200)
	p99 us	305	306	7.283
	CPU us/req	12,07	12,56	104,88
	RSS máx. KiB	16.240	16.328	33.764

Tabla 19 Modo proxy inverso, detrás de `caddy respond`: medianas de tres ejecuciones de `wrk` de cinco segundos (`benchmarks/results/comparison-latest.md`, generado 2026-10-16T00:55:19Z).

Carga de trabajo	Cifra	Sibuna Gate	Sibuna Shield	Anubis 1.27.0
`admitted`	req/s (estado)	106.376 (200)	103.298 (200)	19.398 (200)
	p99 us	3.331	3.321	11.062
	CPU us/req	27,02	27,92	190,40
	RSS máx. KiB	16.528	16.444	39.980
`challenged`	req/s (estado)	198.402 (200)	197.327 (200)	28.842 (200)
	p99 us	325	328	9.546
	CPU us/req	13,05	13,78	137,73
	RSS máx. KiB	16.544	16.464	516.872
`allowed_static`	req/s (estado)	105.427 (200)	106.819 (200)	28.862 (200)
	p99 us	3.320	3.316	30.124
	CPU us/req	26,24	26,52	135,41
	RSS máx. KiB	16.544	16.464	616.548
`attack`	req/s (estado)	105.616 (200)	210.483 (403)	23.399 (200)
	p99 us	3.327	306	14.124
	CPU us/req	26,98	12,28	169,04
	RSS máx. KiB	16.544	16.464	599.516

En modo forward-auth, Sibuna sirvió de 193.087 a 212.569 peticiones por segundo con un p99 de 304 a 342 us, y Anubis de 36.509 (`admitted`) a 113.170 (`challenged`) con un p99 de 3.781 a 7.477 us. Los núcleos ocupados de Sibuna se mantuvieron entre 2,49 y 2,90 de cuatro, así que el límite lo puso el generador de carga; los de Anubis fueron de 3,73 a 3,88, así que Anubis estaba limitado por la CPU. La cifra más clara es el tiempo de CPU por petición: una petición admitida costó 12,09 us a Sibuna Gate y 106,30 us a Anubis. Una parte probable de esa diferencia es el token de sesión. Anubis firma las sesiones con Ed25519 por defecto, y Sibuna comprueba un MAC BLAKE3 con clave (155,8 ns en la tabla de primitivas); las

mediciones de admisión de más abajo, en las que los tokens HS512 aumentan en más de la mitad las comprobaciones de sesión de Anubis, respaldan esa interpretación.
En modo proxy inverso, las filas retransmitidas de Sibuna sirvieron de 103.298 a 106.819 peticiones por segundo con 2,77 a 2,98 núcleos ocupados. Quedan por debajo de las 138.965 a 147.303 del origen sustituto en `product-latest.json`, y el daemon no está limitado por la CPU, así que el límite está fuera de él: el origen, `caddy respond`, comparte las CPU de carga con `wrk`. Su cola se ve en el p90 de 2,1 a 2,4 ms. Anubis sirvió de 19.398 a 28.862 peticiones por segundo con 3,68 a 3,97 núcleos ocupados y de 135 a 190 us de CPU por petición. Su memoria residente subió de 39.980 KiB en la carga `admitted` a 516.872 KiB en `challenged` y 616.548 KiB en `allowed_static`, y retenía 589.276 KiB después de las cargas, frente a 21.552 KiB en reposo. La subida empezó en `challenged`, donde cada petición recibe un desafío nuevo con su propio id; esto sugiere un registro guardado por cada desafío emitido, pero el programa de medición no confirmó la causa. El máximo de Sibuna se mantuvo entre 16.228 y 16.544 KiB en todas las filas de ambos modos.
Varias filas tampoco son comparables de forma directa:

- En modo forward-auth, Anubis responde por sí mismo `/robots.txt` y la ruta `attack` con `200`, sin origen; Sibuna las responde como responde a cualquier petición en ese modo.
- Anubis no inspecciona las peticiones, así que admite la inyección SQL de la carga `attack`. Shield la rechaza con `403`, tras más trabajo; Gate la admite, como hace Anubis.
- Los productos firman las sesiones de forma distinta (Ed25519 por defecto en Anubis, un MAC BLAKE3 con clave en Sibuna) y generan páginas de desafío distintas.

Operaciones de admisión

El programa de medición de admisión cronometra las cuatro operaciones de las que se compone la admisión, en modo forward-auth, con cada producto en dos CPU (dos workers de Sibuna, `GOMAXPROCS=2`). Dos hilos cliente en las CPU de carga, cada uno con una conexión keep-alive, se reparten cada lote: 200 operaciones, o 64 pruebas para la verificación, preparadas antes del lote. Damos la mediana de siete lotes.

- Comprobación de sesión: el endpoint de comprobación con una sesión válida (`200`).
- Comprobación sin autenticar: lo mismo sin cookie (`401`).
- Arranque del desafío: lo que descarga un navegador para empezar un desafío. Para Sibuna son dos peticiones, la página intermedia y el registro de desafío; para Anubis una, la página con el desafío integrado.
- Verificación de la prueba: una prueba nueva por operación. Los desafíos de prueba de Sibuna se emiten con 25 ms de separación, por debajo de los 50 por segundo a partir de los cuales su dificultad adaptativa añade bits, de modo que el trabajo igualado se mantiene. Sibuna se ejecuta con `--challenge-rate-limit 100000000`, así que su limitador se comprueba pero nunca se agota.

Anubis se ejecutó dos veces, con sus tokens Ed25519 por defecto y con `--hs512-secret`.

Tabla 20 Operaciones de admisión por segundo, dos clientes, producto en dos CPU (`benchmarks/results/admission-latest.md`, generado 2026-10-16T00:55:44Z)

Producto (token)	Comprobación de sesión	Comprobación sin autenticar	Arranque del desafío	Verificación de la prueba	RSS después, KiB
Sibuna (BLAKE3 con clave)	88.104	78.997	42.664	57.924	7.792
Anubis 1.27.0 (Ed25519)	10.951	28.619	7.893	12.668	33.400
Anubis 1.27.0 (HS512)	16.976	27.785	7.952	19.261	29.840

Los rangos de los siete lotes están en el archivo. Los de Sibuna quedan dentro de un 11% de sus medianas; los de Anubis son más amplios (sus comprobaciones sin autenticar con tokens Ed25519 abarcan de 25.394 a 32.741), y un lote de 200 operaciones dura solo de 2 a 25 ms, así que leemos estas cifras con una precisión de aproximadamente una décima. El tiempo por operación del cliente más lento da el tiempo de ida y vuelta: 22,7 us para una comprobación de sesión de Sibuna, 182,5 us para una comprobación de Anubis con tokens Ed25519 y 117,7 us con HS512.

En ninguna de las filas anteriores Anubis sirvió más peticiones, ni usó menos CPU o memoria, que Sibuna. Esa es una afirmación sobre estas cargas de trabajo en este host, y nada más.

> **Cómo leer las filas de Anubis** Anubis es un producto capaz y ampliamente desplegado, y estas filas no afirman que sea lento. Se ejecuta sobre un runtime con recolección de basura, firma las sesiones con Ed25519 por defecto para que la verificación no necesite un secreto compartido, ofrece varios métodos de desafío y un lenguaje de políticas rico, y guarda el estado de los desafíos en un almacén; son decisiones de diseño con beneficios que este programa de medición no mide. Las filas muestran lo que costó una petición a cada producto en un host con una forma de carga, en octubre de 2026. Sibuna es nuevo, y estas son sus primeras mediciones junto a un producto consolidado.

Sobrecarga de la consola

El SID 0007 fija el contrato de aislamiento de la consola. Activar la consola puede costar como máximo un 1% del rendimiento del plano de datos y como máximo un 10% de la latencia p99, medidos sobre una matriz de configuraciones (excluida de la compilación, compilada pero desactivada, en reposo y con ocho paneles activos) y de cargas de trabajo, con ejecuciones repetidas e intercaladas y indicando la incertidumbre. Las ejecuciones no concluyentes o ruidosas no demuestran el cumplimiento, y las referencias de primitivas y las pruebas de humo no pueden sustituir a la prueba de aceptación.

El SID 0010 registra una medición: con la consola compilada, ya sea desactivada, solo con almacenamiento o activada y en reposo, las 16 cargas de trabajo de producto se mantienen a menos de aproximadamente un 1% del daemon compilado sin ella, lo que está dentro del ruido. El registro no nombra ningún archivo de resultados para ella. La configuración con ocho

paneles activos bajo carga no se ha medido, así que la prueba de aceptación formal del SID 0007 sigue abierta.

La ejecución con ocho paneles comprueba el funcionamiento, no la sobrecarga. `sibuna-browser dashboards` arrancó un daemon release en un nodo SQLite con su consola, cuatro workers y una política que permite `/static/*`, `/about`, `/missing/*` y `/broken/*` y desafía todo lo demás; un origen de prueba responde `/missing/*` con 404 y `/broken/*` con 500. Un generador envió 200 peticiones por segundo durante 30 segundos, rotando diez rutas (una de ellas una inyección SQL), tres User-Agents y 500 direcciones reenviadas. Cada panel se ejecutó en su propio Chromium con su propia sesión: los paneles 1 a 3 como sesiones de administrador, y los paneles 4 a 8 como pantallas murales abiertas con códigos que generó el primero, porque la consola permite cinco inicios de sesión por minuto por dirección y por cuenta.

Tabla 21 Ocho paneles de tráfico en un nodo (`benchmarks/results/console-8-dashboards-2026-10-15.md`, registrado 2026-10-15, Chromium 155.0.8059.39). Resultado: PASS

Medida	Valor
Peticiones del plano de datos	6.000 en 30,0 s (200 por segundo, 4 hilos, una conexión por petición)
Respondidas	6.000: 200 x 1.200, 401 x 3.000, 403 x 600, 404 x 600, 500 x 600
Errores (sin respuesta)	0
Latencia p50 / p90 / p99 / máx	246 / 311 / 373 / 552 us (conexión incluida)
VmRSS del daemon al inicio	36.152 KiB
VmRSS del daemon con 8 paneles abiertos	38.028 KiB
VmRSS del daemon después del tráfico	41.496 KiB (VmHWM 41.496 KiB)

Los indicadores se leyeron cada dos segundos durante el tráfico y 3 segundos después. Todos los paneles terminaron con los mismos totales (Requests 6.000, Admitted 2.400, Challenged 3.000, Policy denied 600, Origin 4xx 600, Origin 5xx 600), recibieron de 134 a 139 deltas `stats`, mantuvieron abierto su WebSocket y no informaron de ninguna excepción, error de consola ni violación de Content-Security-Policy. Ocho suscriptores de navegador quedan muy por debajo del límite de 64 de la consola. El daemon de esta ejecución tiene almacenamiento SQLite y la consola, a diferencia de los daemons en memoria del programa de medición de producto, así que su memoria residente no es comparable con la de ellos.

Lo que no se mide

Tabla 22 Lo que esta parte no mide, y lo que ocupa el lugar de una medición

Tema	Por qué no se mide aquí	Qué ocupa su lugar
SafeLine community edition 9.x	Se distribuye solo como pila de Docker Compose; el host no tenía daemon de Docker, y la imagen del detector no es una compilación de código abierto que pueda compilarse allí	Datos publicados e informados por el fabricante (comprobados en septiembre de 2026): tasa de detección del 71,65% (balance) y del 76,17% (strict), falsos positivos del 0,07% (balance) y del 0,22% (strict); host mínimo Linux x86_64 o arm64 con SSSE3, 1 núcleo de CPU, 1 GB de RAM, 5 GB de disco, Docker 20.10.14+ y Compose 2.0+
Cloudflare WAF	Un servicio alojado; una medición por loopback mediría la ruta de red, no el WAF	Cuotas publicadas de los planes (comprobadas en septiembre de 2026): reglas personalizadas Free 5, Pro 20, Business 100, Enterprise 1.000; reglas de limitación de tasa Free 1, Pro 2, Business 5, Enterprise 100
Un clúster bajo carga	Todavía ningún programa de medición controla varios nodos sobre una base de datos PostgreSQL	Las propiedades del protocolo que especifica el SID 0011 (una semilla compartida, desafíos ligados al emisor) y su prueba funcional con tres nodos (criterio 2); ninguna medición
Almacenamiento en el camino de la petición	Ambos daemons se ejecutan en memoria en todos los programas de medición	SID 0011: un solo hilo de almacenamiento es dueño de todo el SQL, y el camino de la petición nunca lo espera
TLS, WAN, navegadores reales	Los programas de medición usan loopback y HTTP; `sibuna-pow` resuelve las sesiones una vez por caso	Nada; cada despliegue mide su propia ruta de red
El tiempo de resolución del solucionador de navegador	El programa de medición resuelve a 8 bits con el solucionador nativo	El tamaño del módulo indicado arriba; la parte VII explica por qué los tiempos nativos no representan los tiempos de resolución en el navegador
Recuentos de asignaciones	El programa de medición no instrumenta el asignador	Las pruebas de estado estacionario de la parte V

Tema	Por qué no se mide aquí	Qué ocupa su lugar
Aislamiento de la consola con ocho paneles activos bajo carga	La prueba de aceptación formal del SID 0007 no se ha ejecutado	La cifra del SID 0010 (a menos de aproximadamente un 1%, con la consola en reposo) y la ejecución funcional anterior

Una fila de esta tabla no es una afirmación sobre el rendimiento del producto. Las cifras propias de SafeLine tratan de la calidad de detección, no del rendimiento, y Cloudflare publica cuotas de planes en lugar de costes por petición. Cualquiera con un host Docker puede ejecutar SafeLine detrás de las mismas cargas de `wrk`.

Ejercicio 8.1. Ejecuta `cargo xtask bench primitives` en otra máquina y compara las filas de verificación Hashcash y de verificación de tokens con las del host de referencia. ¿Cuál de las dos depende más de las instrucciones de hash por hardware, y por qué?

Ejercicio 8.2. En la tabla de forward-auth, calcula los núcleos ocupados de una fila: sus microsegundos de CPU por petición multiplicados por sus peticiones por segundo, divididos entre un millón. ¿Estaba el daemon limitado por la CPU? Después compara sus microsegundos de CPU por petición con el coste de clasificación de la primera tabla. ¿Por qué el coste de CPU por petición es tanto mayor, y qué componentes explican la diferencia?
Pista: enumera todo lo que ocurre entre `accept` y la respuesta vaciada que el conjunto de primitivas no cronometra.

Explica el invariante. Cada cifra de esta parte es una mediana sobre repeticiones, con su dispersión al lado. Explica la diferencia entre una cifra de mediana de lotes y un p99.9 por petición, por qué la tabla de primitivas da la primera y la tabla de latencia de la segunda, y por qué solo la segunda podía revelar la pausa por rotación de hilos descrita arriba.

PARTE IX

Operaciones y despliegue

Tratamos los dos modos de protección, todas las opciones de la línea de comandos, las recetas de forward-auth, los archivos de políticas, el almacenamiento persistente en SQLite o PostgreSQL, el clúster sobre una única base de datos compartida y el empaquetado.

Modos de protección y configuración

Objetivos Al terminar este capítulo, deberíamos poder ejecutar Sibuna como proxy inverso o como validador forward-auth, elegir un modo de protección, fijar el nivel y la dificultad de la prueba de trabajo y gestionar el secreto maestro.

Elegir un modo de protección

- *Gate* (`--gate`, alias `--no-waf`): admisión por prueba de trabajo, sesiones, reglas declarativas, reputación, límites GCRA y bloqueos.
- *Shield* (por defecto, `--shield`, alias `--waf`): Gate más el cortafuegos semántico.
- *Despliegue distribuido* (una opción para cualquiera de los dos modos): `--data-dir` guarda políticas, reputación y datos forenses en un archivo SQLite local; `--database-url` con `--cluster-node` conecta varios nodos a una única base de datos PostgreSQL (SID 0011).

Referencia de la línea de comandos

La tabla enumera todas las opciones de `sibuna --help`. Los rangos son los que impone el analizador.

Tabla 23 Opciones del daemon

Opción	Valor por defecto	Significado
`--port, -p`	`8080`	Puerto de escucha, de 1 a 65535
`--host, -h`	`0.0.0.0`	Dirección de escucha, un literal de dirección IP
`--upstream-host`	`127.0.0.1`	Host del origen (modo proxy inverso)
`--upstream-port, -u`	`3000`	Puerto del origen
`--mode, -m`	`reverse_proxy`	`reverse_proxy` o `forward_auth` (también `proxy` y `auth`), indicado una sola vez
`--workers, -w`	uno por CPU	Hilos de aceptación que comparten el socket de escucha, de 0 (uno por CPU) a 64; después, cada conexión aceptada recibe su propio hilo

Opción	Valor por defecto	Significado
`--max-connections`	`1024`	Conexiones atendidas a la vez, de 1 a 8192; las siguientes reciben `503`
`--max-connections-per-ip`	`64`	Conexiones que puede mantener una dirección de par del socket, de 0 (sin límite) a 8192; un par IPv6 cuenta con su red `--ipv6-prefix`, y las cabeceras reenviadas no intervienen; las siguientes reciben `503`
`--idle-timeout`	`15`	Silencio máximo en segundos, de 0 a 86400. El encabezado de una petición debe llegar dentro de ese plazo. Una respuesta del origen lo renueva en cada lectura, así que un flujo lento sigue vivo mientras fluyan bytes y un origen silencioso se corta en ambos sockets. Una subida a través del proxy debe entregar 16 KiB por periodo (una tasa mínima contra los ataques de cuerpo lento). El socket del cliente lleva un plazo de escritura de la misma duración, así que un cliente que no acepta ningún byte durante ese tiempo pierde la conexión
`--websocket-idle-timeout`	`300`	Plazo de inactividad de una conexión actualizada con Upgrade, en segundos, de 0 a 86400
`--trust-forwarded`	desactivada; activada en forward--auth	Acepta `X-Forwarded-For` / `X-Real-IP` del par
`--no-trust-forwarded`	desactivada	No las acepta nunca, ni siquiera en modo forward-auth
`--trusted-hops`	`1`	Si la identidad reenviada es de confianza, el cliente es la entrada de `X-Forwarded-For` situada a este número de posiciones desde la derecha, de 1 a 8
`--ja4-header`	desactivada	La cabecera de petición en la que un proxy de entrada de confianza envía la huella TLS JA4 del cliente, como `cf-ja4`; de 1 a 64 caracteres de token, que no sea `X-Forwarded-*`, `X-Real-IP`, `X-Sibuna-*` ni `Forwarded`. Solo se lee si la identidad reenviada es de confianza (véase "Huellas JA4 desde el proxy de entrada")

Opción	Valor por defecto	Significado
`--ipv6-prefix`	`64`	Longitud del prefijo IPv6, de 32 a 128, a la que se aplican los límites de tasa, los bloqueos y el límite de conexiones por cliente
`--algorithm, -a`	`posw`	`posw` o `hashcash`
`--difficulty, -d`	`16`	Bits de trabajo, de 1 a 64: bits a cero de Hashcash, o profundidad de PoSW más tres
`--posw-challenges`	`16`	Aperturas por prueba de trabajo secuencial, de 1 a 32
`--token-scheme`	`mac`	`mac` (BLAKE3 con clave) o `ed25519`
`--token-ttl`	`86400`	Duración de la sesión, en segundos (de 1 a 31.536.000)
`--challenge-ttl`	`300`	Duración del desafío, en segundos (de 1 a 86.400)
`--secret-file, -s`	aleatorio	64 caracteres hexadecimales o 32 bytes en bruto; también se acepta `SIBUNA_SECRET` (64 caracteres hexadecimales)
`--cookie-name`	`__sibuna_token`	Nombre de la cookie de sesión, de 1 a 64 caracteres de token
`--secure-cookie`	desactivada	Añade el atributo `Secure`
`--gate` / `--shield`	shield	Selección del modo de protección
`--rate-limit`	`100`	Peticiones por ventana y por cliente (ráfaga GCRA)
`--rate-window`	`10`	Ventana en segundos, de 1 a 86400
`--challenge-rate-limit`	`30`	Emisiones y verificaciones de desafíos por ventana y por cliente, aparte del presupuesto de peticiones
`--ban-seconds`	`3600`	Duración del bloqueo del honeypot, de 0 a 31.536.000
`--policy-file, -P`	ninguno	Política JSON declarativa
`--rate-table-cells`	`262144`	Celdas por tabla del limitador de tasa, de 16 bytes cada una, de 4.096 a 16.777.216, redondeadas hacia arriba a una potencia de dos

Opción	Valor por defecto	Significado
`--ban-table-slots`	`65536`	Posiciones de la tabla de bloqueos, de 24 bytes cada una, de 1.024 a 16.777.216, redondeadas hacia arriba a una potencia de dos
`--spent-tags`	`1048576`	Desafíos resueltos que se recuerdan para impedir su reutilización, de 24 bytes cada uno, de 65.536 a 16.777.216
`--data-dir, -D`	ninguno	Un nodo: persiste en el archivo SQLite `<path>/sibuna.db`
`--database-url`	ninguno	Un clúster: la base de datos PostgreSQL compartida (`postgres://` o `postgresql://`)
`--database-url-file`	ninguno	Lee la URL de la base de datos de un archivo accesible solo por su propietario
`--cluster-node`	ninguno	El id de este nodo; vincula las claves de desafío y los bits altos de los identificadores de incidentes; obligatorio con una URL de base de datos
`--storage-poll-ms`	`500`	Intervalo del ciclo de almacenamiento, de 1 a 60.000 ms
`--verbose, -v`	desactivada	Registro detallado
`--help` / `--version`		Imprime la ayuda (primero las opciones de la consola, si la consola está incluida en la compilación) o la versión, y termina

`--data-dir` y una URL de base de datos se excluyen mutuamente, igual que `--database-url` y `--database-url-file`. Se rechaza una URL de base de datos sin `--cluster-node`. Las opciones de clúster de Zaxonlite `--cluster-listen`, `--cluster-peer`, `--cluster-secret-file` y `--cluster-tls-cert`, `--cluster-tls-key` y `--cluster-tls-ca` ya no existen; si se indica alguna, el arranque se detiene con `RemovedOption` y una pista que remite a SID 0011. Las opciones de la consola se enumeran en el capítulo siguiente.

El secreto maestro Sin `--secret-file` ni `SIBUNA_SECRET`, el daemon genera una semilla aleatoria y el banner imprime "random (set –secret-file)". Los tokens y los desafíos desaparecen entonces con el proceso, y un clúster cuyos nodos tengan semillas distintas rechazará las cookies de los demás nodos. Genera uno con `head -c 32 /dev/urandom | xxd -p -c 64 > /etc/sibuna/secret` y dale el modo 0600.

