SIBUNA

Un motor de defensa web sin asignaciones y condicionado al trabajo

Admisión condicionada al trabajo, inspección en tiempo lineal, estado acotado y almacenamiento mediante SQL

Vikrant Rathore

Con la colaboración de Ronak Rathore y Kanak Rathore
Insan AI | Versión 1.0.0-beta.3

RESUMEN

Los cortafuegos de aplicaciones web y las defensas perimetrales suelen adolecer de tres problemas de arquitectura. El primero es la *inversión de costes*: a un remitente enviar una petición no le cuesta casi nada, mientras que el defensor la analiza, la recorre con expresiones regulares que pueden hacer retroceso y consulta por ella un almacén externo. El segundo es la *imprevisibilidad en tiempo de ejecución*: las asignaciones en el heap en el camino de la petición, las pausas de la recolección de basura y los despliegues de varios contenedores hacen difícil acotar la cola de la distribución de latencias. El tercero es el *peso operativo*: el estado compartido de bloqueos, tokens y límites suele residir en servicios que el operador debe mantener junto al cortafuegos. **Sibuna** es nuestro intento de resolver estos problemas con unos pocos mecanismos sencillos en un único binario autónomo, y este documento explica el razonamiento que hay detrás de cada uno. (i) *Admisión condicionada al trabajo*: un navegador obtiene una sesión resolviendo un desafío Hashcash o una prueba de trabajo secuencial de Cohen-Pietrzak, elegida por regla, y el servidor verifica la respuesta en 70,2 ns o 22,3 us frente a un desafío que nunca almacenó. (ii) *Un camino de la petición acotado y sin asignaciones*: un autómata Aho-Corasick por clases de bytes compara 40 firmas de bots en 55,7 ns, 13,7 veces más rápido que la búsqueda secuencial; un limitador GCRA decide en 6,2 ns con 16 bytes de estado por cliente; y un conjunto Robin Hood de desafíos gastados registra una solución en 21,0 ns. La inspección de inyección SQL, cross-site scripting, inyección de comandos y recorrido de rutas se ejecuta en la misma pasada, en tiempo lineal en la entrada. (iii) *Estado compartido sin un protocolo de consenso propio*: un archivo SQLite integrado sirve a un nodo y una única base de datos PostgreSQL compartida sirve a un clúster; los lee un único hilo de almacenamiento al que las peticiones nunca esperan. En un AMD Ryzen 9 5950X, con el daemon en cuatro CPU, Sibuna responde unas 200.000 peticiones forward-auth por segundo (de 197.532 a 211.636 según la carga de trabajo) con un p99 de 305 a 327 us, mantiene su p99.9 en 62 us bajo una carga en lazo abierto de 100.000 peticiones por segundo y alcanza como máximo unos 16 MiB de memoria residente. Explicamos cómo se obtuvo cada cifra y enumeramos lo que no hemos medido.

6,2 ns	55,7 ns	22,3 µs	16 MiB
LÍMITE DE TASA GCRA	**DETECTOR DE BOTS**	**VERIFICACIÓN PoSW**	**RSS MÁXIMO**
cliente en hash | 16 particiones	Aho-Corasick | 40 firmas	profundidad 13 | 16 aperturas	daemon con carga de wrk | 4 CPU

Sibuna en un núcleo de un AMD Ryzen 9 5950X, o en cuatro CPU para el daemon, 15 de octubre de 2026: benchmarks/results/primitives-latest.md y product--latest.md; la sección de evaluación describe el método.

1 Prólogo: la economía de una petición

Cada petición que llega a un servidor reclama su atención. Un método, una ruta y unas cuantas cabeceras no demuestran que merezca la pena atenderla. Por eso una defensa sitúa una decisión entre la petición y el origen: admitirla, rechazarla o pedir antes a su remitente un trabajo verificable. Este documento sigue esa decisión en Sibuna, desde sus matemáticas hasta los bytes en memoria.

> **Feynman sobre la intuición física: ¿quién paga el trabajo?**
>
> *Pensemos en un torniquete que gira con el toque más leve. Detrás hay un guardia que, con cada toque, debe levantarse, revisar tres libros de registro y telefonear a la oficina central. Un niño puede pasarse la tarde tocando el torniquete desde fuera sin ningún esfuerzo, mientras el guardia acaba agotado antes de comer. Así están dispuestos muchos cortafuegos: el lado barato es el del atacante. La física sugiere una disposición mejor. Conectemos el torniquete a una bomba, de modo que quien quiera pasar tenga que subir un cubo de agua a un depósito situado sobre la entrada. Subirlo exige un pequeño esfuerzo honrado; al guardia le basta con echar un vistazo al depósito para ver que el agua ha subido. Mirar no le cuesta casi nada al guardia, y subir el agua le cuesta algo real al visitante. Quien pasa a diario sube el cubo una vez y entra y sale todo el día; el bromista pronto se vuelve a casa.*

Los recuadros con voz propia de este documento son palabras nuestras, escritas en el espíritu de Richard Feynman, Donald Knuth y Leslie Lamport, cuya manera de explicar las cosas intentamos seguir. No son citas, salvo cuando un recuadro nombra su fuente.

Un remitente automatizado paga muy poco por cada petición que envía. Cuando una petición llega a un cortafuegos convencional, el defensor puede completar un handshake TLS, asignar memoria para copiar y dividir la petición, recorrerla con una lista de expresiones regulares y hacer un viaje de ida y vuelta por la red hasta un almacén compartido para actualizar un contador. No atribuimos a esta asimetría una proporción universal, porque no existe: depende del



Fig. 1 Adónde va el trabajo. Arriba, una defensa que inspecciona primero gasta su propio trabajo en cada petición que envía un remitente. Abajo, Sibuna pide al remitente 2^b intentos de hash esperados y verifica la respuesta con un solo hash. Los medidores son un modelo; el tiempo de verificación está medido (verificación Hashcash con 16 bits, `benchmarks/results/primitives-latest.md`).

origen, de la caché y del ataque. En su lugar, un modelo sencillo hace visible la decisión. Sean c_o el trabajo del origen por petición admitida, c_g el trabajo del control de admisión por petición y a la fracción de peticiones que el control admite. El control de admisión compensa solo cuando

$$c_g < (1 - a)\, c_o,$$

es decir, cuando el trabajo que ahorra al origen supera su propio coste[1]. De ello se siguen dos consecuencias. El coste propio del control, c_g, debe ser pequeño y estar acotado sea cual sea la entrada, y la decisión de admitir debe resultar cara para el remitente, no para el servidor.

Sibuna invierte la asimetría condicionando la admisión a una prueba de trabajo. Para obtener una sesión, un navegador encuentra un nonce cuyo resumen SHA-256 tiene b bits iniciales a cero, o calcula una prueba de trabajo secuencial. La búsqueda esperada es de 2^b intentos de hash, 65.536 con $b = 16$; el servidor comprueba un solo candidato. Además, la comprobación del servidor no tiene estado: emitir un desafío no escribe nada, y solo una solución pagada ocupa memoria.

Una prueba de trabajo no demuestra humanidad, identidad ni intención. Un remitente puede alquilar máquinas, compartir una sesión mientras sea válida o repartir el trabajo entre varios equipos. El desafío cambia el precio de la admisión; la política y la inspección siguen decidiendo qué puede hacer una petición admitida.

2 Visión general del diseño: tres decisiones en una sola pasada

Sibuna mantiene la admisión, la inspección y la reputación como tres decisiones separadas, y las toma en una sola pasada sobre la petición, en un único proceso, antes de que el origen haga ningún trabajo. Las ofrece en dos modos de protección. *Gate* ofrece admisión por prueba de trabajo con reglas, reputación por CIDR y controles locales frente a inundaciones de peticiones. *Shield*, el modo por defecto, añade una inspección acotada de la carga útil de la petición. Cualquiera de los dos funciona como proxy inverso delante de un origen o como endpoint forward-auth al que consulta un proxy de entrada como Caddy o Nginx. El almacenamiento persistente de políticas, reputación e historial de incidentes es una opción para ambos: local a un nodo o compartido por un clúster[2].

[1] Libro, parte I, "La admisión es una decisión económica". La desigualdad es un modelo de capacidad, no una medición; omite el ancho de banda, el estado de las conexiones y el retardo del cliente.

[2] SID 0002, "Decision"; libro, parte II, "Tres ideas en un solo ejecutable"; libro, parte VI, "El motor declarativo de políticas". El ejemplo por regla, el ticket de requisito y la sesión con nivel de trabajo proceden de la parte IV del libro; la referencia adaptativa, de la parte IV del libro y del SID 0006 (nota de revisión, parte H).

Tabla 1 Los dos modos de protección (SID 0002; parte II del libro)

Capacidad	Gate (`--gate`)	Shield (`--shield`, por defecto)
Admisión	Desafío de navegador Hashcash o de trabajo secuencial; un token de sesión vinculado al cliente y al trabajo que pagó	Igual
Política	Reglas declarativas (ruta, User-Agent, cabeceras, CIDR IPv4 e IPv6, una huella TLS JA4 de un proxy de entrada de confianza) con ALLOW, DENY, CHALLENGE y WEIGH; firmas de bots; reputación	Igual
Controles frente a inundaciones	Límites de tasa GCRA, honeypot, bloqueos temporales	Igual
Inspección de la carga útil	Desactivada	Inyección SQL, cross-site scripting, inyección de comandos y recorrido de rutas, sobre los primeros 8 KiB del cuerpo
Estado opcional	Políticas y reputación en SQLite, o compartidas mediante PostgreSQL	Además, análisis forense de incidentes: búsqueda de texto completo y agrupación en campañas

2.1 Las decisiones de diseño

La mayoría de las ideas que siguen proceden de la literatura de investigación o de sistemas anteriores, y la parte II del libro traza su linaje. Lo que hemos intentado es combinarlas en un único camino de la petición acotado, mantener barato cada paso y medir el resultado con honestidad. Enumeramos las decisiones con la razón de cada una.

1. *Desafíos que al servidor no le cuesta nada emitir.* Un desafío es un registro autoautenticado sellado con un hash con clave, así que emitirlo no escribe estado; solo una solución pagada ocupa memoria, y la libera cuando el desafío caduca (Los algoritmos y sus garantías).
2. *Dos niveles de trabajo, elegidos por regla.* Junto a Hashcash, Sibuna ofrece la prueba de trabajo secuencial de Cohen-Pietrzak, cuya solidez se ha demostrado incluso frente a probadores cuánticos y cuya prueba se verifica en microsegundos. Una ruta de pago puede exigir 20 bits de trabajo secuencial mientras un blog exige 16 bits de Hashcash (Pruebas de trabajo: búsqueda paralela y trabajo secuencial).
3. *Sesiones que recuerdan el trabajo pagado.* Un token de sesión registra el mecanismo y los bits de trabajo que resolvió su titular. Solo admite rutas que no exigen más, así que una sesión barata no puede abrir una ruta cara.
4. *Tokens simétricos.* Las sesiones se sellan con BLAKE3 con clave, que se verifica en 155,8 ns, unas 260 veces menos trabajo que una firma Ed25519, y frente al que la búsqueda cuántica solo ofrece una aceleración cuadrática.
5. *Una inspección que la sesión no puede eludir.* La admisión y la inspección se ejecutan en la misma pasada, y una sesión válida solo satisface un desafío: una petición con sesión y `q=' OR 1=1--` se sigue rechazando.
6. *Inspección en tiempo lineal y atenta a la estructura.* Un recorrido por clases de bytes, una pasada del autómata y pequeños analizadores léxicos estructurales deciden en tiempo lineal en la entrada, sin expresiones regulares que puedan hacer retroceso.
7. *Estado acotado y colocación con clave.* Cada tabla tiene una capacidad fija y un comportamiento declarado para cuando está llena, y las claves se colocan mediante un hash con clave, de modo que un atacante no puede dirigir colisiones contra una víctima (El hash con clave y la revisión de seguridad).
8. *Ninguna asignación por petición y actualizaciones de políticas sin pausas.* Una conexión sirve todas las decisiones desde un único búfer fijo, y un motor de políticas nuevo se publica sin que espere ninguna petición ni ningún escritor (Simpatía mecánica: sin asignaciones y con estado acotado).
9. *Una dificultad que sigue a la carga.* Por encima de una base de 50 soluciones verificadas por segundo, cada duplicación de la tasa añade un bit de trabajo, hasta seis, de modo que una inundación paga exponencialmente más mientras el tráfico tranquilo conserva el coste base.
10. *Campañas a partir de incidentes.* Las peticiones denegadas se registran con un pequeño embedding, y los incidentes similares se agrupan en campañas que un operador puede investigar desde la consola.

