SIBUNA

零分配、工作量约束型 Web 防御引擎

工作量准入、线性时间检查、有界状态与基于 SQL 的存储

Vikrant Rathore

Ronak Rathore 与 Kanak Rathore 协助
Insan AI | 版本 1.0.0-beta.3

摘要

Web 应用防火墙和边缘防御往往有三个架构上的问题。第一是 成本倒挂：发送方发出一个请求几乎不花任何代价，防御方却要解析它，用可能回溯的正则表达式扫描它，还要为它查询外部存储。第二是 运行时不可预测：请求路径上的堆分配、垃圾回收停顿以及由多个容器组成的部署，使延迟分布的尾部难以设定上界。第三是 运维负担：封禁、令牌和限速的共享状态往往存放在运维人员必须在防火墙之外另行运行的服务中。**Sibuna** 是我们为解决这些问题所做的尝试：它把少量朴素的机制放进一个自包含的可执行文件，本文逐一说明每种机制背后的道理。(i) 工作量准入：浏览器通过求解 Hashcash 谜题或 Cohen-Pietrzak 顺序工作量证明（按规则选择）来换取会话，服务器根据一个从未存储过的挑战，在 70.2 ns 或 22.3 us 内验证解答。(ii) 有界且无分配的请求路径：字节类 Aho-Corasick 自动机在 55.7 ns 内匹配 40 条机器人签名，比顺序搜索快 13.7 倍；GCRA 限制器在 6.2 ns 内作出决定，每个客户端只占 16 字节状态；Robin Hood 已用集合在 21.0 ns 内记录一个解答。SQL 注入、跨站脚本、命令注入和路径遍历的检查在同一遍处理中完成，耗时与输入长度成线性关系。(iii) 无需自研共识协议的共享状态：单个节点使用一个嵌入式 SQLite 文件，集群共享一个 PostgreSQL 数据库，由一个存储线程读取，请求从不等待它。在一台 AMD Ryzen 9 5950X 上，守护进程使用四个 CPU 时，Sibuna 每秒应答约 200,000 个 forward-auth 请求（各工作负载为 197,532 到 211,636 个），p99 为 305 到 327 us；在每秒 100,000 个请求的开环负载下，p99.9 保持在 62 us；常驻内存峰值约为 16 MiB。我们说明每个数字是如何得到的，并列出尚未测量的内容。

6.2 ns	55.7 ns	22.3 µs	16 MiB
GCRA 速率检查	**机器人匹配器**	**PoSW 验证**	**守护进程峰值 RSS**
客户端哈希 | 16 个分片	Aho-Corasick | 40 条签名	深度 13 | 16 次揭示	wrk 负载下 | 4 个 CPU

Sibuna 运行在一颗 AMD Ryzen 9 5950X 的单个核心上（守护进程则用四个 CPU），2026 年 10 月 15 日：benchmarks/results/primitives-latest.md 和 product-latest.md；测量方法见“评估”一节。

1 序章：一次请求的经济学

每个到达服务器的请求都在索取服务器的注意力。一个方法、一个路径和几个请求头，并不能说明这个请求值得服务。因此，防御系统在请求和源站之间设置一个决策：准入它，拒绝它，或者先要求发送方完成可验证的工作。本文以 Sibuna 为例，从数学一直到内存中的字节，追踪这一决策。

> **费曼谈物理直觉：谁为工作买单？**
>
> 设想一个轻轻一碰就会转动的十字转门。门后坐着一名门卫，每碰一下，他就得站起来，核对三本账簿，再给总部打一个电话。门外的孩子可以毫不费力地碰上一整个下午，门卫却不到中午就筋疲力尽。许多防火墙就是这样安排的：省力的一方是攻击者。物理学提示了一种更好的安排。把转门连到一台水泵上，谁想通过，就得把一桶水提到门上方的水箱里。提水需要付出一点实实在在的力气；门卫只需瞥一眼水箱，看看水位是否升高。瞥一眼对门卫几乎没有成本，提水却让访客付出了真实的代价。通勤的人提一次水，就能整天进出；恶作剧的人很快就回家了。

本文中借人物之口的方框是我们自己写的文字，模仿的是理查德 · 费曼、高德纳和莱斯利 · 兰波特的风格，我们努力效仿他们讲解问题的方式。除非方框注明了出处，否则其中的文字都不是引语。

自动化发送方发出每个请求的成本都很低。请求到达传统防火墙时，防御方可能要完成一次 TLS 握手，分配内存来复制和拆分请求，在请求上运行一串正则表达式，还要通过网络往返一次共享存储来更新计数器。我们不给这种不对称性赋予一个通用的比例，因为这样的比例并不存在：它取决于源站、缓存和攻击本身。不过，一个简单的模型就能让这一决策变得清晰可见。令 c_o 表示源站处理每个获准请求的工作，c_g 表示网关处理每个请求的工作，a 表示网关准入的请求比例。只有当

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

即网关为源站节省的工作超过网关自身的成本时，网关才值得部署[1]。由此得出两个推论。网关自身的成本 c_g 必须很小，并且无论输入如何都有上界；准入的决定必须让发送方而不是服务器付出代价。

[1] 《Sibuna 之书》第 I 部分，“准入是一个经济决策”。这个不等式是一个容量模型，不是测量结果；它省略了带宽、连接状态和客户端延迟。



图 1 工作落在哪一方。上图：先检查的防御系统为发送方发出的每个请求付出自己的工作。下图：Sibuna 要求发送方完成期望 2^b 次的哈希尝试，并用一次哈希验证解答。工作量条只是模型；验证时间是实测值（16 比特的 Hashcash 验证，`benchmarks/results/primitives-latest.md`）。

Sibuna 以工作量证明作为准入的条件，从而扭转了这种不对称。为了换取会话，浏览器要找到一个 nonce，使其 SHA-256 摘要具有 b 个前导零比特，或者计算一个顺序工作量证明。期望的搜索量为 2^b 次哈希尝试，$b = 16$ 时为 65,536 次；服务器只检查一个候选解。服务器的检查也是无状态的：签发挑战不写入任何内容，只有付出了工作的解答才会占用内存。工作量证明不能证明对方是人，也不能证明其身份或意图。发送方可以租用计算机，在会话有效期内共享会话，或者把工作分散到多台机器上。谜题改变的是准入的价格；获准的请求可以做什么，仍然由策略和检查决定。

2 设计概览：一遍处理中的三个决策

Sibuna 把准入、检查和信誉作为三个独立的决策，并在源站做任何工作之前，在同一个进程中对请求一遍处理完这三个决策。它以两种产品形态提供这些功能。*Gate* 提供工作量证明准入，以及规则、CIDR 信誉和本地洪泛防护。默认形态 *Shield* 在此基础上增加了对请求载荷的有界检查。两种形态都可以作为反向代理运行在源站之前，也可以作为 forward-auth 端点，供 Caddy 或 Nginx 等入口代理查询。策略、信誉和事件历史的持久存储是两者都可选用的选项：可以是单个节点本地的，也可以由集群共享[2]。

2.1 设计选择

下面的大多数思想来自研究文献或更早的系统，《Sibuna 之书》第 II 部分追溯了它们的源流。我们所做的，是把它们组合进一条有界的请求路径，让每一步都保持低成本，并如实测量结果。下面列出这些选择，并给出各自的理由。

1. 签发挑战不花服务器任何成本。 挑战是一条用带密钥哈希封装的自认证记录，因此签发挑战不写入任何状态；只有付出了工作的解答才会占用内存，并在挑战过期时释放（算法及其保证）。

表 1 两种产品形态（SID 0002；《Sibuna 之书》第 II 部分）

能力	Gate（`--gate`）	Shield（`--shield`，默认）
准入	Hashcash 或顺序工作量的浏览器挑战；会话令牌与客户端及其付出的工作绑定	相同
策略	声明式规则（路径、User-Agent、请求头、IPv4 和 IPv6 CIDR、来自受信任入口代理的 JA4 TLS 指纹），动作为 ALLOW、DENY、CHALLENGE 和 WEIGH；机器人签名；信誉	相同
洪泛防护	GCRA 速率限制、蜜罐、临时封禁	相同
载荷检查	停用	SQL 注入、跨站脚本、命令注入和路径遍历，检查请求体的前 8 KiB
可选状态	策略和信誉存于 SQLite，或通过 PostgreSQL 共享	另有事件取证：全文搜索和攻击活动聚类

[2]SID 0002，“Decision”；《Sibuna 之书》第 II 部分，“一个可执行文件中的三个思想”；第 VI 部分，“声明式策略引擎”。按规则选择的示例、要求票据和带工作等级的会话出自第 IV 部分；自适应基线出自第 IV 部分和 SID 0006（修订说明，Part H）。

2. 两层工作量，按规则选择。 除 Hashcash 外，Sibuna 还提供 Cohen-Pietrzak 顺序工作量证明：它的可靠性即使面对量子证明者也已得到证明，其证明的验证只需几微秒。结账路由可以要求 20 个工作比特的顺序工作量，而博客只要求 16 比特的 Hashcash（工作量证明：并行搜索与顺序工作）。
3. 会话记住付出的工作。 会话令牌记录其持有者所求解的机制和工作比特。它只能用于要求不高于此的路由，因此廉价的会话打不开昂贵的路由。
4. 对称令牌。 会话用带密钥 BLAKE3 封装，验证只需 155.8 ns，工作量约为 Ed25519 签名的 1/260；量子搜索对它只能带来平方级加速。
5. 会话无法绕过的检查。 准入和检查在同一遍处理中运行，有效会话只能满足挑战：一个带有会话和 `q=' OR 1=1--` 的请求仍然会被拒绝。
6. 线性时间、感知结构的检查。 一次字节类扫描、一遍自动机匹配和几个小型结构化分词器，在与输入长度成线性的时间内作出决定，不使用可能回溯的正则表达式。
7. 有界状态与带密钥的放置。 每张表都有固定容量，并规定了表满时的行为；键按带密钥哈希放置，因此攻击者无法针对某个受害者制造碰撞（带密钥哈希与安全审查）。
8. 每个请求零分配，策略更新无停顿。 一个连接用一块固定缓冲区完成所有决策；新的策略引擎发布时，没有任何请求或写者需要等待（机械同理心：零分配与有界状态）。
9. 难度随负载变化。 在每秒 50 个已验证解答的基线之上，速率每翻一番就增加一个工作比特，最多增加六个；洪泛流量因此要付出指数级增长的代价，平静的流量则保持基础成本。
10. 从事件中发现攻击活动。 被拒绝的请求连同一个小型嵌入向量一起记录，相似的事件被归入攻击活动，运维人员可以在控制台中调查。

3 机械同理心：零分配与有界状态

> **高德纳谈机械精度：缓存行与具体数学**
>
> 在实时系统的内层循环里调用通用分配器的程序员，等于把一部分设计交给了陌生人。分配器平均来看也许很快，但它的成本取决于堆的历史，而这段历史是由发送请求的人书写的。在优化任何东西之前，我们应当先计数：每一步触及多少字节，多少缓存行，多少次相互依赖的加载。然后还要测量，因为计数也可能出错。一个程序是优雅的，当且仅当它需要的每个字节在第一个套接字打开之前就有了确定的位置，并且每个循环所做的工作恰好是其数学所要求的。