Vista previa de la consola

La consola se compila en el daemon mediante la feature por defecto `console`, pero solo arranca con `--console`. Necesita almacenamiento: `--console` sin `--data-dir` ni `--database-url` se rechaza con `CONSOLE001` (`StorageRequired`). Usa `cargo build --release -p sibuna --no-default-features --features sqlite` para excluirla de la compilación; una compilación solo con `--no-default-features` no tiene ni almacenamiento ni consola. Inicializa localmente con el daemon detenido:

```
sibuna init-admin admin --data-dir ./data
sibuna --data-dir ./data --console 127.0.0.1:19446
```

`sibuna console init-admin` es el mismo comando. Abre `http://127.0.0.1:19446/console/` y sustituye la contraseña temporal generada, que debe cambiarse en el primer inicio de sesión y caduca al cabo de una hora. La navegación con sesión iniciada ofrece el globo animado de países, estadísticas de peticiones y desafíos, investigación de incidentes, editores de políticas y de inspección, usuarios, tokens de API con ámbito, auditoría y controles del nodo que sirve. La interfaz de autenticación no carga geometría del globo ni telemetría. La interfaz es un módulo WebAssembly compilado desde `cmd/console-ui` (crate `sibuna-console-ui`) por el script de compilación del daemon; el módulo y la hoja de estilos incluida en el repositorio se distribuyen con las compilaciones ordinarias.

Tabla 24 Opciones de la consola

Opción	Significado
`--console <ip:port>`	Inicia la consola en esta dirección de escucha (IPv6 entre corchetes, puerto distinto de cero); requiere `--data-dir` o `--database-url`
`--console-key-file <path>`	64 caracteres hexadecimales, permisos solo para el propietario; protege los secretos guardados del segundo factor y de los webhooks
`--console-origin <https-origin>`	El origen HTTPS canónico detrás de un proxy
`--console-behind-proxy`	Un proxy que termina TLS entrega todas las peticiones
`--console-trusted-proxy <cidr>`	Una red cuyas conexiones pueden llevar campos reenviados; repetible, como máximo 16
`--console-advertise <origin>`	El enlace que otras consolas muestran para esta
`--console-location <latitude,longitude>`	La posición declarada de este nodo en el globo
`--console-probe <node-id>=<http://ip:port>`	La dirección de escucha del plano de datos de un par, cuyo estado se comprueba; repetible, como máximo ocho
`--console-peer <node-id>=<https://origin>`	Un par de gestión; repetible, como máximo ocho

Opción	Significado
`--console-peer-key-file <path>`	La clave de los pares: solo para el propietario, 64 caracteres hexadecimales, independiente de la clave de la consola
`--console-peer-ca-file <path>`	Anclas de confianza PEM opcionales para los pares, que sustituyen a las raíces del sistema
`--console-query-steps <n>`	Pasos de SQLite por lectura agregada, de 100.000 a 50.000.000, por defecto 4.000.000
`--console-capture-heads`	Guarda por incidente los encabezados enmascarados de la petición y de la respuesta del origen; desactivada por defecto
`--console-capture-header <name>`	Conserva el valor de esta cabecera en los encabezados guardados; repetible, como máximo 16; requiere `--console-capture-heads`
`--console-cookie-secure`	Da a las cookies de la consola el atributo `Secure` sin proxy (el analizador la acepta, pero no aparece en `--help`)

Una opción de consola sin `--console`, una opción repetida o un valor mal formado detienen el arranque con `CONSOLE002` y el nombre de error de la opción. Una configuración que incumple una regla de la consola (por ejemplo, una dirección de escucha fuera de loopback sin proxy de confianza) lo detiene con `CONSOLE001`.

Un recorrido por la consola

Las capturas siguientes proceden de una compilación release en un único nodo en loopback con almacenamiento SQLite y una política de cuatro reglas `ALLOW` sobre un desafío por defecto, tras unos cuatro minutos de tráfico sintético: direcciones de rangos de documentación, sondas de inyección SQL y de cross-site scripting, tres accesos al honeypot y siete desafíos resueltos en Chromium sin interfaz gráfica. Por tanto, todas las cifras son ilustrativas. Las tomamos con `cargo xtask browser console --viewport-only`, que recorre la consola en Chromium a 1440 por 960 píxeles y evalúa cada página. Cada página responde a una pregunta enunciada en su título y muestra el producto, el nodo que sirve, la página y el camino de vuelta en el mismo lugar.

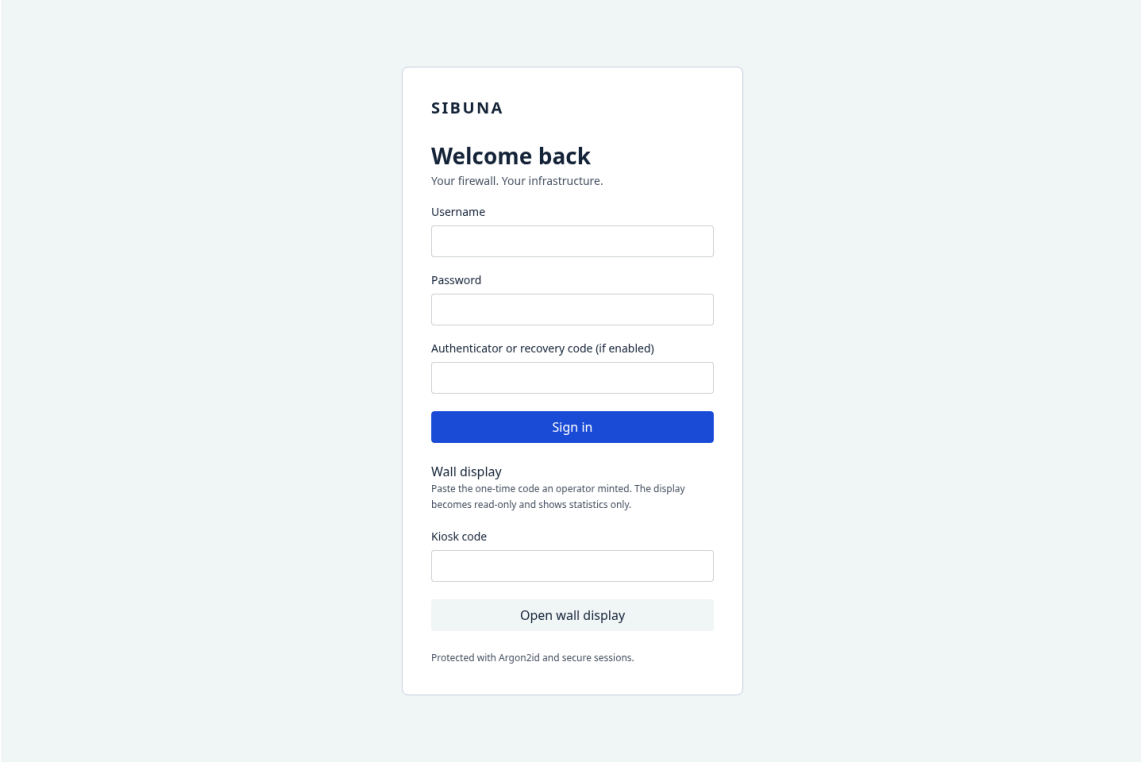


Fig. 13 Inicio de sesión. La interfaz pública no carga geometría del globo, datos GeoIP ni telemetría; el segundo formulario canjea un código de pantalla mural por una sesión de estadísticas de solo lectura.

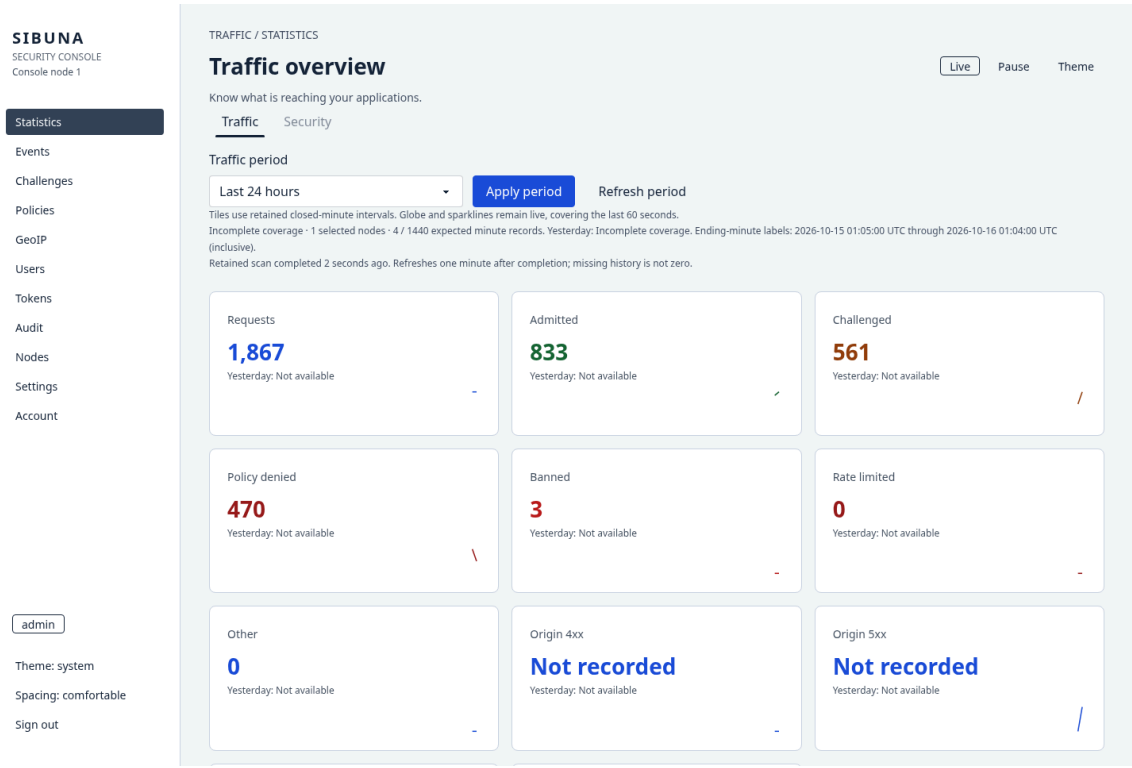


Fig. 14 Statistics, Traffic, en las últimas 24 horas. La nota bajo el control del periodo indica la cobertura: solo existen cuatro de los 1.440 registros por minuto esperados, así que cada "Yesterday" muestra "Not available". Los recuadros muestran los recuentos conservados de minutos cerrados con una minigráfica en vivo de 60 segundos; los colores de las decisiones (admitida en verde, desafiada en ámbar, denegada en rojo) son los mismos en todos los paneles. Los 4xx y 5xx del origen muestran "Not recorded" para un periodo conservado, porque los registros por minuto no guardan el modo de proxy que da sentido a un recuento del origen; los totales en vivo desde el arranque sí los muestran. Debajo de los recuadros visibles hay dos recuadros de nivel, entradas de bloqueo activas y nodos sanos, la cronología y paneles muestreados que clasifican las rutas de las peticiones y los hosts de referencia, y cuentan los sistemas operativos, los navegadores y los estados de respuesta de los clientes.

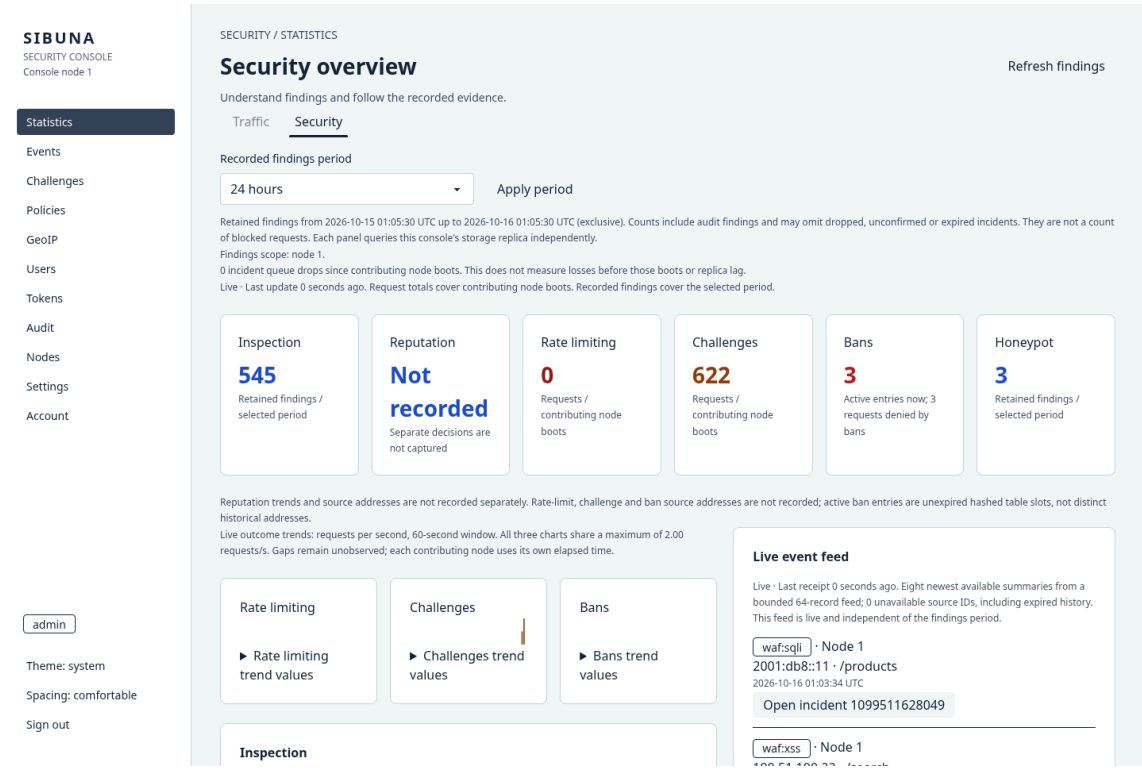


Fig. 15 Statistics, Security. Un recuadro por módulo durante el periodo seleccionado, tendencias de resultados en vivo y el flujo de eventos en vivo con sus hallazgos más recientes (aquí `waf:sqli` y `waf:xss`). Los valores que no se registran lo indican en lugar de mostrar cero: el recuadro de Reputation muestra "Not recorded". Más abajo en la página aparecen los hallazgos en sesenta intervalos iguales, las categorías de ataque y las rutas atacadas.

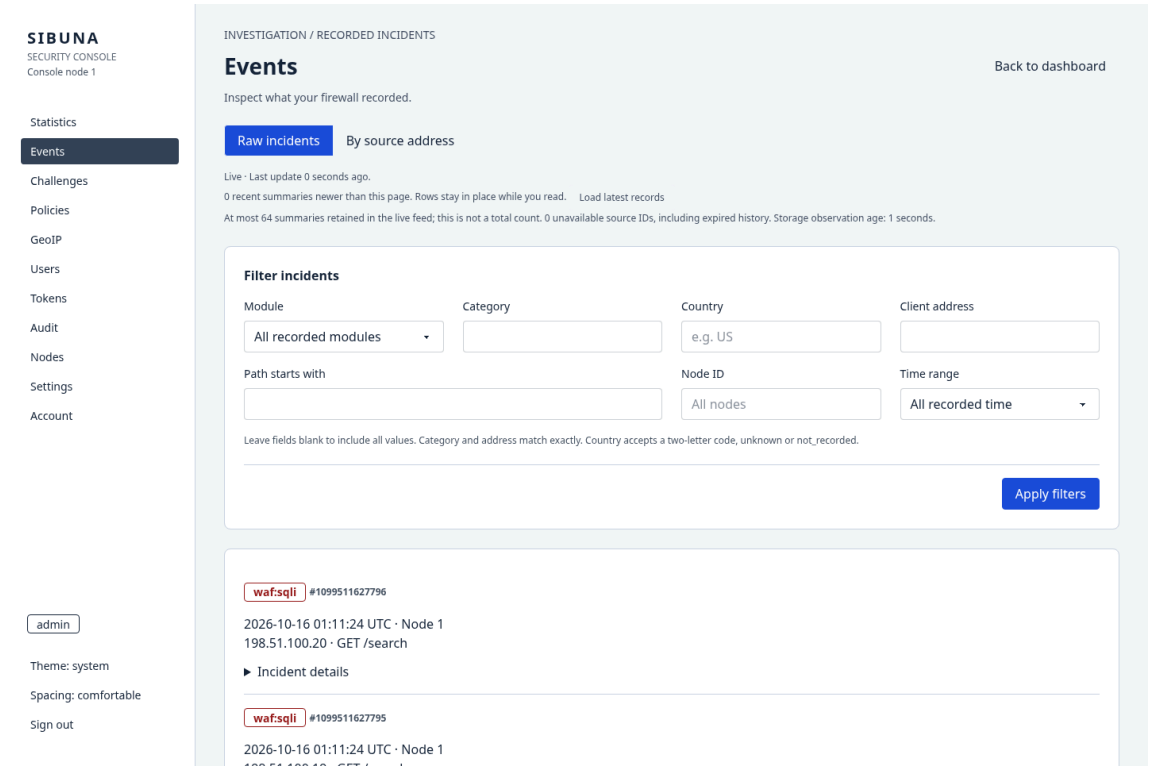


Fig. 16 Events. El formulario de filtros sobre los incidentes registrados, del más reciente al más antiguo. Cada incidente se despliega en línea con sus pruebas: la respuesta local seleccionada, las longitudes en bytes, la campaña candidata y entradas explícitas "not recorded" para la regla coincidente, los términos de la puntuación y la huella JA4. Con `--console-capture-heads`, el encabezado enmascarado de la petición (y, para las admisiones auditadas a través del proxy inverso, el encabezado de la respuesta del origen) se abre bajo demanda, mostrado como UTF-8 o Latin-1, con la condición de la respuesta indicada (capturada, local, no observada en forward-auth, origen no disponible) y un comando para copiarlo como cURL; solo se conservan los valores de las cabeceras enumeradas, y `--console-capture-header <name>` amplía esa lista. En otro caso, los encabezados figuran como no registrados. Las acciones de denegar y permitir abren el formulario de grupos IP con la dirección ya rellenada.

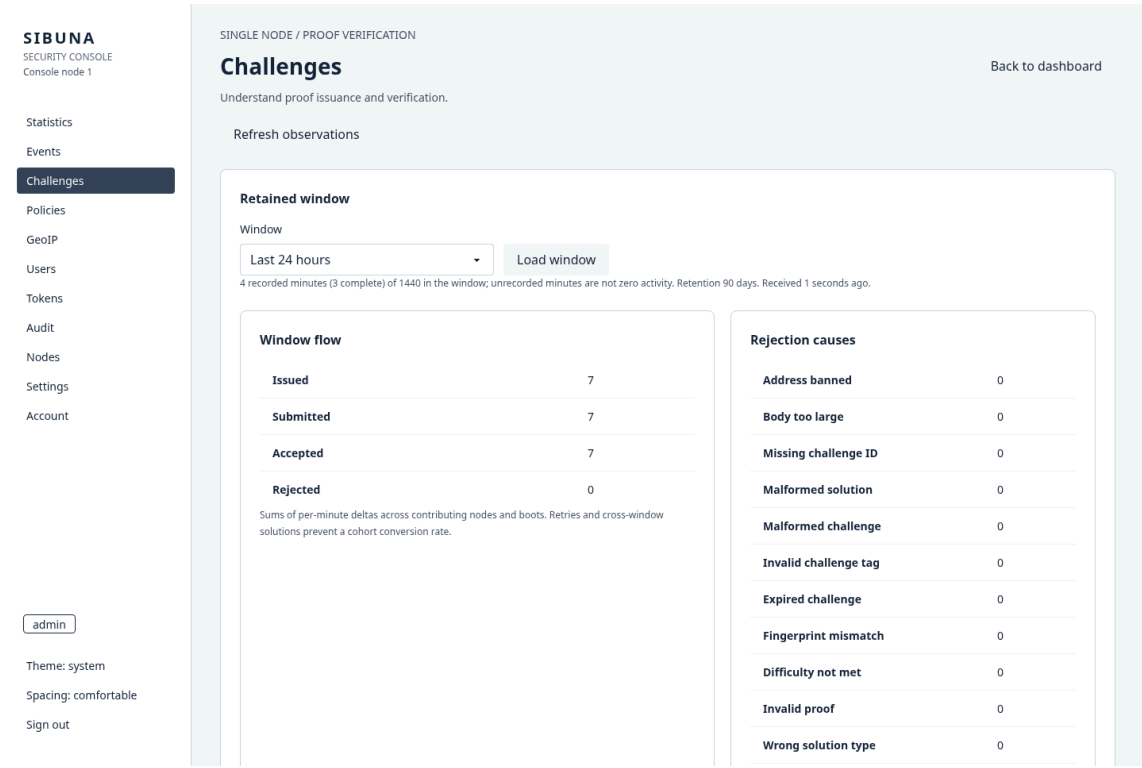


Fig. 17 Challenges. Una ventana conservada (una hora, un día o siete días) suma registros duraderos por minuto e indica cuántos minutos se registraron y estaban completos; aquí se emitieron, enviaron y aceptaron siete desafíos, y todas las causas de rechazo están a cero. Más abajo aparecen los totales en vivo desde este arranque, los parámetros configurados y los emitidos más recientemente, el histograma de tiempos de resolución aceptados desglosado por algoritmo y por intervalo de parámetros, las transiciones de dificultad adaptativa y los registros por dirección con atajos para denegar y permitir, conservados siete días.

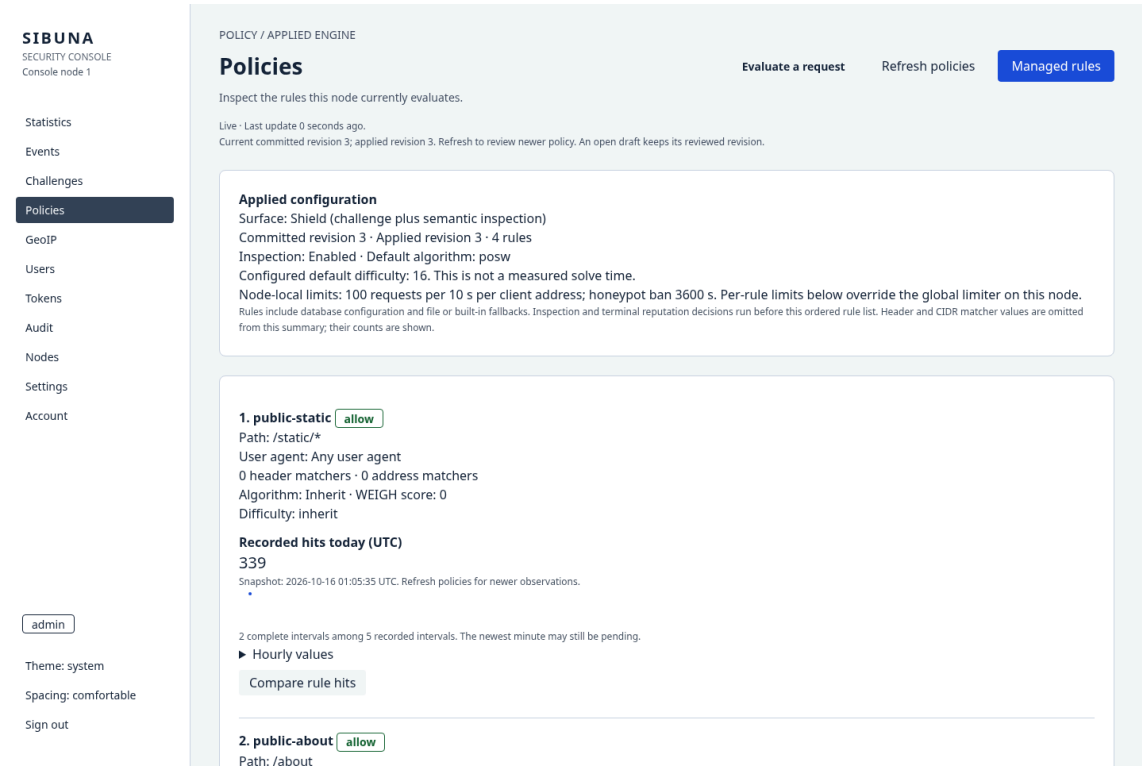


Fig. 18 Policies. El motor aplicado nombra su modo de protección, sus revisiones, su algoritmo y dificultad por defecto y sus límites locales del nodo, y después enumera las reglas en orden de evaluación con sus coincidencias de hoy y la cobertura de esos recuentos. Más abajo aparecen la matriz de modos de inspección, el probador de peticiones y los grupos IP. *Managed rules* abre el editor; guardar una edición muestra primero una revisión de los campos cambiados, y confirmarla valida el candidato completo y crea una nueva revisión.