3 Simpatía mecánica: sin asignaciones y con estado acotado

> **Knuth sobre la precisión mecánica: líneas de caché y matemáticas concretas**
>
> Un programador que llama a un asignador de propósito general dentro del bucle interno de un sistema de tiempo real ha cedido parte del diseño a un desconocido. El asignador puede ser rápido en promedio, pero su coste depende de la historia del heap, y esa historia la escribe quien envía las peticiones. Antes de optimizar nada deberíamos contar: cuántos bytes toca cada paso, cuántas líneas de caché, cuántas cargas dependientes. Después deberíamos medir, porque también contar puede fallar. Un programa es elegante cuando cada byte que necesita tiene un lugar conocido antes de abrir el primer socket, y cuando cada bucle hace exactamente el trabajo que exigen sus matemáticas.

3.1 El invariante de cero asignaciones

Cada asignación dinámica en el camino de una petición es una palanca de denegación de servicio: fragmentación a lo largo de los días, contención en los cerrojos del asignador y amplificación mediante peticiones diseñadas para provocar asignaciones. La regla de Sibuna es que la clasificación, la verificación y el traspaso al proxy se ejecuten sin asignaciones en el heap. Una conexión posee un búfer de lectura de 64 KiB y un búfer de escritura de 16 KiB durante toda la vida de su socket, y un espacio de trabajo del proxy de 40 KiB que se asigna una sola vez al empezar la conexión. Todo lo demás son slices sobre ellos: el analizador localiza el método, la ruta, la consulta, las cabeceras y las cookies de una petición dentro del búfer de lectura y no copia nada[3]. El sistema de tipos de la implementación garantiza que una petición analizada no puede sobrevivir a los bytes que describe, de modo que el búfer no puede rellenarse mientras una petición todavía lo lee.

La regla es un contrato del camino de la petición, no una afirmación sobre todo el proceso. La creación de hilos, el arranque y el almacenamiento sí asignan memoria. El contrato lo comprueban pruebas que cuentan asignaciones: la prueba de régimen estable del daemon cuenta sus asignaciones, y la del proxy no encuentra ninguna a lo largo de 500 peticiones keep-alive a través del proxy[4].

3.2 Hilos acotados y reutilizados

Cada conexión aceptada se atiende en un hilo propio, de modo que un cliente lento solo se retrasa a sí mismo y el camino de la petición sigue siendo síncrono. Consideramos un bucle de eventos y conservamos el modelo de hilos: la velocidad del camino de la petición procede de un analizador sin copias, de un búfer fijo por conexión y de decisiones sin asignaciones, y un bucle de eventos no mejora ninguna de ellas[5]. El número de hilos de conexión está acotado por `--max-connections` (1.024 por defecto); por encima de esa cifra, el bucle de aceptación responde `503` sin crear nada. Una misma dirección de par del socket puede mantener como máximo 64 conexiones por defecto, y un par IPv6 se cuenta con su red /64.

Los hilos proceden de una tabla fija de posiciones que admiten join. Un hilo terminado conserva su pila hasta que se le hace join, así que una tarea nueva hace join de la posición terminada *más baja* y la ocupa, en lugar de avanzar un cursor; la memoria residente sigue entonces a las conexiones abiertas, no a las atendidas. Un hilo que termina su conexión se aparca y toma la siguiente. La sección Qué cambió la medición cuenta cómo una medición nos enseñó a añadir esta última regla.

3.3 La cadena de procesamiento de la petición

Una petición pasa las comprobaciones siguientes en un orden fijo. Las comprobaciones baratas van primero, y cada decisión que termina la petición lo hace antes de que se ejecute cualquier etapa más cara.

Una sesión válida satisface un desafío; nunca anula una denegación ni un hallazgo del WAF. El proxy, por su parte, lee el encabezado de la respuesta del origen, clasifica su delimitado (sin cuerpo, una longitud, codificación chunked o hasta el cierre), retransmite el cuerpo con exactitud y devuelve a un pool de 256 sockets dividido en franjas los sockets del origen que quedaron en un límite limpio, de modo que una conexión de cliente a través del proxy sigue abierta y no hace falta reconectar con el origen en cada petición.

3.4 Publicar políticas sin detener las peticiones

El motor de políticas se reconstruye fuera del camino de la petición cuando cambian las políticas, y las peticiones nunca deben ver un motor a medio construir. El hilo de almacenamiento construye un motor completo y lo publica, con recuento de referencias, en un espacio con un contador de generación. Cada conexión mantiene una caché: carga la generación con una única carga acquire y solo toma una referencia nueva después de una publicación. Un motor antiguo se libera cuando lo suelta la última caché que lo retiene, y ningún escritor espera nunca a que terminen los lectores. Con contención, la lectura en caché es más de 400 veces más barata que tomar un cerrojo de lectura-escritura. El propio motor ocupa unos 200 KiB de tablas compactas (209.052 bytes), con capacidad para 128 reglas y 8.192 nodos del trie.

3.5 Cerrojos por partición

Las tablas que comparten los hilos de conexión (el conjunto de desafíos gastados, el limitador y la tabla de bloqueos) se dividen en 16 particiones según un hash de la clave, y cada partición ocupa su propia línea de caché. Sus secciones críticas tienen unas decenas de instrucciones, así que medimos un spin lock frente a un mutex adaptativo, que gira brevemente y después se duerme. Un spin lock ahorra alrededor de un nanosegundo cuando nadie compite, pero un servidor perimetral ejecuta más hilos que CPU, y un spin lock cuyo poseedor es desalojado por el planificador hace que cada hilo en espera consuma su porción de tiempo. El mutex ganó con contención, así que lo conservamos.

[3]Libro, parte V, "Un hilo por conexión, con un límite" y "Un búfer por conexión".
[4]Libro, parte V, "El bucle de conexión"; `cmd/sibuna/tests/steady_state.rs` y `crates/net/tests/steady_state.rs`.
[5]SID 0010, "Concurrency model" y "Rationale and alternatives"; SID 0002, "Request contract" y sus notas de revisión (conexiones por dirección, prefijo IPv6, el conjunto de conexiones al origen dividido en franjas).

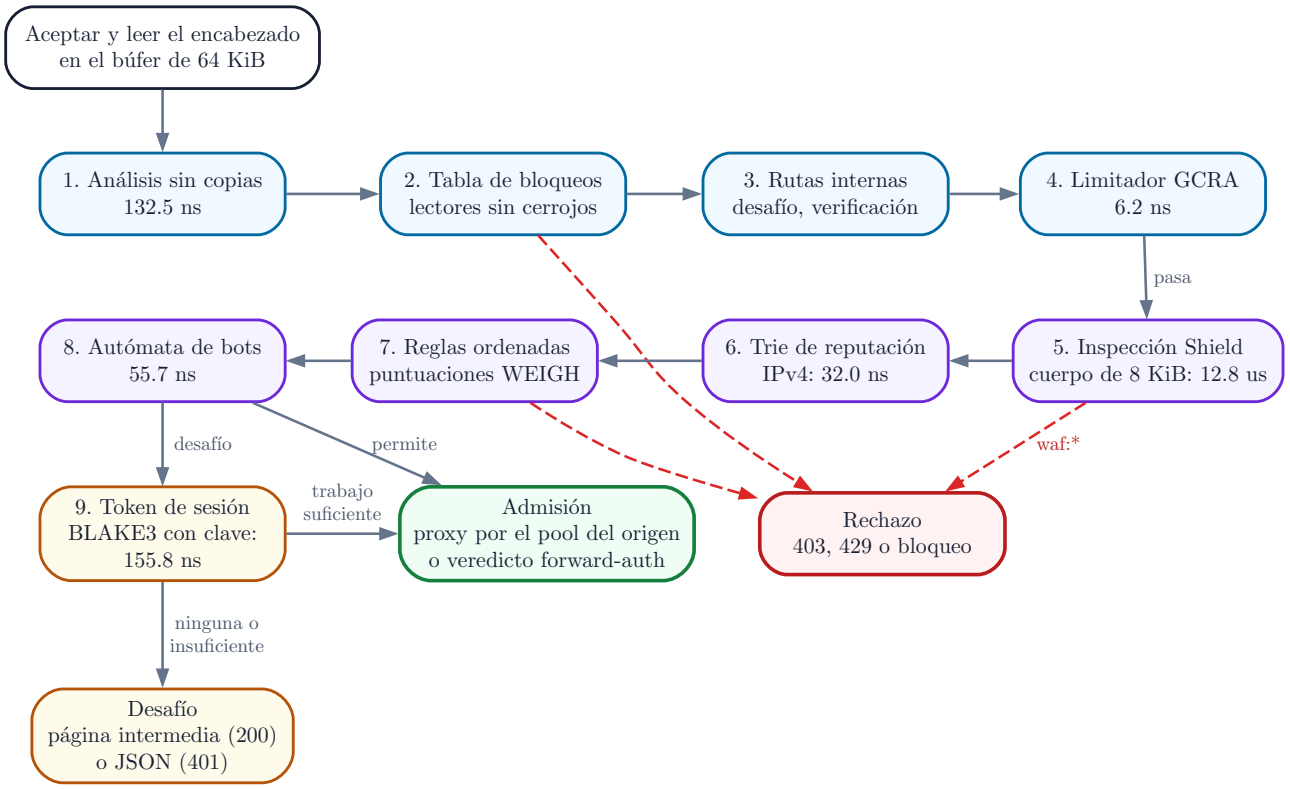


Fig. 2 La cadena de procesamiento de la petición, con la mediana de cada primitiva en un núcleo allí donde la mide el conjunto de primitivas (`benchmarks/results/primitives-latest.md`). Las etapas se ejecutan en el orden numerado; las flechas discontinuas muestran los rechazos tempranos (el limitador GCRA y el trie de reputación rechazan de la misma forma). La clasificación completa de una petición de navegador tarda 75,9 ns en el perfil Gate y 790,8 ns en el perfil Shield.

4 Los algoritmos y sus garantías

Cada primitiva del camino de la petición tiene una declaración de lo que cuesta y de lo que garantiza. Presentamos tres de ellas como teoremas, con un breve argumento; el SID 0006 y las partes III a VI del libro de Sibuna dan las demostraciones completas, las definiciones y las referencias.

Teorema 1 (inspección en tiempo lineal con un autómata Aho-Corasick por clases de bytes). Sea $\mathcal{P}$ un conjunto de patrones de longitud total m cuyos bytes se reparten en $c \leq 256$ clases, y sea un texto de longitud M. Sibuna construye el autómata en tiempo y espacio $O(mc)$ e informa de si algún patrón aparece en el texto en tiempo $O(M)$, con una carga dependiente de la tabla por byte de entrada, sea cual sea el número de patrones. Ninguna entrada puede hacerlo retroceder.

Esbozo de la demostración. El autómata de Aho y Corasick recorre un texto una sola vez tras construir un trie con enlaces de fallo. Sibuna integra los enlaces de fallo en la tabla de transiciones al construirla, así que cada byte

Tabla 2 El coste de una lectura del motor con 1 y 16 hilos leyendo en bucle, en un Ryzen 9 5950X (parte V del libro, octubre de 2026)

Lector	1 hilo	16 hilos
Cerrojo de lectura-escritura, solo la guarda de lectura	4,2 ns	752 ns
Cerrojo de lectura-escritura y una referencia nueva	7,6 ns	833 ns
Comprobación de la generación con una referencia en caché	0,8 ns	1,8 ns

Tabla 3 Rendimiento de los cerrojos de partición, en millones de operaciones por segundo, en un Ryzen 9 5950X (parte V del libro; SID 0010)

Carga de trabajo	Mutex adaptativo	Spin lock
Conjunto de desafíos gastados, 16 hilos	10,8	8,6
Conjunto de desafíos gastados, 64 hilos	11,2	6,9
GCRA sobre seis direcciones calientes, 16 hilos	44,9	19,3
GCRA sobre seis direcciones calientes, 64 hilos	37,4	6,5

de entrada es una transición y nunca se sigue una cadena de fallos. Una clase de bytes es un byte que aparece en algún patrón, con la mayúscula y la minúscula de una letra en la misma clase, o bien la clase única de todos los demás bytes; por tanto, una fila de la tabla tiene c entradas en lugar de 256. Los estados de coincidencia se numeran después de todos los demás, así que detectar una coincidencia es una sola comparación.
Medición. Con 40 firmas de bots, el autómata tarda 55,7 ns, frente a los 763,0 ns de la búsqueda secuencial de subcadenas con los mismos patrones y la semántica de cualquier coincidencia: 13,7 veces más rápido. Con clases de bytes, la tabla completa de firmas de bots ocupa 42.244 bytes[6].

