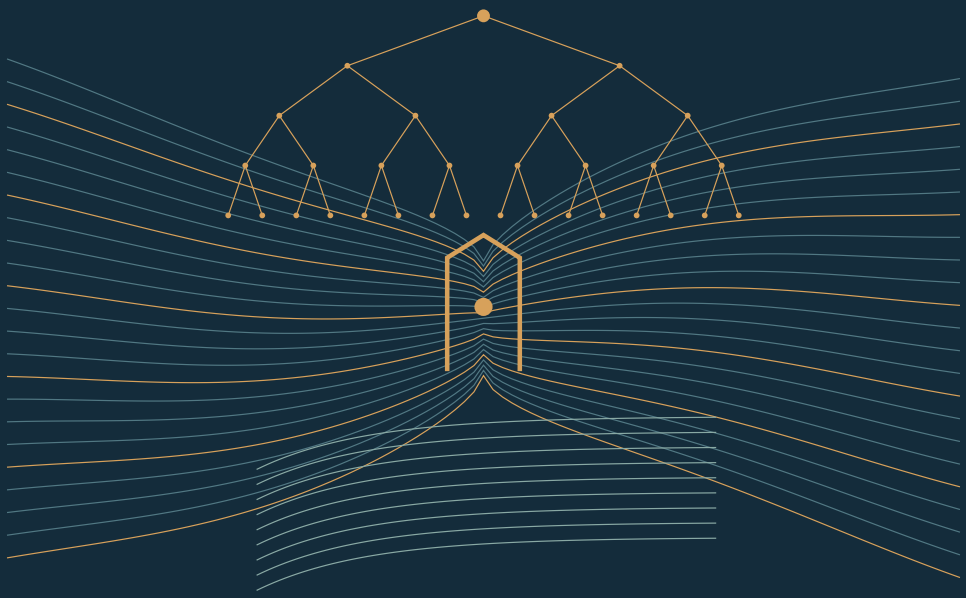
技术手册

SIBUNA

Web 防御的数学与工程



工作量 · 信任 · 有界状态

Vikrant Rathore 著，Ronak Rathore 与 Kanak Rathore 协助

版本 1.0.0-beta.3 · 11 月 13, 2026

目录

前言 **6**

如何阅读本书 **7**

三条阅读路径 7

约定 7

第 I 部分：一次请求的成本 **9**

准入是一个经济决策 9

工作量证明能证明什么 9

把哈希搜索看作随机变量 10

会话分摊工作 10

有界状态是第二种预算 11

我们将贯穿全书的不变量 11

第 II 部分：前人工作与设计空间 **12**

Sibuna 的来历 12

一个可执行文件中的三个思想 14

功能比较 14

第 III 部分：来自密码学研究的基础 **19**

Hashcash：第一层的数学 19

顺序工作量证明：第二层 20

对称认证：挑战与会话 23

第 IV 部分：Sibuna 协议 **26**

挑战的往返过程 26

无状态挑战记录 27

验证顺序与一次性使用 29

会话 Cookie 30

第 V 部分：Rust 中的零分配工程 **32**

连接循环 32

无分配的并发状态 42

第 VI 部分：自动机、字典树与语义防火墙 **51**

单遍多模式匹配 51

IPv4 和 IPv6 的基数字典树 53

语义防火墙 53

声明式策略引擎 55

第 VII 部分：浏览器端的工作量证明引擎 **57**

一份源码，两个目标 57

Worker 协议 60

校准 61

验证中间页 62

第 VIII 部分：实证评估 **63**
方法 63
原语延迟 67
完整产品比较 70
相同施加负载下的延迟 72
Sibuna 与 Anubis 74
控制台开销 77
未测量的内容 78
第 IX 部分：运维与部署 **80**
形态与配置 80
控制台预览 83
部署拓扑 99
声明式策略 105
持久存储与集群 105
打包 109
第 X 部分：案头参考与诊断 **113**
端点 113
指标 114
状态码 115
错误目录 116
策略模式 117
存储表 118
构建目标 118
第 XI 部分：速查卡 **120**
构建 120
运行 120
控制台（可选启用的预览功能） 120
最先需要关注的选项 121
`/__sibuna/` 下的端点 121
状态码 122
请求头与响应头 122
策略文件骨架 123
用 SQL 操作存储（Edge） 123
诊断 123
需要记住的数字 124
入口路由 124
部分练习解答 **125**
第 I 部分：成本与概率 125
第 II 部分：前人工作 125
第 III 部分：密码学 125
第 IV 部分：协议 126
第 V 部分：状态与顺序 126

第 VI 部分：算法 . . . 126
第 VII 部分：浏览器引擎 . . . 126
第 VIII 部分：评估 . . . 126
第 IX 部分：运维 . . . 127
第 X 部分：参考 . . . 127
继续探究 . . . 127
术语表 . . . **128**
参考文献 . . . **131**
索引 . . . **132**

© 2026, Vikrant Rathore 与 Sibuna 贡献者

Web 防御的数学与工程。

前言

一个请求到达时，带着一个方法、一个路径、一些请求头，以及对服务器注意力的索取。这些字节中没有任何一个能证明值得为它提供服务。Sibuna 在请求和源站之间放置一个决策：准入、拒绝，或者要求发送方完成可验证的工作。本书沿着这一决策，从数学模型一直讲到内存中的字节。

系统有两种形态。*Gate* 提供工作量证明准入和本地洪泛防护。*Shield* 增加应用检查和策略决策。两者都可以使用持久存储来保存策略、信誉和事件历史，存储可以位于单个节点本地，也可以由集群共享；我们把这种部署称为 *Edge*。有效会话确立准入，但不会让请求免于检查。在集群中共享持久状态会分发这些状态，但不会把本地速率计数器变成全局配额。