3.1 零分配不变量

请求路径上的每一次动态分配，都是拒绝服务攻击的一根杠杆：日积月累的碎片化、分配器锁上的争用，以及专门构造出来诱发分配的请求所造成的放大。Sibuna 的规则是：分类、验证和移交给代理的过程都不进行堆分配。连接在其套接字的整个生命周期内拥有一块 64 KiB 的读缓冲区和一块 16 KiB 的写缓冲区，以及在连接开始时一次性分配的 40 KiB 代理工作区。其他一切都是指向这些缓冲区的切片：解析器在读缓冲区中定位请求的方法、路径、查询、请求头和 Cookie，一个都不复制[3]。实现语言的类型系统保证，已解析的请求不能比它所描述的字节活得更久，因此只要还有请求在读取缓冲区，缓冲区就不能被重新填充。

这条规则是请求路径的契约，而不是对整个进程的断言。创建线程、启动和存储都会分配内存。这份契约由统计分配次数的测试来检查：守护进程的稳态测试统计其分配次数，代理的测试在 500 个经代理的 keep-alive 请求中没有发现任何分配[4]。

3.2 有界且复用的线程

每个接受的连接都由一个专属线程服务，因此慢速客户端只会拖慢自己，请求路径也保持同步。我们考虑过事件循环，最终保留了线程模型：请求路径的速度来自零拷贝解析器、每个连接固定的缓冲区和无分配的决策，而事件循环对这几点都没有改进[5]。连接线程的数量受 `--max-connections` 限制（默认 1,024）；超出时，accept 循环直接应答 `503`，不创建任何线程。默认情况下，一个套接字对端地址最多持有 64 个连接，IPv6 对端按其所在的 /64 网络计数。

线程来自一张由可 join 的槽组成的固定表。结束的线程在被 join 之前一直保留着它的栈，因此新任务会 join 并占用编号最小 的已结束槽，而不是推进游标；这样，常驻内存跟随的是当前打开的连接数，而不是已服务的连接数。完成连接的线程会停驻下来，接手下一个连接。测量带来的改变 一节讲述了一次测量如何让我们加上了最后这条规则。

3.3 请求流水线

请求按固定顺序通过下面的检查。廉价的检查排在前面，每个终结请求的决策都在任何更昂贵的阶段运行之前作出。有效会话可以满足挑战，但绝不推翻拒绝或 WAF 检出结果。代理则读取源站的响应头部，判定其分帧方式（无响应体、固定长度、分块编码或读到连接关闭为止），原样转发响应体，并把停在干净边界上的源站套接字归还给一个有 256 个槽、分条带的连接池；这样，经代理的客户端连接可以保持打开，也不必为每个请求重新连接源站。

[3] 《Sibuna 之书》第 V 部分，“每个连接一个线程，并且有界”和“每个连接一个缓冲区”。
[4] 《Sibuna 之书》第 V 部分，“连接循环”；`cmd/sibuna/tests/steady_state.rs` 和 `crates/net/tests/steady_state.rs`。
[5] SID 0010，“Concurrency model”和“Rationale and alternatives”；SID 0002，“Request contract”及其修订说明（每个地址的连接数、IPv6 前缀、分条带的源站连接池）。

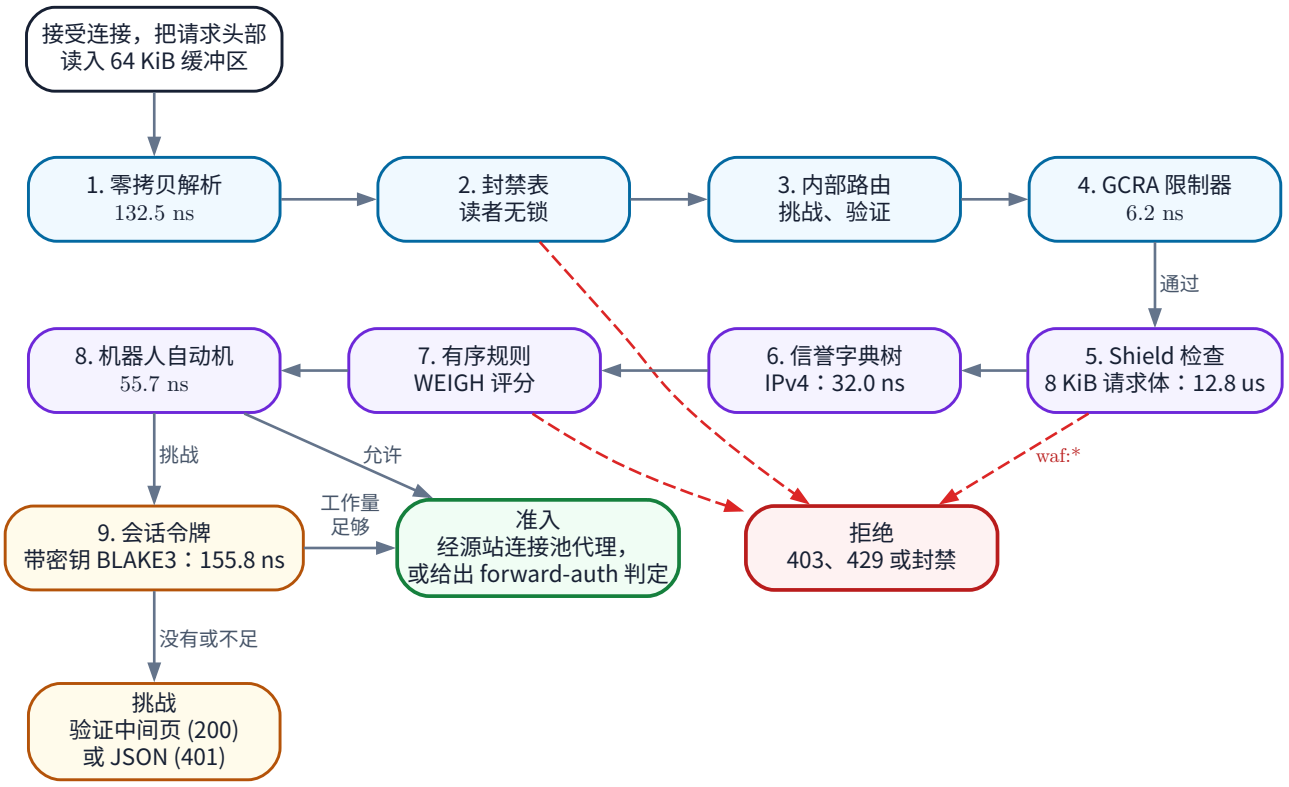


图 2 请求流水线，并标出原语测试套件测得的每个原语在单核上的中位时间（benchmarks/results/primitives-latest.md）。各阶段按编号顺序运行；虚线箭头表示提前拒绝（GCRA 限制器和信誉字典树也以同样的方式拒绝）。对一个浏览器请求的完整分类，在 Gate 配置下耗时 75.9 ns，在 Shield 配置下耗时 790.8 ns。

3.4 发布策略而不中断请求

策略变化时，策略引擎在请求路径之外重建，请求绝不能看到构建了一半的引擎。存储线程构建一个完整的引擎，并以引用计数的形式把它发布到一个带代次计数器的槽中。每个连接都保留一个缓存：它用一次 acquire 加载读取代次，只有在发生发布之后才获取新的引用。旧引擎在最后一个持有它的缓存放手时被释放，写者从不需要等待读者退出。在争用下，缓存读取比获取读写锁便宜 400 多倍。引擎本身约有 200 KiB 的紧凑表（209,052 字节），可容纳 128 条规则和 8,192 个字典树节点。

3.5 分片锁

连接线程共享的表（已用集合、限制器、封禁表）按键的哈希分成 16 个分片，每个分片独占一条缓存行。它们的临界区只有几十条指令长，因此我们测量了自旋锁与自适应互斥锁的对比；后者先短暂自旋，然后休眠。无人争用时，自旋锁大约节省一纳秒；但边缘服务器运行的线程多于 CPU 数，而持有者被抢占的自旋锁会让每个等待者白白耗尽自己的时间片。互斥锁在争用下胜出，因此我们保留了它。

表 2 1 个和 16 个线程循环读取时，一次引擎读取的成本，Ryzen 9 5950X（《Sibuna 之书》第 V 部分，2026 年 10 月）

读取方式	1 个线程	16 个线程
读写锁，仅获取读守卫	4.2 ns	752 ns
读写锁，并获取新引用	7.6 ns	833 ns
代次检查，使用缓存的引用	0.8 ns	1.8 ns

表 3 分片锁吞吐量，单位为每秒百万次操作，Ryzen 9 5950X（《Sibuna 之书》第 V 部分；SID 0010）

工作负载	自适应互斥锁	自旋锁
已用集合，16 个线程	10.8	8.6
已用集合，64 个线程	11.2	6.9
在六个热点地址上运行 GCRA，16 个线程	44.9	19.3
在六个热点地址上运行 GCRA，64 个线程	37.4	6.5

4 算法及其保证

请求路径上的每个原语都有一份说明，写明它的成本和它所保证的性质。我们把其中三条写成定理，并简要给出论证；SID 0006 和《Sibuna 之书》第 III 到 VI 部分给出了完整的证明、定义和参考文献。

定理 1（字节类 Aho-Corasick 自动机的线性时间检查）. 设 $\mathcal{P}$ 是总长度为 m 的模式集合，其字节分为 $c \le 256$ 个类，文本长度为 M。Sibuna 以 $O(mc)$ 的时间和空间构建自动机，并在 $O(M)$ 时间内报告文本中是否出现任一模式；每个输入字节只需一次相互依赖的表读取，与模式数量无关。任何输入都无法使它回溯。

证明概要。 Aho 和 Corasick 的自动机先构建带失败链接的字典树，然后只扫描文本一遍。Sibuna 在构建时把失败链接折叠进转移表，因此每个输入字节只是一次转移，从不沿失败链前进。一个字节类要么是某个模式中出现的字节（字母的大小写归入同一类），要么是由其余所有字节组成的那一个类；因此表的每一行只有 c 个条目，而不是 256 个。匹配状态排在所有其他状态之后编号，因此检测匹配只需一次比较。

测量。 在 40 条机器人签名上，自动机耗时 55.7 ns；使用相同模式、同样采用任一匹配语义的顺序子串搜索耗时 763.0 ns，自动机快 13.7 倍。借助字节类，整张机器人签名表只占 42,244 字节[6]。

□

定理 2（GCRA 区间界）. 固定每个窗口 W 内 N 个请求的限制，发射间隔 $T = \max(1, \lceil W/N \rceil)$，突发容限 $\tau = (N-1)T$。每个客户端保存一个理论到达时间 TAT。时刻 t 的一次到达是合规的，当且仅当 $\mathrm{TAT} \le t + \tau$，此后 $\mathrm{TAT} \leftarrow \max(\mathrm{TAT}, t) + T$。那么，任何以一次合规到达开始、长度为 L 的区间，至多包含 $N + \lfloor L/T \rfloor$ 次合规到达。

证明概要。 把合规到达依次编号为 $1, \ldots, k$，时刻为 $t_1 \le \ldots \le t_k$。每次合规到达至少把 TAT 提高 T，因此第 j 次之后 $\mathrm{TAT}_j \ge t_1 + jT$。第 k 次到达是合规的，因此 $\mathrm{TAT}_{k-1} \le t_k + \tau \le t_1 + L + \tau$。两式合起来得到 $(k-1)T \le L + (N-1)T$，即 $k \le N + L/T$，而 k 是整数。