□

Teorema 2 (la cota del intervalo de GCRA). Fijemos un límite de N peticiones por ventana W, con intervalo de emisión $T = \max(1, \lceil W/N \rceil)$ y tolerancia de ráfaga $\tau = (N-1)T$. Un cliente guarda un único tiempo teórico de llegada, TAT. Una llegada en el instante t es conforme si y solo si $\mathrm{TAT} \le t + \tau$, tras lo cual $\mathrm{TAT} \leftarrow \max(\mathrm{TAT}, t) + T$. Entonces, todo intervalo de longitud L que empiece con una llegada conforme contiene como máximo $N + \lfloor L/T \rfloor$ llegadas conformes.
Esbozo de la demostración. Numeremos las llegadas conformes $1, ..., k$ en los instantes $t_1 \le ... \le t_k$. Cada llegada conforme eleva el TAT en al menos T, así que tras la j-ésima se cumple $\mathrm{TAT}_j \ge t_1 + jT$. La llegada k fue conforme, así que $\mathrm{TAT}_{k-1} \le t_k + \tau \le t_1 + L + \tau$. Juntas, estas desigualdades dan $(k-1)T \le L + (N-1)T$, de modo que $k \le N + L/T$, y k es un número entero.
Implementación. El estado de un cliente es una celda de 16 bytes, sin historial por petición ni efecto de frontera de ventana fija. Las 16 particiones comparten `--rate-table-cells` celdas (262.144 por defecto, 4 MiB), colocadas mediante un hash con clave, y una celda agotada se recupera en su sitio sin un hilo de barrido. Una comprobación sobre un cliente cuyo hash ya ha calculado el daemon tarda 6,2 ns[7].

□

Teorema 3 (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 que se haya aceptado una prueba válida para su desafío, y sale cuando el desafío caduca. Si r acota la tasa de soluciones distintas aceptadas durante la vida de un desafío, entonces $|\ S\ | \le r \cdot \mathrm{TTL}$. Un desafío que se obtuvo pero no se resolvió no ocupa memoria.
Esbozo de la demostración. 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 mediante una función pseudoaleatoria y el hash de la regla que lo exigió; son 36 bytes seguidos de una etiqueta BLAKE3 con clave de 16 bytes, 70 caracteres en base64 apto para URL. Emitirlo no escribe nada. La verificación comprueba la etiqueta en tiempo constante, después la antigüedad y el vínculo con el cliente, después la prueba, y solo entonces inserta la etiqueta en el conjunto de desafíos gastados. Si se modela BLAKE3 en modo con clave como una función pseudoaleatoria, un adversario que hace q consultas de verificación falsifica una etiqueta con probabilidad como máximo $q \cdot 2^{-128}$ más la ventaja frente a la PRF.
Implementación. El conjunto de desafíos gastados consta de 16 particiones de direccionamiento abierto Robin Hood. Una entrada caducada solo puede sobrescribirse en su sitio con una clave que esté al menos igual de lejos de su posición de origen; de lo contrario, una búsqueda posterior podría detenerse antes de tiempo y no encontrar una etiqueta viva. Con esa regla no hace falta ningún hilo de barrido, y una inserción rechazada nunca expulsa una etiqueta viva. Un gasto y una búsqueda juntos tardan 21,0 ns[8].

□

4.1 El cortafuegos semántico

Shield inspecciona la ruta, la consulta, el User-Agent, las cabeceras que no son estructurales y los primeros 8 KiB del cuerpo. Su primera versión nos dio una lección. Comparaba una lista plana de subcadenas con cada cabecera, la lista contenía `/*`, y todos los navegadores envían `Accept: text/html,...,*/*;q=0.8`. Todos los navegadores reales se clasificaban como inyección SQL. Desde entonces, un detector solo puede dispararse ante una *estructura*, y una prueba hace pasar por el motor una petición completa de Chrome y exige que salga limpia[9].
La inspección tiene cuatro etapas, cada una lineal en el campo.

[6] `benchmarks/results/primitives-latest.md` (55,7 y 763,0 ns); el tamaño de la tabla, de `primitives-latest.json` tal como figura en la parte VIII del libro, "Consumo de recursos"; SID 0006, parte F, y su nota de revisión sobre las clases de bytes.
[7] SID 0006, parte D (definición 22, teorema 23); SID 0004, notas de revisión (ubicación con clave, `--rate-table-cells`); `benchmarks/results/primitives-latest.md` (`gcra_check_hashed`).
[8] Libro, parte III (lema 3 y lema 4, formatos de transmisión); libro, parte V, "El conjunto Robin Hood de desafíos gastados"; SID 0006, parte E (lema 28); `benchmarks/results/primitives-latest.md` (`robin_hood_spend_and_lookup`).
[9] SID 0004, "Revision note (2026-09-07)" y "Verification"; libro, parte VI, "Una lección sobre falsos positivos".

1. *Una pasada previa por clases de bytes.* Una pasada con una tabla de 256 entradas registra qué clases de bytes aparecen: comillas, `=`, `<`, separadores de shell, `%`, NUL y las señales de canonicalización. Todos los detectores posteriores consultan estas clases en lugar de volver a recorrer el campo, de modo que un campo sin comillas ni signo igual nunca entra en el analizador léxico SQL.
2. *Un autómata de firmas con etiquetas.* Todas las firmas fuertes de todas las categorías residen en un único autómata (teorema 1). Una coincidencia es una denegación terminal con el nombre `waf:sqli`, `waf:xss`, `waf:path-traversal` o `waf:rce`.
3. *Analizadores léxicos estructurales.* El analizador SQL busca cada palabra corta en una tabla de palabras clave y comprueba las tautologías `literal = literal` tras `or` y `and`; en otro caso, una comilla es obligatoria y la evidencia debe alcanzar una pequeña puntuación. La frase "it's a group order from the shop" puntúa 3 y pasa. El cross-site scripting exige una etiqueta capaz de alojar un script junto con un atributo de manejador de eventos en posición de atributo, y la inyección de comandos exige un separador seguido de un nombre de comando que no sea una asignación, así que `a=1&id=5` es un parámetro, no un comando.
4. *Canonicalización.* Un campo con escapes de porcentaje, aperturas de comentario, escapes JSON o HTML, o espacios en blanco colapsables se decodifica y se vuelve a inspeccionar, de modo que `1%27/**/UnIoN/**/SeLeCt` y `%252e%252e/` se reducen a las firmas originales.

Un cuerpo de 8 KiB se recorre en 12.784,3 ns, unos 1,56 ns por byte. Un operador puede auditar una categoría en lugar de aplicarla, de modo que sus hallazgos se registren pero no se actúe sobre ellos. Entonces un campo debe juzgarse por categoría, para que un hallazgo auditado no pueda ocultar uno aplicado. En lugar de recorrer el campo una vez por categoría, Sibuna lo recorre una sola vez para obtener el conjunto de categorías cuyas firmas aparecen y después busca, solo entre ellas, las categorías deseadas; el mismo cuerpo de 8 KiB con la inyección SQL en modo de auditoría tarda 15.485,4 ns[10]. Shield sigue siendo un inspector heurístico acotado, no un analizador completo de SQL ni de HTML, y los bytes que quedan más allá del límite de inspección no se inspeccionan.

4.2 Huellas TLS desde el proxy de entrada

Un User-Agent es una cadena que cualquier cliente puede escribir a su gusto. Una huella JA4 es más difícil de falsificar. Resume cómo abre un cliente una conexión TLS: protocolo, versión, conjuntos de cifrado, extensiones y el primer valor ALPN. La biblioteca TLS del cliente fija todo esto antes de que se envíe el primer byte HTTP. Solo el componente que termina TLS ve el handshake, así que solo él puede calcular JA4. Sibuna no termina TLS entrante, así que no calcula JA4. Lee la huella que un proxy de entrada de confianza calculó y reenvió en una cabecera de la petición, cuyo nombre se indica con `--ja4-header`[11].

- *Confianza.* La cabecera solo se lee cuando se confía en las cabeceras reenviadas, según la misma decisión que rige `X-Forwarded-For`. Si procede de un par que no es de confianza, se ignora, y en modo proxy inverso se eliminan todas sus copias antes de que la petición llegue al origen.
- *Validación.* Un valor solo se acepta en la forma JA4 de 36 bytes, y solo cuando hay exactamente un campo con ese nombre. Un valor malformado o repetido se trata como ausente y se contabiliza.
- *Política.* Una regla coincide con una huella completa, o con un prefijo seguido de `*`; una petición sin huella nunca coincide con una regla así. La comparación es byte a byte y, una vez en régimen estable, el camino de la petición sigue sin hacer ninguna asignación en el heap.
- *Registros.* Los incidentes guardan la huella de su petición, y la consola la muestra en la evidencia del incidente y permite filtrar eventos por ella. Los bloqueos y la reputación siguen indexados por dirección, porque muchos clientes legítimos comparten la misma biblioteca TLS.

4.3 El hash con clave y la revisión de seguridad

Una revisión adversarial del camino de la petición, en octubre de 2026, encontró debilidades en mecanismos que dábamos por resueltos. Corregimos cada una y dejamos constancia de la corrección en el documento de diseño del mecanismo[12].

- *Colocación predecible en las tablas.* El limitador y la tabla de bloqueos colocaban las claves con un hash bajo semillas escritas en el código fuente, así que cualquiera podía calcular direcciones que compartieran la ventana de sondeo de una víctima. Ahora la colocación usa SipHash-1-3 con una clave obtenida de la aleatoriedad del sistema operativo, y cada tabla deriva su propia clave. El daemon calcula el hash del cliente una vez por petición (8,7 ns) y lo pasa a todas las tablas; el camino de escritura de una petición bajó de 43 a 26 ns.
- *Expulsión de bloqueos.* Un bloqueo vigente nunca se expulsa. Cuando todas las posiciones que puede ocupar una clave contienen un bloqueo vigente, el bloqueo nuevo se rechaza y se contabiliza, de modo que un cliente bloqueado no puede expulsar su propio bloqueo.

[10] `benchmarks/results/primitives-latest.md` (`8kb_body_semantic_scan`, `8kb_body_sqli_audit_other_enforce`); SID 0004, nota de revisión "inspection modes"; libro, parte VIII, "Inspección WAF".
[11] SID 0014, "Specification" y "Security and operational considerations"; libro, parte IX, "Huellas JA4 desde el proxy de entrada"; `.github/release-notes.txt`.
[12] SID 0002, notas de revisión (identidad reenviada; valores de las cookies); SID 0004, notas de revisión (ubicación con clave, tabla de bloqueos, costes medidos, tablas puestas a cero); SID 0006, notas de revisión (capacidad del conjunto de desafíos gastados, entrada del controlador adaptativo); SID 0010, historial de revisiones (revisión de seguridad, 2026-10-15); `benchmarks/results/primitives-latest.md` (`gcra_check` y `gcra_check_hashed`).

- *Identidad reenviada.* El cliente puede escribir la entrada situada más a la izquierda de `X-Forwarded-For`. Sibuna toma la entrada que está `--trusted-hops` posiciones desde la derecha, la que añadió el proxy de confianza más cercano.
- *Cookie tossing.* Una petición puede llevar varios valores de la cookie de sesión; el daemon prueba hasta ocho, de modo que una cookie plantada por un dominio hermano no puede mantener a un visitante en un bucle de desafíos.
- *Capacidad del conjunto de desafíos gastados.* Una tabla de 65.536 etiquetas se agotaba con unas 220 soluciones verificadas por segundo dentro de una vida de desafío de cinco minutos, y a partir de ahí se rechazaban soluciones honestas. Ahora la capacidad es configurable, con 1.048.576 etiquetas (24 MiB) por defecto.
- *Dificultad adaptativa.* El controlador que eleva el trabajo bajo carga cuenta ahora las soluciones verificadas en lugar de los desafíos emitidos, que a quien inunda no le cuestan nada.

La colocación con clave tiene un precio, y lo pagamos una vez por petición, no una vez por tabla. Una comprobación del limitador que calcula ella misma el hash de los bytes del cliente tarda 12,8 ns; la comprobación sobre un cliente con el hash ya calculado, el camino que usa el daemon, tarda 6,2 ns. Como el limitador y el conjunto de desafíos gastados se asignan puestos a cero, las tablas inactivas no son residentes, así que su capacidad configurada no cuesta memoria hasta que se usa.

5 Pruebas de trabajo: búsqueda paralela y trabajo secuencial

5.1 El nivel Hashcash

El primer nivel es Hashcash con granularidad de bit. Para una cadena de desafío C y una dificultad b, el cliente encuentra un nonce N tal que SHA-256 ($C \parallel$ `":"` $\parallel$ dec (N)) tenga b bits iniciales a cero. La representación decimal del nonce mantiene idéntico el formato de transmisión entre el solucionador WebAssembly, la alternativa en JavaScript y el verificador nativo. Para resúmenes ideales, el número de intentos K sigue una distribución geométrica con $p = 2^{-b}$:

$$P(K > k) = (1-p)^k, \qquad E[K] = 2^b, \qquad \text{median } (K) \approx 2^b \ln 2.$$

La búsqueda tiene dos límites. Su varianza es grande, así que una resolución lenta no indica que el worker esté averiado. Y los intentos independientes se ejecutan en paralelo, así que un remitente con más núcleos termina antes, aunque el trabajo total esperado sea el mismo[13].