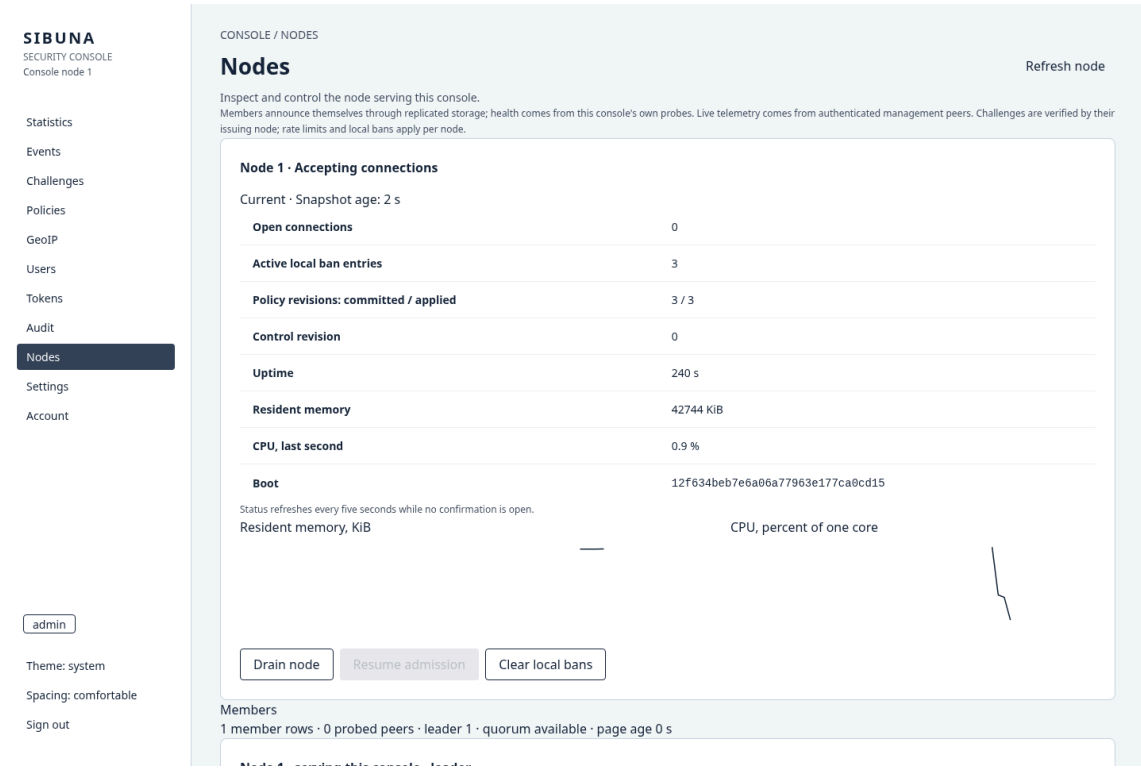


Fig. 19 Nodes. El estado y los comandos del nodo que sirve, con la memoria residente y la CPU del último segundo y sus minigráficas de sesenta segundos; debajo, cada miembro anunciado con su revisión de política aplicada frente a la revisión confirmada, su dirección de gestión y el resultado de su sondeo (SID 0007, SID 0013).

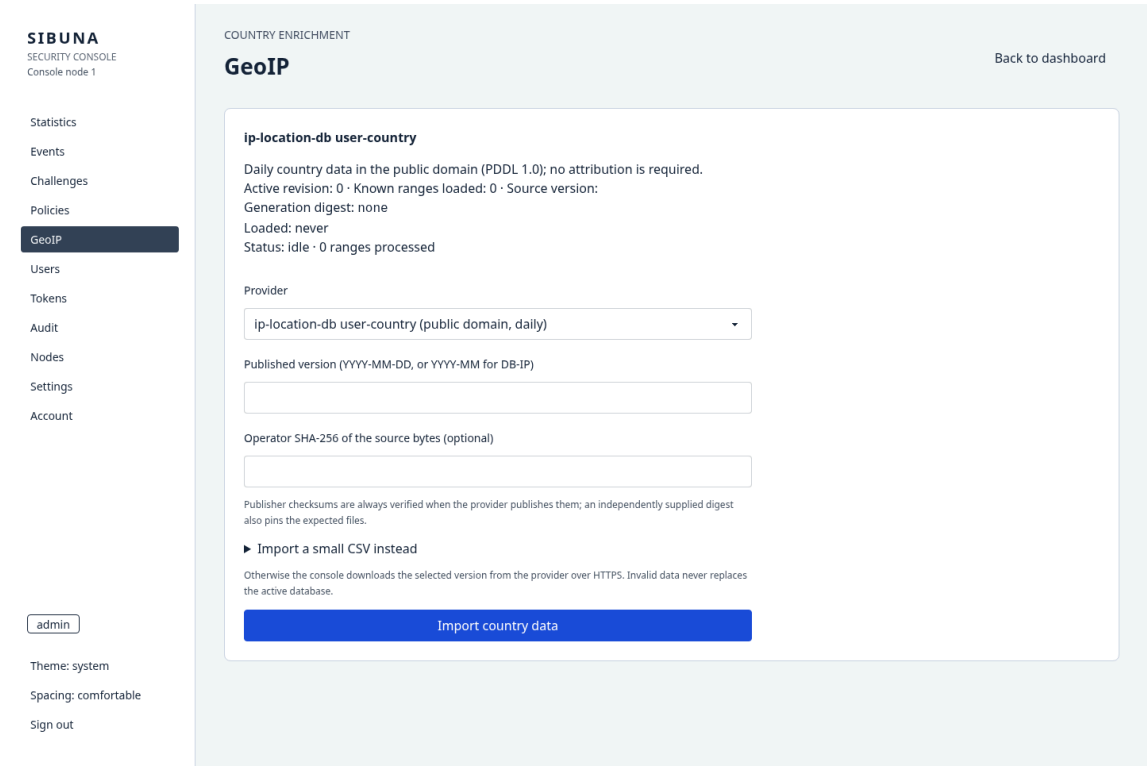


Fig. 20 GeoIP. Proveedor, licencia, resumen criptográfico de la generación activa, hora de carga y el formulario de importación; aquí aún no se ha cargado ninguna generación. Una importación fallida nunca sustituye a la generación activa.

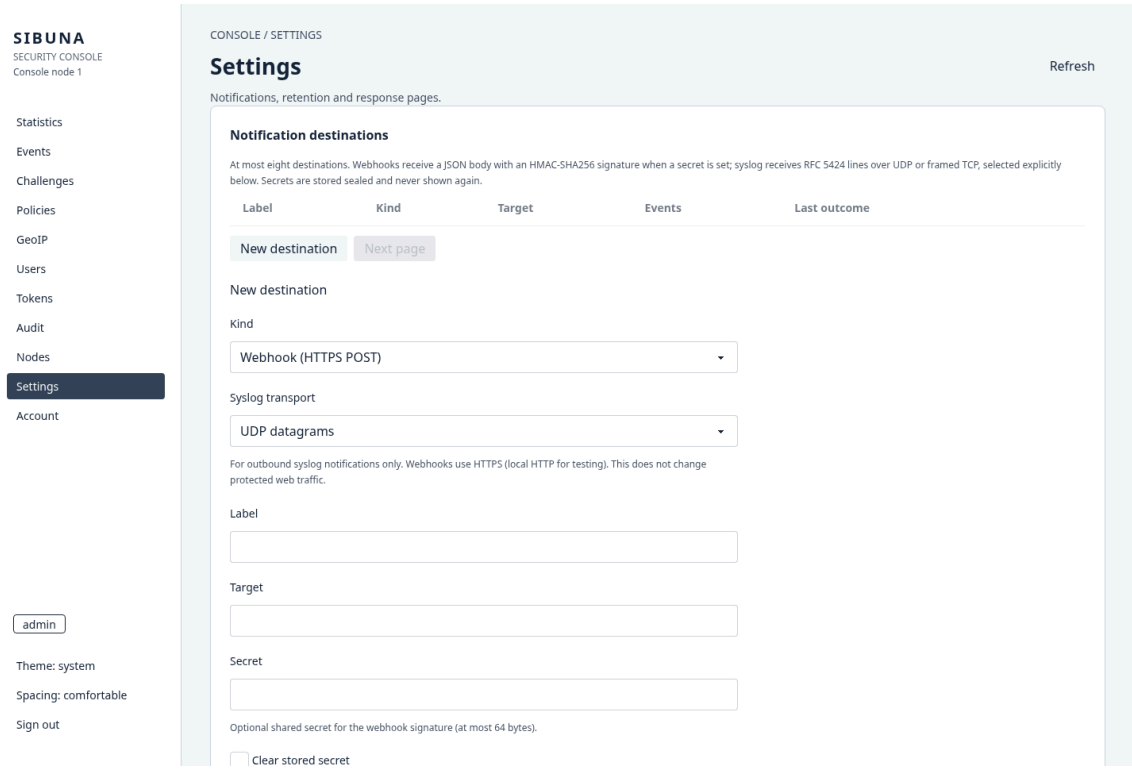


Fig. 21 Settings, todavía sin destino de notificaciones y con el formulario para crear uno. Más abajo en la página aparecen los umbrales de picos de denegaciones, la retención con una confirmación explícita por ventana, las plantillas de páginas de respuesta y About.

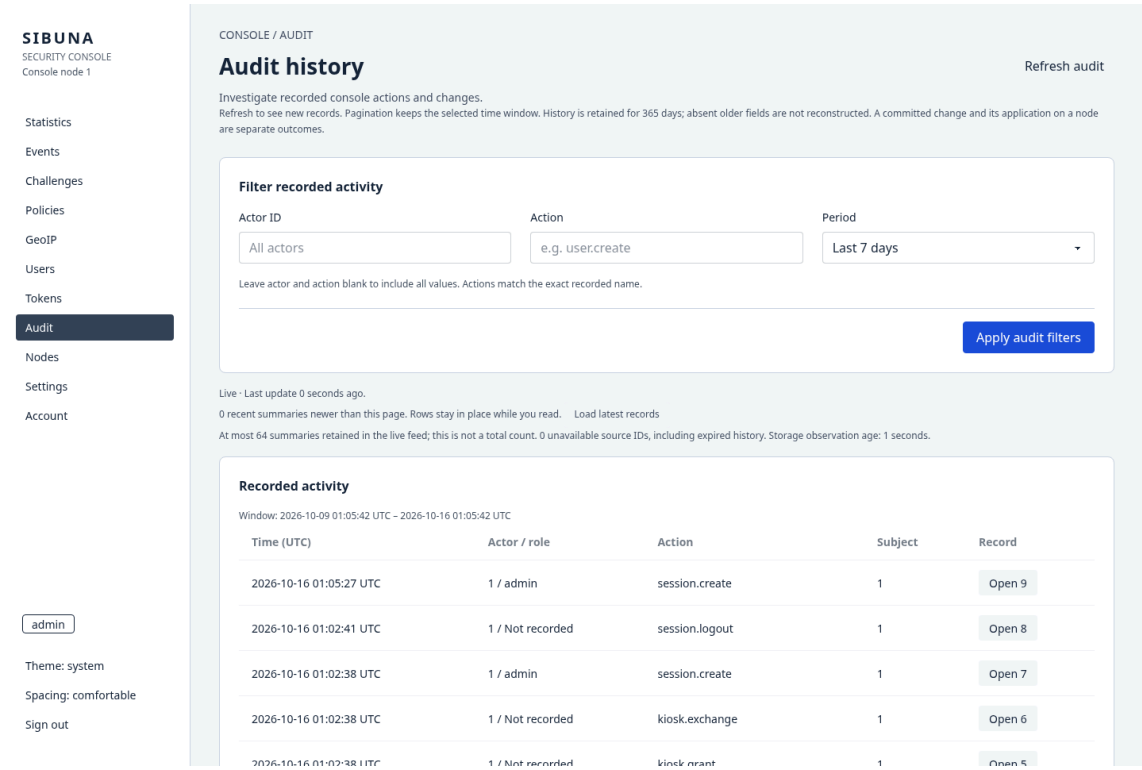


Fig. 22 Audit. Historial de solo adición con actor y rol, acción, sujeto y una vista de detalle con resúmenes enmascarados del antes y el después; también se registran los inicios de sesión rechazados. Cada modificación de gestión y cada fila de autenticación registran la dirección del cliente que presentó la credencial; las filas que escribe el propio sistema la muestran como no registrada. Un registro de política ofrece *Revert this change*, que restaura el documento registrado antes de esa revisión como una revisión nueva y auditada tras una confirmación, y se rechaza si el conjunto de reglas ha cambiado desde entonces.

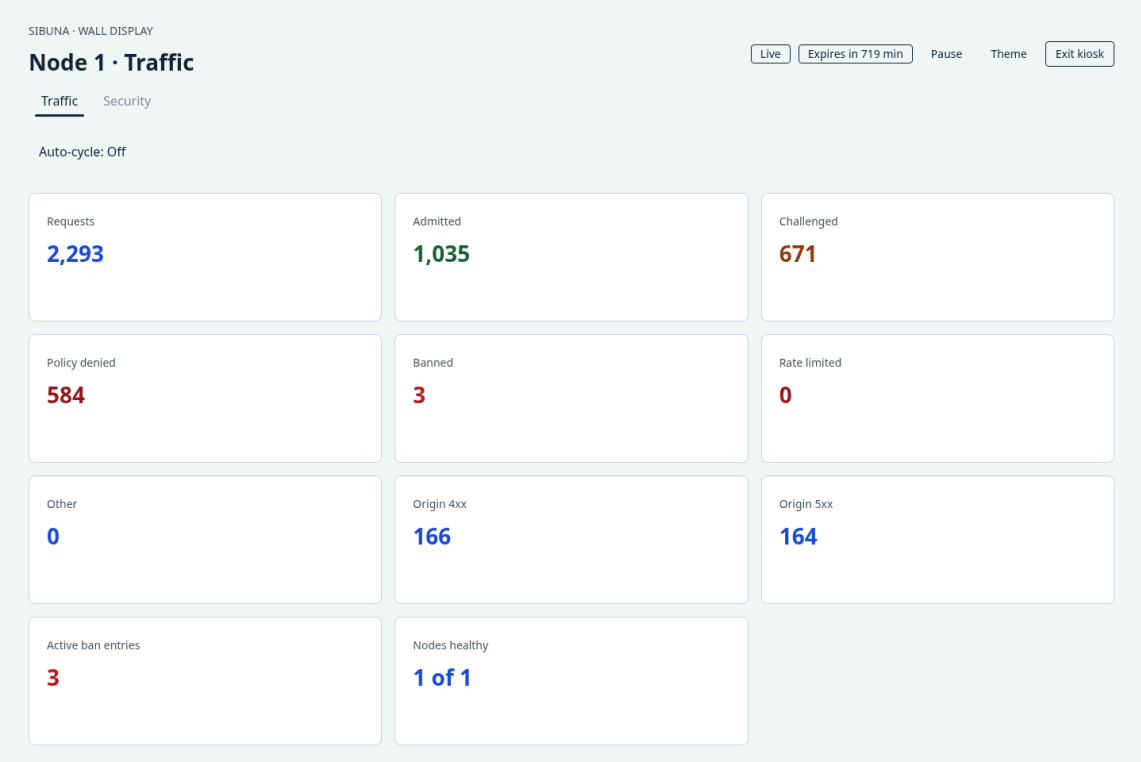


Fig. 23 Una pantalla mural. Una sesión de quiosco muestra solo las vistas Traffic y Security, sin navegación, e indica cuándo caduca.

La opción `--console-location <latitude,longitude>` declara la posición de este nodo, por ejemplo `1.3521,103.8198` para un despliegue en Singapur. El globo se centra inicialmente allí; Center Sibuna vuelve a ese punto después de girarlo. La actividad de los países sigue arcos animados de círculo máximo hacia el nodo, con recorte del hemisferio posterior y de la línea de cambio de fecha en el mapa plano. Las posiciones de los países son centroides representativos, y las flechas representan la ventana observada de sesenta segundos, no conexiones individuales. Una posición del servidor sin configurar sigue siendo explícitamente desconocida. El marcador configurado sigue visible durante las caídas de la telemetría; el tráfico obsoleto no se anima.

Las vistas previas de políticas construyen un candidato privado que incluye las reglas configuradas en archivo. Al guardar se compara la revisión esperada y se exige confirmar una comparación de campos. Las reversiones históricas se comparan con la regla actual y crean una revisión nueva. Las revisiones confirmadas y las aplicadas localmente son distintas. Los detalles de auditoría conservan cambios de decisión acotados y el rol efectivo que actuó. Los valores de los comparadores se enmascaran, y los registros antiguos sin contexto siguen marcados explícitamente como ausentes. Drenar, reanudar y borrar los bloqueos locales requieren una vista previa del comando. Un comando que llega mientras otro se ejecuta en el nodo se rechaza con `503 CONSOLENODE` en lugar de ponerse en cola. Inspecciona el recibo duradero tras una respuesta perdida antes de reintentar; una intención confirmada no demuestra por sí sola que el efecto en tiempo de ejecución haya terminado.

Para operar en remoto, usa un proxy HTTPS y configura `--console-origin`, `--console-behind-proxy`, CIDR explícitos de `--console-trusted-proxy` y `--console-key-file`. El archivo de clave

contiene 64 caracteres hexadecimales, tiene permisos solo para el propietario y protege los secretos guardados del segundo factor; consérvalo entre reinicios, separado de la semilla de desafíos. El origen HTTPS configurado y la lista de proxies permitidos son obligatorios para el acceso fuera de loopback. Sibuna tampoco termina TLS entrante en el puerto de escucha de la consola; la consola usa TLS solo para sus conexiones salientes, mediante rustls (SID 0012). La CLI nativa accede a la consola en ejecución mediante los mismos contratos de autorización y almacenamiento. Lee las credenciales de archivos privados, no de valores en la línea de comandos:

```
sibuna console geoip update --version 2026-09-09 \
    --origin http://127.0.0.1:19446 --username admin \
    --password-file ./admin-password
```

Usa un archivo de contraseña accesible solo por su propietario, completa primero la configuración de la contraseña y añade `--factor-file` para un código de autenticador o de recuperación cuando se exija; `--token-file` sustituye al nombre de usuario y la contraseña en las operaciones de cuentas y de GeoIP. El cliente habla HTTP/1.1, exige HTTPS salvo para una dirección loopback literal, nunca sigue redirecciones y limita cada petición a 20 segundos y cada respuesta a 16 KiB. `geoip status` lee la generación activa. Las actualizaciones descargan por HTTPS los archivos publicados del proveedor seleccionado (por defecto el conjunto de datos de dominio público `user-country`, o DB-IP Lite con `--provider dbip --version YYYY-MM`), verifican las sumas de comprobación que publica el proveedor (o la indicada con `--checksum`), validan rangos acotados mediante `sibuna-geoip`, persisten la nueva generación y la activan localmente. Las actualizaciones fallidas conservan la generación anterior. Un tiempo de espera de la CLI (`--timeout`, por defecto 1200 segundos) deja de esperar; no cancela el trabajo de almacenamiento enviado. DB-IP Lite exige la atribución CC BY 4.0, que se muestra solo mientras sus datos están activos. Una compilación con `SIBUNA_GEOIP_DATA=<snapshot>` incorpora una instantánea validada (escrita por la herramienta `geoip-snapshot`) hasta la primera importación duradera; una compilación sin la consola la rechaza con `GEOIP001`. Las direcciones desconocidas, la pérdida de muestras y los datos obsoletos siguen siendo visibles; importar datos de países no crea tráfico ni enriquece muestras ya caducadas.

Los miembros del clúster se anuncian mediante la tabla de pertenencia de la base de datos compartida (`console_nodes`) al arrancar, cada minuto y en los dos segundos siguientes a un cambio de la revisión de política o de control aplicada, y aparecen en la página Nodes de todas las consolas. No hay elección de líder: un nodo SQLite informa del rol `single` y se declara líder a sí mismo, y un nodo PostgreSQL informa del rol `unknown` y que no hay líder, porque la conmutación por error de PostgreSQL no es visible para Sibuna (SID 0007). Configura `--console-probe <node-id>=<http://ip:port>` para los pares cuyas direcciones de escucha del plano de datos debe comprobar esta consola y `--console-advertise <origin>` para el enlace que muestran las otras consolas. La telemetría directa en vivo usa `--console-peer <node-id>=<https://origin>` (repetible, como máximo ocho) y un `--console-peer-key-file` dedicado. Los pares requieren `--console-behind-proxy`. Ambos extremos deben incluirse mutuamente en su lista y usar un proxy de entrada HTTPS de confianza. La clave de los pares es un archivo accesible solo por su propietario que contiene 64 caracteres hexadecimales, aprovisionado con independencia de la clave de la consola y de la semilla de desafíos. Los pares se comunican por

un WebSocket TLS autenticado, mediante rustls con el proveedor `ring` (SID 0012). `--console-peer-ca-file` aporta opcionalmente anclas de confianza PEM para una PKI de gestión privada; en otro caso, el cliente usa las raíces del sistema. La verificación del nombre de host se aplica siempre y no se puede desactivar: el host de cada origen de gestión, un nombre DNS o un literal de dirección IP, debe figurar en el certificado del par. Un certificado rechazado marca al par como `rejected` y escribe una fila de auditoría `peer.rejected`. La rotación de claves y un cambio del archivo de la CA requieren un reinicio coordinado; el certificado de un par se puede renovar sin él. La API de Nodes y su suscripción informan de la antigüedad de la recepción, el desfase de reloj, los cambios de arranque y la pérdida de muestras. Las observaciones que faltan siguen sin estar disponibles y los valores desconectados siguen marcados como obsoletos; las estadísticas recibidas nunca pasan a ser la contribución propia de otro nodo. El *Live traffic scope* del panel selecciona un nodo o combina los nodos configurados. *Node coverage and locations* identifica las fuentes ausentes, obsoletas y con desfase de reloj, las horas de observación y los destinos configurados. Las clasificaciones por país muestran la incertidumbre debida a las filas de origen omitidas. Las flechas conservan su nodo receptor, y no se inventan las ubicaciones de nodo que faltan. La tasa combinada sigue sin observarse cuando alguna fuente que contribuye carece de un intervalo consecutivo. Selecciona un nodo para inspeccionar sus segundos conservados y las clasificaciones de rutas del minuto actual. Las consultas remotas usan la conexión autenticada con el par; la respuesta a una consulta debe caber en un mensaje WebSocket de 4 KiB, así que la consola que sirve reduce a la mitad las filas de una página que no cabe. Los pares ausentes siguen sin estar disponibles y los cursores del historial no pueden cruzar un reinicio de nodo. El historial por minuto mantiene su propia selección de nodos y sus filas por nodo. La navegación y la cabecera móvil identifican el nodo de consola que sirve con independencia de la fuente de tráfico seleccionada. Los comandos locales siguen siendo por nodo.