实现。 一个客户端的状态是一个 16 字节的单元，没有逐请求的历史，也没有固定窗口的边界效应。16 个分片共用 `--rate-table-cells` 个单元（默认 262,144 个，4 MiB），按带密钥哈希放置；排空的单元就地回收，无需清扫线程。对守护进程已经算好哈希的客户端做一次检查，耗时 6.2 ns[7]。

□

定理 3（状态只随付出的工作增长）. 设 S 是守护进程记住的挑战标签集合。一个标签只有在其挑战的有效证明被接受之后才进入 S，并在挑战过期时离开。若 r 是一个挑战有效期内被接受的不同解答的速率上界，则 $|S| \le r \cdot \mathrm{TTL}$。已获取但未求解的挑战不占用任何内存。

证明概要。挑战标识符是一条自认证记录：版本、算法、难度、揭示数、签发时间、客户端的带密钥指纹、由伪随机函数派生的 nonce，以及要求这次挑战的规则的哈希，共 36 字节，后跟 16 字节的带密钥 BLAKE3 标签，用 URL 安全的 base64 编码为 70 个字符。签发挑战不写入任何内容。验证时先以常量时间检查标签，再检查时效和客户端绑定，然后检查证明，最后才把标签插入已用集合。把带密钥模式的 BLAKE3 建模为伪随机函数时，发起 q 次验证查询的对手伪造标签的概率至多为 $q \cdot 2^{-128}$ 加上 PRF 优势。

实现。 已用集合由 16 个采用 Robin Hood 开放寻址的分片组成。一个过期条目只能被距其原位槽至少同样远的键就地覆盖，否则后续查找可能提前停止而漏掉仍然有效的标签；有了这条规则，就不需要清扫线程，而且被拒绝的插入从不驱逐有效标签。一次花费加一次查找共耗时 21.0 ns[8]。

□

4.1 语义防火墙

Shield 检查路径、查询、User-Agent、非结构性的请求头，以及请求体的前 8 KiB。它的第一个版本给了我们一个教训。那个版本把一张扁平的子串列表与每个请求头逐一匹配，列表中有 `/*`，而每个浏览器都会发送 `Accept: text/html,...,*/*;q=0.8`。结果，每个真实的浏览器都被判为 SQL 注入。从那以后，检测器只能依据 结构 触发，并且有一项测试会让一个完整的 Chrome 请求通过引擎，要求它干净地通过[9]。

检查分为四个阶段，每个阶段的耗时都与字段长度成线性关系。

1. 字节类预扫描。 用一张 256 项的表扫描一遍，记录出现了哪些类别的字节：引号、`=`、`<`、shell 分隔符、`%`、NUL，以及触发规范化的字节。后续每个检测器都查询这些类别，而不是重新扫描，因此不含引号或等号的字段永远不会进入 SQL 分词器。

[6] `benchmarks/results/primitives-latest.md`（55.7 和 763.0 ns）；表的大小取自 `primitives-latest.json`，见《Sibuna 之书》第 VIII 部分，“资源占用”；SID 0006, Part F，及其关于字节类的修订说明。

[7] SID 0006, Part D（Definition 22, Theorem 23）；SID 0004，修订说明（带密钥的放置，`--rate-table-cells`）；`benchmarks/results/primitives-latest.md`（`gcra_check_hashed`）。

[8] 《Sibuna 之书》第 III 部分（引理 3 和引理 4，线路格式）；第 V 部分，“Robin Hood 已用集合”；SID 0006, Part E（Lemma 28）；`benchmarks/results/primitives-latest.md`（`robin_hood_spend_and_lookup`）。

[9] SID 0004，“Revision note (2026-09-07)”和“Verification”；《Sibuna 之书》第 VI 部分，“关于误报的一课”。

2. 带标签的签名自动机。 所有类别的全部强签名都放在同一个自动机中（定理 1）。一次命中就是终局性的拒绝，名为 `waf:sqli`、`waf:xss`、`waf:path-traversal` 或 `waf:rce`。
3. 结构化分词器。 SQL 分词器在关键字表中查找每个短词，并检查 `or` 和 `and` 之后形如 `literal = literal` 的恒真式；否则，必须出现引号，并且证据的分数要达到一个不高的阈值。短语 “it's a group order from the shop” 得 3 分，可以通过。跨站脚本要求出现一个能承载脚本的标签，并在属性位置上带有事件处理属性；命令注入要求一个分隔符后跟一个不是赋值的命令名，因此 `a=1&id=5` 是参数，而不是命令。
4. 规范化。 含有百分号转义、注释开头、JSON 或 HTML 转义，或可折叠空白的字段，会被解码后再检查一次，因此 `1%27/**/UnIoN/**/SeLeCt` 和 `%252e%252e/` 都会还原为原始签名。

一个 8 KiB 的请求体扫描耗时 12,784.3 ns，约合每字节 1.56 ns。运维人员可以对某个类别只审计而不强制执行，此时它的检出结果只记录，不处置。这样一来，字段就必须按类别分别判定，以免一个被审计的检出结果掩盖一个需要强制执行的结果。Sibuna 并不对每个类别各扫描一遍字段，而是只扫描一遍，找出其签名出现过的类别集合，再只在其中搜索需要的类别；同样的 8 KiB 请求体在 SQL 注入设为审计时耗时 15,485.4 ns[10]。Shield 仍然是一个有界的启发式检查器，而不是完整的 SQL 或 HTML 解析器；超出检查范围的字节不会被检查。

4.2 来自入口代理的 TLS 指纹

User-Agent 是任何客户端都可以随意填写的字符串。JA4 指纹则更难伪造。它概括了客户端建立 TLS 连接的方式：协议、版本、密码套件、扩展和第一个 ALPN 值。客户端的 TLS 库在发送第一个 HTTP 字节之前就已确定了这些内容。只有终止 TLS 的组件能看到握手，因此只有它能计算 JA4。Sibuna 不终止任何入站 TLS，所以它不计算 JA4，而是读取受信任的入口代理计算后在请求头中转发的指纹，请求头名称由 `--ja4-header` 指定[11]。

- 信任。 只有在信任转发头时才读取该请求头，这一判断与决定是否信任 `X-Forwarded-For` 的判断相同。来自不受信任对端的该请求头会被忽略；在反向代理模式下，请求到达源站之前，它的所有副本都会被移除。
- 校验。 只有 36 字节 JA4 格式的值才会被接受，并且必须恰好只有一个同名字段。格式错误或重复的值被视为不存在，并计数。
- 策略。规则匹配完整的指纹，或匹配后跟 `*` 的前缀；没有指纹的请求永远不会匹配这样的规则。匹配只是字节比较，预热后的请求路径仍然不做任何堆分配。
- 记录。 事件保存其请求的指纹，控制台在事件证据中显示它，并可以按它筛选事件。封禁和信誉仍然以地址为键，因为许多合法客户端共用同一个 TLS 库。

4.3 带密钥哈希与安全审查

2026 年 10 月，一次针对请求路径的对抗性审查发现了我们原以为已经定型的机制中的弱点。我们逐一修复，并把修复记录在相应机制的设计记录中[12]。

- 可预测的表内放置。限制器和封禁表原先用写在源代码中的种子计算哈希来放置键，因此任何人都能算出与受害者共用同一探测窗口的地址。现在，放置使用 SipHash-1-3，密钥取自操作系统的随机源，每张表派生自己的密钥。守护进程对每个请求只计算一次客户端哈希（8.7 ns），并把哈希传给每张表；一个请求的存储路径从 43 ns 降到了 26 ns。
- 封禁驱逐。生效中的封禁永远不会被驱逐。当一个键可能占用的每个槽都持有有效的封禁时，新的封禁会被拒绝并计数，因此被封禁的客户端无法驱逐自己的封禁。
- 转发的身份。 `X-Forwarded-For` 最左边的条目可以由客户端写入。Sibuna 取从右数第 `--trusted-hops` 个位置的条目，即最近的受信任代理所追加的那一项。
- *Cookie* 投掷。 一个请求可以携带会话 Cookie 的多个值；守护进程最多尝试八个，因此兄弟域名植入的 Cookie 无法让访问者陷入挑战循环。
- 已用集合的容量。 一张 65,536 个标签的表，在五分钟的挑战有效期内，只需每秒约 220 个已验证的解答就会被耗尽，此后诚实的解答都会被拒绝。现在容量可以配置，默认为 1,048,576 个标签（24 MiB）。
- 自适应难度。 在负载下提高工作量的控制器现在统计的是已验证的解答，而不是已签发的挑战，因为后者对洪泛者毫无成本。

带密钥放置是有代价的，而我们每个请求只付一次，而不是每张表付一次。由限制器自己对原始客户端字节计算哈希的检查耗时 12.8 ns；对已算好哈希的客户端做检查（守护进程实际走的路径）耗时 6.2 ns。由于限制器和已用集合以清零的内存分配，空闲的表不会驻留在内存中，因此配置的容量在被使用之前不占用内存。

[10] `benchmarks/results/primitives-latest.md`（`8kb_body_semantic_scan`，`8kb_body_sqli_audit_other_enforce`）；SID 0004，修订说明“inspection modes”；《Sibuna 之书》第 VIII 部分，“WAF 检查”。

[11] SID 0014，“Specification”和“Security and operational considerations”；《Sibuna 之书》第 IX 部分，“来自入口代理的 JA4 指纹”；`.github/release-notes.txt`。

[12] SID 0002，修订说明（转发的身份；Cookie 值）；SID 0004，修订说明（带密钥的放置、封禁表、实测成本、清零的表）；SID 0006，修订说明（已用集合容量、自适应控制器的输入）；SID 0010，审查历史（安全审查，2026-10-15）；`benchmarks/results/primitives-latest.md`（`gcra_check` 和 `gcra_check_hashed`）。

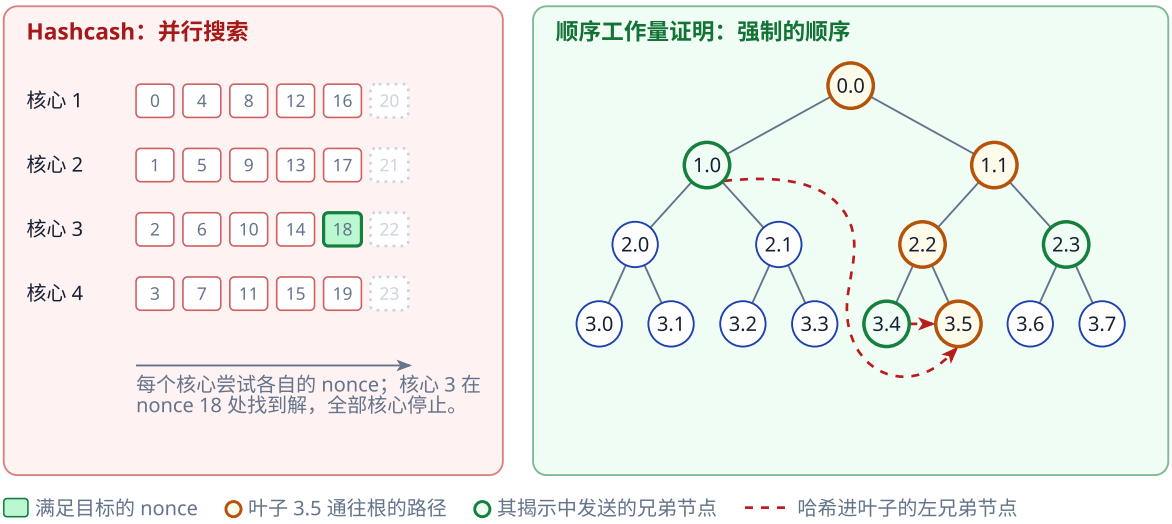


图 3 并行搜索与顺序工作的对比。左图为 Hashcash：各次 nonce 尝试相互独立，因此 M 个核心大约快 M 倍完成。右图为深度 3 的 Cohen-Pietrzak 树：虚线表示的左兄弟边使叶子 3.5 必须等待其左边的子树完成；揭示叶子 3.5 时，发送它的标签和三个绿色的兄弟节点（据《Sibuna 之书》第 III 部分）。