5.2 El nivel de trabajo secuencial

El segundo nivel sigue la prueba de trabajo secuencial de Cohen y Pietrzak (EUROCRYPT 2018). Sus etiquetas forman un árbol binario completo de profundidad n. Un nodo interno calcula el hash de sus dos hijos. Una hoja calcula el hash de los *hermanos izquierdos de sus antepasados* allí donde su camino desde la raíz giró a la derecha. Esas aristas hacia los hermanos izquierdos hacen secuencial el cálculo: una hoja no puede etiquetarse hasta que esté completo cada subárbol a su izquierda, así que la etiqueta de la raíz depende, en orden, de las $2^{n+1} - 1$ etiquetas. Tras calcular la raíz φ, el probador deriva t hojas mediante Fiat-Shamir y, para cada una, envía la etiqueta de la hoja y las n etiquetas hermanas a lo largo de su camino. El verificador recalcula cada hoja abierta y sube calculando hashes hasta φ, con un coste de $O(tn)$ cálculos de etiquetas en un espacio de trabajo fijo.

Los parámetros se eligen para que las cifras sigan siendo pequeñas. Los bits de trabajo se convierten en profundidad mediante $n = b - 3$, limitada al intervalo $[4, 24]$. Una prueba tiene $32(1 + t(n+1))$ bytes, que con profundidad 13 y 16 aperturas son $32(1 + 16 \times 14) = 7200$ bytes. El probador guarda solo los niveles superiores del árbol y recalcula el subárbol que debe abrir, así que todo su espacio de trabajo, prueba incluida, se mantiene por debajo de 100 KB; una pestaña del navegador y las pruebas nativas usan el mismo probador[14]. Blocki, Lee y Zhou (ITC 2021) demostraron que la construcción es sólida frente a probadores cuánticos en el modelo de oráculo aleatorio cuántico.

Conviene ser prudentes con lo que esto demuestra. Los resultados de seguridad valen bajo los supuestos de oráculo aleatorio de los artículos, para las construcciones y las cotas que enuncian. No son una auditoría independiente de nuestro protocolo SHA-256 concreto, de su correspondencia de parámetros ni de sus tiempos en el navegador. La cota elemental de muestreo $(1-\alpha)^t$, la probabilidad de que t aperturas independientes esquiven todas una fracción mala fija α, explica por qué ayudan más aperturas; no es por sí misma una demostración frente a un probador que prueba muchas raíces hasta dar con una favorable. No afirmamos incondicionalmente que hacer trampa nunca sea más barato.

5.3 El viaje de ida y vuelta y la sesión

Una navegación sin sesión válida recibe la página intermedia, que 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 y válidos durante la vida de un desafío. La página obtiene su desafío con el ticket, de modo que el trabajo lo fija la decisión que lo exigió. Un Web Worker resuelve el desafío con el probador WebAssembly, o con una alternativa en JavaScript donde WebAssembly no está disponible, y envía la respuesta. Si tiene éxito, el servidor establece una cookie de sesión de 75 caracteres, vinculada mediante una huella con clave a la dirección y al User-Agent del cliente, y la página se recarga.

[13] Libro, parte III, "Hashcash: las matemáticas del primer nivel"; libro, parte I, "La búsqueda de hashes como variable aleatoria".
[14] Libro, parte III, "Prueba de trabajo secuencial: el segundo nivel", "Memoria del probador" y "Los parámetros no son tiempos"; SID 0006, parte B, y su revisión de la implementación del 2026-09-07.

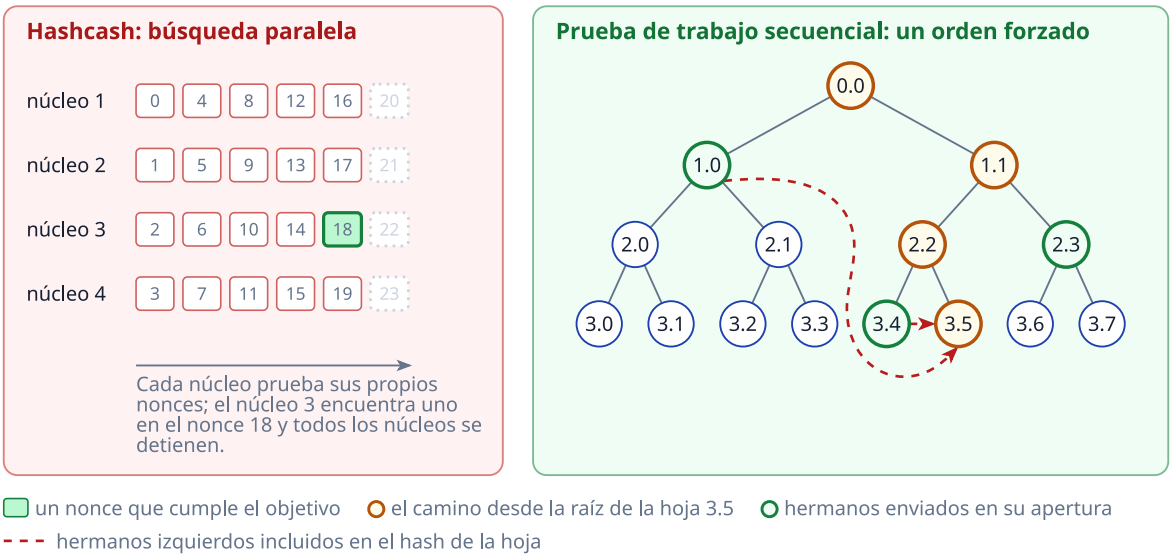


Fig. 3 Búsqueda paralela frente a trabajo secuencial. A la izquierda, Hashcash: intentos de nonce independientes, así que M núcleos terminan unas M veces antes. A la derecha, el árbol de Cohen-Pietrzak con profundidad 3: las aristas discontinuas hacia los hermanos izquierdos hacen que la hoja 3.5 espere a los subárboles de su izquierda, y una apertura de la hoja 3.5 envía su etiqueta y los tres hermanos verdes (según la parte III del libro).

El probador del navegador se compila a partir del mismo código fuente que el verificador del servidor, así que los dos no pueden divergir[15].

5.4 Lo que cuesta la verificación

El coste de ambos niveles lo determina SHA-256, y Sibuna selecciona en tiempo de ejecución las instrucciones SHA del procesador cuando existen, así que un único binario es rápido en todas las máquinas que las tienen. La verificación Hashcash con 16 bits tarda 70,2 ns, y verificar una prueba de profundidad 13 con 16 aperturas tarda 22.272,4 ns. El probador del navegador es un módulo WebAssembly de 14.744 bytes. En la comprobación de extremo a extremo en el navegador, Chromium sin interfaz gráfica resolvió el desafío de la página intermedia en 29 a 34 ms con WebAssembly y en 84 a 91 ms con la alternativa en JavaScript[16]. El conjunto de primitivas solo mide la verificación; un tiempo nativo no representa el tiempo de resolución de un navegador.

Las sesiones usan un token simétrico. El verificador es el emisor, o un clúster que comparte una misma semilla, así que una función pseudoaleatoria con clave aporta exactamente la autenticación necesaria. Un token BLAKE3 con clave se verifica en 155,8 ns y un token Ed25519 en 40.587,5 ns, de modo que verificar una firma cuesta aproximadamente lo mismo que 260 verificaciones MAC. Hay una segunda razón para esta elección: el algoritmo de Shor rompe los logaritmos discretos en los que se basa Ed25519, mientras que frente a un hash con clave la búsqueda cuántica da como mucho una aceleración cuadrática. Ed25519 sigue siendo una opción para los despliegues que necesitan verificadores incapaces de emitir tokens.

6 Estado compartido sin un protocolo de consenso propio

> **Lamport sobre los invariantes distribuidos: seguridad, vivacidad y quién es dueño del registro**
>
> En 1987, Leslie Lamport describió un sistema distribuido como aquel en el que el fallo de una máquina cuya existencia ni siquiera conocías puede dejar inservible la tuya. Leído como consejo, nos pide contar las máquinas de las que dependemos y enunciar qué debe cumplirse cuando cualquiera de ellas falla. Bastan dos tipos de enunciado. *Seguridad*: nunca ocurre nada malo; por ejemplo, un nodo nunca actúa según una política que no haya escrito ninguna transacción confirmada. *Vivacidad*: algo bueno acaba ocurriendo; por ejemplo, un nodo que puede llegar a la base de datos acaba sirviendo la política más reciente. Un protocolo de consenso es una manera elegante de obtener ambas. También es algo grande que mantener, y la pregunta para un cortafuegos no es si el consenso es bueno, sino quién debe mantener el registro.

[15] Libro, parte IV, "El intercambio del desafío" y "La cookie de sesión"; SID 0010, "The browser solver".

[16] `benchmarks/results/primitives-latest.md`; el tamaño del módulo WebAssembly, de `primitives-latest.json` tal como figura en la parte VIII del libro, "Consumo de recursos"; los tiempos de resolución en el navegador, del criterio 3 del SID 0010 (Chromium 155, `cargo xtask browser`); la detección en tiempo de ejecución de las instrucciones SHA, de la nota de revisión del SID 0006 (partes A a C).

6.1 Por qué SQL, y por qué no un protocolo de consenso propio

Sibuna almacena poco: unos cientos de filas de políticas, una tabla de reputación e incidentes forenses que los operadores también exportan a otros sistemas. Ese estado no necesita un motor replicado a medida. Necesita un almacén SQL fiable, local para un nodo y compartido para varios. Por eso un nodo único guarda su estado en un archivo SQLite integrado en su directorio de datos, en modo WAL, y un clúster apunta todos sus nodos a una misma base de datos PostgreSQL[17].

Decidimos no ejecutar un protocolo de consenso dentro del cortafuegos. Un motor de consenso es un sistema grande que mantener junto a un cortafuegos, con sus propios formatos y sus propios modos de fallo, y añade trabajo a la parada y a cada actualización. La disponibilidad y la replicación del estado de un clúster quedan, en cambio, a cargo de PostgreSQL, mediante su replicación en streaming y sus herramientas de conmutación por error, o de un servicio gestionado. Creemos que es la opción más humilde: la replicación de PostgreSQL se ha puesto a prueba en muchísimos más despliegues de los que podría tener la nuestra, y la tasa de escritura de Sibuna (incidentes por lotes y cambios de política poco frecuentes) es lo bastante baja para que un único primario no sea un cuello de botella.

Además, el almacenamiento sigue siendo opcional. Sin `--data-dir` ni `--database-url`, el daemon es un cortafuegos en memoria cuyas políticas proceden de un archivo.

6.2 Un único hilo de almacenamiento

Un único hilo de almacenamiento posee las conexiones a la base de datos; nada más toca SQL. Los hilos de petición nunca lo esperan. Insertan registros de incidentes en un anillo acotado de 512 posiciones, que descarta y contabiliza un registro cuando está lleno, y leen el motor publicado. Cada 500 ms por defecto, el hilo de almacenamiento ejecuta un ciclo: escribe hasta 32 incidentes en cola en una transacción, lee la revisión de las políticas y, si ha cambiado, reconstruye el motor y lo publica. Una reconstrucción lee la revisión antes y después de cargar; si un escritor confirmó entre medias, el candidato se descarta y el motor antiguo sigue activo. Los operadores pueden escribir directamente en las tablas de políticas: los disparadores avanzan la revisión en la transacción del escritor, y con PostgreSQL un `NOTIFY` permite a los nodos ejecutar un ciclo de inmediato, mientras que el sondeo sigue siendo la garantía[18].

Cada nodo de un clúster tiene un identificador `--cluster-node` distinto entre 1 y 8.388.607. Las claves de los desafíos están vinculadas al nodo emisor, así que una solución debe volver al nodo que emitió su desafío, mientras que los tokens de sesión y los tickets de requisito son válidos en todos los nodos que comparten la semilla. Los identificadores de incidente llevan el nodo en sus bits superiores, y cada nodo mantiene un recibo duradero que avanza en la misma transacción que su lote, de modo que un lote repetido tras un fallo ambiguo no escribe nada dos veces.

> **Invariantes de la capa de almacenamiento (SID 0011).** **S1 (un único propietario).** Todo el SQL se ejecuta en el hilo de almacenamiento; ningún hilo de petición espera nunca a la base de datos. **S2 (motores completos).** Todo motor que lee una petición se construyó por completo antes de publicarse, a partir de una única revisión coherente de las tablas de políticas. **L1 (publicación a la larga).** Un cambio confirmado en las tablas de políticas llega en su siguiente ciclo a todos los nodos que pueden llegar a la base de datos. **F1 (pérdida controlada).** Cuando la base de datos no está disponible, el último motor publicado sigue sirviendo, los incidentes se encolan en el anillo acotado y se descartan con un contador cuando se llena, y el hilo de almacenamiento reintenta con esperas crecientes.