核心的工程问题不是哈希是否够快，而是当不可信客户端选择输入时，完整的处理路径是否仍然有界。解析器最多检查多少字节？一次挑战最多消耗多少内存？一个客户端最多能保持多少个打开的连接？哪些写入可以重试？节点或数据库消失之后，还有哪些功能可用？每个回答都必须同时说明一个不变量及其边界。

数学服务于同一目的。期望成本不是延迟保证。针对固定承诺的抽样论证，并不是能抵御所有自适应证明者的证明。队列可以吸收突发流量，却无法弥补一个永久更慢的消费者。我们推导小模型，手工演算示例，然后追问实现在哪些地方偏离了这些模型。

本书有三种用法。按顺序阅读时，第 I 至 VII 部分是一门课程：每章开头说明学完之后我们应当能做什么，接着手工演算一个示例，最后给出一道练习和一个需要用自己的话解释的不变量；部分参考解答放在书末。第 VIII 至 X 部分是参考资料：附带来源的测量结果，每个选项和拓扑，每个端点、状态、错误和表。第 XI 部分是一份两页的终端速查卡。全书以术语表和参考文献结尾。

第 II 部分逐项比较 Sibuna 与 Anubis、SafeLine 和 Cloudflare WAF 的功能；第 VIII 部分给出 Sibuna 自己的测量结果，从单个原语到整个产品的运行，并在同一负载生成器下把 Sibuna 与 Anubis 作为完整产品进行比较。两部分的写法都便于读者核查结论：比较列出了来源，每项测量都注明了数据所取自的已提交结果文件。

读者需要了解基础编程、对数和条件概率。本书通过所有权和数据布局来介绍 Rust，而不是概览整门语言。各章可以按顺序阅读：成本模型引出协议，协议决定状态，状态又决定并发与存储设计。后面的章节把这些不变量转化为测量方法和运维流程。

这是一本讲实现的书，并不声称计算谜题能够消除自动化流量。客户端可以购买算力，地址可以共享，应用语法也比任何有界检测器所能覆盖的更丰富。有用的成果是这样一个系统：它的成本和局限可以被审视、测试和修改，而不必藏在口号背后。

适用于 *Sibuna 1.0.0-beta.3* 的版本，*2026* 年 *11* 月

源代码、练习和可复现的测量结果随 Sibuna 仓库提供。

如何阅读本书

我们从一个问题开始：服务器此刻能做出的、成本最低的安全决策是什么？在阅读协议和实现章节时，我们始终把这个问题放在身边。一个成本低廉却准入了错误请求的操作是缺陷；一个正确却成本无界的操作是另一种缺陷。

问题	在哪些部分展开
谜题换来了什么?	第 I 至 III 部分：成本、概率和证明的局限。
Sibuna 与其他方案相比如何?	第 II 部分：技术源流与功能表；第 VIII 部分：测量。
会话意味着什么?	第 IV 部分：绑定、验证顺序、过期和重放。
内存用在了哪里?	第 V 和 VI 部分：缓冲区、线程、表、自动机和发布。
浏览器怎样解题?	第 VII 部分：worker 消息、内存和校准。
我们怎样知道它有效?	第 VIII 部分：三套测量工具以及各自不包含的内容。
我们怎样运行它?	第 IX 部分：选项、拓扑、策略、存储、集群和打包。
那个错误是什么意思?	第 X 部分：端点、状态码、错误和数据库模式；第 XI 部分：速查卡。

三条阅读路径

- 学习设计。按顺序阅读第 I 至 IV 部分，然后阅读第 VI 部分的逐步追踪，再阅读第 VII 部分。先完成练习，之后再与书末的解答比较。当追踪涉及共享表或借用切片时，回到第 V 部分。
- 运维部署。 先读第 IX 部分，再读第 XI 部分，同时打开第 X 部分以便查阅。选择形态之前，阅读第 II 部分的“功能比较”；选择 worker 数量和连接限制之前，阅读第 VIII 部分的“完整产品比较”。
- 修改代码。 阅读第 V 和 VI 部分、每章中的源码定位，以及它们引用的 SID 记录。第 VIII 部分说明哪套测量工具能发现我们所改动部分的性能退化；第 II 部分列出修改必须保持的约束。

约定

实例分析 展示中间状态，而不只是答案。练习 要求我们改变一个假设。提示给出第一步；书末的部分解答使推理可以核查。图示区分请求路径和后台工作。当正文与代码看起来不一致时，源码定位指出应当查看的实现。标题为“实现”的框给出一个函数及其所在文件；标题为“解释这个不变量”的框要求我们用自己的话把这个思想讲出来。

公式中，b 是以比特计的 Hashcash 难度，n 是顺序工作量的深度，t 是揭示的数量，p 是每次尝试的成功概率，K 是尝试次数，T 是速率限制器的发射间隔，τ 是它的突发容限，Q 是队列容量。除非另有说明，符号只在所在小节内有效。第 VIII 部分中的纳秒数和每秒请求数是来自某次具名运行的测量结果。实例分析中的数字是选定的输入，不是基准测试结论。统计模型先说明假设，再得出结论。

第 I 部分

一次请求的成本

选择算法之前，先确定它要保护的资源。本章建立几个小模型，后面的每个设计决策都以它们为衡量标准：源站工作、谜题工作、会话分摊，以及不可信客户端能让服务器占用的内存。

学习目标 学完本章，我们应当能够写出判断网关是否值得部署的不等式，说明工作量证明能证明什么、不能证明什么，推导哈希搜索的期望与尾部，解释会话如何分摊工作，并列出本书其余部分所维护的五个不变量。