5 工作量证明：并行搜索与顺序工作

5.1 Hashcash 层

第一层是比特粒度的 Hashcash。对于挑战字符串 C 和难度 b，客户端寻找一个 nonce N，使 SHA-256 $(C \,\|\, \texttt{":"} \,\|\, \mathrm{dec}\,(N))$ 具有 b 个前导零比特。nonce 以十进制表示，使 WebAssembly 求解器、JavaScript 后备实现和原生验证器之间的线路格式完全一致。对于理想的摘要，尝试次数 K 服从 $p = 2^{-b}$ 的几何分布：

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

搜索有两个局限。它的方差很大，所以一次缓慢的求解并不说明 worker 出了故障。而且各次独立尝试可以并行进行，因此核心更多的发送方完成得更快，尽管期望的总工作量相同[13]。

5.2 顺序工作层

第二层采用 Cohen 和 Pietrzak 的顺序工作量证明（EUROCRYPT 2018）。它的标签构成一棵深度为 n 的完全二叉树。内部节点对其两个子节点做哈希。叶子则在其通往根的路径每次向右转的地方，对 祖先的左兄弟 做哈希。正是这些左兄弟边使计算成为顺序的：一个叶子必须等它左边的每棵子树都完成之后才能计算标签，因此根标签按顺序依赖于全部 $2^{n+1}-1$ 个标签。算出根 φ 之后，证明者用 Fiat-Shamir 推导出 t 个叶子，并为每个叶子发送叶子标签及其路径上的 n 个兄弟标签。验证者重新计算每个被揭示的叶子，再逐层哈希直到 φ，代价为在固定工作区内计算 $O(tn)$ 个标签。

参数的选择让数字保持很小。工作比特按 $n = b - 3$ 映射为深度，并限制在 $[4, 24]$ 之内。一个证明有 $32(1 + t(n+1))$ 字节，在深度 13、16 次揭示时为 $32(1 + 16 \times 14) = 7200$ 字节。证明者只保留树的顶部几层，并重新计算它必须揭示的子树，因此包括证明在内的整个工作区都不超过 100 KB；浏览器标签页和原生测试使用同一个证明者[14]。Blocki、Lee 和 Zhou（ITC 2021）在量子随机预言机模型中证明了该构造对量子证明者也是可靠的。

我们应当谨慎看待这些结果究竟证明了什么。这些安全性结果在相关论文的随机预言机假设下成立，适用于论文所陈述的构造和界。它们并不是对我们具体的 SHA-256 协议、参数映射或浏览器耗时的独立审计。初等的抽样界 $(1-\alpha)^t$，即 t 次独立揭示全部错过一个固定的坏比例 α 的概率，解释了为什么更多的揭示有帮助；但它本身并不能防御一个反复尝试大量根的证明者。我们不作“作弊永远不会更便宜”这样的无条件断言。

5.3 往返流程与会话

没有有效会话的导航请求会收到验证中间页，页中携带一张 要求票据：本次决策所要求的挑战（算法、工作比特、揭示数）和规则的哈希，为该客户端封装，在一个挑战有效期内有效。页面凭票据获取挑战，因此工作量由提出要求的那次决策确定。Web Worker 用 WebAssembly 证明器求解挑战（在 WebAssembly 不可用时改用 JavaScript 后备实现），然后提交答案。成功后，服务器设置一个 75 个字符的会话 Cookie，它通过带密钥的指纹与客户端的地址和 User-Agent 绑定，随后页面重新加载。浏览器中的证明器与服务器的验证器由同一份源代码编译而成，因此两者不会逐渐偏离[15]。

[13]《Sibuna 之书》第 III 部分，“Hashcash：第一层的数学”；第 I 部分，“把哈希搜索看作随机变量”。
[14]《Sibuna 之书》第 III 部分，“顺序工作量证明：第二层”、“证明者的内存”和“参数不是耗时”；SID 0006，Part B，及其 2026-09-07 的实现审查。
[15]《Sibuna 之书》第 IV 部分，“挑战的往返过程”和“会话 Cookie”；SID 0010，“The browser solver”。

5.4 验证的成本

两层的耗时都取决于 SHA-256；Sibuna 在运行时选用处理器的 SHA 指令（如果有的话），因此同一个可执行文件在每台具备这些指令的机器上都很快。16 比特的 Hashcash 验证耗时 70.2 ns，验证一个深度 13、16 次揭示的证明耗时 22,272.4 ns。浏览器证明器是一个 14,744 字节的 WebAssembly 模块。在端到端浏览器检查中，无头 Chromium 求解验证中间页的挑战，使用 WebAssembly 时耗时 29 到 34 ms，使用 JavaScript 后备实现时耗时 84 到 91 ms[16]。原语测试套件只测量验证；原生环境下的耗时不能代表浏览器的求解时间。

会话使用对称令牌。验证者就是签发者，或者是共享同一个种子的集群，因此带密钥的伪随机函数恰好提供所需的认证。带密钥 BLAKE3 令牌的验证耗时 155.8 ns，Ed25519 令牌耗时 40,587.5 ns，一次签名验证的成本约等于 260 次 MAC 验证。选择对称令牌还有第二个理由：Shor 算法能攻破 Ed25519 所依赖的离散对数，而对带密钥哈希，量子搜索至多带来平方级加速。对于需要验证者无法签发令牌的部署，Ed25519 仍是一个可选项。

6 无需自研共识协议的共享状态

> **兰波特谈分布式不变量：安全性、活性，以及谁拥有日志**
>
> 1987 年，莱斯利 · 兰波特这样描述分布式系统：在这种系统中，一台你甚至不知道其存在的计算机出了故障，就能让你自己的计算机无法使用。把它当作忠告来读，它要求我们清点自己依赖的计算机，并说明其中任何一台出故障时必须成立的性质。两类陈述就够了。安全性：坏事永远不会发生，例如节点永远不会依据一个没有任何已提交事务写入过的策略行事。活性：好事最终会发生，例如能连上数据库的节点最终会采用最新的策略。共识协议是同时获得这两者的一种优美方式。它也是一个需要长期维护的庞大系统；对防火墙来说，问题不在于共识好不好，而在于应该由谁来维护日志。

6.1 为什么用 SQL，为什么不自研共识协议

Sibuna 存储的东西不多：几百行策略、一张信誉表，以及运维人员也会导出到其他系统的取证事件。这些状态不需要专门定制的复制引擎。它需要的是一个可靠的 SQL 存储，单个节点用本地的，多个节点用共享的。因此，单个节点把状态保存在其数据目录中的一个嵌入式 SQLite 文件里（WAL 模式），集群则让每个节点都指向同一个 PostgreSQL 数据库[17]。

我们选择不在防火墙内部运行共识协议。共识引擎是一个要在防火墙之外另行维护的庞大系统，有它自己的格式和故障模式，还会给关停和每次升级增加工作。集群状态的可用性和复制改由 PostgreSQL 负责，借助其流复制和故障切换工具或托管服务来实现。我们认为这是更谦逊的选择：PostgreSQL 的复制经受过的部署远比我们的多，而 Sibuna 的写入速率（分批写入的事件和少有的策略变更）足够低，一个主库不会成为瓶颈。

存储也仍然是可选的。不使用 `--data-dir` 或 `--database-url` 时，守护进程是一个纯内存的防火墙，策略来自文件。

6.2 单一存储线程

一个存储线程拥有所有数据库连接；其他任何部分都不接触 SQL。请求线程从不等待它。请求线程把事件记录推入一个有 512 个槽的有界环形缓冲区（缓冲区满时丢弃记录并计数），并读取已发布的引擎。存储线程默认每 500 ms 运行一个节拍：在一个事务中写入最多 32 个排队的事件，读取策略修订版本，如果版本有变，就重建并发布引擎。重建时会在加载前后各读一次修订版本；如果期间有写者提交，就丢弃这个候选引擎，旧引擎继续生效。运维人员可以直接写策略表：触发器在写者的事务中推进修订版本；在 PostgreSQL 下，`NOTIFY` 让各节点立即执行节拍，而轮询仍然是最终的保证[18]。

集群中的每个节点都有一个互不相同的 `--cluster-node` 标识符，取值在 1 到 8,388,607 之间。挑战密钥与签发节点绑定，因此解答必须回到签发其挑战的节点；而会话令牌和要求票据在共享同一种子的每个节点上都有效。事件标识符的高位携带节点编号，每个节点都维护一个持久回执，回执与其批次在同一事务中推进，因此在结果不明的故障之后重放一个批次，不会把任何内容写两次。

> **不变量：存储层（SID 0011）。** **S1（唯一所有者）。** 所有 SQL 都在存储线程上运行；没有任何请求线程等待数据库。**S2（完整的引擎）。** 请求读取的每个引擎，在发布之前都已根据策略表的同一个一致修订版本完整构建。**L1（最终发布）。** 对策略表的一次已提交修改，会在下一个节拍之前到达每个能连上数据库的节点。**F1（平稳降级）。** 数据库宕机时，最后发布的引擎继续提供服务，事件在有界环形缓冲区中排队，缓冲区满后被丢弃并计数，存储线程则以退避方式重试。

6.3 带类型的模式

数据库模式用 SQL 表达程序类型所表达的内容，因此数据库会拒绝存储层无法表示的行。SQLite 表都是 `STRICT` 的；枚举列带有一个 `CHECK`，恰好列出程序会写入的名称；有界整数带有范围检查；一列为 `NOT NULL`，当且仅当其值永远不会缺失。每条语句都以参数绑定其值，因此不会通过加引号来拼接 SQL。控制台的存储遵循同样的规则：一次基线迁移，33

[16] `benchmarks/results/primitives-latest.md`；WebAssembly 模块的大小取自 `primitives-latest.json`，见《Sibuna 之书》第 VIII 部分，“资源占用”；浏览器求解时间取自 SID 0010，标准 3（Chromium 155，`cargo xtask browser`）；SHA 指令的运行时检测取自 SID 0006，修订说明（Parts A to C）。

[17] SID 0011，“Abstract”“Motivation and scope”和“Why PostgreSQL for clusters”；SID 0005 描述了早先的复制式设计。