Los recuadros de tráfico se abren sobre las últimas 24 horas de registros conservados de minutos cerrados de los nodos seleccionados. El control del periodo también ofrece una hora, siete o noventa días y los totales en vivo desde el arranque. Las desviaciones respecto a ayer requieren una cobertura completa y coincidente de todos los nodos seleccionados. El globo y las minigráficas siguen describiendo su propia ventana en vivo de 60 segundos. Un escaneo conservado se actualiza un minuto después de terminar, y mantiene visibles sus valores anteriores y su antigüedad hasta que se sustituyen; cambiar la fuente o el periodo descarta el ámbito anterior. El historial ausente nunca se convierte en cero.

Compare retained traffic muestra dos ventanas de minutos cerrados una junto a otra: la misma ventana de ayer, el periodo anterior o un segundo nodo. Los controles de duración y de desplazamiento final fijan ambos rangos UTC. Cada acción lee como máximo dieciséis páginas compactas por lado, con 96 registros guardados por página. Esto cubre una ventana de 24 horas sin reinicios solapados en un solo lote; Continue conserva los mismos límites para escaneos más largos. La cobertura completa, la cobertura incompleta y el historial ausente siguen siendo explícitos. Los porcentajes de tasa requieren intervalos completos y sin solapamiento en ventanas UTC iguales, normalizados por los milisegundos observados; una referencia cero con tráfico nuevo muestra "New". El periodo principal de Traffic sigue siendo independiente. El formato por minuto no registra el modo de proxy histórico, así que las comparaciones de respuestas del origen siguen sin estar disponibles en lugar de interpretar los ceros de forward--auth como respuestas observadas.

Compare retained path rankings selecciona dos ventanas cerradas o dos nodos. El nodo 0 incluye todos los nodos conservados, incluidos los miembros retirados. Una acción lee como máximo dieciséis archivos inmutables, comprueba sus identidades y sumas de comprobación, y combina todos los contadores antes de seleccionar las filas que se muestran. Los archivos se sellan tras la ventana de 60 segundos para muestras tardías, así que el minuto cerrado más reciente aún puede estar pendiente. Continue conserva las ventanas y el cursor originales. Los límites describen prefijos de ruta muestreados, no totales exactos de peticiones. La página muestra los escaneos parciales, el truncamiento, los límites de retención y la pérdida de cola notificada; los archivos ausentes no pueden demostrar tráfico cero ni una cobertura completa. Security se abre sobre las últimas 24 horas y usa sesenta intervalos de tiempo iguales sobre el periodo seleccionado. Sus agregados se ejecutan en el hilo de almacenamiento bajo una cota de trabajo. En SQLite la cota es `--console-query-steps` pasos de la máquina virtual (de 100.000 a 50.000.000, por defecto cuatro millones). Fíjala junto con `--console`; los valores no válidos o repetidos se rechazan. En PostgreSQL la cota es un tiempo límite de sentencia de 2.000 ms fijado en la transacción de la lectura (SID 0013); cambiar `--console-query-steps` no lo modifica. Las consultas agotadas piden un periodo más estrecho. Los recuentos de pasos acotan el trabajo de la consulta, no el tiempo transcurrido. Los gráficos comparten escala; despliega los valores de Trend para ver los inicios de los intervalos UTC y los recuentos agrupados exactos. Son hallazgos conservados, así que la falta de cobertura de incidentes no puede interpretarse como ausencia de ataques.

Las pantallas murales usan sesiones de quiosco. En Account, un operador da nombre a la pantalla y selecciona Create display code. La pantalla pega el código de un solo uso en el formulario de inicio de sesión en un plazo de diez minutos; su sesión es de solo lectura, se limita a las estadísticas y caduca en un máximo de doce horas. La página de la cuenta borra el código mostrado al navegar o al ocultarse. Ocultarla no revoca una concesión sin usar; el plazo de canje sigue vigente. Los códigos nunca deben ir en las URL. Traffic y Security comparten la pantalla: Security muestra tendencias agregadas de los módulos y los resultados de las peticiones sin direcciones de incidentes ni pruebas de carga útil. La rotación automática es opcional, está desactivada al principio y se suspende con movimiento reducido, datos obsoletos o Pause.

Los destinos de notificación (webhooks firmados y syslog para picos de denegaciones, bloqueos, miembros inalcanzables y cambios de líder) están en Settings; los secretos de los webhooks necesitan `--console-key-file`. El tipo de cambio de líder sigue definido para que los destinos guardados conserven su significado, pero ningún nodo lo genera, porque no hay elección de líder. Cada destino tiene su propio periodo de espera y un presupuesto de tres intentos. Los registros de auditoría de entrega muestran los resultados; el historial de la cola completada se limita a siete días y 4.096 eventos. Los webhooks incluyen una `Idempotency-Key` estable para que los receptores puedan suprimir efectos repetidos tras una finalización de red incierta. Los webhooks se envían por HTTPS, o por HTTP sin cifrar solo a una dirección loopback literal, y el cuerpo de la respuesta se lee hasta 64 KiB. La selección UDP/TCP de syslog se aplica solo a las notificaciones salientes, con independencia del tráfico web protegido. Las pruebas manuales registran la intención y la finalización en Audit y actualizan el resultado del destino; una finalización de auditoría no confirmada se muestra explícitamente antes de que un operador reintente.

Settings también controla la retención: de uno a 90 días para el historial por minuto, de uno a siete para las clasificaciones, de uno a 30 para los incidentes y de uno a 365 para la auditoría. Los valores superiores son los valores por defecto; las clasificaciones mantienen su cuota de 512 MiB. Guardar un valor de retención requiere confirmación porque la limpieza elimina definitivamente los registros más antiguos en lotes acotados. Aumentar el valor después no restaura el historial borrado. Las ediciones obsoletas se rechazan y los cambios correctos aparecen en Audit.

Las páginas de respuesta (desafío, denegada, tasa limitada, bloqueada, sobrecargada) se pueden editar en Settings como HTML acotado con marcadores fijos; los borradores se previsualizan en una pestaña aislada y las páginas guardadas se sirven a partir de la siguiente instantánea de políticas.

Flujos de trabajo de políticas en la página Policies: reordenar reglas gestionadas, reproducir un borrador contra los hallazgos de inspección conservados, gestionar grupos IP y bloqueos de países fijados a la generación GeoIP activa, y exportar o importar de forma atómica el conjunto gestionado (también `sibuna console policies export` y `sibuna console policies import --file <set.json>`). Una importación es una petición que lleva de 1 a 128 documentos de 4 KiB como máximo cada uno y 256 KiB como máximo en total; la nota de la página indica la importación por línea de comandos para conjuntos de más de unos 96 KiB. Las reglas aplicadas muestran las coincidencias registradas hoy, minigráficas por hora y una tabla accesible. Una coincidencia es una evaluación correcta de un comparador declarativo: cuentan las reglas WEIGH que coinciden y la primera coincidencia terminal; una decisión anterior de inspección o de reputación puede impedir la evaluación. Las pruebas privadas no incrementan estos contadores. *Compare rule hits* fija dos periodos cerrados para un nodo registrado. El historial de revisiones ofrece *Compare hits around this edit*, que excluye el minuto de la edición y usa periodos disponibles iguales hasta la duración elegida. Los filtros opcionales por revisión aplicada mantienen las generaciones no relacionadas fuera de la comparación. El arranque, la conmutación, las escrituras ausentes y la retención dejan huecos de cobertura visibles; los porcentajes requieren cobertura completa en ambos lados. Las observaciones de reglas y los resúmenes por hora y por día siguen el ajuste de retención por minuto. Un cambio observado en torno a una edición es una comparación, no una prueba de que la edición causara el cambio de tráfico.

Importar una generación GeoIP posterior no actualiza automáticamente las filas de reputación existentes derivadas de países. Previsualiza la acción del país para comparar los prefijos añadidos, conservados y eliminados; recorre página a página la diferencia revisada antes de aplicarla. La sustitución elimina las filas obsoletas que pertenecen a ese país y conserva los prefijos gestionados de forma independiente. Una generación o una revisión de política cambiadas requieren una vista previa nueva, y las ediciones independientes que se solapan se rechazan. La página Events contiene los hallazgos WAF conservados y los incidentes del honeypot; no es un registro de accesos completo. Con GeoIP cargado, el hilo de almacenamiento registra el país y la generación de cada incidente en la transacción del lote de incidentes. Es una atribución en el momento de persistir, no una ubicación reconstruida en el momento de la petición. Las importaciones posteriores no modifican las asignaciones registradas. El filtro de país acepta un código de dos letras en mayúsculas, `unknown` para una dirección ausente de la generación cargada, o `not_recorded` para un incidente sin datos de asignación. Los grupos de

origen indican explícitamente los países mixtos o la cobertura. La acción *View events* del globo mantiene el nodo seleccionado y abre los incidentes conservados del país durante la última hora. La vista *Security* de la página Statistics combina las tasas en vivo de límites de tasa, desafíos y bloqueos con los hallazgos conservados de inspección y del honeypot durante una hora, un día, siete días o treinta días. Los enlaces de categoría, origen y ruta abren el flujo normal de incidentes con el mismo nodo y un límite temporal fijo. Aplicar filtros de incidentes inicia un periodo nuevo. Los hallazgos incluyen registros de auditoría y son distintos de los totales de peticiones bloqueadas; la atribución ausente de reputación y de coincidencias de reglas se muestra como no registrada. Los filtros de eventos y de auditoría mantienen las etiquetas junto a sus controles, alinean la acción Apply por separado y adaptan sus columnas al ancho de contenido disponible. Un único WebSocket autenticado sobrevive a la navegación y transporta estadísticas, resúmenes de incidentes, el estado de los nodos, revisiones de políticas, desafíos y resúmenes de auditoría. Los nuevos registros de incidentes y de auditoría esperan detrás de *Load latest records* para que la tabla no se mueva mientras se lee. Las actualizaciones de políticas muestran las revisiones confirmadas y aplicadas actuales sin sustituir un borrador abierto. Los contadores de flujo se actualizan en vivo; actualiza explícitamente una partición de tiempos de desafío seleccionada que no sea la predeterminada. Las consultas históricas, las lecturas de detalle y las modificaciones siguen siendo peticiones HTTP.
Los fragmentos de página se pueden guardar como marcadores; Back y Forward vuelven a abrir páginas autenticadas. La barra lateral también recuerda en este navegador las preferencias de tema y de espaciado, con System como tema por defecto. SID 0007 está en estado Committed. La consola supera sus fixtures de referencia y sus flujos de trabajo en el navegador en Chromium sin interfaz gráfica (`cargo xtask browser console`, criterio 5 de SID 0010). Su batería de pruebas de extremo a extremo es `cmd/sibuna/tests/console_*.rs`, cuyos escenarios de clúster PostgreSQL se ejecutan cuando se define `SIBUNA_TEST_DATABASE_URL`. SID 0010 registra que, con la consola compilada y desactivada, solo con almacenamiento, o activada y en reposo, las 16 cargas de trabajo del producto se mantienen a menos de un 1% del daemon compilado sin ella, lo que está dentro del ruido. El criterio formal de aislamiento de SID 0007, con ocho paneles activos bajo carga, aún no se ha medido (parte VIII). La aplicación WebAssembly completa tiene un techo de 768 KiB sin comprimir, que el script de compilación del daemon impone haciendo fallar la compilación, y un umbral de revisión de 640 KiB. En la 1.0.0-beta.3 el módulo mide 724.330 bytes (272.568 con gzip), por encima del umbral de revisión, lo que SID 0007 registra para su revisión. Estos límites del proyecto dejan margen para los flujos de trabajo de la consola; no sustituyen a las mediciones de carga y de capacidad de respuesta en el navegador. La memoria lineal del módulo crece bajo demanda hasta un máximo declarado de 16 MiB.

Topologías de despliegue

> **Objetivos** Desplegamos el proxy inverso autónomo y el validador forward-auth detrás de Nginx o Caddy.

Proxy inverso

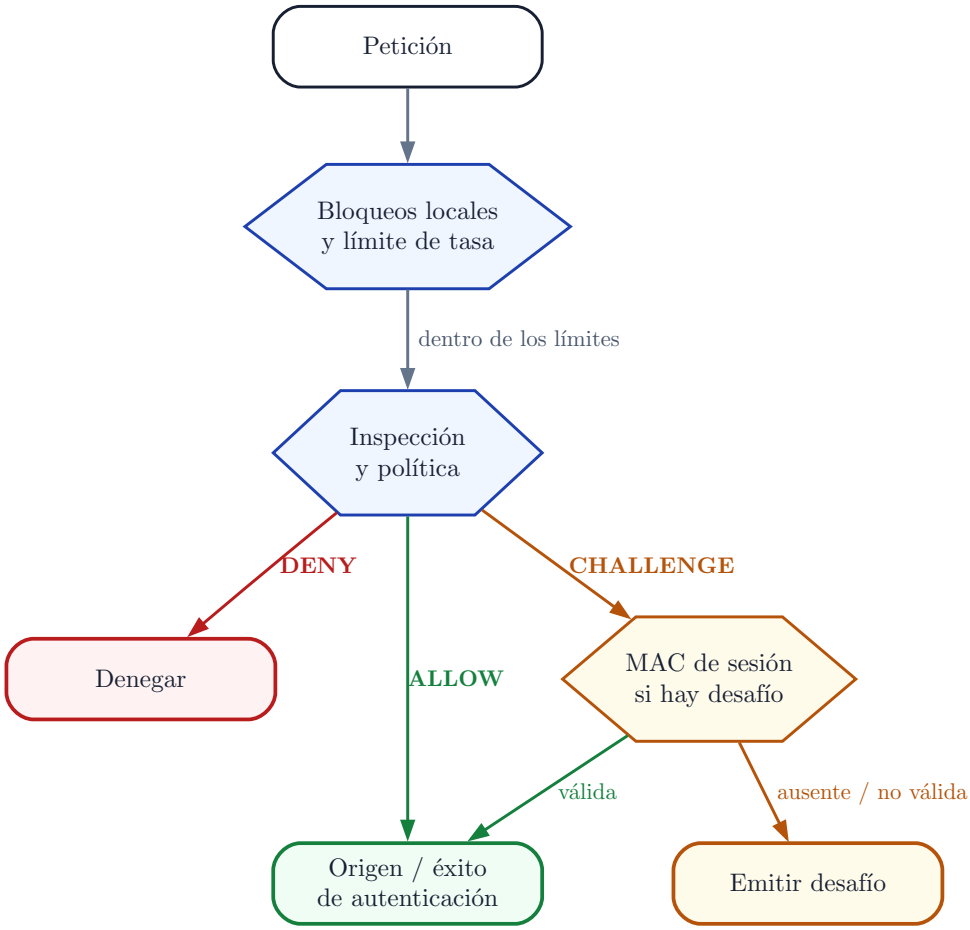


Fig. 24 Enrutamiento de peticiones en el modo de protección Shield

```
sibuna --port 80 --upstream-host 127.0.0.1 --upstream-port 3000 \
       --secret-file /etc/sibuna/secret --difficulty 16 --algorithm posw
```

Las peticiones admitidas llegan al origen con las cabeceras `X-Forwarded-For`, `X-Forwarded-Proto`, `X-Real-IP`, `X-Sibuna-Status` y `X-Sibuna-Rule`. Se eliminan las cabeceras hop-by-hop, los campos que escribe el propio Sibuna y los campos de reenvío alternativos (`Forwarded`, `X-Forwarded-Host`, `X-Forwarded-Port`), también cuando un cliente escribe sus nombres con guiones bajos (`X_Forwarded_For`). `X-Real-IP` es el cliente según el cual se juzgó la petición. `X-Forwarded-For` es solo el par del socket, nunca una entrada escrita por el cliente; cuando la identidad reenviada es de confianza, es la cadena recibida seguida del par del socket. Por tanto, los orígenes deben leer el cliente de `X-Real-IP` o desde la derecha de `X-Forwarded-For`, nunca de su entrada más a la izquierda. La ruta que recibe el origen es la forma normal que juzgó la política (SID 0009); una ruta sin forma normal recibe `400`.

Las actualizaciones a WebSocket de HTTP/1.1 pasan por las comprobaciones de admisión y de políticas antes de que se acepte el handshake del origen. La negociación de subprotocolos

y extensiones sigue siendo de extremo a extremo. Dos hilos retransmiten los bytes, uno por dirección, cada uno con un búfer fijo, incluidos los bytes leídos por adelantado y los cierres parciales; por tanto, una conexión actualizada ocupa dos hilos, y el número de hilos de conexión puede llegar al doble de `--max-connections` cuando todas las conexiones están actualizadas. Los sockets actualizados siguen dentro de la cuota normal de conexiones y del registro de apagado, pero usan `--websocket-idle-timeout` (300 segundos por defecto) con independencia del plazo de inactividad HTTP. El tráfico en cualquiera de las dos direcciones renueva esa cota, pero un par que no acepta ningún byte durante todo el periodo termina la retransmisión, por muy activa que esté la otra dirección. Las tramas posteriores al handshake no son entradas de la inspección WAF. HTTPS/WSS usa un proxy de entrada que termina TLS delante del puerto de escucha HTTP/1.1 privado de Sibuna; Sibuna no termina por sí mismo el TLS del navegador.

El proxy de entrada puede negociar HTTP/2 con los navegadores y reenviar HTTP/1.1 a Sibuna. El soporte nativo de HTTP/2 en Sibuna se aplaza a una actualización posterior; este esquema no ofrece semántica HTTP/2 de extremo a extremo para aplicaciones como gRPC nativo.

Los cuerpos de petición con Content-Length fluyen a través de búferes fijos, conservando los bytes y las cabeceras MIME. Las subidas multipart conservan los delimitadores, los nombres de campo repetidos, los nombres de archivo y los tipos de las partes. El WAF examina como máximo los primeros 8 KiB del cuerpo: los metadatos multipart y los campos que no son archivos siguen siendo entradas de la inspección de texto, mientras que el contenido de los archivos es opaco. Una parte cuyo `filename` está vacío es un campo ordinario. Un cuerpo de un tipo MIME de nivel superior binario reconocido (imágenes salvo SVG, audio, vídeo, PDF, ZIP, gzip, 7z, protobuf y octet-stream) es opaco solo cuando su prefijo también parece binario, es decir, cuando al menos un byte de cada dieciséis es un byte de control distinto de tabulador, salto de línea, tabulador vertical, salto de página y retorno de carro; en otro caso se inspecciona como texto (SID 0004). JSON, XML, SVG, los formularios URL-encoded y los tipos desconocidos mantienen la inspección de texto. Los metadatos MIME ambiguos o mal formados recurren a la inspección de texto. El análisis multipart se limita a 32 partes y 2 KiB de cabeceras por parte dentro de ese mismo prefijo, y acepta el relleno de transporte de RFC 2046 tras un delimitador.

Esto es compatibilidad con las subidas, no validación de archivos ni análisis de malware. El backend debe imponer los tipos MIME aceptados en lugar de fiarse de la declaración Content-Type del cliente. Los campos posteriores al prefijo de inspección, incluidos los que siguen a un archivo subido grande, no se inspeccionan. Las cargas útiles comprimidas no se descomprimen para inspeccionarlas. Las subidas siguen pasando las comprobaciones de admisión, ruta, consulta y cabeceras, y siguen sujetas a los plazos de conexión. Los cuerpos de petición chunked se decodifican dentro del búfer de la conexión antes de la inspección (SID 0009). Un cuerpo que termina dentro del búfer llega al backend con Content-Length. Si es más largo, Sibuna lo vuelve a dividir en chunks, uno por lectura, así que el backend nunca ve los tamaños de chunk, las extensiones ni los trailers del cliente. Una línea de chunk con un CR o un LF sueltos, espacios en blanco alrededor del tamaño, una extensión mal formada o más de 4 KiB se rechaza con 400. Lo mismo ocurre con una sección de trailers de más de 16 KiB. `Transfer-Encoding` con otra codificación recibe 501. Los backends que no pueden analizar peticiones chunked siguen

recibiendo Content-Length para cuerpos de menos de unos 44 KiB. Una conexión al origen que transportó un cuerpo de petición se cierra tras su intercambio y nunca vuelve al conjunto de conexiones. Los clientes que esperan `100-continue` lo reciben localmente antes de enviar el cuerpo; las expectativas no admitidas reciben 417. La cabecera Expect se consume antes de reenviar al backend.