准入是一个经济决策

假设源站为每个获准的请求执行一次数据库查询并渲染一个页面。令 c_o 表示这项工作，c_g 表示网关处理每个请求的工作，r 表示到达率。没有网关时，源站每秒必须承担 rc_o 个单位的工作。准入比例为 a 的网关把这一需求变为 rac_o，而自身消耗 rc_g。只有当节省下来的源站工作超过网关自身的成本以及它给合法访问者带来的成本时，网关才有用。

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

这是一个容量模型，不是测量结果。它省略了带宽、连接状态、存储和客户端延迟。它的价值在于每个缺失的项都清晰可见。缓存命中可能使 c_o 很小；代价高昂的查询可能使它很大。请求方与服务器之间不存在通用的成本比例。

实例分析：网关何时值得部署 取 $c_o = 100$ 个工作单位，$c_g = 1$。如果一半请求被拒绝，带网关的系统每次到达消耗 $1 + 0.5 \times 100 = 51$ 个单位，而不是 100。如果只有千分之一的请求被拒绝，它消耗 100.9：在这个模型中，网关的成本超过了它节省的成本。这些选定的数字不是耗时。它们说明了为什么每一项性能结论都必须交代工作负载。

工作量证明能证明什么

客户端谜题证明的是：某个人或程序找到了一个满足公开验证规则的输入。它不能证明对方是人、对方的身份或良好意图。请求方可以租用算力，在会话有效期内复用有效会话，或者把工作分散到多台机器上。谜题改变的是准入成本；获准的请求可以做什么，仍然由策略和检查决定。基于地址的限制与之互补。它们约束单个地址，但一个共享地址可能代表很多人，一个请求方也可以使用很多地址。谜题和地址都不是可靠的身份判据。因此，Sibuna 把准入、检查和信誉作为彼此独立的决策。

把哈希搜索看作随机变量

对于理想的 256 位摘要，要求 b 个前导零比特，则每次独立尝试的成功概率为 $p = 2^{-b}$。令 K 表示直到首次成功（含）为止的尝试次数。于是

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

期望可由尾部求和得到：

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

难度每增加一级，期望尝试次数翻倍。但这并不保证每道谜题的耗时都翻倍。有些谜题第一次尝试就成功；有些的耗时远超平均值。只测量一次求解的客户端，测到的主要是这种随机性。

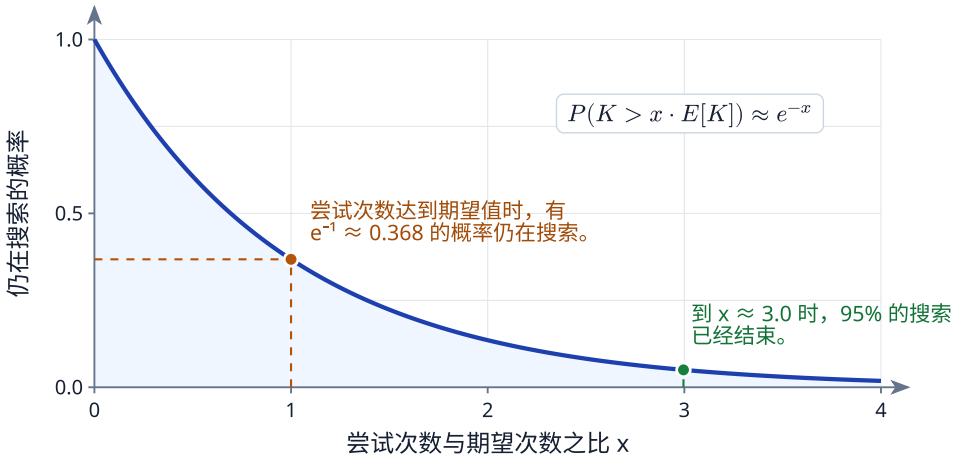


图 1 哈希搜索的生存概率：在大工作量近似 $P(K > x/p) \approx e^{-x}$ 下，尝试次数达到期望次数的 x 倍之后，搜索仍在进行的概率。

实例分析：期望不是截止时间 当 $b = 16$ 时，期望工作量为 65,536 次哈希。尝试这么多次之后仍在搜索的概率约为 $e^{-1} = 0.368$。第 95 百分位的尝试次数为 $\lceil \ln(0.05) / \ln(1 - 2^{-16}) \rceil \approx 196327$，接近平均值的三倍。用户界面应当容忍这种离散，而不是在到达期望完成时间时就宣告失败。

会话分摊工作

假设一个会话在过期前允许 m 个请求，一道谜题的成本为 c_p，验证成本为 c_v，一次会话检查的成本为 c_s。忽略失败的尝试，网关分摊到每个请求的成本为 $c_v/m + c_s$，而客户端的谜题成本为 c_p/m。延长会话有效期对真人和自动化客户端同样有利。选择有效期是一个策略决策，而不是密码学优化。

练习 1.1. 一个客户端每个会话可以发出 100 个请求，并完成一道需要一百万次尝试的谜题。分摊到每个请求的工作量是多少？如果它与第二个进程共享这个会话，会有什么变化？
提示：区分记账模型与令牌实际的绑定。

有界状态是第二种预算

把工作移出请求路径并不会让它消失。设事件以速率 λ 到达，存储线程以平均速率 μ 持久化它们，队列可容纳 Q 条记录。当 $\lambda > \mu$ 时，流体近似给出

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

更大的队列换来的是时间，而不是吞吐量。批处理分摊了事务成本，但被保留的记录仍然需要内存，最终也需要处理。过载时，Sibuna 丢弃新的事件记录并对其计数，同时继续依据内存中的状态作出请求决策。运维人员既要监控 HTTP 成功率，也要监控丢失的证据。