[18] SID 0011，“The storage thread”“Policy revision”和“Incidents and campaigns”。

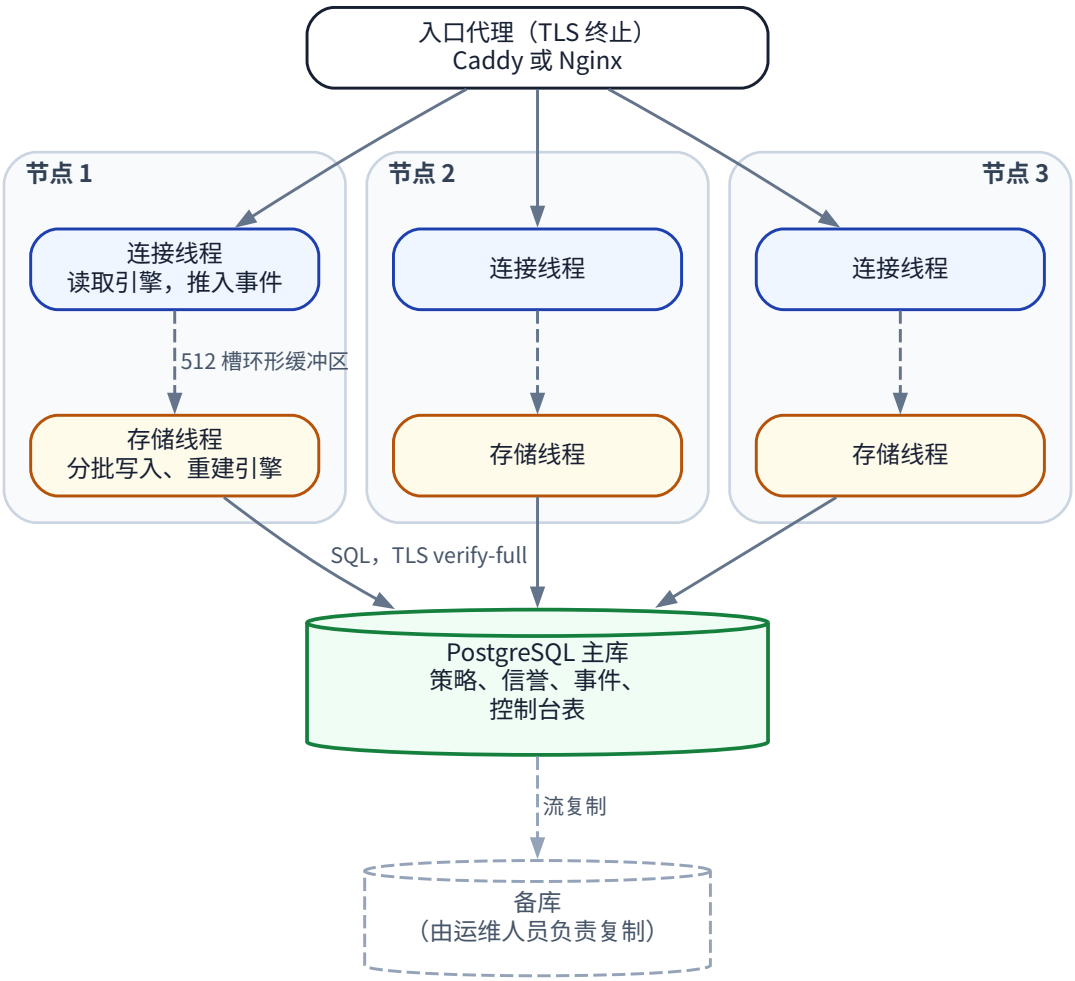


图 4 Sibuna 集群。每个节点都把请求路径保持在内存中，并把所有 SQL 交给一个存储线程；各节点共享一个 PostgreSQL 数据库，其复制和故障切换由 PostgreSQL 负责（SID 0011）。单个节点则改用一个嵌入式 SQLite 文件。

张带类型的表，没有触发器。控制台的每个不变量，例如“对令牌的每次修改都写入一行审计记录”，都恰好由一个函数保证，该函数在一个事务中运行其语句，在 SQLite 和 PostgreSQL 上都是如此[19]。

守护进程以 `postgres-tls` 特性构建时，到 PostgreSQL 的连接按 URL 的要求，用 libpq 的 `sslmode` 和 `sslrootcert` 加以保护。不带该特性的构建在启动时拒绝 `require`、`verify-ca` 和 `verify-full`，绝不会退回到明文连接。生产集群使用 `sslmode=verify-full`。看起来相似的事件会被归入攻击活动。每个事件携带一个 64 维的嵌入向量，以 256 字节存储。存储线程在内存中保存所有节点最近的 16,384 个嵌入向量（4 MiB）；当余弦距离不超过 0.35 时，把事件归入其最近邻所在的攻击活动。不与更早的事件做聚类是有意的取舍：攻击活动本质上是近期的，而这个上界让内存和工作量保持固定。

6.4 这一设计不做什么

我们应当坦白地说明这一选择的局限。Sibuna 不包含自研的共识协议，集群的一致性由 PostgreSQL 保证：失去数据库连接的节点会一直沿用最后的策略，直到重新连上。由于各节点共享同一个种子，而挑战密钥与签发节点绑定，在一个节点上签发的会话会被所有节点接受，而在一个节点上已花费的解答重放到另一个节点时会被拒绝；这两条性质都不需要数据库。SID 0011 的状态为 Committed：一项在同一 PostgreSQL 数据库上运行三个节点的功能测试表明，在一个节点上插入的策略会在一个节拍内被全部三个节点执行，一个节点上的蜜罐封禁会到达其他节点，事件标识符也从不冲突。目前还没有测量工具在负载下驱动多个节点共用一个 PostgreSQL 数据库，因此本文不报告任何集群测量结果。发布的可执行文件不含 PostgreSQL 支持；集群需要从源代码以 `postgres-tls` 特性构建 Sibuna[20]。

7 控制台

Sibuna 包含一个需要显式启用的管理控制台：带实时流量的认证仪表板，从一次拒绝追溯到其记录证据的调查路径，策略编辑，地址组，基于开放 GeoIP 数据的国家信息补充，使用 Argon2id 密码和 TOTP 第二因素的用户，审计记录，

[19]SID 0011，“Schema and migrations”及其审查历史（传输安全）；SID 0013，“Abstract”。
[20]SID 0011，“Security and operational considerations”和“Acceptance criteria and implementation status”（标准 2，`crates/storage/tests/multi_node.rs`）；`.github/release-notes.txt`，“Known limitations”；SID 0002，“Authentication and work”（与节点绑定的挑战密钥）；《Sibuna 之书》第 VIII 部分，“未测量的内容”。

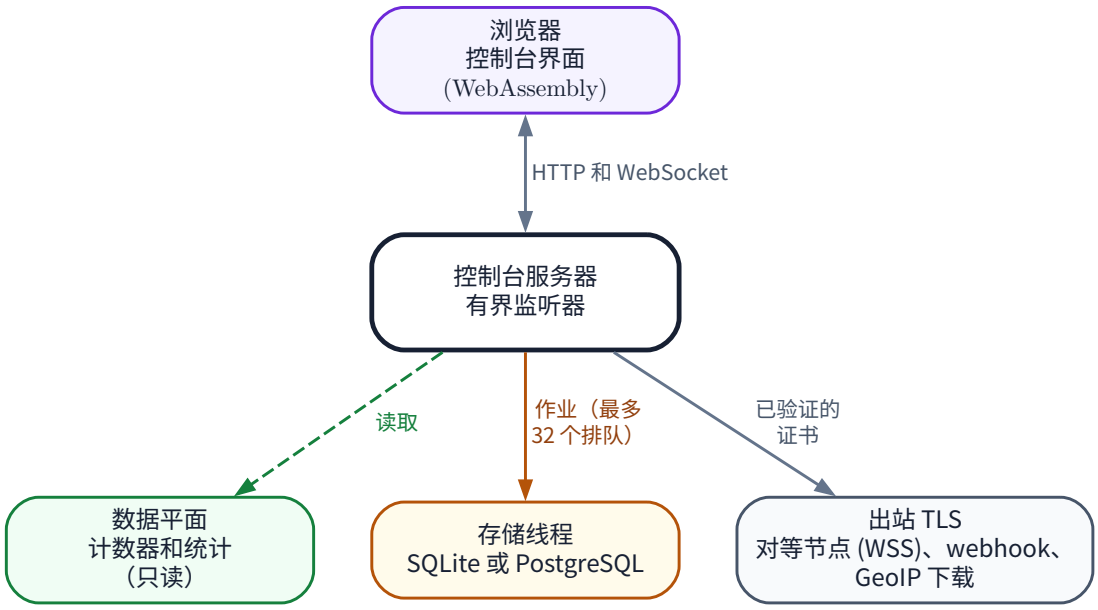


图 5 控制台。浏览器以 WebAssembly 模块的形式运行界面；控制台服务器读取数据平面的计数器，并把 SQL 交给存储线程；到对等节点、webhook 和 GeoIP 发布方的出站连接使用 TLS，并验证证书（SID 0007、0012、0013）。

webhook，以及集群管理。它编译进同一个可执行文件，界面以 WebAssembly 模块的形式在浏览器中运行[21]。

控制台最根本的约束是与请求路径隔离。它可以读取数据平面的计数器和数据库，也可以写数据库，但它从不进入请求线程，从不在请求线程上分配内存，也从不持有连接线程需要的锁。它的监听器最多接受 96 个连接，其中 16 个保留给 HTTP，64 个可用于浏览器流，16 个用于对等节点流。在 1.0.0-beta.3 中，完整的界面模块大小为 724,330 字节（gzip 压缩后 272,568 字节）。这低于构建过程强制执行的 768 KiB 上限，但高于 640 KiB 的审查阈值，SID 0007 已将其记录待审[22]。

控制台会为长期保持的对等节点流、webhook 和 GeoIP 下载建立出站 TLS 连接。证书依据系统根证书或运维人员提供的 CA 文件验证；主机名验证无法关闭；只有字面量回环地址才允许使用明文 HTTP；证书错误也绝不会退回到明文连接。Sibuna 仍然不终止任何入站 TLS，这仍是入口代理的职责。

SID 0007 规定了控制台的性能目标：启用控制台至多使数据平面吞吐量下降 1%，使 p99 延迟增加至多 10%。目前只有一项测量：无论控制台是编译进来但停用、只启用存储，还是启用但空闲，16 个完整产品工作负载与不含控制台的守护进程构建之间的差异都在约 1% 以内，处于噪声范围。在负载下运行八个活动仪表板的正式门槛测试尚未进行，因此我们不声称已达到该目标。一次运行八个实时仪表板的功能测试已经通过。

表 4 在一个带控制台的 SQLite 节点上运行八个流量仪表板，使用无头 Chromium 155（`benchmarks/results/console-8-dashboards-2026-10-15.md`，记录于 2026-10-15）。结果：PASS

指标	数值
数据平面请求	30.0 秒内 6,000 个（每秒 200 个，每个请求一个连接）
已应答	6,000: 200 x 1,200, 401 x 3,000, 403 x 600, 404 x 600, 500 x 600
错误（无回复）	0
延迟 p50 / p90 / p99 / 最大值	246 / 311 / 373 / 552 us（含建立连接）
守护进程的常驻内存：启动时、打开 8 个仪表板时、流量结束后	36,152 KiB; 38,028 KiB; 41,496 KiB
仪表板	3 个管理员会话和 5 个大屏显示；每个都收到了 134 到 139 个 `stats` 增量，最终总数相同，没有异常、控制台错误或 Content-Security-Policy 违规

[21]SID 0007，“Abstract”及其修订说明（监听器准入、界面模块、GeoIP 和密码）；SID 0010，审查历史（控制台移植、TOTP）；SID 0012；SID 0013。隔离测量记录在 SID 0010 的审查历史中（控制台移植）；《Sibuna 之书》第 VIII 部分“控制台开销”一节指出，该记录没有注明对应的结果文件。