Forward-auth detrás de un proxy de entrada

En `--mode forward_auth` el daemon responde a la subpetición del proxy de entrada con `200` (más las cabeceras de auditoría `X-Sibuna-Status`, `X-Sibuna-Rule` y `X-Sibuna-Rule-Hash`), `401` para un desafío, `403` para una denegación o `429` cuando se limita la tasa. El proxy de entrada debe reenviar la dirección del cliente y la URL original; el modo forward-auth confía en ellas por defecto. Asocia este puerto de escucha a una dirección privada para que solo pueda conectarse el proxy de entrada. `X-Forwarded-Uri` (Caddy) o `X-Original-URI` (Nginx), y `X-Forwarded-Method`, restauran la petición de la aplicación para evaluar las políticas; la ruta original se lleva a su forma normal, igual que la ruta de una línea de petición. Los campos de URL original duplicados o contradictorios, y una URI original que no está en forma de origen, se rechazan con `400`. Las rutas internas del daemon usan siempre la URI real de la petición. Forward-auth no inspecciona un cuerpo que el proxy de entrada omite.

```
map $http_upgrade $sibuna_connection_upgrade {
    default upgrade;
    '' close;
}
server {
    listen 443 ssl;
    location / {
        auth_request /__sibuna_auth;
        auth_request_set $sibuna_auth_status $upstream_status;
        auth_request_set $sibuna_retry_after $upstream_http_retry_after;
        auth_request_set $sibuna_status $upstream_http_x_sibuna_status;
        auth_request_set $sibuna_rule $upstream_http_x_sibuna_rule;
        auth_request_set $sibuna_rule_hash $upstream_http_x_sibuna_rule_hash;
        error_page 401 = @sibuna_challenge;
        error_page 500 = @sibuna_auth_error;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $sibuna_connection_upgrade;
        proxy_set_header Host $host;
        proxy_set_header X-Sibuna-Status $sibuna_status;
        proxy_set_header X-Sibuna-Rule $sibuna_rule;
        proxy_set_header X-Sibuna-Rule-Hash $sibuna_rule_hash;
        proxy_read_timeout 300s;
        proxy_pass http://127.0.0.1:3000;
    }
    location = /__sibuna_auth {
        internal;
        proxy_pass http://127.0.0.1:8080/;
        proxy_pass_request_body off;
        proxy_set_header Content-Length "";
        proxy_set_header X-Original-URI $request_uri;
        proxy_set_header X-Forwarded-Method $request_method;
```

```
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-Uri "";
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header User-Agent $http_user_agent;
        proxy_set_header Cookie $http_cookie;
    }
    location @sibuna_challenge {
        rewrite ^ /__sibuna/challenge break;
        proxy_pass http://127.0.0.1:8080;
        proxy_pass_request_body off;
        proxy_set_header Content-Length "";
        proxy_set_header X-Original-URI $request_uri;
        proxy_set_header X-Forwarded-Method $request_method;
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
    location @sibuna_auth_error {
        add_header Retry-After $sibuna_retry_after always;
        if ($sibuna_auth_status = 429) { return 429; }
        return 503;
    }
    location /__sibuna/ {
        proxy_pass http://127.0.0.1:8080;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
}
```

```
example.com {
    handle /__sibuna/* {
        reverse_proxy localhost:8080
    }
    handle {
        forward_auth localhost:8080 {
            uri /
            header_up X-Forwarded-For {remote_host}
            header_up -X-Original-URI
            copy_headers X-Sibuna-Status X-Sibuna-Rule X-Sibuna-Rule-Hash
        }
        reverse_proxy localhost:3000
    }
}
```

El espacio de nombres `/__sibuna/*` (página intermedia, desafío, verificación, recursos del solucionador) debe llegar directamente al daemon en ambas recetas. La página de error de nginx transmite la URI y el método originales para que la página intermedia lleve el requisito que decidió la autorización; sin ellos la página sigue funcionando, y la emisión recurre a evaluar la URL que indica el navegador. Caddy aporta por sí mismo los metadatos de URI y método originales. Los bloques `handle` exclusivos garantizan que las rutas de desafío y de verificación no entren en la comprobación previa de forward-auth. Nginx requiere las cabeceras Upgrade/ Connection explícitas que se muestran arriba para los WebSockets de la aplicación. El gestor de errores de Nginx sirve la ruta interna del desafío sin tomar una segunda decisión de autorización ni consumir el cuerpo de la subida. Su módulo de autenticación acepta directamente solo 2xx,

401 y 403, así que el ejemplo traduce un error de autenticación por límite de tasa de vuelta a 429 con Retry-After; los demás fallos de autorización siguen denegándose con 503. Nginx aporta su propio cuerpo 403, mientras que Caddy reenvía la página de denegación de Sibuna. Ambas recetas sustituyen las cabeceras de auditoría de Sibuna entrantes por la respuesta de autorización real. Configura en el proxy de entrada los certificados TLS normales y los límites de subida y de plazos de la aplicación; los ejemplos muestran la lógica de enrutamiento. El repositorio aún no prueba estas recetas contra proxies de entrada reales; las reglas de metadatos reenviados de las que dependen están cubiertas por pruebas unitarias en `sibuna-http`.

Huellas JA4 desde el proxy de entrada

Una huella JA4 resume cómo abre un cliente una conexión TLS. Se calcula a partir del primer mensaje de handshake del cliente, el ClientHello. La biblioteca TLS del cliente fija ese mensaje antes de que se envíe ningún byte HTTP, así que un cliente no puede cambiar su huella desde dentro de sus peticiones HTTP. Los navegadores, las bibliotecas y los bots difieren en ella. Esto hace de JA4 una señal más fuerte que un valor de `User-Agent`, que cualquier cliente puede escribir como quiera (SID 0014). JA4 lo definió FoxIO. Una huella tiene 36 caracteres en tres partes unidas por `_`:

```
t13d1516h2_8daaf6152771_02713d6af862
^^^^^^^^^^ ^^^^^^^^^^^^ ^^^^^^^^^^^^
a          b            c

a  protocolo (t TCP, q QUIC, d DTLS), versión de TLS (13 es TLS 1.3),
   d con SNI o i sin él, número de cifrados (dos dígitos),
   número de extensiones (dos dígitos), código ALPN (h2 es HTTP/2)
b  hash hexadecimal en minúsculas, de 12 dígitos, de los conjuntos
   de cifrado ordenados
c  hash hexadecimal en minúsculas, de 12 dígitos, de las extensiones
   y los algoritmos de firma ordenados
```

Solo el componente que termina TLS ve el ClientHello, así que solo él puede calcular JA4. Sibuna no termina TLS: sirve HTTP/1.1 sin cifrar detrás de un proxy de entrada, y el proxy de entrada habla TLS, HTTP/2 y HTTP/3 con los navegadores. Varios proxies de entrada ya calculan JA4. Por eso dejamos que el proxy de entrada calcule la huella y la envíe en una cabecera de petición, y Sibuna la lee de ahí.

`--ja4-header <name>` nombra esa cabecera. Está desactivada por defecto y no tiene nombre por defecto; siempre indicamos el nombre de la cabecera que usa el proxy de entrada. Sibuna guarda el nombre en minúsculas y lo compara sin distinguir mayúsculas. Rechaza los nombres de los campos que él mismo escribe o elimina (`X-Forwarded-*`, `X-Real-IP`, `X-Sibuna-*` y `Forwarded`, también con `_` en lugar de `-`). Cuando la opción está activa, el banner de arranque imprime una línea como `JA4: from cf-ja4 (trusted)`.

La cabecera solo se lee si los campos reenviados son de confianza. Es la misma decisión que rige `X-Forwarded-For`: `--trust-forwarded`, o el modo forward-auth sin `--no-trust-forwarded`. Si el par no es de confianza, la cabecera se ignora. Si es de confianza, una petición lleva una huella cuando está presente exactamente un campo con ese nombre y su valor tiene la forma de JA4. Un valor mal formado, o un campo repetido, deja la petición sin huella. Dos contadores lo reflejan: `sibuna_ja4_present_total` cuenta las peticiones con huella, y

`sibuna_ja4_invalid_total` cuenta las peticiones cuya cabecera estaba mal formada o repetida. Una petición de un par que no es de confianza no incrementa ninguno de los dos contadores. En modo proxy inverso, el origen solo ve la cabecera cuando Sibuna confía en ella. Si no confía en ella, Sibuna elimina todos los campos con ese nombre de la petición que envía al origen, también cuando un cliente escribe el nombre con `_` en lugar de `-`. Por tanto, un cliente no puede entregar al origen una huella falsificada con ese nombre. Si confía en ella, el campo pasa sin cambios con su grafía exacta, y un alias con `_` se sigue eliminando. Sin `--ja4-header`, un campo con ese nombre es un campo corriente.

> **El proxy de entrada debe controlar la cabecera** Una huella solo es tan fiable como el proxy de entrada que la escribió. Sibuna no puede distinguir un valor que calculó el proxy de entrada de un valor que envió el cliente. Por tanto, el proxy de entrada debe sobrescribir, o eliminar, cualquier copia de la cabecera que aporte el cliente, en todas las rutas que llegan a Sibuna. El puerto de escucha de Sibuna debe aceptar conexiones solo del proxy de entrada, igual que para `X-Forwarded-For`.

Cloudflare. La transformación gestionada "Add bot protection headers" añade `cf-ja4` a la petición enviada al origen, junto a `cf-bot-score`, `cf-verified-bot` y `cf-ja3-hash`. Requiere un plan Enterprise con Bot Management. Un proxy de entrada situado entre Cloudflare y Sibuna debe pasar el campo sin cambios. Arrancamos Sibuna con:

```
sibuna --ja4-header cf-ja4 --trust-forwarded -s /etc/sibuna/secret
```

El proxy de entrada situado delante de Sibuna debe aceptar conexiones solo de las direcciones de Cloudflare; de lo contrario, un cliente puede saltarse Cloudflare y enviar su propio `cf-ja4`.
nginx. El `ja4-nginx-module` de FoxIO calcula JA4 dentro de nginx y lo expone como `$http_ssl_ja4`. Necesita nginx 1.26 o posterior con el parche de captura del ClientHello del módulo, compilado con `--with-http_ssl_module`. `proxy_set_header` sustituye el campo del cliente con el mismo nombre, así que cada location que llega a Sibuna fija la cabecera por sí misma:

```
# Proxy inverso: Sibuna se ejecuta con --ja4-header x-ja4 --trust-forwarded
location / {
    proxy_set_header X-JA4 $http_ssl_ja4;
    proxy_set_header X-Forwarded-For $remote_addr;
    proxy_pass http://127.0.0.1:8080;
}

# Forward-auth: Sibuna se ejecuta con --mode forward_auth --ja4-header x-ja4
location = /__sibuna_auth {
    internal;
    proxy_pass http://127.0.0.1:8080/;
    proxy_set_header X-JA4 $http_ssl_ja4;
    # ... los demás campos de la receta de forward-auth anterior
}
```

En la receta de forward-auth, la misma línea `proxy_set_header X-JA4 $http_ssl_ja4;` va también en `@sibuna_challenge` y en `location /__sibuna/`. Una location que la omita pasa a Sibuna el propio `X-JA4` del cliente, y Sibuna confía en él.

Apache Traffic Server. El plugin `ja4_fingerprint` se activa con la línea `ja4_fingerprint.so` en `plugin.config`. No tiene opciones. Añade un campo `ja4` con la huella, y un campo `x-ja4-via` con el nombre del proxy, a las peticiones de cada conexión TLS. Arrancamos Sibuna con `--ja4-header ja4 --trust-forwarded`. La documentación del plugin no dice si sustituye un campo `ja4` que haya enviado el cliente. Debemos comprobarlo para nuestra versión, o eliminar el campo del cliente antes de que el plugin añada el suyo, antes de confiar en el valor.

Una regla comprueba una huella con el campo `ja4` (parte X, Esquema de políticas). El valor es o bien una huella completa, que debe ser igual, o bien un prefijo seguido de `*`, por el que debe empezar la huella. `"*"` por sí solo coincide con cualquier huella. Una petición sin huella nunca coincide con una regla `ja4`. El cargador rechaza un patrón con el que ninguna huella podría coincidir, como `T13D*` en mayúsculas, con `INVALID POLICY JA4`. La comparación opera sobre bytes y no asigna memoria.

```
{ "rules": [
    { "name": "deny-known-scanner", "action": "DENY",
      "ja4": "t13i010400_0f2cb44170f4_5c4c70b73fa0" },
    { "name": "weigh-no-sni", "ja4": "t13i*", "action": "WEIGH", "weight": 15 },
    { "name": "weigh-headless", "user_agent": "Headless", "action": "WEIGH",
      "weight": 30 },
    { "name": "challenge-tls12-checkout", "path": "/checkout/*", "ja4": "t12*",
      "action": "CHALLENGE" }
] }
```

La primera regla deniega exactamente una huella; el valor es un ejemplo, no un escáner conocido. Las dos siguientes añaden peso a la puntuación de la petición en lugar de decidir: un cliente TLS 1.3 que no envió nombre de servidor, y un navegador sin interfaz gráfica por su `User-Agent`. Si ninguna regla posterior decide antes, una petición que muestra ambas señales puntúa 45 y alcanza el umbral de denegación por defecto de 40, mientras que una petición que muestra solo una recibe un desafío. La última regla desafía, por prefijo, a todo cliente que negoció TLS 1.2 sobre TCP en la ruta de pago. Desafía en lugar de denegar porque muchos clientes legítimos más antiguos siguen usando TLS 1.2. Las reglas deberían combinar JA4 con otras señales de este modo. Un valor JA4 nombra una biblioteca TLS y su configuración, no una persona ni un dispositivo. Muchos clientes comparten una biblioteca, y las bibliotecas especializadas pueden imitar el ClientHello de un navegador. JA4 eleva el coste de la automatización; no demuestra la identidad. Por el mismo motivo, Sibuna asocia los bloqueos y la reputación a la dirección y nunca a la huella.

Los incidentes (denegaciones del WAF, accesos al honeypot y hallazgos de auditoría) registran la huella de su petición cuando esta llevaba una, en la columna `security_incidents.ja4`. La consola la muestra en la evidencia del incidente como "JA4 fingerprint", con el valor o "Not recorded". Los filtros de Events aceptan una huella o un prefijo. El editor de reglas gestionadas tiene un campo JA4, guardado en `policies.ja4_pattern`, y el formulario de prueba de políticas acepta una huella.

Política declarativa

> **Objetivos** Escribimos un archivo de políticas con reglas, pesos, umbrales, reglas IP y el interruptor del WAF.

```
{
  "default_action": "CHALLENGE",
  "waf": true,
  "thresholds": { "challenge_at": 10, "deny_at": 40, "bits_step": 5 },
  "ip_rules": { "10.0.0.0/8": "ALLOW", "192.0.2.0/24": "DENY", "2001:db8::/32":
"DENY" },
  "rules": [
    { "name": "deny-cf-workers", "headers": { "CF-Worker": ".*" }, "action":
"DENY" },
    { "name": "deny-amazonbot", "user_agent": "Amazonbot", "action": "DENY" },
    { "name": "api-with-key", "path": "/api/*", "headers": { "X-Api-Key": ".*" },
    "action": "ALLOW" },
    { "name": "protect-checkout", "path": "/checkout/*", "action": "CHALLENGE",
      "challenge": { "difficulty": 20, "algorithm": "posw" } },
    { "name": "headless", "user_agent": "Headless", "action": "WEIGH", "weight":
30 },
    { "name": "internal-vpc", "remote_addresses": ["10.0.0.0/8", "fd00::/8"],
    "action": "ALLOW" }
  ]
}
```

El cargador falla de forma segura (fail closed): una clave desconocida, una acción mal escrita, una dirección mal formada o un número fuera de rango rechazan todo el archivo, el diagnóstico nombra la regla y el campo, y el daemon no arranca. Una línea de comandos con una opción desconocida o un valor fuera de rango se rechaza de la misma manera. `rules` sustituye a la tabla integrada cuando está presente; `ip_rules` alimenta el trie de reputación, que escala a miles de prefijos; `waf: false` selecciona el modo de protección Gate desde el archivo. Otras dos claves forman parte del esquema (parte X): `inspection` fija cada categoría del WAF en `disabled`, `audit` o `enforce`, y el objeto `limits` de una regla le da una cuota propia. El campo `ja4` de una regla compara la huella JA4 que reenvió un proxy de entrada de confianza, de forma exacta o por un prefijo seguido de `*`; necesita `--ja4-header` (véase "Huellas JA4 desde el proxy de entrada"). Los patrones de ruta coinciden con la ruta de la petición en forma normal (SID 0003), así que un patrón debe escribirse en esa forma.

Almacenamiento persistente y clúster

> **Objetivos** Activamos el almacenamiento SQLite o PostgreSQL, añadimos una política dinámica y un bloqueo con SQL, consultamos los datos forenses de incidentes, entendemos la agrupación en campañas y ejecutamos varios nodos sobre una única base de datos PostgreSQL.

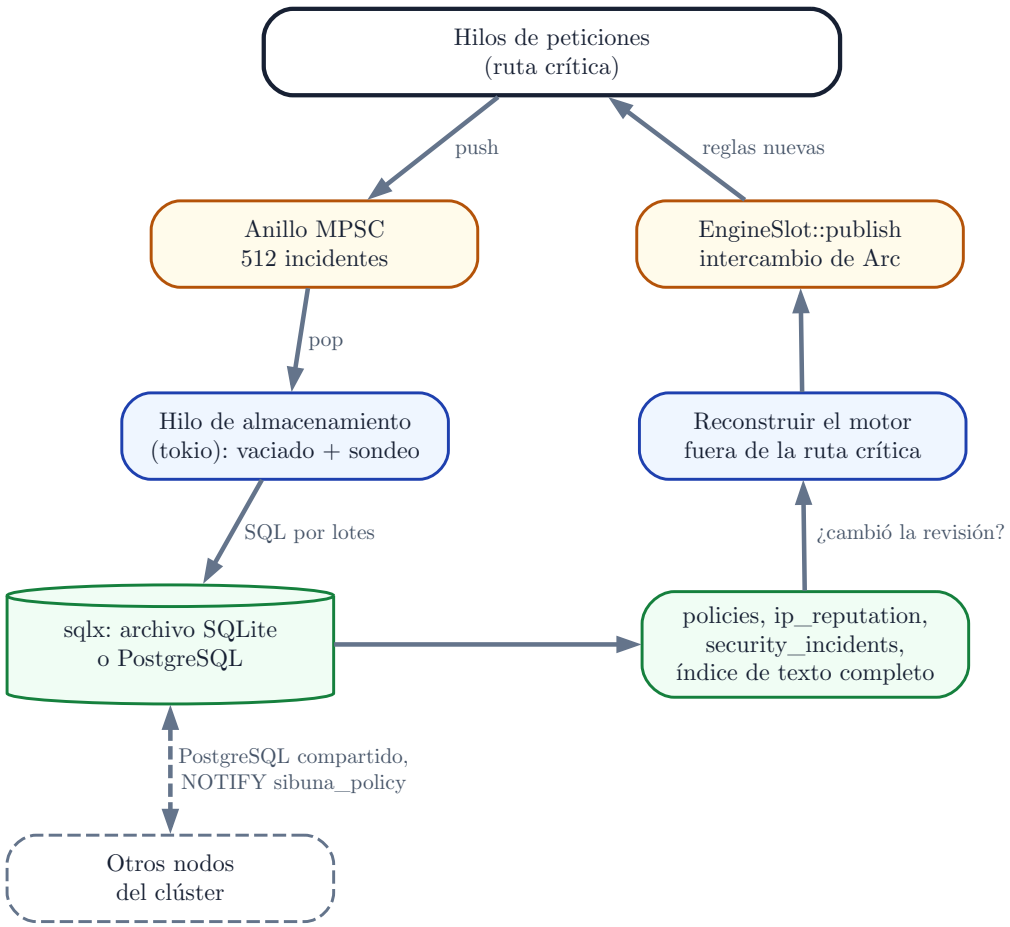


Fig. 25 La arquitectura de almacenamiento: los workers nunca se bloquean esperando a la base de datos

`--data-dir /var/lib/sibuna` abre el archivo SQLite `/var/lib/sibuna/sibuna.db` (modo WAL, `synchronous=NORMAL`, un tiempo de espera ante bloqueo (busy timeout)) e inicia el hilo de almacenamiento. `--database-url` (o `--database-url-file`) se conecta en su lugar a una base de datos PostgreSQL. El hilo de almacenamiento posee un runtime tokio de un solo hilo y un pequeño conjunto de conexiones; es el único lugar que ejecuta SQL. Al arrancar, cada vez que cambia la revisión de la política (una escritura en `policies`, `ip_reputation` o `policy_inspection`) y cuando caduca una entrada de reputación, el hilo reconstruye el motor a partir del archivo de políticas y de la base de datos y lo publica; las peticiones nunca esperan a SQL. Una reconstrucción lee la revisión antes y después de cargar, y descarta su candidato si entretanto un escritor confirmó una transacción.

Esquema

Tabla 25 Tablas principales

Tabla	Propósito y clave
`policy_revision`	La revisión de la política, una fila; la hacen avanzar disparadores en las tres tablas de políticas.
`policies`	Reglas dinámicas ordenadas, con clave `id`. La columna `ja4_pattern`, que admite nulos, guarda el criterio JA4 de una regla, de 37 bytes como máximo.
`policy_inspection`	El modo WAF guardado por categoría, una fila.
`ip_reputation`	Puntuaciones, recuentos de coincidencias y caducidad, con clave por dirección o prefijo.
`incident_receipts`	El recibo duradero de incidentes de cada nodo (la secuencia siguiente a su último lote confirmado), con clave por nodo.
`security_incidents`	Un registro de evento acotado con su embedding de 64 componentes (256 bytes), con clave `node_id << 40 | sequence`. La columna `ja4`, que admite nulos, guarda la huella JA4 de la petición, de exactamente 36 bytes.
`incidents_fts`	En SQLite: una tabla FTS5 sobre los tokens canónicos de las rutas y las cargas útiles. En PostgreSQL los mismos tokens son la columna `tsvector search` de `security_incidents` con un índice GIN.
`sibuna_schema`	Cada migración aplicada, con clave por componente y versión.

El esquema principal es `crates/storage/src/schema.rs`, escrito una vez por dialecto. Con la consola, el componente `console` añade 33 tablas cuyos nombres empiezan por `console_` (SID 0013; `crates/console-storage/src/schema/`). Las migraciones se numeran por componente y se compilan en el binario. La migración 2 del núcleo añade las dos columnas JA4; una base de datos creada antes de ella se migra al arrancar. Todas las migraciones pendientes se ejecutan en una sola transacción, y un nodo rechaza una base de datos cuya versión de un componente es más nueva que la que conoce. Un esquema impreso abreviado no es un script de actualización. Las tablas SQLite son `STRICT`, y las restricciones `CHECK` reflejan los tipos de Rust, así que la base de datos rechaza una fila que el daemon no podría representar. Los valores enumerados se guardan en minúsculas: una escritura directa de `'DENY'` se rechaza.

Operación

Usa el shell `sqlite3` sobre `sibuna.db` (el modo WAL le permite leer mientras el daemon se ejecuta), o `psql` sobre la base de datos del clúster:

```
-- Añade una regla dinámica; cada nodo reconstruye el motor en su siguiente ciclo.
INSERT INTO policies (id, name, priority, path_pattern, action, difficulty,
algorithm,
  header_matchers, cidr_matchers, weight, enabled, created_at, updated_at)
VALUES ('p-checkout', 'protect-checkout', 10, '/checkout/*', 'challenge', 20,
'posw',
```

```
  '{"X-Api": "v2"}', '["10.0.0.0/8"]', 0, 1, unixepoch(), unixepoch());

-- Bloquea una dirección en todo el clúster durante un día.
INSERT INTO ip_reputation (ip_or_cidr, reputation_score, banned_until,
  trigger_rule, hits, last_seen)
VALUES ('198.51.100.7', -100, unixepoch() + 86400, 'analyst', 1, unixepoch());

-- Análisis forense: búsqueda de texto completo en las cargas útiles registradas.
SELECT s.id, s.client_ip, s.violation_category, s.path, s.campaign_id
FROM incidents_fts f JOIN security_incidents s ON s.id = f.rowid
WHERE incidents_fts MATCH 'union' ORDER BY rank LIMIT 20;
```

Las sentencias están escritas para SQLite. En PostgreSQL, `enabled` es `TRUE`, la hora es `CAST(extract(epoch FROM now()) AS BIGINT)` en lugar de `unixepoch()`, y la búsqueda lee `WHERE search @@ CAST('union' AS tsquery)` sobre `security_incidents`. `path` y `offending_payload` contienen los bytes en bruto de la petición (`BLOB` o `BYTEA`). La base de datos rechaza una escritura directa que incumple una restricción, y esa escritura nunca llega a un nodo. En PostgreSQL el disparador de revisión también emite `NOTIFY sibuna_policy`, así que los demás nodos ejecutan su ciclo de inmediato; el sondeo sigue siendo la garantía.
Las puntuaciones iguales o inferiores a −50 se convierten en prefijos `deny` en el trie, las iguales o superiores a 50 se convierten en `allow`, y `banned_until` acota el bloqueo. Los accesos al honeypot insertan automáticamente un registro de −100, así que, con una base de datos PostgreSQL compartida, una trampa activada en un nodo bloquea la dirección en todos ellos.