练习 1.2. 一个队列可容纳 512 条记录。到达率为 1,000/s，排空速率为 600/s。一个初始为空的队列能吸收多长时间的超额流量？为什么这个答案只是近似值？
提示：先用两个速率之差计算，再考虑突发和批量提交。

我们将贯穿全书的不变量

1. 不可信的请求不能创建无界的服务器状态。
2. 有效的准入令牌不能绕过应用层的拒绝。
3. 只要还有从缓冲区借出的切片，该缓冲区就保持存活。
4. 持久化重试不能让一个事件或它对信誉的影响成倍增加。
5. 已发布的策略快照在最后一个读者释放它之前保持不可变。

本书其余部分推导使这些陈述成立的机制，以及能够揭示违反情况的测试。只有在这些陈述保持成立时，快速路径才有用。

解释这个不变量. 用文字表述网关不等式 $c_g < (1 - a)c_o$，然后解释为什么仅凭测得的验证延迟，运维人员无法判断这个网关是否值得运行。

第 II 部分

前人工作与设计空间

我们把 *Sibuna* 放到它所借鉴的系统之中：密码学文献中的客户端谜题和顺序工作量，基于签名和基于语义的应用防火墙，工作量证明验证中间页，以及托管的边缘网络。然后，我们逐项比较运维人员最可能相互权衡的四种工具，并记录哪些事实经过了核查。

Sibuna 的来历

> **学习目标** 学完本章，我们应当能够追溯客户端谜题从 1992 年到 2018 年顺序工作量构造的发展脉络，说出 Sibuna 结合的三个架构思想，把 Sibuna、Anubis、SafeLine 和 Cloudflare WAF 放进同一个设计空间，并说明哪些比较结论经过了验证、哪些只是从文档中读到的。

客户端谜题与顺序工作量

Dwork 和 Naor 的 “Pricing via Processing”（CRYPTO 1992）提出：服务在为请求方工作之前，先要求对方完成一项难度适中、易于检查的计算。Back 的 Hashcash（1997 年提出，2002 年成文）把谜题变成部分哈希求逆，其难度由一个整数决定；Juels 和 Brainard 的客户端谜题（NDSS 1999）把这一思想用于连接洪泛，此时服务器不能为未解出的谜题记住任何东西。三者共有 Sibuna 保留的形态：由服务器选定的陈述，成本可调的解答，以及比求解便宜几个数量级的验证。

哈希搜索是易并行的，因此核心更多的请求方完成得更快。Mahmoody、Moran 和 Vadhan（CRYPTO 2013）定义了 顺序 工作量证明：无论证明者有多少核心，都必须执行一长串相互依赖的步骤；Cohen 和 Pietrzak（EUROCRYPT 2018）给出了简单的哈希图构造，其验证者只需 $O(t \log N)$ 的工作量。Blocki、Lee 和 Zhou（ITC 2021）在量子随机预言机模型中证明了同一构造对量子证明者也是可靠的。第 III 部分从这一系列工作推导出 Sibuna 的第二层；SID 0006 记录了相关假设。