[22]SID 0007，审查历史（Rust 移植、整合；守护进程集成），其中记录的 719,406 字节早于 SID 0014 加入的 JA4 字段。1.0.0-beta.3 的大小是按守护进程构建脚本的方式构建的模块（`cargo rustc --lib -p sibuna-console-ui --target wasm32-unknown-unknown --profile wasm -- -C link-arg=--compress-relocations`），再用 `gzip -9` 压缩后的大小。

8 实证评估

8.1 方法

三个通过 `cargo xtask bench` 运行的测量工具测量 Sibuna。原语 测试套件在一次不计时的预热之后，用七个计时批次为请求路径上的各项操作计时，在计时之外重建可变状态，并报告每次操作的批次中位数，旁边列出最快和最慢的批次。完整产品 测量工具用 `wrk`（2 个线程，64 个连接，5 秒，重复 3 次）驱动守护进程，覆盖 forward-auth 和反向代理两种模式、Gate 和 Shield 两种形态，以及四种工作负载：`admitted`（有效会话）、`challenged`（没有会话）、`allowed_static`（`/robots.txt`）和 `attack`（会话加上 `q=' OR 1=1--`，只有 Shield 会拒绝）。延迟 测量工具使用开环生成器，无论守护进程表现如何都以固定速率发送，并从每个请求的预定发送时刻开始计时，从而把守护进程中的排队时间计入守护进程[23]。

这些运行于 2026 年 10 月 15 日在一台主机上记录：AMD Ryzen 9 5950X（16 核 32 线程），运行 Linux 7.0.0-28。守护进程运行在 CPU 0 到 3 上，使用四个 worker；负载生成器和一个桩源站运行在 CPU 8 到 15 上（另一个 CCD）；原语测试套件运行在 CPU 4 上。守护进程是默认的 release 构建，编译进了 SQLite 和控制台，以纯内存方式运行，不使用存储。

测量范围. 每次运行都在一台主机上通过回环接口进行：没有 TLS，没有广域网，也没有真实的浏览器。测量工具不统计分配次数；无分配的请求路径改由稳态测试检查。它们运行未经修改的 release 可执行文件，以其 SHA-256 标识。在这台主机上的排名并不代表在另一台主机上的排名。

8.2 原语延迟

表 5 Sibuna 的原语耗时，单位为每次操作的纳秒数，在一个 Ryzen 9 5950X 核心上测得：七个批次的中位数，以及最快和最慢的批次（`benchmarks/results/primitives-latest.md`，生成于 2026-10-15T22:34:59Z）

子系统	工作负载	中位数	最小值-最大值
工作量证明	Hashcash 验证，16 比特	70.2	69.9-71.0
工作量证明	PoSW 验证，深度 13，16 次揭示	22,272.4	22,257.0-22,316.4
机器人匹配器	Aho-Corasick，40 条签名	55.7	55.3-57.2
机器人匹配器	顺序子串搜索，40 条签名（作为参照）	763.0	762.2-763.5
IP 分类器	IPv4 CIDR 查找	32.0	31.8-32.2
IP 分类器	IPv6 CIDR 查找	79.3	78.5-79.6
令牌	带密钥 BLAKE3 会话令牌	155.8	155.3-156.4
令牌	Ed25519 会话令牌	40,587.5	40,544.6-40,631.1
已用集合	Robin Hood 花费与查找	21.0	20.8-21.3
速率限制器	GCRA，已哈希的客户端	6.2	6.1-6.3
速率限制器	GCRA，四个规则作用域，已哈希的客户端	6.3	6.2-6.4
速率限制器	GCRA，客户端为原始字节（含带密钥哈希）	12.8	12.7-12.8
速率限制器	GCRA，四个规则作用域，客户端为原始字节	13.1	13.0-13.2
HTTP 解析器	零拷贝解析请求和 Cookie	132.5	132.3-133.2
策略引擎	完整分类，Shield 配置	790.8	787.2-793.1
策略引擎	分类，Gate 配置	75.9	75.6-76.1
WAF	8 KiB 请求体，语义扫描	12,784.3	12,771.5-12,809.2
WAF	8 KiB 请求体，SQL 注入设为审计	15,485.4	15,460.9-15,530.1

离散范围很窄：任何一行的任何批次与该行中位数的偏差都不超过 4.2%[24]。守护进程采用的是已哈希客户端的那几行，因为它对每个请求只计算一次客户端哈希；传入原始字节的那几行在每次检查中都包含这次带密钥哈希。计时器包围整个批次，从不包围单次操作，因此表中不报告每次操作的百分位。

[23] `benchmarks/results/README.md`，“Method”和“Provenance and limits”（rustc 1.99.0；守护进程为 `cargo build --release -p sibuna`，49,113,856 字节）；《Sibuna 之书》第 VIII 部分，“方法”。

[24] 《Sibuna 之书》第 VIII 部分，“原语延迟”，由 `primitives-latest.json` 计算得出。

8.3 完整产品

表 6 wrk 下的 Sibuna：每秒请求数、以微秒计的延迟、以微秒计的每请求 CPU 时间和峰值常驻内存；三次运行的中位数（benchmarks/results/product-latest.md，生成于 2026-10-15T23:15:24Z）

模式	形态	工作负载（状态码）	req/s	p50	p99	CPU	峰值 KiB
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
反向代理	Gate	admitted (200)	142,498	427	768	27.68	15,896
反向代理	Gate	challenged (200)	199,465	160	324	12.98	16,160
反向代理	Gate	allowed_static (200)	147,303	411	759	26.88	16,324
反向代理	Gate	attack (200)	143,900	422	745	27.67	16,348
反向代理	Shield	admitted (200)	138,965	438	857	28.57	16,064
反向代理	Shield	challenged (200)	195,188	162	328	13.73	16,328
反向代理	Shield	allowed_static (200)	145,673	415	746	27.27	16,328
反向代理	Shield	attack (403)	209,507	151	306	12.17	16,328

在 forward-auth 模式下，由入口代理询问 Sibuna 是否准入请求，没有源站参与；每种工作负载都在每秒 197,532 到 211,636 个请求之间，p99 为 305 到 327 us。此时守护进程的四个 CPU 并未饱和（只有 2.47 到 2.72 个处于忙碌状态），这说明上限来自负载生成器。在反向代理模式下，admitted 行和静态资源行包含桩源站和多出的一次回环转发，守护进程的 CPU 已经饱和（四个中有 3.95 到 3.98 个忙碌）；而由 Sibuna 自己应答的请求（挑战，以及 Shield 对攻击的拒绝）仍以每秒约 195,000 到 210,000 个的速度运行。在每种工作负载下，峰值常驻内存都保持在 15,896 到 16,348 KiB 之间[25]。

在三个反向代理行中，约两百万个应答里有少数几个是 503：wrk 在一个正在关闭的连接归还其在按地址连接限制下的名额之前就重新连接了。守护进程现在会在写出关闭响应之前归还名额；这些结果文件早于这一修复。

此后对 release 配置的一次修改去掉了内置 SQLite 的调试信息，据构建清单中的说明，约占 49 MB 中的 13 MB；结果文件此后没有重新生成，因此我们引用的是实测大小[26]。

8.4 相同施加负载下的延迟

wrk 始终保持 64 个请求在途，因此请求速率取决于守护进程，而且 wrk 的百分位遗漏了它在等待期间没有发送的请求。开环测量工具则固定施加速率，并把每一处延迟都计入守护进程，这正是访问者实际感受到的。

在 forward-auth 模式下，直到每秒 150,000 个请求，p99.9 都低于 75 us。在反向代理模式下，由于路径中有源站，每秒 100,000 个请求时 p99 为 100 us，p99.9 为 150 us。在反向代理模式下达到每秒 150,000 个请求时，守护进程已经

表 7 Sibuna 守护进程的资源占用（product-latest.json 和 primitives-latest.json，2026 年 10 月 15 日，按《Sibuna 之书》第 VIII 部分的列表）

指标	数值
守护进程可执行文件，带 SQLite 和控制台的默认构建	49,113,856 字节
空闲常驻内存，产品运行	5,592 到 5,788 KiB
工作负载期间的峰值常驻内存	15,896 到 16,348 KiB
工作负载之后的常驻内存	16,176 到 16,348 KiB
策略引擎表	209,052 字节
机器人签名表	42,244 字节
WAF 签名表	164,908 字节
浏览器证明器（WebAssembly）	14,744 字节

[25] benchmarks/results/product-latest.md；忙碌的 CPU 数取自 product-latest.json 中的 cores_busy_median；503 应答及其修复取自 benchmarks/results/README.md 和 SID 0002 的修订说明（缺陷修复）。

[26] 《Sibuna 之书》第 VIII 部分，“资源占用”；根目录的 Cargo.toml，配置段 [profile.release.package.libsqlite3-sys]。

表 8 Sibuna Shield 在固定施加速率下的已准入请求：以微秒计的延迟和每请求 CPU 时间，三次 10 秒运行的中位数（benchmarks/results/latency-latest.md，生成于 2026-10-15T23:07:14Z）

模式	施加 req/s	实际 req/s	p50	p90	p99	p99.9	错误	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
反向代理	10,000	10,000	56	61	65	76	0	32.25
反向代理	25,000	25,000	51	54	58	72	0	28.63
反向代理	50,000	50,000	47	50	60	74	0	27.45
反向代理	100,000	99,996	55	75	100	150	0	28.42
反向代理	150,000	134,344	583,934	1,544,160	1,906,273	1,967,134	4	29.22

超出其容量：它每秒完成 134,344 个请求，接近 wrk 对同一工作负载测得的 138,965 个，而延迟增长到秒级，这正是开环测试在超出容量时应当显示的。该行的四个错误发生在过载期间[27]。

8.5 测量带来的改变

延迟测量工具改变了连接线程的设计。在最初的几次运行中，64 个连接各自在 4,096 个请求后被守护进程关闭，每秒 100,000 个已准入请求时的第 99.9 百分位为 0.3 到 2 ms，这是 wrk 没有显示出的尾部。创建和结束一个线程会映射和解除映射它的信号栈，并归还它的栈页面，而这些地址空间的变化拖住了其他线程。一个实验就确认了诊断：在一个从不关闭连接、因而运行期间不创建任何线程的构建中，相同速率下的结果是 54 到 60 us。修复方法是让完成连接的线程停驻并接手下一个连接；停驻 10 秒的线程会结束，因此一次突发不会让它的栈一直驻留在内存中。修复之后，每秒 100,000 个请求时的 p99.9 在 forward-auth 模式下为 62 us，在反向代理模式下为 150 us[28]。

一种罕见的、周期性的卡顿，早在影响 p99 之前就会影响远端尾部。在每秒 100,000 个请求、64 个连接的条件下，每个连接持续 4096/1562.5 ≈ 2.6 秒，因此每秒约有 24 个线程启动和结束；在一次一百万个请求的运行中，最高的 0.1% 只有 1,000 个请求，每次线程更替延迟少数几个请求就足以填满它。要发现这样的缺陷，需要 p99.9 和对每个请求的计时。