6.3 Un esquema tipado

El esquema expresa en SQL lo mismo que expresan los tipos del programa, de modo que la base de datos rechaza una fila que la capa de almacenamiento no podría representar. Las tablas SQLite son `STRICT`; una columna de tipo enumerado tiene un `CHECK` que lista exactamente los nombres que escribe el programa; un entero acotado tiene una comprobación de rango; y una columna es `NOT NULL` exactamente cuando el valor nunca falta. Cada sentencia vincula sus valores como parámetros, así que nunca se construye SQL entrecomillando valores. El almacenamiento de la consola sigue las mismas reglas: una migración base de 33 tablas tipadas y ningún disparador. Cada invariante de la consola, como "todo cambio en un token escribe una fila de auditoría", lo impone exactamente una función que ejecuta sus sentencias en una transacción, igual en SQLite que en PostgreSQL[19].

Las conexiones a PostgreSQL se protegen como indica la URL, con `sslmode` y `sslrootcert` de libpq, cuando el daemon se compila con la feature `postgres-tls`. Una compilación sin esa feature rechaza `require`, `verify-ca` y `verify-full` al arrancar; nunca recurre a texto plano. Los clústeres de producción usan `sslmode=verify-full`.

Los incidentes que se parecen se agrupan en campañas. Cada incidente lleva un embedding de 64 dimensiones almacenado en 256 bytes. El hilo de almacenamiento mantiene en memoria los 16.384 embeddings más recientes de todos los nodos (4 MiB) y asigna un incidente a la campaña de su vecino más cercano cuando la distancia coseno es como máximo 0,35. No agrupar con incidentes más antiguos es una omisión deliberada: las campañas son recientes por naturaleza, y la cota mantiene fijos la memoria y el trabajo.

[17] SID 0011, "Abstract", "Motivation and scope" y "Why PostgreSQL for clusters"; el SID 0005 describe el diseño replicado anterior.
[18] SID 0011, "The storage thread", "Policy revision" y "Incidents and campaigns".
[19] SID 0011, "Schema and migrations" y su historial de revisiones (seguridad del transporte); SID 0013, "Abstract".

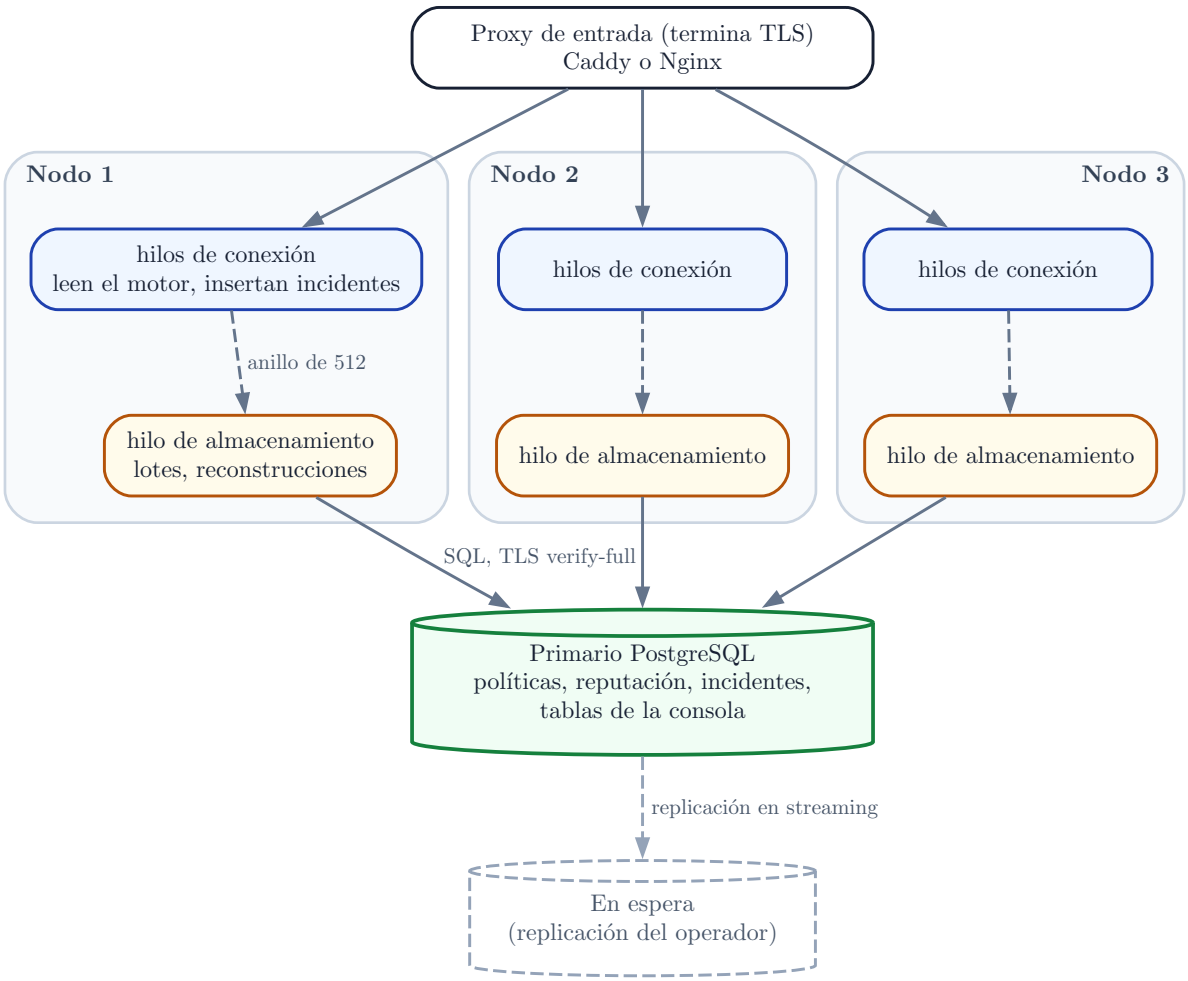


Fig. 4 Un clúster de Sibuna. Cada nodo mantiene en memoria su camino de la petición y entrega todo el SQL a un único hilo de almacenamiento; los nodos comparten una misma base de datos PostgreSQL, cuya replicación y conmutación por error corresponden a PostgreSQL (SID 0011). Un nodo único usa en su lugar un archivo SQLite integrado.

6.4 Lo que este diseño no hace

Debemos ser claros sobre los límites de esta decisión. Sibuna no contiene ningún protocolo de consenso propio, y la consistencia de un clúster es la de PostgreSQL: un nodo que pierde la base de datos conserva su última política hasta que se reconecta. Una sesión emitida en un nodo la aceptan todos los nodos, y una solución gastada en un nodo se rechaza si se reutiliza en otro, porque los nodos comparten una misma semilla y las claves de los desafíos están vinculadas al nodo emisor; ninguna de las dos propiedades necesita la base de datos. El SID 0011 está en estado Committed: una prueba funcional con tres nodos sobre una misma base de datos PostgreSQL muestra que una política insertada en un nodo la aplican los tres en un ciclo, que un bloqueo del honeypot en un nodo llega a los demás y que los identificadores de incidente nunca colisionan. Ningún programa de medición ejecuta todavía varios nodos sobre una misma base de datos PostgreSQL bajo carga, así que este documento no presenta mediciones de clúster. Los ejecutables publicados se compilan sin soporte de PostgreSQL; un clúster compila Sibuna desde el código fuente con la feature `postgres-tls`[20].

7 La consola

Sibuna incluye una consola de gestión opcional: paneles autenticados con tráfico en vivo, un camino de investigación desde una denegación hasta su evidencia registrada, edición de políticas, grupos de direcciones, enriquecimiento por país a partir de datos GeoIP abiertos, usuarios con contraseñas Argon2id y un segundo factor TOTP, un registro de auditoría, webhooks y la gestión de un clúster. Se compila en el mismo binario, y su interfaz se ejecuta en el navegador como un módulo WebAssembly[21].

[20] SID 0011, "Security and operational considerations" y "Acceptance criteria and implementation status" (criterio 2, `crates/storage/tests/multi_node.rs`); `.github/release-notes.txt`, "Known limitations"; SID 0002, "Authentication and work" (claves de desafío vinculadas al nodo); libro, parte VIII, "Lo que no se mide".

[21] SID 0007, "Abstract" y sus notas de revisión (admisión en el puerto de escucha, módulo de la interfaz, GeoIP y contraseñas); SID 0010, historial de revisiones (migración de la consola, TOTP); SID 0012; SID 0013. La medición del aislamiento está en el historial de revisiones del SID 0010 (migración de la consola); la parte VIII del libro, "Sobrecarga de la consola", señala que el registro no nombra ningún archivo de resultados para ella.

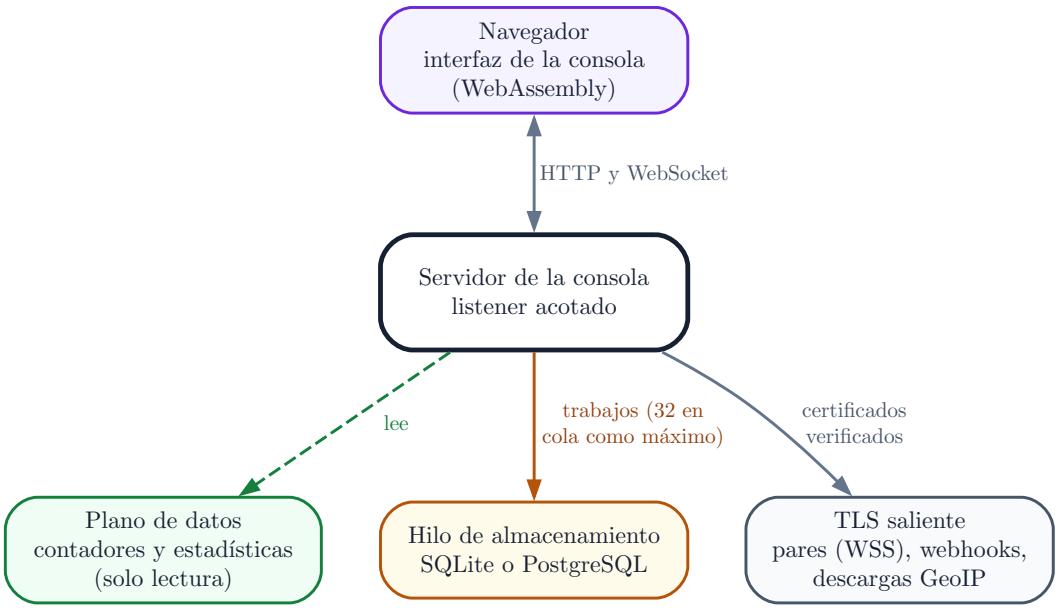


Fig. 5 La consola. El navegador ejecuta la interfaz como un módulo WebAssembly; el servidor de la consola lee los contadores del plano de datos y entrega su SQL al hilo de almacenamiento; las conexiones salientes hacia pares, webhooks y proveedores de GeoIP usan TLS con certificados verificados (SID 0007, 0012 y 0013).

La restricción que define la consola es su aislamiento del camino de la petición. Puede leer los contadores del plano de datos y su base de datos, y escribir en la base de datos, pero nunca entra en un hilo de petición, nunca asigna memoria en uno y nunca retiene un cerrojo que necesite un hilo de conexión. Su puerto de escucha admite como máximo 96 conexiones, de las que 16 se reservan para HTTP, 64 pueden ser flujos de navegador y 16, flujos de pares. En 1.0.0--beta.3, el módulo completo de la interfaz mide 724.330 bytes (272.568 bytes comprimido con gzip). Está por debajo del techo de 768 KiB, que impone la compilación, y por encima del umbral de revisión de 640 KiB, que el SID 0007 registra para su revisión[22].

La consola establece conexiones TLS salientes para los flujos de larga duración con pares, los webhooks y las descargas GeoIP. Los certificados se verifican con las raíces del sistema o con el archivo de CA del operador, la verificación del nombre de host no puede desactivarse, el HTTP sin cifrar solo se permite hacia direcciones de loopback literales, y un error de certificado nunca recurre a texto plano. Sibuna sigue sin terminar TLS entrante; esa sigue siendo tarea del proxy de entrada.

El SID 0007 fija el objetivo de rendimiento de la consola: activarla puede costar como máximo un 1% del rendimiento del plano de datos y como máximo un 10% de la latencia p99. Existe una medición: con la consola compilada pero desactivada, solo con almacenamiento, o activada y en reposo, las 16 cargas de trabajo del producto completo se mantuvieron a menos de un 1% de un daemon compilado sin ella, lo que está dentro del ruido. El criterio formal, con ocho paneles activos bajo carga, no se ha ejecutado, así que no afirmamos que se cumpla el objetivo. Una ejecución funcional con ocho paneles en vivo fue satisfactoria.