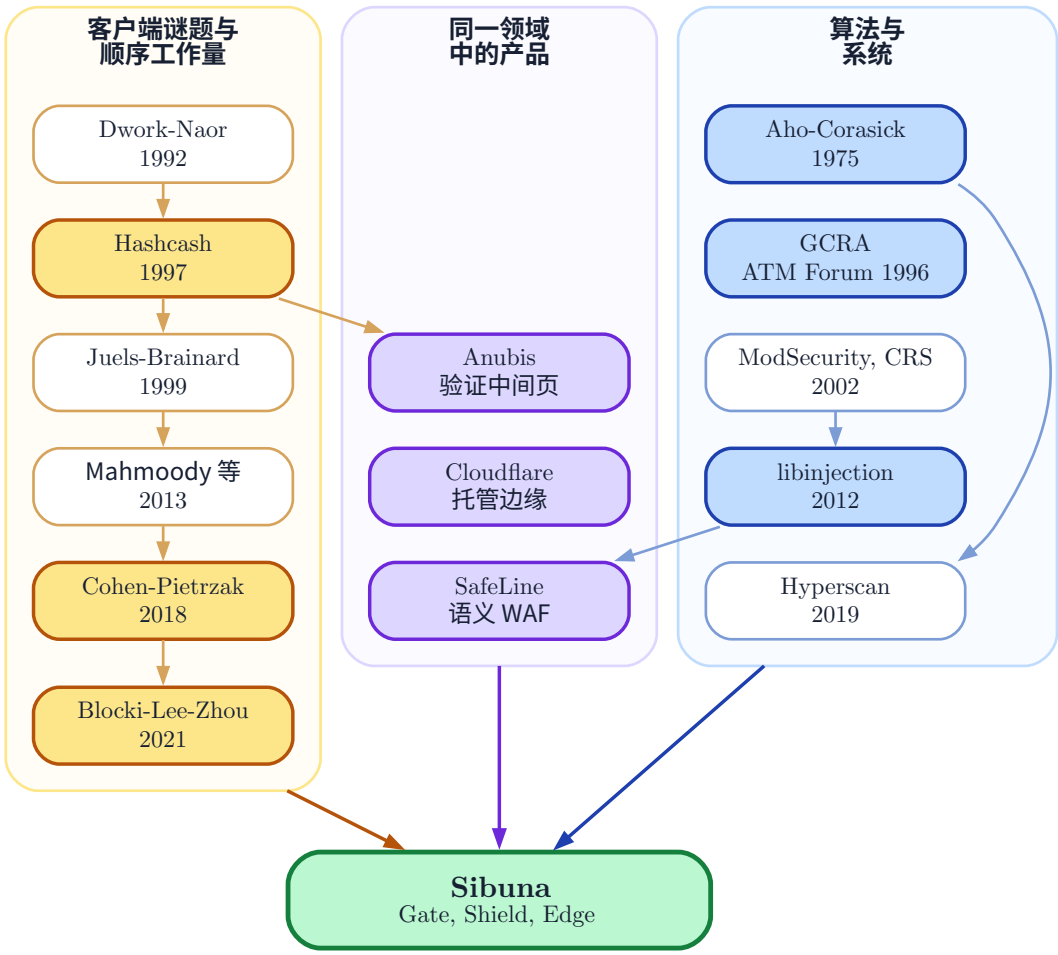


图 2 Sibuna 所结合思想的源流。左列是客户端谜题与顺序工作量的密码学，右列是 Sibuna 所依托的算法与系统，中间一列是同一领域中的产品；时间自上而下推移。细箭头表示直接的传承关系，填充的方框是 Sibuna 直接采用的思想。较早的一种存储设计借鉴了 Multi-Paxos（Lamport 1998；SID 0005）；Sibuna 现在把状态存放在 SQLite 或 PostgreSQL 中（SID 0011）。

应用防火墙

第一代 Web 应用防火墙用正则表达式匹配请求字节。ModSecurity（2002）和 OWASP Core Rule Set 使这种方法开放并得到广泛部署，也让它的代价广为人知：每个请求要执行数百个带回溯的模式，异常分数需要手工调整，普通输入上还有大量长尾误报。libinjection（Hanson, 2012）用一个分词器取代模式列表，判断一个字符串能否 解析 为 SQL，同时降低了成本和误报。长亭科技的 SafeLine 把这种语义思想带进了完整的产品，Hyperscan（Wang 等，NSDI 2019）则展示了在输入很长时，向量化的字面量匹配能走多远。第 VI 部分解释为什么对于它所检查的短字段，Sibuna 选择了稠密自动机加小型结构分词器，而不是正则表达式或 SIMD 引擎。

验证中间页与边缘网络

网站前面的工作量证明验证中间页在 2025 年变得普遍，当时为语言模型训练提供数据的爬虫流量，让普通源站的运行成本难以承受。Anubis（Techaro）是最知名的开源实现：一个 Go 编写的反向代理，提供 JavaScript 谜题，签发会话令牌，其余流量直接放行。Sibuna 的 Gate 形态用无状态挑战、对称令牌和第二层工作量解决同一问题；下面的比较记录了两者的差异。

在规模的另一端，Cloudflare 这样的托管边缘网络把检查、速率限制和机器人评分部署在数百个接入点中，并由全局控制平面统一管理。Sibuna 的 Edge 部署借鉴了这种形态（由共享的策略和信誉平面向本地不可变快照供数），同时保持自托管。因此，它的存储层是运维人员自己运行的数据库（单个节点上的一个 SQLite 文件，或由集群共享的一个 PostgreSQL 数据库），由存储线程在请求路径之外读取，而不是一次服务调用。

一个可执行文件中的三个思想

Sibuna 结合了三样东西：用于准入的浏览器工作量挑战，有界的应用载荷检查，以及发布本地策略快照的共享控制平面。Gate 和 Shield 是两种产品形态；存储和分布式部署是两者都可选用的选项。

属性	Gate	Shield
准入	原生 Hashcash 或顺序工作量验证	相同
会话	带密钥 BLAKE3；可选 Ed25519	相同
本地控制	规则、CIDR、GCRA、封禁	相同
检查	停用	有界签名匹配和结构分词器
状态	可选的持久策略和信誉，可在集群中共享	另有检查取证

浏览器求解器与原生验证器由同一份证明源码编译而成。稠密的字面量匹配避免了回溯，使扫描工作量与输入长度呈线性关系。不可变的策略快照把请求处理与 SQL 以及数据库的可用性隔离开。这些选择都不能证明算法具有普遍的最优性：短请求头、长请求体、模式的多样性、核心数和缓存大小都会改变取舍。

采用的设计约束

1. 请求路径上没有堆分配；每张表都有固定容量和明确的饱和行为。
2. 签发挑战不写入任何内容；只有付出了工作的解答才占用内存。
3. 会话只负责准入；它绝不让请求免于检查或明确的拒绝。
4. 每个引用的数字要么由已提交的测量工具测得，要么标明为模型。
5. 密码学选择注明发表出处和证明，并说明其抗量子能力。

功能比较

下面四张表把 Sibuna 与最常和它一同被提及的三种工具作比较。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：Anubis 1.27.0（MIT 许可证，Go），SafeLine 社区版 9.x

（GPL-3.0 的管理代码，外加闭源的检测器镜像，Docker Compose），以及 Cloudflare WAF 的开发者文档。Anubis 还在基准测试主机上实际运行过；SafeLine 和 Cloudflare 没有（原因见第 VIII 部分）。短横线表示该产品不提供这项功能；套餐名称表示该功能存在，但只在该套餐中提供。每项 Sibuna 测量都注明了它所来自的已提交结果文件。TLS 指纹一行于 2026 年 10 月核对，依据是 Anubis 的变更日志（1.21.0 和 1.26.0）、SafeLine 的发布说明（8.10.0 和 9.2.0），以及 Cloudflare 的“JA3/JA4 fingerprint”页面。

表 3 功能比较，第 1 部分：部署

功能	**Sibuna**	**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
语言与运行时	Rust，无垃圾回收器；关闭存储时不依赖外部 crate	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
主机资源占用	35.7 MiB 可执行文件（`benchmarks/results/comparison-latest.json`）；空闲 RSS 5.9 MiB（`benchmarks/results/primitives-latest.json`）；负载下峰值 RSS 最多 16.0 MiB（`benchmarks/results/product-latest.md`）	38.1 MiB 可执行文件；空闲 RSS 21 MiB；Linux 测试环境，控制台关闭	最低 1 个 CPU 核心、1 GB 内存、5 GB 磁盘，Docker 20.10+	本地无资源占用