8.6 要点

1. 分类不到一微秒。 在 Gate 配置下，策略引擎对一个浏览器请求的分类耗时 75.9 ns；在执行完整检查的 Shield 配置下耗时 790.8 ns。
2. 对称验证。 带密钥 BLAKE3 会话令牌的验证耗时 155.8 ns，Ed25519 令牌则为 40,587.5 ns：在同一信任域内获得同样的完整性，工作量约为后者的 1/260。
3. 占用小而稳定。 使用四个 CPU 时每秒约处理 200,000 个 forward-auth 请求，每种工作负载下的峰值常驻内存都在 16 MiB 左右，空闲时低于 6 MiB。
4. 尾部需要开环测量。 线程更替造成的卡顿在闭环测量工具下不可见，在开环生成器下的 p99.9 中却一目了然。

9 市场比较：Sibuna 及其同类产品

我们把 Sibuna 与运维人员最可能拿来与它比较的工具进行对照。其他产品的情况取自它们 2026 年 9 月的公开文档和发布制品，《Sibuna 之书》第 II 部分记录了这些情况及其来源[31]。这些产品都很好地服务着各自的用户，也都作出了一些设计选择，而这些选择的好处是我们的测量工具无法测量的。我们描述的是架构上的差异，而不是排名。

9.1 与 Anubis 的实测比较

我们用 完整产品 中的测量工具，把官方 Anubis v1.27.0 发布版可执行文件与 Sibuna 放在一起测量：相同的四种工作负载，每个产品的所有线程都使用相同的四个 CPU，相同的 wrk 负载，对等的 Hashcash 工作量（Sibuna 为 8 个零比特，Anubis 为 2 个十六进制零位），像浏览器那样求解各产品的挑战来获得会话，并以 caddy respond 作为源站[29]。这里 Sibuna 运行的是 Shield 形态，因此它还会检查每个载荷，而 Anubis 不做这件事。

[27] benchmarks/results/latency-latest.md；每秒 25,000 个请求的各行和每秒 150,000 个请求的反向代理行都在该文件中。
[28] SID 0002，修订说明（性能，复用连接线程）；《Sibuna 之书》第 VIII 部分，“测量带来的改变”；benchmarks/results/latency-latest.md。
[31] 《Sibuna 之书》第 II 部分，“功能比较”和“验证中间页与边缘网络”；Sibuna 的许可证是 Cargo.toml 中工作区的 license。
[29] benchmarks/results/comparison-latest.md 和 admission-latest.md，于 2026 年 10 月 16 日用 cargo xtask bench product --anubis 和 cargo xtask bench admission --anubis 记录；《Sibuna 之书》第 VIII 部分“Sibuna 与 Anubis”一节列出了方法以及两个产品之间的每一处差异。

表 9 功能比较（《Sibuna 之书》第 II 部分；其他产品的情况读自 2026 年 9 月的公开文档，JA4 一行为 2026 年 10 月）

功能	Sibuna 1.0.0-beta.3	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
运行形式	一个可执行文件；反向代理或 forward auth；可选的 SQLite 文件或共享的 PostgreSQL 数据库	一个 Go 可执行文件；反向代理或 forward auth	七个 Docker 容器（tengine、detector、mgt、luigi、fvm、chaos、PostgreSQL）	托管网络；DNS 指向 Cloudflare
运行时	原生代码，无垃圾回收器；关闭存储时没有第三方依赖	Go，带垃圾回收	C/C++ 代理和检测器，Go 管理端，PostgreSQL	专有边缘软件
许可证	LGPL-3.0-or-later	MIT	GPL-3.0 管理代码；闭源检测器镜像	专有服务
工作量证明准入	Hashcash（比特粒度）和 Cohen-Pietrzak 顺序工作量，按规则选择	以十六进制半字节计的 SHA-256 Hashcash；不涉及工作量的挑战	JavaScript 挑战和验证码；不受工作量约束	托管质询、JS 质询、Turnstile
服务器上的挑战状态	解出之前没有；已解出的标签保存在固定大小的 Robin Hood 表中	存储后端：内存、bbolt、Valkey 或 S3	由整套组件管理	由 Cloudflare 管理
会话令牌	带密钥 BLAKE3，75 个字符，与付出的工作绑定；可选 Ed25519	用 Ed25519 签名的 JSON Web Token；可选 HS512	由整套组件签发的 Cookie	`cf_clearance` Cookie
载荷检查	Shield：带标签的自动机加结构化分词器，检查请求体的前 8 KiB	无	覆盖 OWASP 各类别的语义分析	托管规则集（Pro 及以上）
TLS 指纹（JA4）	规则匹配受信任的入口代理转发的 JA4 值；Sibuna 不计算	文档未提及；策略可以引用 Anubis 计算的 HTTP 指纹 JA4H	在自身的 TLS 终止处计算；自定义规则和内置的拒绝列表	自定义规则和分析中的 JA3 与 JA4（Enterprise，需 Bot Management）
共享状态	一个 PostgreSQL 数据库，存放策略、信誉和取证数据	共享签名密钥；共享 Valkey 存储	每台主机一套组件	全球任播网络
TLS 终止	否（由入口代理负责）	否（由入口代理负责）	是	是

如何解读这些数据行。 这些数字描述的是一台机器上的四种工作负载，不代表所有部署，而且有几行并不是同类对比。在 forward-auth 模式下，Anubis 自己应答 `/robots.txt` 和攻击路径；Anubis 不检查载荷，因此 Shield 的拒绝做了更多的工作；挑战的引导过程对 Sibuna 是两个请求，对 Anubis 是一个页面；会话检查的差距在很大程度上可以由对称令牌解释，因为 Anubis 自己的 HS512 方案比其默认的 Ed25519 快 55%。Anubis 是一个被广泛使用且能力很强的工具，它的维护者出于充分的理由作出了不同的选择。这项测量真正表明的是：一条小而有界的请求路径，可以让准入对防御方来说成本很低。

在 forward-auth 模式下，Anubis 为每个已准入请求花费 106 us 的 CPU 时间，Sibuna 只花 13 us；Anubis 的峰值常驻内存在 forward-auth 模式下约为 32 MiB，在反向代理模式下增长到约 600 MiB，而 Sibuna 始终保持在 16 MiB 左右。以 `caddy respond` 为源站时，Sibuna 的代理行受限于源站，因为源站与负载生成器共用 CPU：换用 完整产品 中更轻量的桩源站时，同样的行可达到每秒约 139,000 个请求。

表 10 `wrk` 下的 Sibuna（Shield）与 Anubis 1.27.0：每秒请求数和以微秒计的 p99 延迟，三次五秒运行的中位数（`benchmarks/results/comparison-latest.md`）。除非另有标注，状态码均为 200；Shield 以 403 拒绝攻击。

模式	工作负载	Sibuna 请求/秒	Sibuna p99	Anubis 请求/秒	Anubis p99
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
反向代理	`admitted`	103,298	3,321	19,398	11,062
反向代理	`challenged`	197,327	328	28,842	9,546
反向代理	`allowed_static`	106,819	3,316	28,862	30,124
反向代理	`attack`	210,483 (403)	306	23,399	14,124

表 11 两个 CPU 上每秒的准入操作次数，七个批次的中位数（benchmarks/results/admission-latest.md）。Anubis 对其每种令牌方案各运行一次。

产品（令牌）	会话检查	未认证	引导	证明验证	常驻内存
Sibuna（带密钥 BLAKE3）	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

9.2 详细的架构比较

9.2.1 SafeLine（长亭科技）

SafeLine 把 libinjection 的语义思想带进了一个带管理控制台的完整产品；其厂商报告的检出率为 71.65%（平衡模式）和 76.17%（严格模式），误报率分别为 0.07% 和 0.22%[32]。这些是对检测质量的测量，而 Sibuna 尚未发布自身引擎的此类数据。

- 部署。 社区版以七个 Docker 容器的形式发布，运行在至少有一个 CPU 核心、1 GB 内存和 5 GB 磁盘的 Linux 主机上。它自己终止 TLS，Sibuna 则不这样做。
- 状态。 它把攻击日志保存在栈内的 PostgreSQL 中，每台主机一个栈。
- *Sibuna* 的不同之处。 Sibuna 是一个可执行文件，启动时可选地使用一个 SQLite 文件；集群共享一个运维人员可能已经在运行的 PostgreSQL 数据库。在产品测试中，它空闲时的常驻内存为 5,592 到 5,788 KiB。我们的主机上没有 Docker 守护进程，因此没有在其上运行 SafeLine，也就不作性能比较。

9.2.2 Anubis 与 Coraza

工作量证明验证中间页在 2025 年变得普遍：当时为训练语言模型而抓取网页的流量，使普通源站的运行成本变得高昂，而 Anubis 是这类方案中最知名的开源实现。它是一个能力很强、部署广泛的 Go 反向代理：提供一道 JavaScript 谜题，用 Ed25519 签名 JSON Web Token，并把挑战状态保存在一个存储中。Coraza 实现了 ModSecurity 规则语言（SecLang），运行 OWASP Core Rule Set；它是在这类网关旁边增加基于规则的检查的一种方式。

- 工作落在哪一方。 Anubis 和 Sibuna Gate 都让客户端在源站做任何事之前先付出代价。Sibuna 另外提供第二层顺序工作量，按规则选择，供需要更强门槛的路由使用，而且它的会话会记住付出了多少工作。
- 服务器记住什么。 Anubis 存储挑战；Sibuna 签发一条自认证记录，只在一张固定的表中记住已求解的标签（定理 3）。它的默认令牌是对称的，验证成本更低，量子攻击者对它只能获得平方级加速。
- 检查。 Sibuna Shield 在准入的同一遍处理中完成自己的结构化检查，因此一个进程、一个解析器同时承担两种角色。它不运行 OWASP Core Rule Set。我们评估过现有的 SecLang 引擎，包括 libmodsecurity 和 Coraza 的绑定，但一个都没有采用：边缘服务器上的检查引擎是安全关键代码，我们宁愿自己掌握并审查它。如果 Sibuna 将来增加对 Core Rule Set 的兼容，会先由一份设计记录规定所支持的子集[33]。
- 测量。 在同一台主机、同样的负载下，Sibuna 在 forward-auth 模式下应答的已准入请求是 Anubis 的五倍多，p99 更低，每个请求的 CPU 时间约为其八分之一（与 Anubis 的实测比较）。

9.2.3 云端边缘 WAF（Cloudflare 等）

托管边缘网络把检查、速率限制和机器人评分放在众多的接入点中，并配有全局控制平面。它们能吸收任何单台服务器都无法承受的大流量攻击，也不需要运维人员提供硬件。Sibuna 的集群部署借鉴了它们的形态，即由共享的策略平面向各处本地的不可变快照供给策略，同时保持自托管。

- 数据驻留。 托管 WAF 会终止运维人员的 TLS，并在共享的基础设施上看到请求载荷，这可能与数据驻留要求相冲突。
- 套餐与配额。 功能与套餐挂钩：根据 Cloudflare 公布的配额，Free、Pro、Business 和 Enterprise 套餐分别允许 5、20、100 和 1,000 条自定义规则，以及 1、2、5 和 100 条速率限制规则。
- *Sibuna* 的不同之处。 Sibuna 运行在运维人员自己的机器上，没有按请求计算的费用，它的策略变更会在下一个存储节拍（默认 500 ms）到达每个节点。它不缓解大流量网络攻击，不用训练好的模型给机器人打分，也不终止 TLS；需要这些功能的部署，仍然要在它前面放置入口代理或边缘网络。

10 实现：小型 crate 与少量依赖

Sibuna 用 Rust 编写，是一个由若干小型 crate 组成的工作区，依赖关系只朝一个方向：先是可复用的库，然后是配置和诊断的核心，再是应用库，最后是守护进程和浏览器证明器。有三项工程选择对运维人员很重要，我们在此简要记录[34]。