Tabla 4 Ocho paneles de tráfico en un nodo SQLite con su consola, en Chromium 155 sin interfaz gráfica (`benchmarks/results/console-8-dashboards-2026-10-15.md`, registrado el 2026-10-15). Resultado: PASS

Medida	Valor
Peticiones del plano de datos	6.000 en 30,0 s (200 por segundo, 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)
Memoria residente del daemon al arrancar, con 8 paneles y después del tráfico	36.152 KiB; 38.028 KiB; 41.496 KiB
Paneles	3 sesiones de administrador y 5 pantallas murales; cada una recibió entre 134 y 139 deltas de `stats` y terminó con los mismos totales, sin excepciones, errores de consola ni infracciones de la Content-Security-Policy

[22]SID 0007, historial de revisiones (migración a Rust, consolidación; integración en el daemon), que registra 719.406 bytes antes de los campos JA4 del SID 0014. El tamaño para 1.0.0-beta.3 es el del módulo compilado tal como lo compila el script de compilación del daemon (`cargo rustc --lib -p sibuna-console-ui --target wasm32-unknown-unknown --profile wasm -- -C link-arg=--compress-relocations`), comprimido con `gzip -9`.

8 Evaluación empírica

8.1 Método

Tres programas de medición, que se ejecutan con `cargo xtask bench`, miden Sibuna. La batería de *primitivas* mide las operaciones del camino de la petición en siete lotes cronometrados tras un calentamiento sin cronometrar, reconstruye el estado mutable fuera del cronómetro y da para cada operación el lote mediano junto con el más rápido y el más lento. El programa de *producto completo* somete el daemon a `wrk` (2 hilos, 64 conexiones, 5 segundos, 3 repeticiones) en los modos forward-auth y proxy inverso, con los modos de protección Gate y Shield, sobre cuatro cargas de trabajo: `admitted` (una sesión válida), `challenged` (sin sesión), `allowed_static` (`/robots.txt`) y `attack` (una sesión más `q=' OR 1=1--`, que solo Shield rechaza). El programa de *latencia* usa un generador en lazo abierto, que envía a una tasa fija haga lo que haga el daemon y mide cada petición desde el momento en que debía enviarse, de modo que la espera en las colas del daemon se imputa al daemon[23].

Las ejecuciones se registraron el 15 de octubre de 2026 en un único host: un AMD Ryzen 9 5950X (16 núcleos, 32 hilos) con Linux 7.0.0-28. El daemon se ejecutó en las CPU 0 a 3 con cuatro workers; el generador de carga y un origen simulado, en las CPU 8 a 15 (el otro CCD), y la batería de primitivas, en la CPU 4. El daemon era la compilación release por defecto, con SQLite y la consola incluidos, y se ejecutó en memoria, sin almacenamiento.

> **Alcance de las mediciones.** Cada ejecución usa un único host a través de loopback: sin TLS, sin red de área extensa y sin navegadores reales. Los programas de medición no cuentan asignaciones; que el camino de la petición no asigna memoria lo comprueban en su lugar las pruebas de régimen estable. Ejecutan el binario de la versión sin modificar, identificado por su SHA-256. Una clasificación obtenida en este host no vale para otro.

8.2 Latencias de primitivas

Tabla 5 Tiempos de las primitivas de Sibuna en nanosegundos por operación en un núcleo de un Ryzen 9 5950X: la mediana de siete lotes, con el lote más rápido y el más lento (`benchmarks/results/primitives-latest.md`, generado el 2026-10-15T22:34:59Z)

Subsistema	Carga de trabajo	Mediana	Mín-máx
Prueba de trabajo	Verificación Hashcash, 16 bits	70,2	69,9-71,0
Prueba de trabajo	Verificación PoSW, profundidad 13, 16 aperturas	22.272,4	22.257,0-22.316,4
Detector de bots	Aho-Corasick, 40 firmas	55,7	55,3-57,2
Detector de bots	Búsqueda secuencial de subcadenas, 40 firmas (como referencia)	763,0	762,2-763,5
Clasificador de IP	Búsqueda CIDR IPv4	32,0	31,8-32,2
Clasificador de IP	Búsqueda CIDR IPv6	79,3	78,5-79,6
Token	Token de sesión BLAKE3 con clave	155,8	155,3-156,4
Token	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, cliente en hash	6,2	6,1-6,3
Limitador de tasa	GCRA, cuatro ámbitos de regla, cliente en hash	6,3	6,2-6,4
Limitador de tasa	GCRA, cliente en bytes (hash con clave incluido)	12,8	12,7-12,8
Limitador de tasa	GCRA, cuatro ámbitos de regla, cliente en bytes	13,1	13,0-13,2
Analizador HTTP	Petición y cookie sin copia	132,5	132,3-133,2
Motor de políticas	Clasificación completa, perfil Shield	790,8	787,2-793,1
Motor de políticas	Clasificación, perfil Gate	75,9	75,6-76,1
WAF	Cuerpo de 8 KiB, exploración semántica	12.784,3	12.771,5-12.809,2
WAF	Cuerpo de 8 KiB, inyección SQL en auditoría	15.485,4	15.460,9-15.530,1

Las dispersiones son pequeñas: ningún lote de ninguna fila se aleja más de un 4,2% de la mediana de su fila[24]. El daemon usa las filas con el cliente en hash, porque calcula el hash del cliente una vez por petición; las filas que pasan los bytes en bruto incluyen ese hash con clave en cada comprobación. Los cronómetros abarcan lotes completos, nunca operaciones sueltas, así que la tabla no da percentiles por operación.

[23] `benchmarks/results/README.md`, "Method" y "Provenance and limits" (rustc 1.99.0; el daemon es `cargo build --release -p sibuna`, 49.113.856 bytes); libro, parte VIII, "Metodología".
[24] Libro, parte VIII, "Latencias de primitivas", calculado a partir de `primitives-latest.json`.

8.3 Productos completos

Tabla 6 Sibuna bajo `wrk`: peticiones por segundo, latencia en microsegundos, tiempo de CPU por petición en microsegundos y memoria residente máxima; medianas de tres ejecuciones (`benchmarks/results/product-latest.md`, generado el 2026-10-15T23:15:24Z)

Modo	Modo de protección	Carga de trabajo (estado)	req/s	p50	p99	CPU	KiB máx.
forward-auth	Gate	`admitted` (200)	211.345	151	305	11,84	15.912
forward-auth	Gate	`challenged` (401)	198.315	160	326	13,15	16.176
forward-auth	Gate	`allowed_static` (200)	211.053	152	305	11,73	16.176
forward-auth	Gate	`attack` (200)	211.618	151	305	11,84	16.176
forward-auth	Shield	`admitted` (200)	208.101	152	308	12,87	15.928
forward-auth	Shield	`challenged` (401)	197.532	161	327	13,77	16.192
forward-auth	Shield	`allowed_static` (200)	211.636	151	305	11,78	16.192
forward-auth	Shield	`attack` (403)	210.549	152	307	12,32	16.192
proxy inverso	Gate	`admitted` (200)	142.498	427	768	27,68	15.896
proxy inverso	Gate	`challenged` (200)	199.465	160	324	12,98	16.160
proxy inverso	Gate	`allowed_static` (200)	147.303	411	759	26,88	16.324
proxy inverso	Gate	`attack` (200)	143.900	422	745	27,67	16.348
proxy inverso	Shield	`admitted` (200)	138.965	438	857	28,57	16.064
proxy inverso	Shield	`challenged` (200)	195.188	162	328	13,73	16.328
proxy inverso	Shield	`allowed_static` (200)	145.673	415	746	27,27	16.328
proxy inverso	Shield	`attack` (403)	209.507	151	306	12,17	16.328

En el modo forward-auth, el proxy de entrada pregunta a Sibuna si debe admitir una petición, y no hay origen; todas las cargas de trabajo quedan entre 197.532 y 211.636 peticiones por segundo, con un p99 de 305 a 327 us. Las cuatro CPU del daemon no llegan a saturarse en este modo (están ocupadas entre 2,47 y 2,72), lo que señala al generador de carga como el techo. En el modo proxy inverso, las filas de peticiones admitidas y estáticas incluyen el origen simulado y un salto loopback más, y las CPU del daemon se saturan (de 3,95 a 3,98 de cuatro ocupadas); las peticiones que Sibuna responde por sí mismo (los desafíos y el rechazo del ataque por parte de Shield) siguen en unas 195.000 a 210.000 por segundo. La memoria residente máxima se mantiene entre 15.896 y 16.348 KiB en todas las cargas de trabajo[25].

Unas pocas respuestas de cada dos millones, en tres filas de proxy inverso, fueron `503`: `wrk` volvía a conectarse antes de que una conexión que se estaba cerrando hubiera liberado su plaza bajo el límite de conexiones por dirección. Ahora el daemon libera la plaza antes de escribir la respuesta de cierre; estos archivos son anteriores a esa corrección.

Un cambio posterior del perfil release elimina la información de depuración del SQLite incluido, unos 13 MB de los 49 MB según la nota del manifiesto de compilación; los archivos de resultados no se han regenerado desde entonces, así que citamos el tamaño medido[26].

Tabla 7 Consumo de recursos del daemon de Sibuna (`product-latest.json` y `primitives-latest.json`, 15 de octubre de 2026, tal como se tabula en la parte VIII del libro)

Medida	Valor
Binario del daemon, compilación por defecto con SQLite y la consola	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
Tablas del motor de políticas	209.052 bytes
Tabla de firmas de bots	42.244 bytes
Tabla de firmas del WAF	164.908 bytes
Probador del navegador (WebAssembly)	14.744 bytes

[25] `benchmarks/results/product-latest.md`; las CPU ocupadas, de `cores_busy_median` en `product-latest.json`; las respuestas `503` y su corrección, de `benchmarks/results/README.md` y de la nota de revisión del SID 0002 (corrección de errores).
[26] Libro, parte VIII, "Consumo de recursos"; el `Cargo.toml` raíz, perfil `[profile.release.package.libsqlite3-sys]`.

Tabla 8 Sibuna, Shield, peticiones admitidas a tasas ofrecidas fijas; latencia en microsegundos y tiempo de CPU por petición, medianas de tres ejecuciones de 10 segundos (`benchmarks/results/latency-latest.md`, generado el 2026-10-15T23:07:14Z)

Modo	req/s ofrecidas	req/s logradas	p50	p90	p99	p99.9	Errores	CPU us
forward-auth	10.000	10.000	30	32	37	58	0	16,92
forward-auth	25.000	25.000	28	31	34	45	0	15,83
forward-auth	50.000	50.000	25	27	30	40	0	13,83
forward-auth	100.000	99.998	24	26	41	62	0	13,48
forward-auth	150.000	149.996	25	36	51	74	0	14,16
proxy inverso	10.000	10.000	56	61	65	76	0	32,25
proxy inverso	25.000	25.000	51	54	58	72	0	28,63
proxy inverso	50.000	50.000	47	50	60	74	0	27,45
proxy inverso	100.000	99.996	55	75	100	150	0	28,42
proxy inverso	150.000	134.344	583.934	1.544.160	1.906.273	1.967.134	4	29,22

8.4 Latencia con la misma carga ofrecida

`wrk` mantiene 64 peticiones en curso, así que la tasa de peticiones depende del daemon, y los percentiles de `wrk` omiten las peticiones que no envió mientras esperaba. El programa en lazo abierto fija en cambio la tasa ofrecida e imputa cada retraso al daemon, que es lo que percibe un visitante.

En el modo forward-auth, el p99.9 se mantiene por debajo de 75 us hasta 150.000 peticiones por segundo. En el modo proxy inverso, con el origen en el camino, el p99 es de 100 us y el p99.9 de 150 us a 100.000 peticiones por segundo. A 150.000 peticiones por segundo en modo proxy inverso, el daemon supera su capacidad: completa 134.344 peticiones por segundo, cerca de las 138.965 que midió `wrk` para la misma carga de trabajo, y la latencia crece hasta los segundos, como debe mostrar una prueba en lazo abierto por encima de la capacidad. Los cuatro errores de esa fila se produjeron durante la sobrecarga[27].

8.5 Qué cambió la medición

El programa de medición de latencia cambió el diseño de los hilos de conexión. En sus primeras ejecuciones, con 64 conexiones que el daemon cerraba tras 4.096 peticiones cada una, el percentil 99,9 con 100.000 peticiones admitidas por segundo era de 0,3 a 2 ms, una cola que `wrk` no había mostrado. 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. Un experimento zanjó el diagnóstico: una compilación que nunca cerraba conexiones, y por tanto no creaba ningún hilo durante la ejecución, dio de 54 a 60 us a la misma tasa. La corrección permite que un hilo que termina su conexión se aparque y tome la siguiente; un hilo que lleva 10 segundos aparcado termina, para que una ráfaga no deje sus pilas residentes. Tras la corrección, el p99.9 a 100.000 peticiones por segundo es de 62 us en modo forward-auth y de 150 us en modo proxy inverso[28].