Un recibo de transacción hace que reintentar sea seguro

El hilo de almacenamiento reúne como máximo 32 registros pendientes y construye una transacción SQL. La transacción inserta incidentes, actualiza el índice de texto completo y aplica los cambios de reputación del honeypot. Cada emisor (nodo) tiene un recibo monótonamente creciente, su fila de `incident_receipts`. Cada sentencia que modifica datos está protegida por ese recibo; la transacción lo hace avanzar en último lugar, y nunca hacia atrás.

> **Ejemplo resuelto: el acuse de recibo desaparece** Supongamos que el emisor 2 prepara las secuencias 101 a 132, que terminan en el recibo 133. La base de datos confirma todos los registros y el recibo 133, y después la conexión falla antes de que llegue el acuse de recibo. El hilo de almacenamiento conserva el lote preparado (identificadores, embeddings y campañas) y lo reintenta. El recibo ya vale 133, así que las escrituras protegidas no hacen nada: no se duplican ni los incidentes ni los recuentos de accesos al honeypot. Si la transacción original se deshizo, el recibo antiguo permanece y el reintento aplica todas las escrituras una vez.

Una confirmación fallida conserva el lote pendiente en memoria e incrementa `incident_write_failures`. El ciclo sigue intentando el sondeo de políticas. Una cola llena de 512 registros cuenta `incidents_dropped`; nunca bloquea una respuesta HTTP esperando al disco. El lote pendiente añade espacio para 32 registros. Si el proceso muere antes de confirmar, pueden perderse datos en cola: es un camino forense asíncrono y acotado, no una cola de mensajes duradera en el momento de encolar.

Ejercicio 9.1. Mueve la actualización del recibo fuera de la transacción de datos. Construye una secuencia de caídas que duplique un efecto de reputación y otra que pierda un incidente.

Agrupación en campañas

La carga útil de cada incidente se representa como un vector unitario de 64 dimensiones mediante el truco del hashing sobre trigramas de bytes (con los dígitos unificados y las mayúsculas convertidas en minúsculas) y se guarda como 256 bytes en `security_incidents.embedding`. El hilo de almacenamiento mantiene en memoria los 16.384 embeddings más recientes de todos los nodos (4 MiB), actualizados desde la base de datos en cada ciclo. Antes de insertar, busca el más cercano de ellos, incluidos los registros anteriores del mismo lote; si la distancia coseno es como máximo 0.35, el nuevo incidente se une al `campaign_id` de ese incidente; en otro caso, inicia una campaña con el nombre de su propio identificador. Los incidentes más antiguos que la ventana no se comparan. La similitud es una heurística: los ejemplos de regresión agrupan cargas útiles SQL relacionadas, pero eso no garantiza que cada par de ataques comparta campaña. No se ejecuta ningún modelo ni se entrena nada.

Funcionamiento en clúster

Un clúster es un conjunto de nodos que comparten una única base de datos PostgreSQL. Compila con la feature `postgres-tls` (`cargo build --release -p sibuna --features postgres-tls`; implica `postgres`) e inicia cada miembro con la base de datos y un id de nodo propio:

```
sibuna --database-url-file /etc/sibuna/database-url --cluster-node 1 \
       --secret-file /etc/sibuna/secret
```

Todos los miembros deben usar el mismo secreto maestro y un `--cluster-node` único de 1 a 8.388.607. El archivo de la URL contiene una URL `postgres://` y solo debe poder leerlo su propietario; el banner nunca imprime la URL. Las escrituras van al primario de PostgreSQL. El hilo de almacenamiento de cada nodo ve el cambio de la revisión confirmada (mediante `NOTIFY` o por sondeo) y reconstruye su motor.

Los nodos acceden a la base de datos compartida a través de la red, así que ciframos y verificamos esa conexión. Damos al servidor un certificado firmado por nuestra propia autoridad de certificación, y nombramos esa autoridad en la URL: `postgres://sibuna:...@db.internal:5432/sibuna?sslmode=verify-full&sslrootcert=/etc/sibuna/db-ca.pem`. En el servidor aceptamos TCP solo mediante líneas `hostssl` en `pg_hba.conf`.

Al arrancar, el daemon imprime una línea sobre la conexión a la base de datos. Con `verify-full` indica que el certificado y el nombre de host están verificados. Con cualquier modo más débil imprime una advertencia y una pista. Sin `sslmode` el modo es `prefer`: la conexión solo se cifra si el servidor ofrece TLS, y el servidor no se verifica. Un daemon compilado sin `postgres-tls` rechaza `require`, `verify-ca` y `verify-full` y nombra la feature que falta; nunca recurre a una conexión sin cifrar. Un certificado rechazado detiene el arranque de inmediato; la pista nos pide comprobar `sslrootcert` y que el host de la URL sea un nombre del certificado del

servidor. `sslrootcert` añade nuestra autoridad al almacén de confianza del sistema; no elimina las autoridades del sistema (SID 0011, SID 0012).
Sibuna no replica una base de datos propia y no elige líder. La replicación, la conmutación por error y las copias de seguridad corresponden a PostgreSQL (replicación en streaming, Patroni o un servicio gestionado). La vista de nodos de la consola no tiene líder: muestra el rol `single` para un nodo SQLite y `unknown` para un nodo PostgreSQL, y `quorum` significa solo que el último viaje de ida y vuelta del nodo a la base de datos tuvo éxito. Si la base de datos es inalcanzable, el último motor publicado sigue sirviendo, los incidentes se encolan en el anillo acotado y se descartan con un contador cuando este se llena, y el hilo de almacenamiento reintenta con espera exponencial; un nodo conserva su última política hasta que se vuelve a conectar. No se lee un directorio de datos de Zaxonlite (SID 0005); las políticas se trasladan con `sibuna console policies export` y `sibuna console policies import`, y el historial forense no se migra.

Enrutamiento de desafíos en el clúster

Las claves de desafío están vinculadas a `--cluster-node`; mantén la emisión y la verificación de un desafío en el mismo miembro. Los tokens usan la semilla compartida y funcionan en los demás miembros. Las cuotas de tasa locales no se convierten en cuotas globales, y los reinicios del proceso vacían los conjuntos de desafíos gastados. Con tres nodos sobre un PostgreSQL local, una fila de política escrita directamente llega a los tres en un ciclo, un bloqueo del honeypot en un nodo llega a los demás y los identificadores de incidentes nunca colisionan (`crates/storage/tests/multi_node.rs`, criterio 2 de SID 0011; se ejecuta cuando se define `SIBUNA_TEST_DATABASE_URL`).

Empaquetado

> **Objetivos** Instalamos y verificamos un paquete de una versión publicada, ejecutamos Sibuna como servicio, en un contenedor o en Kubernetes, y compilamos el binario nosotros mismos.

Cada versión publicada en GitHub incluye el mismo ejecutable para cada plataforma, compilado desde la etiqueta de la versión con las features por defecto: almacenamiento SQLite y la consola. Los clústeres PostgreSQL necesitan una compilación propia (más abajo).

Plataforma	Archivos de la versión
Linux x86-64, ARM64	`sibuna-linux-amd64.tar.gz` y `sibuna-linux-arm64.tar.gz` contienen un ejecutable musl estático que funciona en cualquier distribución. `sibuna_<version>-1_amd64.deb`, `sibuna_<version>-1_arm64.deb`, `sibuna-<version>-1.x86_64.rpm` y `sibuna-<version>-1.aarch64.rpm` lo instalan con un servicio systemd; `sibuna-bin-<version>-1-x86_64.pkg.tar.zst` hace lo mismo en Arch Linux.

Plataforma	Archivos de la versión
macOS arm64, x86-64	`sibuna-macos-arm64.tar.gz`, `sibuna-macos-amd64.tar.gz`, para macOS 12 o posterior.
Windows x86-64	`sibuna-windows-amd64.zip` contiene `sibuna.exe`, para Windows 10 o Server 2016 o posterior.
FreeBSD 15.1, OpenBSD 7.9 (amd64)	`sibuna-<version>-freebsd-15.1-amd64.pkg` y `sibuna-<version>.tgz`.
Homebrew, Kubernetes, contenedores	`sibuna.rb`, `helm-sibuna-<version>.tgz` y la imagen `ghcr.io/insanai/sibuna:<version>` para amd64 y arm64, cuyo digest está en `IMAGE-DIGEST.txt`.
Fuentes	`sibuna-<version>-source.tar.gz`.

Cada archivo comprimido y cada paquete incluyen `LICENSE`, `NOTICE`, `LICENSES/` (con `THIRD-PARTY-RUST.txt`, los crates del ejecutable), `README.md`, `SECURITY.md` y `sibuna.build.json`. El manifiesto registra la versión, el commit, el target de Rust, la versión del compilador, las features y el SHA-256 del ejecutable y de sus dos módulos WebAssembly. El flujo de trabajo de publicación arranca cada ejecutable en su propia plataforma antes de publicar nada, e instala, reinstala y elimina los paquetes en Debian, Fedora, Arch Linux, FreeBSD y OpenBSD.
Los archivos no están firmados. Comprobamos cada descarga contra `SHA256SUMS` de la misma versión antes de instalarla:

```
sha256sum --ignore-missing -c SHA256SUMS          # Linux
shasum -a 256 --ignore-missing -c SHA256SUMS      # macOS, FreeBSD
sha256 -C SHA256SUMS sibuna-1.0.0-beta.3.tgz      # OpenBSD
```

En macOS, una descarga hecha con el navegador lleva un atributo de cuarentena que bloquea un ejecutable sin firmar; `xattr -d com.apple.quarantine sibuna` lo elimina después de la comprobación.

El servicio systemd

Los paquetes Debian, RPM y Arch instalan `/usr/bin/sibuna`, la unidad `sibuna.service`, la cuenta `sibuna` (mediante `systemd-sysusers`), el script auxiliar `/usr/lib/sibuna/seed.sh` y el archivo de ajustes `/etc/sibuna/service.env`. La instalación no arranca nada. Fijamos el origen, un literal IP, en el archivo de ajustes y arrancamos el servicio:

```
sudoedit /etc/sibuna/service.env      # SIBUNA_UPSTREAM_HOST, SIBUNA_UPSTREAM_PORT
sudo systemctl enable --now sibuna
curl -s http://127.0.0.1:8080/__sibuna/health
```

El servicio escucha en `127.0.0.1:8080` mientras `SIBUNA_HOST` y `SIBUNA_PORT` no indiquen otra cosa. `SIBUNA_OPTIONS` pasa más opciones, separadas por espacios, por ejemplo `--mode forward_auth` o `--console 127.0.0.1:19446`. Antes de cada arranque, `seed.sh` crea la semilla de admisión `/var/lib/sibuna/admission.seed` (64 dígitos hexadecimales, modo 0600, propiedad

de `sibuna`) si no existe; nunca sustituye una existente, porque una semilla nueva terminaría todas las sesiones. El daemon guarda su base de datos SQLite en `/var/lib/sibuna`, un directorio con modo 0700. La unidad se ejecuta sin capabilities, con un sistema de solo lectura (`ProtectSystem=strict`), `/tmp` y dispositivos privados, y sin nuevos privilegios.
Una actualización conserva el archivo de ajustes, la semilla y el estado, y no reinicia el servicio; lo reiniciamos después de hacer una copia de seguridad. La desinstalación detiene y desactiva el servicio, pero conserva la cuenta, la semilla, los ajustes y el estado, que borramos nosotros mismos cuando retiramos el host.
Los paquetes de FreeBSD y OpenBSD solo instalan `/usr/local/bin/sibuna` y sus documentos (`pkg add`, o `pkg_add -D unsigned` para un paquete que no procede de los servidores espejo de OpenBSD). No crean servicio, cuenta ni estado: ejecutamos el daemon como un usuario sin privilegios, con una semilla y un directorio de datos privados.

Homebrew

`sibuna.rb` instala el archivo de la versión adecuado para la máquina: macOS o Linux, arm64 o x86-64. No compila desde las fuentes, porque el Rust de Homebrew no tiene biblioteca estándar para `wasm32-unknown-unknown`, que necesita la compilación de los dos módulos WebAssembly integrados; los archivos contienen los ejecutables que el flujo de trabajo de publicación ya comprobó. Homebrew solo instala fórmulas desde taps, así que ponemos la fórmula en un tap local:

```
brew tap-new "$USER/local"
cp sibuna.rb "$(brew --repository)/Library/Taps/$USER/homebrew-local/Formula/"
brew install "$USER/local/sibuna"
```

La fórmula no arranca ningún servicio. Sus advertencias (caveats) muestran cómo crear una semilla; su prueba arranca el daemon dos veces sobre el mismo directorio de datos.

Contenedores y Kubernetes

La imagen `ghcr.io/insanai/sibuna:<version>` contiene el ejecutable estático sobre una base distroless: certificados de CA, usuario 65532, sin shell y sin gestor de paquetes. El punto de entrada es `sibuna`, así que los argumentos son opciones del daemon. El flujo de trabajo de publicación la ejecuta en solo lectura, sin capabilities, con una semilla montada en solo lectura y un volumen para el estado, y la reinicia una vez:

```
docker run -d --read-only --cap-drop=ALL --security-opt=no-new-privileges \
  -v "$PWD/admission.seed:/run/admission.seed:ro" -v sibuna-state:/var/lib/sibuna
\
  -p 8080:8080 ghcr.io/insanai/sibuna:1.0.0-beta.3 \
  --host 0.0.0.0 --secret-file /run/admission.seed --data-dir /var/lib/sibuna/data
\
  --upstream-host 10.0.0.5 --upstream-port 3000
```

El chart de Helm `helm-sibuna-<version>.tgz` ejecuta un Pod de esa imagen. Requiere un Secret que creamos nosotros (`secret.existingSecret`, clave `admission.seed`), de modo que ninguna semilla aparece nunca en los valores ni en las plantillas. El Pod se ejecuta como UID 65532 con un sistema de archivos raíz de solo lectura bajo el estándar de seguridad de pods

Restricted. Una sola réplica con actualizaciones `Recreate` mantiene un único escritor en el volumen ReadWriteOnce, donde Sibuna guarda su estado SQLite en `data/`; la reclamación de volumen se conserva al desinstalar. Las sondas llaman a `/__sibuna/health`. El Service es ClusterIP, el chart no crea ingress ni consola, y su NetworkPolicy deniega todo el tráfico hasta que `networkPolicy.ingressFrom` y `networkPolicy.egressTo` permitan los clientes y el origen. `upstream.host` debe ser un literal IP, y `extraArgs` pasa más opciones al daemon. El flujo de trabajo instala, actualiza y desinstala el chart en un clúster kind que aplica el estándar Restricted.

```
kubectl -n sibuna create secret generic sibuna-admission --from-
file=admission.seed
helm install gateway ./helm-sibuna-1.0.0-beta.3.tgz --namespace sibuna \
  --set secret.existingSecret=sibuna-admission \
  --set upstream.host=10.0.0.5 --set upstream.port=3000
```

Compilar desde las fuentes

Compilamos y probamos con Rust 1.100; Rust 1.94 sigue siendo la versión mínima compatible. Instalamos la cadena de herramientas de la versión con `rustup toolchain install 1.100.0 --profile minimal --target wasm32-unknown-unknown` y la seleccionamos con `cargo +1.100.0` en los comandos siguientes.

- `cargo build --release -p sibuna` produce `target/release/sibuna` con las features por defecto: almacenamiento SQLite y la consola. sqlx incluye SQLite, y `ring` (rustls) necesita un compilador de C en tiempo de compilación; el binario enlaza dinámicamente el runtime de C del sistema. El script de compilación también compila el solucionador WebAssembly y la interfaz de la consola, lo que necesita `rustup target add wasm32-unknown-unknown`; `SIBUNA_POW_WASM=<path>` y `SIBUNA_CONSOLE_WASM=<path>` incorporan en su lugar módulos precompilados.
- `cargo build --release -p sibuna --no-default-features` produce el cortafuegos en memoria sin ningún crate externo; `--data-dir` y `--database-url` se rechazan entonces al arrancar con `StorageDisabled`, y `sibuna console` con `CONSOLEBUILD`.
- `cargo build --release -p sibuna --features postgres-tls` añade los clústeres PostgreSQL con TLS verificado hacia la base de datos. No necesita OpenSSL.
- Los ejecutables Linux estáticos de las versiones publicadas proceden de `cargo build --release --locked -p sibuna --target x86_64-unknown-linux-musl` (o `aarch64-unknown-linux-musl`) con `musl-gcc` como compilador de C (`CC_x86_64_unknown_linux_musl=musl-gcc`).

El perfil de release usa LTO completo (fat) y una unidad de generación de código. El flujo de trabajo compila los dos módulos WebAssembly una sola vez y los pasa a todas las plataformas, de modo que todos los ejecutables de una versión incorporan módulos idénticos byte a byte. `cargo xtask licenses` regenera `LICENSES/THIRD-PARTY-RUST.txt` cuando cambia una dependencia.

En Windows, el daemon se detiene con Ctrl+C, Ctrl+Break y los eventos de cierre de la consola; no se ejecuta como servicio de Windows. Los archivos de Windows no tienen bits de modo, así que Sibuna no puede rechazar un archivo de clave, de credenciales o de URL de

base de datos que otros puedan leer: restringimos esos archivos con ACL. Los indicadores de memoria y CPU de la consola se leen de `/proc` y no están disponibles en otros sistemas.

Ejercicio 9.2. Escribe el archivo de políticas y las filas de `ip_reputation` necesarias para que una red de preproducción (`10.20.0.0/16`) evite los desafíos, `/admin/*` exija 20 bits de trabajo secuencial y un scraper de un socio identificado por `X-Partner-Key` se admita a 30 peticiones cada 10 segundos.
Pista: las opciones del daemon fijan el límite por cliente que toda petición encuentra primero; una regla puede llevar una cuota propia en `limits`.

Explica el invariante. Explica a un operador por qué añadir una fila a `policies` surte efecto sin reiniciar y sin que ninguna petición tenga que esperar, en términos del hilo de almacenamiento y del espacio del motor.

PARTE X

Referencia de consulta y diagnóstico

Endpoints, métricas, catálogo de errores, esquema de políticas y tablas de almacenamiento en un solo lugar.

Endpoints

Objetivos Conoce todas las rutas bajo `/__sibuna/` y lo que devuelve cada una.

Tabla 27 Rutas reservadas

Endpoint	Método	Función
`/__sibuna/challenge`	GET	La página intermedia
`/__sibuna/challenge.json?need=&path=`	GET	Emite un desafío sin estado: lo decide el ticket de requisito de la respuesta que exigió el desafío; si no lo hay, se evalúa en forma normal la URL que indica el cliente (`414` por encima de 8 KiB, `400` para una ruta sin forma normal). Devuelve `{id, algorithm, difficulty, challenges, expires_at}`
`/__sibuna/verify`	POST	Acepta `{"challenge_id", "nonce"}` o `{"challenge_id", "proof"}`; responde `200` con `Set-Cookie`, `400` con un diagnóstico o `413` si el cuerpo es mayor que el búfer de la conexión
`/__sibuna/wasm/sibuna-pow.wasm`	GET	El módulo del solucionador, de 14.744 bytes (`wasm_bytes` en `benchmarks/results/primitives-latest.json`), que se puede almacenar en caché durante una hora
`/__sibuna/worker.js`	GET	El Web Worker con los probadores WASM y JavaScript, que se puede almacenar en caché durante una hora
`/__sibuna/honeypot`	GET	Bloquea a quien llama durante `--ban-seconds` y registra un incidente; `403`. Solo bloquea una petición de nivel superior (sin `Sec-Fetch-Dest` o con

Endpoint	Método	Función
		`document`, y sin `Sec-Fetch-Site` entre sitios); cualquier otra recibe `404` y se contabiliza
`/__sibuna/health`	GET	Estado de disponibilidad en JSON con el nombre del motor, la versión, el modo de proxy y el algoritmo de prueba
`/__sibuna/metrics`	GET	Formato de texto de Prometheus (versión 0.0.4)

Cualquier otra ruta bajo `/__sibuna/` recibe `404`. La emisión y la verificación consumen el presupuesto independiente `--challenge-rate-limit`, y una dirección bloqueada se rechaza con `403` en todas las rutas. Las respuestas de desafío llevan `X-Sibuna-Status: CHALLENGE`; las peticiones admitidas llevan `X-Sibuna-Status: PASS` y `X-Sibuna-Rule` hacia el origen (y `X-Sibuna-Rule-Hash`, en hexadecimal, en las respuestas de forward-auth). Un cliente que no acepta HTML recibe `401` con `{"error":"challenge_required","challenge":"/__sibuna/challenge.json?need=<ticket>"}`.
El puerto de escucha de gestión, independiente y opcional, sirve `/console/`. Inicializa su administrador localmente con `sibuna init-admin` y sustituye la contraseña temporal en el primer inicio de sesión. `/console/api/stats` y `/console/ws` requieren una sesión autenticada (incluida una sesión de quiosco). El WebSocket multiplexa estadísticas, eventos, nodos, políticas, desafíos y auditoría con instantáneas acotadas, deltas y recuperación de huecos. Los pares de gestión se conectan a `/console/peer`, y todas las rutas de la API están bajo `/console/api/`. En el modo forward-auth no se observan los contadores de errores del origen. La parte IX documenta los flujos de trabajo de la consola, su despliegue con HTTPS y los límites actuales de SID 0007.