少量依赖。 我们把每个第三方库都视为一项负担：它必须经过审查、持续更新并值得信任。数据平面不使用任何第三方库。哈希、签名、base64、JSON、DEFLATE、Aho-Corasick 和 HTTP/1.1 解析器都是我们自己的小型库，其他项目也可以单独使用。只有当一个依赖成熟、自成一体并且解决的确实是一个大问题时，才会被采用，采用的理由写在构建清单和一份设计记录中。目前采用了两个依赖，各自限定在一个库内：sqlx 用于 SQLite 和 PostgreSQL 存储，带来约一

[32]《Sibuna 之书》第 VIII 部分，“未测量的内容”：厂商报告的数字，于 2026 年 9 月核对。
[33]SID 0010，“The OWASP Core Rule Set”。
[34]SID 0010，“Workspace”“Dependency policy”和“Engineering standards”。

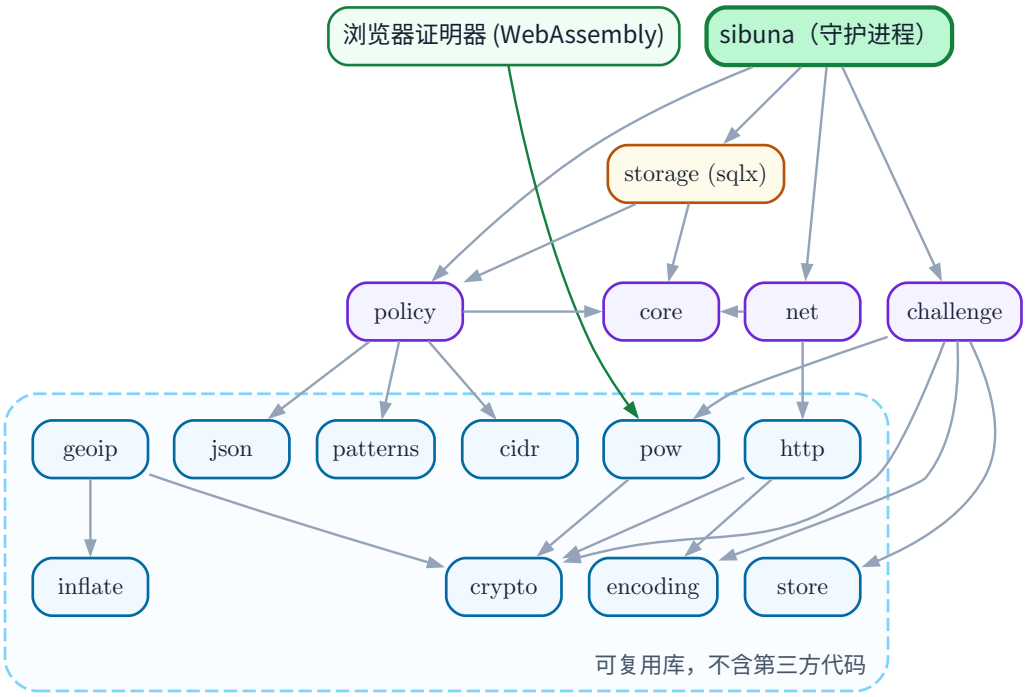


图 6 Sibuna 的库，经过简化（据 SID 0010）。箭头从一个库指向它所使用的库。可复用库不依赖任何应用库，只有存储库和控制台库会引入第三方代码。

百个传递依赖 crate[35]；`rustls` 用于控制台的出站 TLS 和 PostgreSQL 的 TLS，运行时涉及十三个 crate，因为自行编写的 TLS 客户端会是一个比 Sibuna 的数据平面还大的安全关键项目[36]。不带存储和控制台的构建完全没有第三方依赖。内存安全。 绕过编译器内存安全检查的代码默认被禁止；需要这类代码的模块必须按名称显式启用，并在注释中说明每一处用法的理由。只有少数几个模块这样做：使用处理器 SHA 指令的路径、一处擦除密钥的写操作、为处理信号而声明的几个 C 库函数、两个 WebAssembly 模块的导出，以及把事件送往存储线程的环形缓冲区。环形缓冲区是我们出于速度而保留这类代码的唯一地方，而且是在测量之后才这样做的：在 16 个生产者的情况下，标准库的有界通道每次推入的成本要高出 20% 到 45%[37]。在其他地方，所有权完成了原本需要手写内存管理才能完成的工作：已解析的请求不能比它的缓冲区活得更久，引擎随最后一个读者一起释放，只要还有线程能访问某个套接字，它就不能被关闭。规范与测试。 一个函数体最多 70 个逻辑行，一个文件最多 1,408 行，由检查工具强制执行。库代码返回结构化的错误，从不打印输出；用户看到的每个错误都会指出其位置，展示一段摘录，并给出提示。1.0.0-beta.3 的工作区测试套件在 SQLite 上运行约 2,200 个测试，另有约 700 个针对 PostgreSQL 的测试；控制台界面有 298 个测试，其中包括黄金样例；浏览器检查驱动无头 Chromium 走过验证中间页和控制台的每个页面[30]。

11 局限与待完成的工作

我们列出本文没有证明的内容，以免任何结论被解读得超出其证据所及。

- 只实测了一个同类产品。 我们在一台主机上测量了 Anubis（与 Anubis 的实测比较）。SafeLine 和托管 WAF 都没有测量。
- 没有负载下的集群测试。 一项功能测试在同一个 PostgreSQL 数据库上运行三个节点（SID 0011，状态为 Committed），但还没有测量工具在负载下驱动多个节点。
- *PostgreSQL* 需要从源代码构建。 发布的可执行文件带有 SQLite 存储和控制台，但不支持 PostgreSQL；集群需要以 `postgres-tls` 特性构建 Sibuna。
- 控制台的隔离门槛测试，即在负载下运行八个活动仪表板，尚未进行（控制台）。
- 只测了回环接口。 没有任何测量涵盖 TLS、广域网或负载下的真实浏览器；部署者应当测量自己的路径。
- 有界的检查。Shield 是一个只检查请求体前 8 KiB 的启发式检查器，而不是完整的 SQL 或 HTML 解析器；forward-auth 模式下的检查只能看到入口代理发来的字节。
- 协议范围。 Sibuna 支持 HTTP/1.0 和 HTTP/1.1；它不支持 HTTP/2，也不终止入站 TLS。

[35]SID 0011，“Why sqlx”；`Cargo.toml`，其中每个第三方 crate 都注明了采用理由。
[36]SID 0012，“Specification”“Rationale and alternatives”和审查历史（已构建 PostgreSQL TLS）。
[37]SID 0010，审查历史（安全审查，2026-10-15）：标准库有界通道与 Vyukov 环形缓冲区的对比，`tests/timing.rs`。按名称启用的模块是 `crates/crypto/src/sha256/`（x86-64 和 AArch64）、`crates/crypto/src/ct.rs`、`cmd/sibuna/src/signals.rs`、`cmd/sibuna-pow-wasm/src/lib.rs`、`cmd/console-ui/src/abi.rs` 和 `crates/store/src/ring.rs`；工作区设置了 `unsafe_code = "deny"` 和 Clippy 的 `undocumented_unsafe_blocks = "deny"`。
[30]SID 0010，“Acceptance criteria and implementation status”（标准 1、3 和 5），其中记录了 SID 0014 之前约 2,100 个工作区测试和 290 个界面测试；1.0.0-beta.3 的数字取自 `cargo test --workspace -- --list`。

- *TLS* 指纹来自入口代理。 Sibuna 自己无法计算 JA4。指纹的可信程度取决于写入它的入口代理，而该入口代理必须覆盖客户端发来的任何副本。客户端可以借助专门的库模仿浏览器的握手，因此 JA4 提高了自动化的成本，但并不能证明身份[11]。
- 密码学保证。 顺序工作层依赖于随机预言机模型下已发表的证明；我们的具体协议有针对参考标签计算器的测试，但没有经过独立的密码学审计。
- 只有一台机器。 每个数字都是在一台 AMD Ryzen 9 5950X 上测得的。在这台主机上的排名并不代表在另一台主机上的排名。

12 结论

Web 防御往往是层层堆砌起来的：解释型运行时、容器、正则表达式规则集和外部存储，一层层堆在应用前面。在 Sibuna 中，我们尝试了相反的方向，由几条朴素的思想指引：

- 让发送方用可验证的工作为准入付费，让服务器的检查保持廉价和无状态，并让会话记住它付出了多少工作；
- 在准入的同一遍处理中，以线性时间按结构进行检查，使任何会话都无法绕过检查；
- 尊重机器：每个连接一块缓冲区，请求路径上零分配，表有界且采用带密钥的放置，线程被复用而不是反复重建；
- 让成熟的数据库掌管持久的共享状态，由一个请求路径从不等待的存储线程居中访问。

在一台 16 核机器上，这些选择使 Sibuna 在 forward-auth 模式下用四个 CPU 每秒处理超过 200,000 个已准入请求，在该负载下常驻内存约为 16 MiB，在每秒 100,000 个请求时 p99.9 为 62 us。这些是一台主机通过回环接口测得的数字，我们把它们作为设计自洽的证据，而不是作为排名。我们去掉了自研的共识协议，并坦白地说明这一点；作为回报，系统变得更小，其数据平面也没有任何第三方依赖。还有许多内容有待测量，我们已尽力把它们确切地说清楚。

13 来源与复现

本文中的每个数字都来自 Sibuna 仓库中一个已提交的结果文件、一份 Sibuna 讨论（SID）记录或《Sibuna 之书》，每张表或每个脚注都注明了其来源。结果文件由 `cargo xtask bench primitives`、`cargo xtask bench product` 和 `cargo xtask bench latency` 生成，`benchmarks/results/README.md` 给出了命令、CPU 集合和每个文件的来历。

- `benchmarks/results/primitives-latest.md`、`product-latest.md`、`latency-latest.md` 及其 `.json` 文件；`console-8-dashboards-2026-10-15.md`。
- SID 0002（基础架构及其修订说明）、SID 0004（Shield 形态）、SID 0006（数学基础）、SID 0007（控制台）、SID 0010（工作区、crate 与依赖策略）、SID 0011（通过 sqlx 实现的存储）、SID 0012（控制台 TLS）、SID 0013（控制台存储）和 SID 0014（来自入口代理的 JA4 指纹），位于 `docs/sid/records/`。
- 《Sibuna 之书》，`docs/book/`：第 I 部分（一次请求的成本）、第 II 部分（前人工作）、第 III 部分（密码学）、第 IV 部分（协议）、第 V 部分（零分配工程）、第 VI 部分（自动机与语义防火墙）、第 VIII 部分（实证评估）和第 IX 部分（运维）。
- 原始文献：Back，Hashcash（1997）；Cohen 和 Pietrzak，Simple Proofs of Sequential Work（EUROCRYPT 2018）；Blocki、Lee 和 Zhou，On the Security of Proofs of Sequential Work in a Post-Quantum World（ITC 2021）；Aho 和 Corasick，Efficient String Matching（1975）；Celis，Robin Hood Hashing（1986）；ATM Forum 的 Generic Cell Rate Algorithm（1996）。

Sibuna 白皮书 | 2026 年 11 月 | 版本 1.0.0-beta.3
规范、源代码与基准测试： https://github.com/insanai/sibuna