Una parada rara y periódica desplaza la cola extrema mucho antes de mover el p99. A 100.000 peticiones por segundo sobre 64 conexiones, cada conexión dura $4096/1562.5 \approx 2.6$ s, así que cada segundo empiezan y terminan unos 24 hilos; en una ejecución de un millón de peticiones, el 0,1% superior son solo 1.000 peticiones, y unas pocas peticiones retrasadas en cada rotación de hilos bastan para llenarlo. Para ver un defecto así hacen falta el p99.9 y un cronómetro en cada petición.

8.6 Conclusiones principales

1. *Clasificación por debajo del microsegundo.* En el perfil Gate, el motor de políticas clasifica una petición de navegador en 75,9 ns; con la inspección completa del perfil Shield tarda 790,8 ns.
2. *Verificación simétrica.* Un token de sesión BLAKE3 con clave se verifica en 155,8 ns, frente a los 40.587,5 ns de un token Ed25519: unas 260 veces menos trabajo para la misma integridad dentro de un único dominio de confianza.
3. *Un consumo pequeño y estable.* Unas 200.000 peticiones forward-auth por segundo en cuatro CPU, con una memoria residente máxima cercana a 16 MiB en todas las cargas de trabajo y por debajo de 6 MiB en reposo.
4. *Las colas exigen medir en lazo abierto.* La parada por rotación de hilos era invisible para el programa en lazo cerrado y evidente en el p99.9 con un generador en lazo abierto.

9 Comparación de mercado: Sibuna y sus alternativas

Comparamos Sibuna con las herramientas que un operador probablemente sopesará frente a él. Los datos sobre los demás productos se tomaron de su documentación pública y de sus artefactos de publicación en septiembre de 2026,

[27] `benchmarks/results/latency-latest.md`; las filas a 25.000 peticiones por segundo y las filas de proxy inverso a 150.000 están en ese archivo.
[28] SID 0002, nota de revisión (rendimiento, reutilización de los hilos de conexión); libro, parte VIII, "Qué cambió la medición"; `benchmarks/results/latency-latest.md`.

Tabla 9 Comparación de características (parte II del libro; datos de los demás productos tomados de su documentación pública en septiembre de 2026, y la fila de JA4, en octubre de 2026)

Característica	Sibuna 1.0.0-beta.3	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
Entorno de ejecución	Código nativo, sin recolector de basura; sin dependencias de terceros 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
Admisión por prueba de trabajo	Hashcash (a nivel de bit) y trabajo secuencial de Cohen-Pietrzak, elegidos por regla	Hashcash SHA-256 en nibbles hexadecimales; desafíos sin trabajo	Desafío JavaScript y CAPTCHA; no condicionado al trabajo	Desafío gestionado, desafío JS, Turnstile
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	BLAKE3 con clave, 75 caracteres, vinculado al trabajo pagado; Ed25519 opcional	JSON Web Token firmado con Ed25519; HS512 opcional	Cookie emitida por la pila	Cookie `cf_clearance`
Inspección de la carga útil	Shield: autómata etiquetado más tokenizadores estructurales, primeros 8 KiB del cuerpo	Ninguna	Análisis semántico sobre las categorías de OWASP	Conjuntos de reglas gestionados (Pro y superiores)
Huella TLS (JA4)	Las reglas comparan un valor JA4 que reenvía un proxy de entrada de confianza; Sibuna no lo calcula	No documentado; las políticas pueden hacer referencia a la huella HTTP JA4H, que Anubis calcula	Calculada en su propia terminación TLS; reglas personalizadas y una lista de denegación integrada	JA3 y JA4 en reglas personalizadas y analíticas (Enterprise con Bot Management)
Estado compartido	Una base de datos PostgreSQL para políticas, reputación y análisis forense	Clave de firma compartida; almacén Valkey compartido	Una pila por host	Red anycast global
Terminación TLS	No (proxy de entrada)	No (proxy de entrada)	Sí	Sí

y el libro de Sibuna los recoge con sus fuentes en la parte II[31]. Cada uno de estos productos sirve bien a sus usuarios, y cada uno tomó decisiones de diseño cuyas ventajas nuestro programa de medición no mide. Describimos diferencias de arquitectura, no una clasificación.

9.1 Comparación medida con Anubis

Medimos el binario oficial de la versión v1.27.0 de Anubis junto a Sibuna con el programa de medición de Productos completos: las mismas cuatro cargas de trabajo, las mismas cuatro CPU para todos los hilos de cada producto, la misma carga de `wrk`, un trabajo Hashcash equivalente (8 bits a cero para Sibuna, 2 dígitos hexadecimales a cero para Anubis), una sesión obtenida resolviendo el desafío de cada producto como lo haría un navegador, y `caddy respond` como origen[29]. Aquí Sibuna ejecuta su modo de protección Shield, así que además inspecciona cada carga útil, cosa que Anubis no hace.

> **Cómo leer estas filas.** Estas cifras describen cuatro cargas de trabajo en una máquina, no todos los despliegues, y varias filas no comparan lo mismo. En el modo forward-auth, Anubis responde por sí mismo a `/robots.txt` y a la ruta del ataque; Anubis no inspecciona cargas útiles, así que el rechazo de Shield hace más trabajo; el arranque de un desafío son dos peticiones para Sibuna y una página para Anubis; y el token simétrico explica buena parte de la diferencia en la comprobación de sesión, ya que el propio esquema HS512 de Anubis es un 55% más rápido que su Ed25519 por defecto. Anubis es una herramienta capaz y muy utilizada, y sus mantenedores tomaron otras decisiones por buenas razones. Lo que sí muestra la medición es que un camino de la petición pequeño y acotado puede abaratar la admisión para el defensor.

Anubis gasta 106 us de CPU por petición admitida en modo forward-auth, frente a los 13 de Sibuna; su memoria residente máxima fue de unos 32 MiB en modo forward-auth y creció hasta unos 600 MiB en modo proxy inverso,

[31] Libro, parte II, "Comparación de características" y "Páginas intermedias y redes perimetrales"; la licencia de Sibuna es el campo `license` del workspace en `Cargo.toml`.

[29] `benchmarks/results/comparison-latest.md` y `admission-latest.md`, registrados el 16 de octubre de 2026 con `cargo xtask bench product --anubis` y `cargo xtask bench admission --anubis`; la parte VIII del libro, "Sibuna y Anubis", recoge el método y cada diferencia entre los productos.

Tabla 10 Sibuna (Shield) y Anubis 1.27.0 bajo `wrk`: peticiones por segundo y latencia p99 en microsegundos, medianas de tres ejecuciones de cinco segundos (`benchmarks/results/comparison-latest.md`). El estado es 200 salvo que se indique otro; Shield rechaza el ataque con 403.

Modo	Carga de trabajo	req/s de Sibuna	p99 de Sibuna	req/s de Anubis	p99 de Anubis
forward-auth	`admitted`	208.635	308	36.509	7.477
forward-auth	`challenged`	193.087	342	113.170	3.781
forward-auth	`allowed_static`	212.444	304	89.904	4.547
forward-auth	`attack`	211.347 (403)	306	36.853	7.283
proxy inverso	`admitted`	103.298	3.321	19.398	11.062
proxy inverso	`challenged`	197.327	328	28.842	9.546
proxy inverso	`allowed_static`	106.819	3.316	28.862	30.124
proxy inverso	`attack`	210.483 (403)	306	23.399	14.124

Tabla 11 Operaciones de admisión por segundo en dos CPU, mediana de siete lotes (`benchmarks/results/admission-latest.md`). Anubis se ejecuta una vez con cada uno de sus esquemas de token.

Producto (token)	Comprobación de sesión	Sin autenticar	Arranque del desafío	Verificación de la prueba	Memoria residente
Sibuna (BLAKE3 con clave)	88.104	78.997	42.664	57.924	7,6 MiB
Anubis (Ed25519)	10.951	28.619	7.893	12.668	32,6 MiB
Anubis (HS512)	16.976	27.785	7.952	19.261	29,1 MiB

mientras que la de Sibuna se mantuvo cerca de 16 MiB en todo momento. Detrás de `caddy respond`, las filas retransmitidas de Sibuna están limitadas por el origen, que comparte las CPU del generador de carga: con el origen simulado, más ligero, de Productos completos, las mismas filas alcanzan unas 139.000 peticiones por segundo.

9.2 Comparación arquitectónica detallada

9.2.1 SafeLine (Chaitin Technology)

SafeLine lleva la idea semántica de libinjection a un producto completo con consola de gestión, y su fabricante declara tasas de detección del 71,65% (modo equilibrado) y del 76,17% (modo estricto), con tasas de falsos positivos del 0,07% y del 0,22%[32]. Son mediciones de la calidad de detección, que Sibuna no ha publicado para su propio motor.

- *Despliegue.* La edición comunitaria se distribuye como siete contenedores Docker en un host Linux con al menos un núcleo de CPU, 1 GB de memoria y 5 GB de disco. Termina TLS por sí misma, algo que Sibuna no hace.
- *Estado.* Guarda su registro de ataques en PostgreSQL dentro de la pila, una pila por host.
- *En qué se diferencia Sibuna.* Sibuna es un único binario que arranca con un archivo SQLite opcional, y un clúster comparte una base de datos PostgreSQL que el operador quizá ya tenga en marcha. Su memoria residente en reposo fue de 5.592 a 5.788 KiB en las ejecuciones de producto. No hemos ejecutado SafeLine en nuestro host, que no tenía daemon de Docker, así que no hacemos ninguna comparación de rendimiento.

9.2.2 Anubis y Coraza

La página intermedia con prueba de trabajo se generalizó en 2025, cuando el tráfico de scrapers para entrenar modelos de lenguaje encareció el funcionamiento de orígenes corrientes, y Anubis es su implementación abierta más conocida. Es un proxy inverso en Go, capaz y muy desplegado: sirve un desafío en JavaScript, firma un JSON Web Token con Ed25519 y guarda el estado de los desafíos en un almacén. Coraza implementa el lenguaje de reglas de ModSecurity (SecLang) y ejecuta el OWASP Core Rule Set; es una forma de añadir inspección basada en reglas junto a un control de admisión de ese tipo.

- *Adónde va el trabajo.* Anubis y Sibuna Gate hacen que el cliente pague antes de que el origen haga nada. Sibuna añade un segundo nivel de trabajo secuencial, elegido por regla, para las rutas que merecen un control de admisión más fuerte, y sus sesiones recuerdan cuánto trabajo se pagó.
- *Lo que recuerda el servidor.* Anubis almacena los desafíos; Sibuna emite un registro autoautenticado y solo recuerda las etiquetas resueltas, en una tabla fija (teorema 3). Su token por defecto es simétrico, lo que lo abarata de verificar y solo deja a un atacante cuántico una aceleración cuadrática.
- *Inspección.* Sibuna Shield realiza su propia inspección estructural en la misma pasada que la admisión, de modo que un solo proceso y un solo analizador cumplen ambas funciones. No ejecuta el OWASP Core Rule Set. Evaluamos los motores SecLang existentes, incluidos los enlaces a libmodsecurity y a Coraza, y no admitimos ninguno: un motor de inspección en un servidor perimetral es código crítico para la seguridad, que preferimos poseer y revisar

[32] Libro, parte VIII, "Lo que no se mide": cifras declaradas por los fabricantes, comprobadas en septiembre de 2026.

nosotros mismos. Si Sibuna añade compatibilidad con el Core Rule Set, un registro de diseño especificará antes el subconjunto[33].

- *Medición.* En un mismo host y con la misma carga, Sibuna respondió más de cinco veces más peticiones admitidas en modo forward-auth, con un p99 menor y aproximadamente una octava parte de la CPU por petición (Comparación medida con Anubis).

9.2.3 WAF perimetrales en la nube (Cloudflare y otros)

Las redes perimetrales alojadas sitúan la inspección, la limitación de tasa y la puntuación de bots en muchos puntos de presencia con un plano de control global. Absorben ataques volumétricos que ningún servidor aislado puede absorber y no exigen hardware al operador. El despliegue en clúster de Sibuna toma prestada su forma, un plano compartido de políticas que alimenta instantáneas locales e inmutables, sin dejar de ser autoalojado.

- *Residencia de los datos.* Un WAF alojado termina el TLS del operador y ve las cargas útiles de las peticiones en una infraestructura compartida, lo que puede entrar en conflicto con los requisitos de residencia de los datos.
- *Planes y cuotas.* Las funciones dependen del plan: las cuotas publicadas de Cloudflare permiten 5, 20, 100 y 1.000 reglas personalizadas y 1, 2, 5 y 100 reglas de limitación de tasa en los planes Free, Pro, Business y Enterprise.
- *En qué se diferencia Sibuna.* Sibuna se ejecuta en las propias máquinas del operador, sin tarifa por petición, y sus cambios de política llegan a cada nodo en el siguiente ciclo de almacenamiento (500 ms por defecto). No mitiga ataques de red volumétricos, no puntúa bots con un modelo entrenado y no termina TLS; un despliegue que necesite todo eso sigue necesitando delante un proxy de entrada o una red perimetral.