表 4 功能比较，第 2 部分：准入与会话

功能	**Sibuna**	**Anubis 1.27**	**SafeLine CE 9**	**Cloudflare WAF**
工作量证明准入	Hashcash（比特级）和 Cohen-Pietrzak	以十六进制半字节计的	JavaScript 反机器人挑战和验证	托管质询、JS 质询、

功能	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
	顺序工作量，按规则选择；WebAssembly，并以 JavaScript 作为后备	SHA-256 Hashcash（`fast`、`slow`）；不涉及工作量的 `metarefresh` 和 `preact` 挑战	码；不受工作量约束	Turnstile；Bot Fight Mode 会发出计算质询
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 标签，75 字符的 Cookie，与所付出的工作绑定；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
默认令牌的后量子安全性	对称：只有 Grover 算法的平方加速适用	Ed25519：可被 Shor 算法攻破	文档未提及	文档未提及
策略语言	JSON 规则：路径、User-Agent、请求头、IPv4/IPv6 CIDR、JA4 指纹；ALLOW、DENY、CHALLENGE，以及带阈值的 WEIGH；来自数据库的动态规则	YAML/JSON 机器人定义：对 UA、路径、请求头使用正则表达式；CIDR；CEL 表达式；带阈值的 WEIGH	在控制台中定义的自定义规则、ACL、IP 组	Rules 语言表达式；按套餐为 *5 / 20 / 100 / 1000* 条自定义规则
TLS 指纹（JA4）	规则可以精确匹配或按前缀匹配 JA4 指纹；指纹由受信任的入口代理计算，并通过请求头转发（`--ja4-header`）；Sibuna 自己不计算	文档未提及；策略可以引用 HTTP 指纹 JA4H，Anubis 会计算它并添加为请求头 `X-Http-Fingerprint-JA4H`	在自身的 TLS 终止处计算；显示在攻击详情中；自定义规则可以匹配它（等于、不等于）；一条内置规则拒绝其恶意 JA4 情报库中的指纹	WAF 自定义规则和分析中的 JA3 与 JA4 指纹（*Enterprise*，需 *Bot Management*）

表 5 功能比较，第 3 部分：检查与状态

功能	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
SQLi、XSS、RCE、路径遍历检查	Shield：带标记的自动机加结构分词器，检查请求体的前 8 KB	-	覆盖 OWASP 各类别的语义分析引擎，另有 SSRF、XXE、CRLF	托管规则集（*Pro* 及以上）；OWASP Core Ruleset；攻击评分（*Business* 及以上）
速率限制	按客户端的 GCRA，作用于单个节点本地	-（系统负载会暴露给 CEL 规则）	按 IP、路径、会话	按套餐为 *1 / 2 / 5 / 100* 条规则；Enterprise 以下只能以 IP 为键
封禁与信誉	蜜罐封禁；信誉字典树从集群共享的信誉表加载	DNSBL；通过 Thoth 提供 ASN 和 GeoIP（付费，闭源）	IP 组和黑名单；威胁情报（*Pro*）	IP 列表；托管列表和机器人评分（*Enterprise*）
取证	SQLite 或 PostgreSQL：事件全文搜索，基于向量的攻击活动聚类	仅 Prometheus 指标	PostgreSQL 攻击日志，配有控制台	Security Events；Free 套餐为抽样数据
多节点	所有节点共享一个 PostgreSQL 数据库来保存策略和信誉；共享种子；粘性挑战验证	共享签名密钥；共享 Valkey 存储	每台主机一套组件	全球任播

表 6 功能比较，第 4 部分：运维

功能	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
指标	Prometheus，位于 `/__sibuna/metrics`	Prometheus，使用单独的端口	控制台仪表板	仪表板和分析
TLS 终止	-（由入口代理负责）	-（由入口代理负责）	**是**	**是**
管理控制台	可选启用；SID 0007 状态为 Committed	-	**是**	**是**

功能	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
在基准测试主机上测量	**是**（第 VIII 部分）	**是**	- （仅 Docker）	- （托管服务）

在选型决策中，有三个差异最为关键。

- 工作落在哪一方。 Anubis 和 Sibuna Gate 让 客户端 在源站做任何事情之前先付出代价。SafeLine 和 Cloudflare 先检查，再有选择地发起挑战；它们的默认立场是先准入再过滤。Sibuna Shield 两者都做，并且按这个顺序：即使请求已经持有有效会话，检查仍然可以拒绝它。
- 服务器记住什么。 Anubis 把挑战状态保存在一个存储中（内存、bbolt、Valkey 或 S3），并用 Ed25519 签名 JSON Web Token；Sibuna 签发一个自认证记录，只在一张固定大小的表中记住已解出的标签。SafeLine 把攻击日志保存在 PostgreSQL 中；Cloudflare 保存一切，Free 套餐只提供抽样的可见性。
- 运维人员需要运行什么。 Sibuna 是一个可执行文件，加上可选的 SQLite 文件，或者由集群共享的 PostgreSQL 数据库；Anubis 是一个 Go 可执行文件；SafeLine 是七个容器，运行在 Linux 主机的 Docker 守护进程之上，主机至少需要一个核心、1 GB 内存和 5 GB 磁盘；Cloudflare 则是一次 DNS 修改加一份订阅。

当前的产品边界 Sibuna 把入口处的 TLS 终止交给部署中的代理，不用训练好的模型给机器人打分，也不发布付费签名。它匹配受信任的入口代理计算出的 JA4 指纹，自己无法计算 JA4（SID 0014，第 IX 部分）。它可选启用的控制台提供需要认证的仪表板、国家信息补充、策略编辑、调查和集群管理。国家查询和存储工作在请求分类之外运行。SID 0007 的状态为 Committed，控制台也通过了浏览器工作流测试（SID 0010）。它的正式性能隔离门槛，即在负载下运行八个活动仪表板，尚未测量；已经测得的内容见第 VIII 部分。

练习 2.1. 解释为什么把同一份验证器源码同时编译为原生代码和 WebAssembly 可以减少协议漂移。哪些性质仍然需要独立测试或密码学审查?