Métricas

Contadores expuestos como `sibuna_<name>_total` en formato de texto de Prometheus, en este orden:

Tabla 28 Contadores

Contador	Se incrementa cuando
`incidents_persisted`, `incident_batches`	Registros y transacciones cuya confirmación ha verificado el hilo de almacenamiento
`incidents_dropped`, `incident_write_failures`	Inserciones en la cola rechazadas porque el anillo estaba lleno, e intentos de confirmación fallidos
`requests`	Una petición analizada llegó al despacho (incluidas las rutas reservadas)
`allowed`, `denied`, `challenged`	Se tomó una decisión de política (admitida, rechazada, desafío emitido o reemitido)
`challenges_issued`	`/__sibuna/challenge.json` emitió un registro de desafío

Contador	Se incrementa cuando
`solutions_accepted`, `solutions_rejected`	`/__sibuna/verify` aceptó o rechazó una prueba
`rate_limited`	GCRA rechazó una petición (`429`): el límite de peticiones, el presupuesto de desafíos o la cuota de una regla
`banned`	Se rechazó una dirección bloqueada, o el honeypot bloqueó una
`bans_issued`	Se registró un bloqueo (del honeypot o de la cuota de una regla)
`proxied`, `upstream_errors`	Se retransmitió una petición al origen, o el origen falló (`502`)
`parse_errors`	Se rechazó un encabezado de petición mal formado, un `Host` repetido, un destino sin forma normal (`400`) o una codificación de transferencia no admitida (`501`)
`overloaded`	Una conexión por encima de `--max-connections` o `--max-connections-per-ip` recibió `503`
`bans_refused`	No se registró un bloqueo porque todas las posiciones que puede ocupar su clave tenían un bloqueo vigente
`forwarded_invalid`	No se pudo analizar una dirección de cliente reenviada de confianza; la petición se evaluó según el extremo remoto del socket
`honeypot_refused`	Un acceso al honeypot no bloqueó a nadie porque lo solicitó una página, no una persona
`rate_evictions`	Una llegada al limitador desalojó la celda de otro cliente activo
`ja4_present`	Con `--ja4-header` y el reenvío de confianza, la petición llevaba exactamente una cabecera de ese nombre, con una huella JA4
`ja4_invalid`	Con `--ja4-header` y el reenvío de confianza, la cabecera estaba mal formada o repetida; la petición se evaluó sin huella

Un reintento puede incrementar `incident_write_failures` sin perder registros, porque el lote pendiente se conserva. Vigila los incrementos durante un intervalo; los totales por sí solos no indican la profundidad de la cola.

Códigos de estado

Tabla 29 Códigos de estado del plano de datos

Código	Cuándo
200	Admitida (forward-auth), página intermedia (navegación HTML que necesita un desafío), rutas internas
302	No lo usa el protocolo actual; la verificación responde `200` y la página se recarga
400	Petición mal formada, un campo `Host` repetido, un destino sin forma normal, metadatos de reenvío no válidos, un cuerpo mal formado o una solución rechazada con un diagnóstico al estilo de Elm
401	Se requiere un desafío para un cliente que no acepta HTML, o en el modo forward-auth
403	Denegación por política o por el WAF, dirección bloqueada, honeypot
404	Una ruta desconocida bajo `/__sibuna/`, o un acceso al honeypot que no es una petición de nivel superior
413	Cuerpo de la solución mayor que el búfer de conexión de 64 KB
414	La URL indicada para el desafío supera 8 KiB
417	Expectativa de la petición no admitida; `100-continue` se gestiona localmente
429	Límite GCRA superado; `Retry-After` en segundos
431	Encabezado de la petición de más de 16 KB
501	`Transfer-Encoding` con una codificación distinta de chunked
502	Origen inalcanzable, o el encabezado de su respuesta estaba mal formado o superaba 16 KB
503	Se alcanzó el límite de conexiones (`--max-connections` o `--max-connections-per-ip`), o el nodo se está drenando; el socket se cierra tras la respuesta

Catálogo de errores

`INVALID COMMAND LINE` detiene el arranque ante cualquier opción que el daemon no reconoce, cualquier valor fuera de su rango documentado y una selección de modo no válida, ausente o duplicada. El bloque nombra la opción, el valor dado, el rango esperado y el error (`UnknownOption`, `MissingValue`, `InvalidValue`, `InvalidMode`, `MissingMode`, `DuplicateMode`, `RemovedOption` para una opción de clúster de Zaxonlite, `ConflictingOptions` para dos opciones de almacenamiento que se excluyen entre sí, `MissingOption` para una URL de base de datos sin `--cluster-node`), y termina con una pista. Una URL de base de datos se muestra como `[redacted]`. Indica `--mode reverse_proxy` o `--mode forward_auth` una sola vez; `-m` es la opción corta equivalente. Un archivo de políticas rechazado imprime un bloque titulado con su error (por ejemplo `INVALID POLICY ACTION`) con el archivo, el nombre del error, la regla, el campo y el valor. Un patrón `ja4` con el que ninguna huella podría coincidir imprime `INVALID POLICY JA4`

(`InvalidJa4`); un documento de gestión de la consola con un patrón así se rechaza con el mismo nombre de error. Las opciones de la consola se comprueban por separado: `CONSOLE002` señala una opción de consola mal formada o repetida, y `CONSOLE001` una configuración de consola que incumple una regla (`StorageRequired` sin almacenamiento).

Implementación: `ErrorKind::explain` Asocia cada error del dominio a una línea separadora, una explicación y un `Hint:`. En `crates/core/src/errors.rs` (crate `sibuna-core`).

Tabla 30 Errores de desafío y de sesión

Error	Causa	Pista
`MalformedChallenge`	El identificador no es un registro de desafío bien formado	Obtén un desafío nuevo y envíalo sin cambios
`InvalidChallengeTag`	La etiqueta no se verifica: el desafío no lo emitió este clúster o se modificó	Los desafíos no se pueden falsificar; solicita uno nuevo
`ChallengeExpired`	Supera el TTL del desafío, o se emitió en el futuro	Solicita un desafío nuevo
`FingerprintMismatch`	Enviado desde otra dirección u otro User-Agent	Envíalo desde el cliente que lo obtuvo
`DifficultyNotMet`	Al nonce de Hashcash le faltan los bits a cero requeridos	Sigue buscando nonces
`InvalidProof`	La prueba de trabajo secuencial no abre las etiquetas del compromiso	Ejecuta el probador hasta el final con la profundidad y las aperturas emitidas
`WrongSolutionType`	Un nonce para un desafío PoSW o una prueba para Hashcash	Haz coincidir el campo de la solución con el algoritmo
`DoubleSpendAttempt`	El desafío ya se había gastado	Los desafíos son de un solo uso
`StoreFull`	El conjunto de desafíos gastados está lleno	Reduce el TTL de los desafíos o aumenta `--spent-tags`
`InvalidTokenSignature`	Falla la etiqueta o la firma de la cookie	Vuelve a autenticarte mediante la página intermedia
`TokenExpired`	Cookie caducada	Vuelve a autenticarte
`TokenBoundAddressMismatch`	Cookie presentada desde otra identidad de cliente	Las cookies no se pueden compartir

La API de la consola responde a un fallo con `{"error":"<CODE>","hint":"<sentence>"}`; la pista es el texto que muestra la interfaz (`crates/console/src/http/reply.rs`).

Tabla 31 Códigos de fallo de la consola que interesan a los operadores

Código	Estado	Significado
`CONSOLE002`	400	La petición está mal formada, es demasiado grande o no se permite en esa forma
`CONSOLE003`	503 / 429	La cola de tareas de almacenamiento está llena (32 tareas), o el verificador de contraseñas está ocupado
`CONSOLEQUORUM`	503	La base de datos no respondió en los 10 segundos que espera la petición; el resultado se desconoce, y el plano de datos continúa con la última política aplicada
`CONSOLE004`	503	El servicio no puede responder: no está disponible en esta compilación, se está deteniendo o tuvo un fallo interno
`CONSOLENODE`	400 / 409 / 503	Se rechazó un comando dirigido a un nodo: nombra otro nodo (400), una revisión de arranque o de control obsoleta (409), o llegó mientras se ejecutaba otro comando o no se pudo registrar una finalización (503); inspecciona el recibo de la operación antes de reintentar
`CONSOLEMUTATION`	429	Más de sesenta modificaciones de gestión en un minuto desde una misma sesión

Esquema de políticas

```
{
  "default_action": "ALLOW" | "DENY" | "CHALLENGE",
  "waf": true | false,
  "inspection": { "path_traversal" | "sqli" | "xss" | "rce": "disabled" | "audit"
| "enforce" },
  "thresholds": { "challenge_at": int, "deny_at": int, "bits_step": int },
  "ip_rules": { "<cidr>": "ALLOW" | "DENY" | "CHALLENGE", ... },
  "rules": [                                                    // hasta 128
    {
      "name": "<kebab-case>",                                   // de 1 a 128
bytes imprimibles
      "path" | "path_regex": "<pattern>",                       // hasta 512 bytes
      "user_agent" | "user_agent_regex": "<pattern>",
      "headers" | "headers_regex": { "<Header>": "<pattern>" }, // hasta 4
      "remote_addresses" | "cidrs": ["<cidr>", ...],            // hasta 8, IPv4 o
IPv6
      "action": "ALLOW" | "DENY" | "CHALLENGE" | "WEIGH",
      "weight": int,                                            // solo WEIGH,
-1000..1000
      "challenge": { "difficulty": <work bits>, "algorithm": "hashcash" |
"posw" },
      "limits": { "rate": int, "window_seconds": int, "ban_seconds": int },
```

```
      "ja4": "<fingerprint>" | "<prefix>*"                    // 36 bytes, o un
prefijo y *
    }
  ]
}
```

`challenge_at` y `deny_at` están entre -1000 y 1000, con `deny_at` mayor o igual que `challenge_at`, y `bits_step` entre 1 y 1000. Un bloque `challenge` solo pertenece a las reglas `CHALLENGE`, con 1 a 64 bits de trabajo. `limits` da a una regla su propia cuota: `rate` de 1 a 1.000.000 y `window_seconds` de 1 a 86.400 son obligatorios, `ban_seconds` (como máximo 86.400) es opcional, y las reglas `WEIGH` no pueden llevar límites. `ja4` es una huella JA4 completa, a la que la huella debe ser igual, o un prefijo seguido de *, por el que debe empezar la huella; * por sí solo coincide con cualquier huella, y una petición sin huella no coincide con ninguna regla `ja4`. Un patrón con el que ninguna huella podría coincidir se rechaza con `InvalidJa4` (parte IX, "Huellas JA4 desde el proxy de entrada"). El trie de reputación de `ip_rules` contiene como máximo 8.192 nodos. Cada regla puede usar una sola variante de cada criterio.
Gramática de patrones: `.*` o * coinciden con cualquier cosa; `^...$` ancla una ruta exacta; un *, `/*` o `.*` final indica un prefijo; un patrón que empieza por `/` es una ruta exacta; cualquier otra cosa es una subcadena que no distingue mayúsculas de minúsculas. Los patrones de ruta se comparan con la ruta en forma normal (una barra entre segmentos, sin segmentos de punto, caracteres no reservados sin escapar y el resto de escapes con dígitos hexadecimales en mayúsculas), así que un patrón que contiene `;`, `%2F`, `%5C` o `\` no coincide con nada, y la comparación de rutas distingue mayúsculas de minúsculas (SID 0003).

Tablas de almacenamiento

`policy_revision`, `policies` (con el criterio `ja4_pattern`, que admite nulos, de 37 bytes como máximo), `policy_inspection`, `ip_reputation`, `incident_receipts`, `security_incidents` (con el embedding de 256 bytes y la huella `ja4` de 36 bytes, que admite nulos), el índice de texto completo (`incidents_fts`, FTS5 sobre tokens canónicos, en SQLite; la columna `tsvector` `search` con un índice GIN en PostgreSQL) y `sibuna_schema`; la consola añade 33 tablas `console_*`. Los nombres de acción, algoritmo y modo se guardan en minúsculas. Los ids de incidente son `node_id << 40 | sequence`, únicos en todo un clúster sin coordinación; los ids de nodo van de 1 a 8.388.607.

Objetivos de compilación

Tabla 32 Comandos de compilación y del repositorio

Comando	Resultado
`cargo build --release -p sibuna`	Daemon con almacenamiento SQLite y la consola (las features por defecto)
`cargo build --release -p sibuna --no-default-features`	Daemon en memoria sin almacenamiento ni consola y sin ningún crate externo

Comando	Resultado
`cargo build --release -p sibuna --features postgres-tls`	Daemon que además se conecta a una base de datos PostgreSQL compartida, con TLS verificado (la feature `postgres` por sí sola se conecta sin TLS)
`cargo test`	Pruebas unitarias, pruebas de integración contra un daemon en ejecución, pruebas de almacenamiento y de consola (las de PostgreSQL cuando se define `SIBUNA_TEST_DATABASE_URL`)
`cargo clippy --all-targets`, `cargo fmt --check`	Lints y formato (`max_width = 99`)
`cargo xtask check`	Los límites de 70 líneas por función, 1.408 líneas por archivo y 99/108 columnas por línea; sin tabuladores
`cargo xtask browser`	Comprobaciones en Chromium sin interfaz gráfica (`sibuna-browser`)
`cargo xtask bench <primitives|product|latency>`	Pruebas de rendimiento en modo release; resultados en `benchmarks/results/<command>-latest.json`
`cargo build -p sibuna-pow-wasm --target wasm32-unknown-unknown --profile wasm`	Solo el módulo del navegador (`-p sibuna-console-ui` para la interfaz de la consola)
`cargo run -p sibuna-geoip --bin geoip-snapshot -- ...`	Valida los datos de países y escribe una instantánea GeoIP que se puede integrar
`guidedog build` / `guidedog dsc`	Este libro / los registros SID

El script de compilación del daemon (`cmd/sibuna/build.rs`) compila ambos módulos WebAssembly con el perfil `wasm` en el directorio de destino del workspace y hace fallar la compilación cuando la interfaz de la consola supera 768 KiB. `SIBUNA_POW_WASM` y `SIBUNA_CONSOLE_WASM` indican en su lugar módulos precompilados, para compilaciones sin el objetivo `wasm32-unknown-unknown`. `SIBUNA_GEOIP_DATA` indica una instantánea GeoIP que integrar; se decodifica en tiempo de compilación, y una compilación sin la consola la rechaza (`GEOIP001`).

Ejercicio 10.1. Un cliente recibe `400` con el título `CLIENT FINGERPRINT MISMATCH` tras pasar de Wi-Fi a datos móviles en mitad de la resolución. Explica la causa a partir de los formatos del token y del desafío, y propón el cambio más pequeño en la página intermedia que le permita recuperarse sin contratiempos.

Explica el invariante. Sin mirar, enumera los endpoints que un proxy inverso delante de Sibuna debe enrutar al daemon en lugar de al origen, y di por qué se necesita cada uno.

PARTE XI

Tarjeta de referencia rápida

Para tener junto al terminal: compilar, ejecutar, configurar, enrutar, observar y diagnosticar. La parte X contiene las tablas completas; esta tarjeta contiene lo que teclea un operador.

Compilar

```
cargo build --release -p sibuna                          # daemon + SQLite + consola
cargo build --release -p sibuna --no-default-features    # en memoria, sin crates externos
cargo build --release -p sibuna --features postgres-tls  # clúster con PostgreSQL común
rustup target add wasm32-unknown-unknown   # o define SIBUNA_POW_WASM / SIBUNA_CONSOLE_WASM
cargo test            # unitarias, daemon en ejecución, almacenamiento y consola
cargo clippy --all-targets && cargo fmt --check
cargo xtask check     # límites de 70 líneas / 1.408 líneas / 99 columnas
cargo xtask browser   # comprobaciones en Chromium sin interfaz gráfica
cargo xtask bench product   # productos completos bajo wrk -> benchmarks/results/
guidedog build        # este libro (guidedog dsc: los registros SID)
```

Ejecutar

```
# Shield (por defecto): PoW + WAF, proxy hacia :3000
sibuna -p 8080 -u 3000 -s /etc/sibuna/secret
# Gate: solo PoW
sibuna --gate -p 8080 -u 3000 -s /etc/sibuna/secret
# Forward-auth detrás de Nginx/Caddy (confía en X-Forwarded-For)
sibuna -m forward_auth --host 127.0.0.1 -p 8080 -s /etc/sibuna/secret
# Edge: políticas, reputación e incidentes persistentes en <dir>/sibuna.db
sibuna -D /var/lib/sibuna -s /etc/sibuna/secret
# JA4 del proxy de entrada fiable (Cloudflare; módulo nginx: x-ja4; ATS: ja4)
sibuna --ja4-header cf-ja4 --trust-forwarded -p 8080 -u 3000 -s /etc/sibuna/secret
# Miembro de un clúster: PostgreSQL compartido, id de nodo único
sibuna --database-url-file /etc/sibuna/database-url --cluster-node 1 -s /etc/sibuna/secret
head -c 32 /dev/urandom | xxd -p -c 64 > /etc/sibuna/secret
```

Consola (versión preliminar opcional)

```
# Inicializa con el daemon detenido y después arranca la consola
```

```
sibuna init-admin admin --data-dir ./data
sibuna --data-dir ./data --console 127.0.0.1:19446
```

Abre `http://127.0.0.1:19446/console/` y sustituye la contraseña temporal. Añade `--console-location 1.3521,103.8198` para situar el nodo en el globo. El acceso remoto requiere un proxy HTTPS, un origen explícito y los CIDR de los proxies de confianza; sigue la parte IX. SID 0007 está en estado Committed. La consola supera sus flujos de trabajo de navegador en Chromium sin interfaz gráfica (SID 0010, criterio 5); su criterio formal de aislamiento del rendimiento sigue abierto (parte VIII).

Las opciones más importantes

Opción	Por defecto	Significado
`-d, --difficulty`	16	bits de trabajo (profundidad PoSW = bits - 3)
`-a, --algorithm`	posw	`posw` o `hashcash`
`--posw-challenges`	16	aperturas por prueba
`--token-ttl`	86400	segundos de sesión
`--challenge-ttl`	300	segundos de desafío
`--token-scheme`	mac	`mac` o `ed25519`
`--rate-limit / --rate-window`	100 / 10	ráfaga GCRA por ventana
`--ban-seconds`	3600	bloqueo del honeypot
`-w, --workers`	núm. de CPU	hilos de aceptación
`--max-connections`	1024	después `503`
`--max-connections-per-ip`	64	por dirección remota (IPv6: por `--ipv6-prefix`), después `503`
`--trusted-hops`	1	cliente = n-ésima entrada de `X-Forwarded-For` empezando por la derecha
`--ja4-header`	desactivado	cabecera con la huella JA4 del proxy de entrada; solo se lee si es de confianza
`--idle-timeout`	15	segundos del recolector
`-P, --policy-file`	ninguno	política JSON
`--trust-forwarded`	desactivado	activado en forward-auth

Opción	Por defecto	Significado
`--secure-cookie`	desactivado	añade `Secure`

Endpoints bajo `/__sibuna/`

Ruta	Respuesta
`challenge`	HTML de la página intermedia
`challenge.json?need=&path=`	`{id, algorithm, difficulty, challenges, expires_at}`
`verify` (POST)	`{challenge_id, nonce | proof}` -> `200` + `Set-Cookie`; si no, `400`
`wasm/sibuna-pow.wasm`	solucionador de 15.181 bytes
`worker.js`	worker con probadores WASM y JS
`honeypot`	bloquea a quien llama, `403`; `404` para un subrecurso o una petición entre sitios
`health`	`{"status":"ok",...}`
`metrics`	texto de Prometheus

Enruta todos ellos al daemon, nunca al origen.

Códigos de estado

Código	Cuándo
200	admitida (forward-auth), página intermedia, rutas internas
400	petición o destino mal formados, solución rechazada (cuerpo con diagnóstico)
401	se requiere un desafío: cliente sin HTML o forward-auth
403	denegación por política o por el WAF, bloqueo, honeypot
404	ruta `/__sibuna/` desconocida, acceso al honeypot rechazado
413	cuerpo de la solución mayor que el búfer de 64 KB
414	URL indicada para el desafío de más de 8 KiB
417	expectativa de la petición no admitida
429	límite GCRA, segundos en `Retry-After`
431	encabezado de más de 16 KB

Código	Cuándo
501	codificación de transferencia distinta de chunked
502	origen inalcanzable o con respuesta mal formada
503	límites de conexiones alcanzados, o el nodo se está drenando

Cabeceras

- *Hacia el origen (proxy)*: `X-Forwarded-For` (el extremo remoto del socket, tras la cadena recibida si es de confianza), `X-Forwarded-Proto`, `X-Real-IP` (el cliente evaluado), `X-Sibuna-Status: PASS`, `X-Sibuna-Rule: <rule>`; se eliminan los campos hop-by-hop y los campos de reenvío alternativos, también sus variantes con guion bajo.
- *Respuesta de forward-auth*: `X-Sibuna-Status`, `X-Sibuna-Rule`, `X-Sibuna-Rule-Hash`.
- *Respuesta de desafío*: `X-Sibuna-Status: CHALLENGE`.
- *Cookie*: `__sibuna_token=<75 chars, 139 with ed25519>; Path=/; Max-Age=<ttl>; HttpOnly; SameSite=Lax[; Secure]`

Esqueleto del archivo de políticas

```
{ "default_action": "CHALLENGE", "waf": true,
  "thresholds": {"challenge_at": 10, "deny_at": 40, "bits_step": 5},
  "ip_rules": {"10.0.0.0/8": "ALLOW", "2001:db8::/32": "DENY"},
  "rules": [
    {"name": "api", "path": "/api/*", "headers": {"X-Api-Key": ".*"}, "action":
"ALLOW"},
    {"name": "checkout", "path": "/checkout/*", "action": "CHALLENGE",
     "challenge": {"difficulty": 20, "algorithm": "posw"}},
    {"name": "login", "path": "^/login$", "action": "CHALLENGE",
     "limits": {"rate": 10, "window_seconds": 60, "ban_seconds": 600}},
    {"name": "old-tls", "ja4": "t12*", "action": "CHALLENGE"},
    {"name": "headless", "user_agent": "Headless", "action": "WEIGH", "weight":
30} ] }
```

JA4: una huella completa, o un prefijo y *; sin huella, no hay coincidencia. Patrones: * o .*, cualquiera; ^...$, exacto; * final, prefijo; / inicial, ruta exacta; en otro caso, subcadena sin distinguir mayúsculas. Las rutas se comparan en forma normal. Orden: WAF -> trie deny/allow -> reglas -> puntuación -> exención -> trie challenge -> bots -> por defecto.

Almacenamiento (Edge) en SQL

```
INSERT INTO policies (id, name, priority, path_pattern, action, difficulty,
algorithm,
  header_matchers, cidr_matchers, weight, enabled, created_at, updated_at)
  VALUES ('p1', 'protect', 10, '/checkout/*', 'challenge', 20, 'posw', '{}', '[]',
0, 1,
```

```
  unixepoch(), unixepoch());
INSERT INTO ip_reputation (ip_or_cidr, reputation_score, banned_until,
trigger_rule, hits, last_seen)
  VALUES ('198.51.100.7', -100, unixepoch() + 86400, 'analyst', 1, unixepoch());
SELECT s.client_ip, s.violation_category, s.path, s.campaign_id
  FROM incidents_fts f JOIN security_incidents s ON s.id = f.rowid
  WHERE incidents_fts MATCH 'union' ORDER BY rank LIMIT 20;
```

SQLite (`sqlite3 <dir>/sibuna.db`); las acciones van en minúsculas. Puntuación ≤ -50 -> prefijo deny; ≥ 50 -> allow. Clúster: el mismo secreto en cada nodo, una base de datos PostgreSQL, `--features postgres-tls`, `--database-url-file <file> --cluster-node N` con un N único de 1 a 8.388.607; sin lista de miembros, líder ni TLS de clúster (SID 0011).