10 Implementación: crates pequeños y pocas dependencias

Sibuna está escrito en Rust, como un workspace de crates pequeños cuyas dependencias apuntan en un solo sentido: bibliotecas reutilizables, después un núcleo de configuración y diagnóstico, después las bibliotecas de la aplicación y, por último, el daemon y el probador del navegador. Tres decisiones de ingeniería importan a un operador, así que las recogemos brevemente[34].

Pocas dependencias. Tratamos cada biblioteca de terceros como un pasivo: hay que revisarla, actualizarla y confiar en ella. El plano de datos no usa ninguna. Las funciones hash, las firmas, base64, JSON, DEFLATE, Aho-Corasick y el analizador HTTP/1.1 son bibliotecas pequeñas propias, que otros proyectos pueden usar por separado. Una dependencia solo se admite cuando es madura, autocontenida y resuelve un problema realmente grande, y el motivo se escribe junto a ella en el manifiesto de compilación y en un registro de diseño. Se han admitido dos, cada una confinada en una biblioteca: `sqlx`, para el almacenamiento en SQLite y PostgreSQL, que trae un centenar de crates transitivos[35], y `rustls`, para el TLS saliente de la consola y el TLS de PostgreSQL, con trece crates en tiempo de ejecución, porque un cliente TLS escrito por nosotros sería un proyecto crítico para la seguridad mayor que el plano de datos de Sibuna[36]. Una compilación sin almacenamiento ni consola no tiene ninguna dependencia de terceros.

Seguridad de memoria. El código que elude las comprobaciones de seguridad de memoria del compilador está prohibido por defecto, y un módulo que lo necesite debe habilitarlo por su nombre y justificar cada uso en un comentario. Algunos módulos lo hacen: los caminos que usan las instrucciones SHA del procesador, una escritura que borra secretos, las declaraciones de unas pocas funciones de la biblioteca de C para el manejo de señales, las exportaciones de los dos módulos WebAssembly y el anillo que lleva los incidentes al hilo de almacenamiento. El anillo es el único lugar donde mantuvimos ese código por velocidad, y solo después de una medición: el canal acotado de la biblioteca estándar costaba entre un 20 y un 45% más por inserción con 16 productores[37]. En el resto, la propiedad hace el trabajo que de otro modo haría la gestión manual de la memoria: una petición analizada no puede sobrevivir a su búfer, un motor se libera con su último lector y un socket no puede cerrarse mientras algún hilo pueda alcanzarlo todavía.

Disciplina y pruebas. El cuerpo de una función tiene como máximo 70 líneas lógicas y un archivo como máximo 1.408 líneas, límites que impone un comprobador. El código de biblioteca devuelve errores estructurados y nunca imprime, y cada error que ve un usuario indica su ubicación, muestra un fragmento y da una pista. La batería de pruebas del workspace de 1.0.0-beta.3 ejecuta unas 2.200 pruebas en SQLite y unas 700 más contra PostgreSQL; la interfaz de la consola tiene 298 pruebas, incluidos fixtures de referencia, y las comprobaciones en el navegador recorren con Chromium sin interfaz gráfica la página intermedia y todas las páginas de la consola[30].

[33] SID 0010, «The OWASP Core Rule Set».
[34] SID 0010, "Workspace", "Dependency policy" y "Engineering standards".
[35] SID 0011, "Why sqlx"; `Cargo.toml`, donde cada crate de terceros lleva su motivo.
[36] SID 0012, "Specification", "Rationale and alternatives" e historial de revisiones (TLS de PostgreSQL implementado).
[37] SID 0010, historial de revisiones (revisión de seguridad, 2026-10-15): el canal acotado estándar frente al anillo de Vyukov, `tests/timing.rs`. Los módulos que lo habilitan son `crates/crypto/src/sha256/` (x86-64 y AArch64), `crates/crypto/src/ct.rs`, `cmd/sibuna/src/signals.rs`, `cmd/sibuna-pow-wasm/src/lib.rs`, `cmd/console-ui/src/abi.rs` y `crates/store/src/ring.rs`; el workspace fija `unsafe_code = "deny"` y el `undocumented_unsafe_blocks = "deny"` de Clippy.
[30] SID 0010, "Acceptance criteria and implementation status" (criterios 1, 3 y 5), que registraba unas 2.100 pruebas del workspace y 290 pruebas de la interfaz antes del SID 0014; los recuentos para 1.0.0-beta.3 proceden de `cargo test --workspace -- --list`.

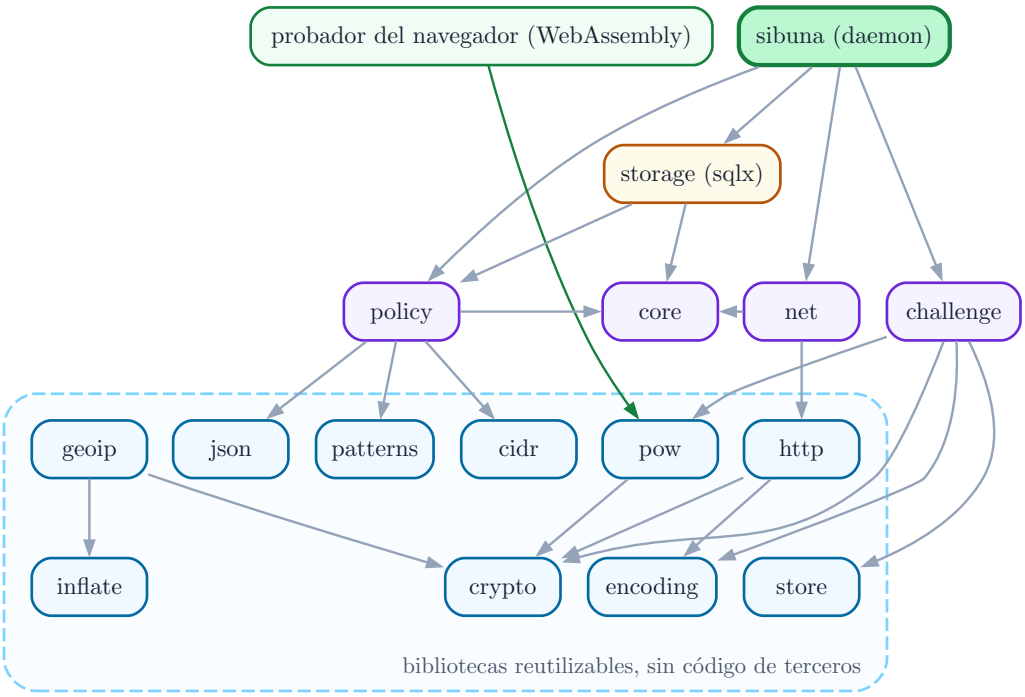


Fig. 6 Las bibliotecas de Sibuna, simplificadas (según el SID 0010). Las flechas van de una biblioteca a lo que usa. Las bibliotecas reutilizables no dependen de ninguna biblioteca de la aplicación, y solo las bibliotecas de almacenamiento y de la consola traen código de terceros.

11 Limitaciones y trabajo pendiente

Enumeramos lo que este documento no establece, para que ninguna afirmación se lea de forma más amplia que su evidencia.

- *Una sola alternativa medida.* Medimos Anubis en un host (Comparación medida con Anubis). No hemos medido ni SafeLine ni un WAF alojado.
- *Sin clúster bajo carga.* Una prueba funcional ejecuta tres nodos sobre una base de datos PostgreSQL (SID 0011, en estado Committed), pero ningún programa de medición somete varios nodos a carga.
- *PostgreSQL requiere compilar desde las fuentes.* Los ejecutables de la versión incluyen el almacenamiento SQLite y la consola, pero no PostgreSQL; un clúster compila Sibuna con la feature `postgres-tls`.
- *El control de aislamiento de la consola* con ocho paneles activos bajo carga no se ha ejecutado (La consola).
- *Solo loopback.* Ninguna medición cubre TLS, una red de área extensa ni navegadores reales bajo carga; cada despliegue debe medir su propio camino.
- *Inspección acotada.* Shield es un inspector heurístico de los primeros 8 KiB de un cuerpo, no un analizador completo de SQL o HTML, y la inspección forward-auth solo ve los bytes que envía el proxy de entrada.
- *Alcance del protocolo.* Sibuna habla HTTP/1.0 y HTTP/1.1; no tiene HTTP/2 ni terminación TLS de entrada.
- *Las huellas TLS vienen del proxy de entrada.* Sibuna no puede calcular JA4 por sí mismo. Una huella es tan fiable como el proxy de entrada que la escribe, y ese proxy debe sobrescribir cualquier copia que haya enviado un cliente. Un cliente puede imitar el handshake de un navegador con bibliotecas especializadas, así que JA4 encarece la automatización, pero no demuestra la identidad[11].
- *Garantía criptográfica.* El nivel de trabajo secuencial se apoya en demostraciones publicadas en el modelo de oráculo aleatorio; nuestro protocolo concreto tiene pruebas frente a un etiquetador de referencia, no una auditoría criptográfica independiente.
- *Una sola máquina.* Todas las cifras se tomaron en un único AMD Ryzen 9 5950X. Una clasificación obtenida en este host no vale para otro.

12 Conclusión

La defensa web ha crecido a menudo por acumulación: entornos de ejecución interpretados, contenedores, conjuntos de reglas de expresiones regulares y almacenes externos apilados delante de una aplicación. Con Sibuna hemos probado la dirección contraria, guiados por unas pocas ideas sencillas:

- hacer que el remitente pague la admisión con trabajo verificable, mantener barata y sin estado la comprobación del servidor, y dejar que una sesión recuerde cuánto trabajo pagó;
- inspeccionar por estructura en tiempo lineal, en la misma pasada que la admisión, para que ninguna sesión pueda eludir la inspección;
- respetar la máquina: un búfer por conexión, ninguna asignación en el camino de la petición, tablas acotadas con ubicación con clave e hilos que se reutilizan en lugar de recrearse; y
- dejar que una base de datos madura sea la dueña del estado compartido duradero, detrás de un único hilo de almacenamiento al que el camino de la petición nunca espera.

En una máquina de 16 núcleos, estas decisiones permiten a Sibuna servir más de 200.000 peticiones admitidas por segundo en modo forward-auth con cuatro CPU, con unos 16 MiB de memoria residente bajo esa carga y un p99.9 de 62 us a 100.000 peticiones por segundo. Son las cifras de un único host a través de loopback, y las ofrecemos como indicio de que el diseño se sostiene, no como una clasificación. Eliminamos nuestro propio protocolo de consenso y lo decimos con claridad; a cambio, el sistema es más pequeño y su plano de datos no tiene dependencias de terceros. Queda mucho por medir, y hemos intentado decir exactamente qué.

13 Fuentes y reproducción

Cada cifra de este documento procede de un archivo de resultados confirmado en el repositorio de Sibuna, de un registro Sibuna Discussion (SID) o del libro de Sibuna, y cada tabla o nota nombra su fuente. Los archivos de resultados los escriben `cargo xtask bench primitives`, `cargo xtask bench product` y `cargo xtask bench latency`, y `benchmarks/results/README.md` da los comandos, los conjuntos de CPU y la procedencia de cada archivo.

- `benchmarks/results/primitives-latest.md`, `product-latest.md`, `latency-latest.md` y sus archivos `.json`; `console-8-dashboards-2026-10-15.md`.
- SID 0002 (arquitectura base y sus notas de revisión), SID 0004 (el modo de protección Shield), SID 0006 (fundamentos matemáticos), SID 0007 (la consola), SID 0010 (workspace, crates y política de dependencias), SID 0011 (almacenamiento con sqlx), SID 0012 (TLS de la consola), SID 0013 (almacenamiento de la consola) y SID 0014 (huellas JA4 desde el proxy de entrada), en `docs/sid/records/`.
- El libro de Sibuna, `docs/book/`: parte I (el coste de una petición), parte II (trabajos previos), parte III (criptografía), parte IV (el protocolo), parte V (ingeniería sin asignaciones), parte VI (autómatas y el cortafuegos semántico), parte VIII (evaluación empírica) y parte IX (operaciones).
- Las fuentes primarias: Back, Hashcash (1997); Cohen y Pietrzak, Simple Proofs of Sequential Work (EUROCRYPT 2018); Blocki, Lee y Zhou, On the Security of Proofs of Sequential Work in a Post-Quantum World (ITC 2021); Aho y Corasick, Efficient String Matching (1975); Celis, Robin Hood Hashing (1986); el Generic Cell Rate Algorithm del ATM Forum (1996).

Documento técnico de Sibuna | noviembre de 2026 | Versión 1.0.0-beta.3
Especificación, código fuente y pruebas de rendimiento: https://github.com/insanai/sibuna