练习 2.2. 只使用比较表，写出能为一个静态网站提供以下功能的最小部署：(a) 工作量证明网关，(b) SQL 注入检查，(c) 集群范围的封禁。每个选择是由哪几行决定的?
提示：从“运行形式”和“多节点”两行开始。

解释这个不变量. 描述 Gate、Shield 和可选的共享存储如何划分准入、检查和策略分发。然后指出比较表中我们在向客户引用之前想要重新核实的一行，并说明我们会如何核实它。

第 III 部分

来自密码学研究的基础

我们推导 *Sibuna* 两层工作量证明背后的成本模型，介绍 *Cohen-Pietrzak* 顺序工作量证明及其安全性论证，并说明为什么用带密钥哈希来认证挑战和会话。*SID 0006* 记录了相关的假设和参考文献。

Hashcash：第一层的数学

学习目标 学完本章，我们应当能够给出比特级 Hashcash 谜题的期望工作量、方差和尾部概率，解释为什么服务器的验证成本限制了攻击者的优势，并描述促使我们引入第二层的两个局限。

定义与成本

对于挑战字符串 C 和 b 比特的难度，Sibuna 的第一层谜题是：找到 $N \in \mathbb{N}$，使 SHA-256 $(C \,\|\, ":" \,\|\, \mathrm{dec}\,(N))$ 具有 b 个前导零比特。nonce 采用十进制表示是有意为之；它使 WebAssembly 求解器、JavaScript 后备实现和原生验证器之间的线路格式保持一致，而不需要二进制编码步骤。难度的单位是一次期望的 哈希尝试，而不是一次压缩函数调用。一个输入可能占用多个 SHA-256 分组。前缀预计算也会改变每个 nonce 的工作量。线路规则始终是摘要谓词；各个实现必须就陈述和 nonce 编码达成一致，而不是就估计的 CPU 成本达成一致。

对于相互独立的理想摘要，尝试次数 X 服从参数为 $p = 2^{-b}$ 的几何分布。它的中位数是 $\lceil \ln(1/2) / \ln(1-p) \rceil$，当 p 很小时约为 $2^b \ln 2$，方差为 $(1-p)/p^2$。验证者只检查一个候选摘要，而不执行搜索。解析、挑战认证、指纹检查、写入已用状态以及响应都是额外的工作。不能从一个原语的耗时反推出服务器抵御洪泛的能力。

搜索的两个局限

第一个局限是方差：一次很长的搜索并不一定意味着 worker 出了故障。第二个局限是并行性：相互独立的 nonce 尝试可以并发执行。即使期望的总尝试次数不变，更多的核心也会缩短搜索的实际耗时。因此，硬件吞吐量对准入策略很重要。无论是声称的哈希速率还是单次基准测试，都无法给每个请求方的优势划定上界。

练习 3.0. 利用几何分布的尾部，估计需要超过 4.6×2^b 次尝试的概率。为什么把 nonce 空间分给多个 worker 不会改变期望尝试次数？
提示： 统计总尝试次数，而不是经过的秒数。

顺序工作量证明：第二层

> **学习目标** 理解 Cohen-Pietrzak 标签计算、它为什么强制顺序计算、验证者检查什么、条件抽样界 $(1-\alpha)^t$ 及其局限、后量子结果、证明者的内存上界，以及工作比特映射背后的校准数据。

为什么需要顺序工作量

顺序工作量构造在计算之间引入依赖：后面的标签需要前面算出的值。它针对的是与总工作量不同的一个量，即计算的深度。相互独立的谜题仍然可以并行运行，硬件的哈希延迟也仍然重要。

Sibuna 采用 Cohen 和 Pietrzak 描述的基于哈希的构造。深度为 n 时，它的图有 $2^{n+1}-1$ 个标签，但固定的标签数并不意味着确定的浏览器延迟：标签可能跨越多个压缩分组，揭示可能需要重新计算子树，调度也会带来波动。不能把预言机模型分析得出的安全性陈述，变成关于每个具体实现或每个对手的无条件结论。

构造

令 n 为深度，考虑一棵完全二叉树，其节点以长度不超过 n 的二进制串命名（根为 ε，子节点为 $v0$ 和 $v1$，叶子的长度为 n）。每个节点 v 带有一个 32 字节的标签 $\ell_v = H(\chi, v, \text{parents}\ (v))$，其中 χ 是挑战陈述，H 是只吸收一次 χ 的 SHA-256（与 Hashcash 求解器相同的前缀技巧）：

- 内部节点对它的两个子节点做哈希：$\ell_v = H(\chi, v, \ell_{v0}, \ell_{v1})$；
- 叶子 $u = u_1 \ldots u_n$ 在根路径每次向右转的地方，对其祖先的左兄弟做哈希：$\ell_u = H(\chi, u, \{\ell_{u_1 \ldots u_{i-1}0} : u_i = 1\})$。