Diagnósticos

Título	Significado
`MALFORMED CHALLENGE`	id editado o truncado; obtén uno nuevo
`INVALID CHALLENGE TAG`	no emitido con esta semilla ni por este nodo
`CHALLENGE EXPIRED`	supera el TTL o desfase de reloj > 60 s
`CLIENT FINGERPRINT MISMATCH`	cambió la dirección o el User-Agent
`DIFFICULTY NOT MET`	al nonce le faltan bits a cero
`INVALID PROOF OF SEQUENTIAL WORK`	las aperturas no llegan a la raíz
`WRONG SOLUTION TYPE`	nonce para PoSW o prueba para Hashcash
`DOUBLE SPEND PREVENTED`	desafío ya usado
`STORE FULL`	conjunto de desafíos gastados saturado; reduce el TTL o aumenta `--spent-tags`
`NETWORK BINDING MISMATCH`	cookie copiada a otro cliente
`INVALID COMMAND LINE`	opción desconocida, valor incorrecto u opción de Zaxonlite eliminada

Métricas para alertas: `denied`, `rate_limited`, `overloaded`, `upstream_errors`, `incidents_dropped`, `incident_write_failures`, `bans_refused`, `forwarded_invalid` (todas `sibuna_*_total`).

Cifras que recordar

Id de desafío de 70 caracteres (carga útil de 36 bytes + etiqueta de 16 bytes); token de 75 caracteres (40 + 16), 139 con Ed25519; bytes de la prueba $32(1 + t(n + 1))$; profundidad PoSW = bits - 3, de 4 a 24; búfer de petición de 64 KB; límite de encabezado de 16 KB; 8 KB de cuerpo inspeccionados; conjunto de desafíos gastados de 1.048.576 etiquetas (`--spent-tags`); 262.144 celdas de tasa por tabla (`--rate-table-cells`); 65.536 posiciones de bloqueo (`--ban-table-slots`); 64 conexiones por dirección de cliente; anillo de incidentes de 512; lote de 32;

distancia de campaña de 0,35 sobre los 16.384 embeddings más recientes; keep-alive de 4.096 peticiones por conexión; pool de 256 sockets hacia el origen; cola de tareas de almacenamiento de 32.

Enrutamiento en el proxy de entrada

Usa la receta de Nginx o Caddy de la parte IX, "Forward-auth detrás de un proxy de entrada". Enruta `/__sibuna/*` directamente; autoriza las peticiones de la aplicación con la URI original, el método y la dirección de cliente de confianza. Sustituye las cabeceras de auditoría de Sibuna entrantes por el resultado de la autorización. Nginx necesita un tratamiento explícito del desafío, del 429 y de la autorización no disponible.

`reverse_proxy` transporta los cuerpos HTTP/1.1 admitidos y los WebSockets; `forward_auth` concede la admisión y el proxy de entrada los transporta. Sibuna no puede inspeccionar ni contar los cuerpos que se omiten en la autorización ni las respuestas del origen. TLS termina en el proxy de entrada; el HTTP/2 nativo queda aplazado. Las subidas usan Content-Length o codificación chunked; la inspección ve como máximo un prefijo de 8 KiB, y los bytes de los archivos son opacos.

Soluciones seleccionadas

Estas soluciones sirven para comprobar el razonamiento, no para sustituir los ejercicios. Si un resultado difiere, primero comparamos los supuestos: los extremos de los intervalos de tiempo, lo que se cuenta como trabajo y si una escritura se ha confirmado son fuentes habituales de desacuerdo.

Parte I: Coste y probabilidad

1.1. Un millón de intentos repartidos entre 100 peticiones son 10.000 intentos por petición. Compartir una sesión no cambia esa aritmética si el número total de peticiones sigue siendo 100. El sistema real debe decidir si un segundo proceso puede satisfacer los vínculos de huella y de enrutamiento; esas condiciones quedan fuera de la ecuación de amortización.
1.2. El exceso es de $1000 - 600 = 400$ registros/s. Una cola vacía de 512 registros da $512/400 = 1.28$ segundos en el modelo de fluido. Los productores reales llegan en ráfagas y el consumidor confirma lotes, así que el momento exacto del primer descarte depende de los tiempos de llegada y de confirmación.

Parte II: Trabajos previos

2.1. Un mismo código fuente compilado dos veces no puede contradecirse a sí misma sobre el enunciado, la codificación del nonce, la forma del árbol o la disposición de las aperturas; una divergencia exigiría un error del compilador. No dice nada sobre si la construcción es sólida (revisión criptográfica), si el entorno de ejecución del navegador la ejecuta correctamente (eso lo cubre una comparación byte a byte entre los probadores WebAssembly y JavaScript) ni si los parámetros están calibrados (medición).
2.2. (a) Sibuna Gate o Anubis, un solo binario; (b) Sibuna Shield, SafeLine o Cloudflare (plan Pro o superior); (c) Sibuna Edge con una base de datos PostgreSQL compartida (`--database-url`), Anubis con un almacén Valkey compartido (para el estado, no para los bloqueos) o Cloudflare. El único producto que cumple las tres condiciones sin un host Docker ni una suscripción es Sibuna Shield con sus nodos compartiendo una base de datos PostgreSQL (`--database-url` y `--cluster-node`); lo deciden las filas "Se ejecuta como", "Inspección" y "Multinodo".

Parte III: Criptografía

3.0. La estimación de la cola es $e^{-4.6} \approx 0.010$. Dividir el espacio de nonces cambia el ritmo al que se completan los intentos, no la probabilidad de éxito de cada intento independiente. Los intentos duplicados o el trabajo de coordinación pueden hacer menos eficiente una implementación paralela real.
3.1. Si cada compromiso tiene éxito con probabilidad q, los g fallan todos con probabilidad $(1-q)^g$. Al menos uno tiene éxito con probabilidad $1-(1-q)^g$. La fórmula no cuenta la

construcción de los compromisos, los presupuestos de consultas, las dependencias entre intentos ni el trabajo necesario para encontrar un compromiso con una probabilidad de aceptación concreta.

3.2. Con profundidad 17 y 16 aperturas, la prueba tiene $32(1 + 16 \times 18) = 9248$ bytes. Con 32 aperturas tiene $32(1 + 32 \times 18) = 18464$ bytes. Los bytes de las aperturas se duplican; la raíz de 32 bytes se sigue enviando una sola vez.

3.3. Sin el nivel de trabajo, una sesión obtenida en una ruta barata admitiría una petición a una ruta cara que exigiera más trabajo; sin `timestamp` se podría emitir un token "en el futuro" que sobreviviera a su política; sin `expiry` nunca caducaría; sin `rule_hash` el origen no podría saber qué regla admitió al cliente; sin `fingerprint` se aceptaría una cookie copiada a otra máquina. La etiqueta debe cubrir los cinco porque cualquier campo que quede fuera podría editarse libremente sin que el verificador lo notara.

Parte IV: Protocolo

4.1. Ambos navegadores comparten la dirección y el User-Agent, así que comparten la huella, la celda del límite de tasa y la posición en la tabla de bloqueos. No comparten el desafío (cada uno obtiene su propio registro con su propio nonce), la entrada del conjunto de desafíos gastados ni la cookie. Para reutilizar la cookie del otro navegador, un atacante detrás del mismo NAT tendría que leerla de la otra máquina: entonces la huella coincidiría, y por eso `HttpOnly` y `SameSite` importan más que el vínculo cuando la dirección es compartida.

Parte V: Estado y orden

5.1. El encabezado de la petición y la primera parte del cuerpo (hasta lo que quepa después del encabezado en 64 KB) están en el búfer de la conexión y se envían con `write_all(request.body)`. Los bytes restantes nunca entran completos en el búfer: `relay_body` los lee del lector del cliente en trozos de 16 KB y escribe cada uno en el origen. El encabezado debe ir primero porque el origen necesita `Content-Length` antes del cuerpo y porque las cabeceras de auditoría forman parte de él.

5.2. El intervalo de emisión es de 100 ms y la tolerancia de ráfaga es de 9.900 ms. Un cliente inactivo puede enviar 100 peticiones de inmediato y después una cada 100 ms. Hasta el extremo inclusivo de 3.000 ms, la cota es 130. Un contador fijo de 10 segundos permitiría 100 en la primera ventana y otras 100 en el instante en que cambia la ventana; un registro deslizante permitiría exactamente 100 en cualquier intervalo de 10 segundos, pero debe guardar 100 marcas de tiempo por cliente.

Parte VI: Algoritmos

6.1. Un apóstrofo en un nombre y un parámetro corriente llamado `order` no constituyen una expresión SQL. Un detector de subcadenas podría confundir cualquiera de los dos con sintaxis. El detector estructural busca más indicios, incluidos operadores y relaciones entre literales. Sigue siendo una inspección acotada, no un analizador de todos los lenguajes de aplicación.

6.2. Los filtros LDAP se construyen con paréntesis, `|`, `&`, `!` y `=`. El detector necesita una clase de bytes para `(` y `)` (nueva) y puede reutilizar las clases existentes de `=` y de separadores de shell para `|` y `&`. Debe activarse ante una *secuencia* como `)(` o `(|(`, no ante un paréntesis aislado, por la misma razón por la que el detector SQL exige una comilla o una tautología.

Parte VII: Motor del navegador

7.1. Con los 16 bits de trabajo por defecto (profundidad 13 con 16 aperturas), los supuestos 20 ms en WebAssembly se convierten en $50 \times 20 = 1000$ ms, un segundo, en JavaScript en el portátil, y en cuatro segundos en el teléfono más lento. Las cifras son suposiciones; medimos los dispositivos de nuestros propios clientes antes de fijar una política. Una política sensata: dejar que la página reenvíe el mensaje `fallback`, registrarlo en el servidor como métrica y reducir la dificultad para una regla que coincida con los clientes que se sabe que bloquean WebAssembly, en lugar de reducirla para todos.

Parte VIII: Evaluación

8.2. Para la fila `admitted` de Gate, $11.84 \times 211,345/10^6 \approx 2.50$ núcleos ocupados de cuatro, así que el daemon no estaba limitado por la CPU y el generador de carga puso el límite. Los 11,84 us de CPU por petición quedan muy por encima del coste de clasificación, inferior a un microsegundo, porque una fila del producto incluye también la lectura y escritura del socket en el kernel, el analizador, la comprobación de la etiqueta de sesión, los incrementos de métricas, el formato de la respuesta, el vaciado del escritor y el traspaso del planificador entre el generador de carga y el hilo de la conexión. La batería de primitivas no mide ninguno de ellos.

Parte IX: Operaciones

9.1. Si los datos se confirman antes de una actualización separada del recibo, una caída entre ambas deja el recibo antiguo y un reintento puede aplicar el efecto otra vez. Si el recibo se confirma primero, una caída antes de escribir los datos puede hacer que el reintento omita datos que nunca existieron. Una única transacción atómica elimina ambos huecos. Si se pierde el acuse de recibo de la transacción, sigue siendo necesario conservar el reintento.

9.2. `ip_rules: {"10.20.0.0/16": "ALLOW"}` (o una fila de `ip_reputation` con puntuación 100) admite el entorno de preproducción (staging); una regla `{"name": "admin", "path": "/admin/*", "action": "CHALLENGE", "challenge": {"difficulty": 20, "algorithm": "posw"}}` exige el trabajo; una regla que coincide con `headers: {"X-Partner-Key": ".*"}` con `ALLOW` y `"limits": {"rate": 30, "window_seconds": 10}` admite al socio a 30 peticiones cada 10 segundos. La cuota de la regla se comprueba en cada petición que coincide con ella, después del límite por cliente del daemon (`--rate-limit`, `--rate-window`), que por tanto debe ser al menos igual de generoso.

Parte X: Referencia

10.1. El registro de desafío lleva la huella con clave de la dirección que lo obtuvo; la solución llega desde una dirección nueva, así que `verify_and_mint` devuelve `FingerprintMismatch` antes de mirar la prueba. La recuperación más sencilla está en la página intermedia: ante un `400` cuyo título es `CLIENT FINGERPRINT MISMATCH`, obtener un desafío nuevo y resolverlo otra vez en lugar de mostrar el error, ya que el trabajo hecho no se puede transferir.

Continuar la investigación

Cambia un invariante cada vez en una prueba desechable. Quita la nueva comprobación de la instantánea, omite el recibo del incidente o redondea hacia abajo el intervalo de tasa. Predice una secuencia de fallos antes de ejecutar la prueba. Una prueba de regresión útil hace algo más que comprobar la salida del caso sin errores: demuestra por qué la condición omitida es necesaria.

Glosario

Admisión
: La decisión de reenviar una petición al origen. En Sibuna la toma una sesión válida, una regla `ALLOW`, una ruta estática exenta o un prefijo de reputación `allow`, y nunca anula una denegación del WAF.

Autómata Aho-Corasick
: Un autómata finito que encuentra cualquiera de un conjunto fijo de patrones en una sola pasada sobre la entrada. Sibuna integra los enlaces de fallo en una tabla de transiciones densa, de modo que cada byte cuesta una carga de la tabla.

Tabla de bloqueos
: 4.096 posiciones sin cerrojos (lock-free) de direcciones de clientes bloqueadas, con caducidad; las escriben el honeypot y la reputación compartida en un clúster, y se leen en cada petición.

Campaña
: Un grupo de incidentes de seguridad cuyos embeddings de carga útil están a una distancia coseno de 0,35 o menos entre sí; una agrupación heurística, no una prueba de origen común.

Registro de desafío
: El identificador sin estado de 70 caracteres que un cliente debe resolver: versión, algoritmo, dificultad, aperturas, hora de emisión, huella del cliente, nonce PRF, hash de regla y una etiqueta BLAKE3 con clave de 16 bytes. Emitir uno no escribe nada en el servidor.

Dificultad, bits de trabajo
: El único entero que escala ambos niveles: el número de bits iniciales a cero para Hashcash, o tres más que la profundidad del árbol para el trabajo secuencial.

Edge
: Un despliegue de Sibuna con `--data-dir` (un archivo SQLite en un nodo), o con `--database-url` y `--cluster-node` (una base de datos PostgreSQL compartida por un clúster), que aporta políticas persistentes, reputación compartida y análisis forense de incidentes a cualquiera de los dos modos de protección.

Espacio del motor
: El lugar donde vive el motor de políticas publicado, como un puntero con recuento de referencias. Los workers mantienen una referencia mientras clasifican; el hilo de almacenamiento construye un motor nuevo fuera del camino de la petición y lo publica intercambiando el puntero, y el motor antiguo se libera cuando su último lector lo libera.

Huella
: Un hash con clave de la dirección del cliente y del User-Agent, vinculado a los desafíos y los tokens para que ninguno pueda reutilizarse desde otro cliente.

Forward auth
: Modo de despliegue en el que un proxy de entrada pregunta a Sibuna si debe admitir una petición (`200`) o desafiarla, denegarla o limitarla (`401`, `403`, `429`), y hace él mismo de proxy hacia el origen.

Gate
: El modo de protección con admisión por prueba de trabajo, sesiones, reglas, reputación, límites GCRA y bloqueos, pero sin inspección de aplicaciones (`--gate`).

GCRA
: El Generic Cell Rate Algorithm: un tiempo teórico de llegada por cliente que admite una ráfaga de N y después una petición por intervalo de emisión, sin fronteras entre ventanas.

Hashcash
: Primer nivel: encontrar un nonce tal que el SHA-256 del desafío, dos puntos y el nonce decimal tenga b bits iniciales a cero. Se esperan 2^b intentos, y se verifica con un hash.

Honeypot
: Un enlace invisible a `/__sibuna/honeypot`; un cliente que lo sigue queda bloqueado y se registra un incidente con reputación -100.

Recolector de conexiones inactivas
: Un hilo que cierra las conexiones que llevan en silencio más de `--idle-timeout`, para que un cliente lento no pueda acaparar un hilo de conexión.

Página intermedia
: La página HTML integrada que se sirve en lugar de una página protegida; obtiene un desafío, lo resuelve en un Web Worker, envía la solución y recarga.

JA4
: Una huella de 36 caracteres, definida por FoxIO, de cómo abre un cliente una conexión TLS: protocolo, versión de TLS, nombre de servidor, número de cifrados y de extensiones, ALPN, y los hashes de las suites de cifrado y de las extensiones, ordenadas. Sibuna no la calcula; lee el valor que un proxy de entrada de confianza reenvía en la cabecera nombrada por `--ja4-header`, y las reglas la comparan con el campo `ja4` (SID 0014). No tiene relación con la huella del cliente vinculada a los desafíos y los tokens.

BLAKE3 con clave
: La función pseudoaleatoria en la que se basan la derivación de claves de Sibuna, las etiquetas de desafíos, las etiquetas de sesión, las huellas y los nonces de los desafíos.

Token MAC
: La cookie de sesión por defecto: una carga útil de 40 bytes (versión, nivel de trabajo, marca de tiempo, caducidad, hash de regla, huella) y una etiqueta BLAKE3 con clave de 16 bytes, verificada en tiempo constante.

Nivel de trabajo
: El mecanismo y los bits de trabajo que el titular de una sesión resolvió realmente, guardados en el token; una ruta admite una sesión solo cuando su nivel alcanza el requisito de la ruta.

Apertura
: En una prueba de trabajo secuencial, la etiqueta de una hoja más las etiquetas hermanas a lo largo de su camino hasta la raíz; el verificador recalcula el camino y lo compara con el compromiso.

Prueba de trabajo secuencial (Proof of Sequential Work, PoSW)
: Segundo nivel: el grafo de hashes de Cohen-Pietrzak cuyas etiquetas dependen de todas las etiquetas anteriores, lo que obliga a calcular $2^{n+1} - 1$ hashes secuenciales con independencia de cuántos núcleos tenga el probador. Se verifica con $t(n + 1)$ hashes.

Trie de reputación
: El trie radix de 128 bits que contiene los prefijos `allow`, `deny` y `challenge` del archivo de políticas y de la tabla `ip_reputation`.

Conjunto Robin Hood de desafíos gastados
: La tabla de direccionamiento abierto y capacidad fija con las etiquetas de los desafíos resueltos, que impide enviar dos veces una solución válida.

Hash de regla
: Un hash de 64 bits del nombre de la regla que exigió un desafío; viaja en el desafío, en el token y en la cabecera `X-Sibuna-Rule-Hash`.

Shield
: El modo de protección por defecto: Gate más el cortafuegos semántico (`--shield`).

SID
: Un registro Sibuna Discussion: un documento de diseño con formato de artículo en `docs/sid/records` que recoge los supuestos, las demostraciones y las mediciones de una parte del sistema.

Modo de protección
: Una de las dos formas del producto, Gate o Shield; el almacenamiento y el clúster son opcionales en ambas.

WEIGH
: Una acción de regla que suma un peso con signo a la puntuación de una petición en lugar de decidir; los totales iguales o superiores a los umbrales desafían con bits de trabajo adicionales o deniegan.

Zaxonlite
: El motor integrado de replicación de SQLite (Multi-Paxos) del SID 0005, sustituido por el almacenamiento en SQLite y PostgreSQL en el SID 0011; sus opciones de clúster ahora detienen el arranque con `RemovedOption`.

Bibliografía

- Aho, A. V. y Corasick, M. J. "Efficient string matching: an aid to bibliographic search." *Communications of the ACM* 18(6), 1975.
- ATM Forum. *Traffic Management Specification Version 4.0*, af-tm-0056.000, 1996 (el Generic Cell Rate Algorithm).
- Back, A. "Hashcash: a denial of service counter-measure." Informe técnico, 2002.
- Blocki, J., Lee, S. y Zhou, S. "On the security of proofs of sequential work in a post--quantum world." *Information-Theoretic Cryptography (ITC)*, 2021.
- Celis, P. *Robin Hood Hashing*. Tesis doctoral, University of Waterloo, 1986.
- Cohen, B. y Pietrzak, K. "Simple proofs of sequential work." *EUROCRYPT*, 2018.
- Dwork, C. y Naor, M. "Pricing via processing or combatting junk mail." *CRYPTO*, 1992.
- Fiat, A. y Shamir, A. "How to prove yourself: practical solutions to identification and signature problems." *CRYPTO*, 1986.
- Fielding, R., Nottingham, M. y Reschke, J. *HTTP Semantics*, RFC 9110, y *HTTP/1.1*, RFC 9112. IETF, 2022.
- FoxIO. *JA4+ Network Fingerprinting*. https://github.com/FoxIO-LLC/ja4, 2023 (la huella de cliente TLS JA4).
- Hanson, N. «libinjection: SQL injection detection by tokenization.» Black Hat USA, 2012.
- Juels, A. y Brainard, J. "Client puzzles: a cryptographic countermeasure against connection depletion attacks." *NDSS*, 1999.
- Lamport, L. "The part-time parliament." *ACM Transactions on Computer Systems* 16(2), 1998; y "Paxos made simple." *ACM SIGACT News* 32(4), 2001.
- Mahmoody, M., Moran, T. y Vadhan, S. "Publicly verifiable proofs of sequential work." *Innovations in Theoretical Computer Science (ITCS)*, 2013.
- O'Connor, J., Aumasson, J.-P., Neves, S. y Wilcox-O'Hearn, Z. *BLAKE3: one function, fast everywhere*. Especificación, 2020.
- Vyukov, D. "Bounded MPMC queue." 1024cores.net, 2011 (el anillo con números de secuencia usado para la cola de incidentes).
- Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J. y Zhu, H. "Hyperscan: a fast multi-pattern regex matcher for modern CPUs." *NSDI*, 2019.
- Sibuna Discussions 0001 a 0014, `docs/sid/records`, 2026: proceso, arquitectura base, motor declarativo de políticas, inspección semántica, arquitectura de almacenamiento Zaxonlite, fundamentos matemáticos, interfaz de gestión de la consola, monitorización del tráfico de bots de IA, cuerpos de petición chunked, la migración a Rust, el almacenamiento mediante sqlx, TLS saliente para la consola mediante rustls, el almacenamiento de la consola y las huellas JA4 de confianza desde el proxy de entrada.

Índice

A
Admisión, **152**
Apertura, **153**
Autómata Aho-Corasick, **152**

B
BLAKE3 con clave, **153**

C
Campaña, **152**
Conjunto Robin Hood de desafíos gastados, **154**

D
Dificultad, bits de trabajo, **152**

E
Edge, **152**
Espacio del motor, **152**

F
Forward auth, **152**

G
Gate, **153**
GCRA, **153**

H
Hash de regla, **154**
Hashcash, **153**
Honeypot, **153**
Huella, **152**

J
JA4, **153**

M
Modo de protección, **154**

N
Nivel de trabajo, **153**

P
Página intermedia, **153**
Prueba de trabajo secuencial (Proof of Sequential Work, PoSW), **153**

R
Recolector de conexiones inactivas, **153**
Registro de desafío, **152**

S
Shield, **154**
SID, **154**

T
Tabla de bloqueos, **152**
Token MAC, **153**
Trie de reputación, **154**

W
WEIGH, **154**

Z
Zaxonlite, **154**