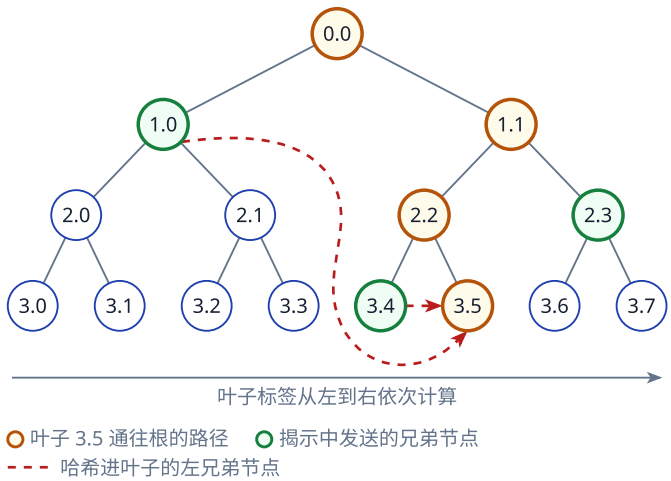


图 3 深度为 3 的树：叶子 3.5 的一次揭示发送该叶子标签和三个兄弟标签；虚线边是使标签计算成为顺序过程的左兄弟依赖。

左兄弟边正是使计算成为顺序过程的原因：在叶子 u 左侧的每棵子树都完成之前，无法计算它的标签，因此只能按深度优先的后序遍历进行，根标签 $\varphi = \ell_\varepsilon$ 依次依赖于全部 $2^{n+1}-1$ 个标签。

揭示。 计算出 φ 之后，证明者通过 Fiat-Shamir 导出 t 个叶子，$\gamma_i = H(\chi \,\|\, \varphi \,\|\, i) \bmod 2^n$，并为每个叶子发送叶子标签和其根路径上的 n 个兄弟标签。验证者用作为父节点的兄弟标签重新计算叶子，沿路径向上做哈希，并检查最终是否得到 φ。验证者的成本是 $O(tn)$ 次标签计算，所需工作空间有界。一次标签计算可能跨越多个哈希分组；统计标签数并不等于统计压缩函数调用次数。

> **实现：posw::verify_opening** 用兄弟标签重新计算一个被揭示的叶子，并沿路径走到根。位于 crates/pow/src/posw.rs（crate sibuna-pow）。

```
fn verify_opening(base: &Sha256, n: u8, phi: &Label, gamma: u32, opening: &[Label]) -> bool {
    let Some((leaf, siblings)) = opening.split_first() else {
        return false;
    };
    // `siblings` 从深度 n 向上排到深度 1；反转之后，它与深度 1..=n 一一配对。
    let mut hasher = node_hasher(base, n, gamma);
    for (d, sibling) in (1..=n).zip(siblings.iter().rev()) {
        if turns_right(gamma, n, d) {
            hasher.update(sibling);
        }
    }
    if hasher.finalize() != *leaf {
        return false;
    }
    let mut current = *leaf;
    for (d, sibling) in (1..=n).rev().zip(siblings) {
        let mut parent = node_hasher(base, d - 1, gamma >> (n - d + 1));
        if turns_right(gamma, n, d) {
            parent.update(sibling);
            parent.update(&current);
        } else {
            parent.update(&current);
            parent.update(sibling);
        }
        current = parent.finalize();
    }
    current == *phi
}
```

抽样、可靠性及其局限

考虑一个固定的承诺，其中有比例为 α 的坏位置，且每个坏位置都可以独立检测出来。如果每次揭示都独立、均匀地抽样，那么 t 次揭示全部错过坏位置的概率是 $(1-\alpha)^t$。这个初等计算解释了为什么更多的揭示有帮助。但它本身并不是顺序工作量协议的安全性证明。

> **实例分析：一个条件抽样界** 当坏位置比例固定为 $\alpha = 0.1$，并进行 $t = 16$ 次独立揭示时，全部错过坏位置的概率是 $0.9^{16} \approx 0.185$。进行 32 次揭示时，该概率约为 0.034。这个前提要求在选择样本之前承诺已经固定。如果证明者可以反复尝试许多个根、选择揭示哪个承诺，或者让失败相互关联，那么这个计算没有考虑这些策略。

所引用的顺序工作量构造在一个预言机模型中有安全性分析。Sibuna 的域分离、参数映射、截断和实现，仍然必须对照该构造的假设进行检查。Fiat-Shamir 抽样使揭示可以从承诺复现；但仅有可复现性并不能证明能够抵御反复尝试（grinding）。我们不作出“作弊永远不会更便宜”这样的无条件声明。

练习 3.1. 如果攻击者尝试 g 个相互独立的承诺，每个被接受的概率为 q，推导至少有一个被接受的概率。这个表达式遗漏了哪些成本？
提示： 先计算全部 g 个都失败的概率。

证明者的内存

朴素的证明者会保存每个标签（共 $2^{n+1}-1$ 个）。Sibuna 的证明者在第一遍计算时维护一个栈，保存当前路径的左兄弟标签（$n+1$ 个标签），并只保留最上面的 $m=\min(n,10)$ 层（$2^{m+1}-1$ 个标签，最多 64 KB）。为了揭示一个叶子，它重新计算该叶子在深度 m 处的祖先的子树，共 $2^{n-m+1}-1$ 次哈希，并用保留的各层初始化栈。证明者的总成本是 $2^{n+1}-1+t\,(2^{n-m+1}-1)$ 次哈希，整个工作空间（包括输出的证明）不到 100 KB：浏览器标签页和原生测试共用同一个 `Workspace` 类型（`sibuna_pow::posw::Workspace`）。

参数不是耗时

Sibuna 用 $n=b-3$ 把工作比特映射为深度，并限制在支持的区间 $[4,24]$ 内。这是一个工作量尺度上的约定，并不保证不同算法的实际耗时相等。一个叶子可能要对多个父标签做哈希，而 Hashcash 是反复测试 nonce。编译器、浏览器、处理器和参数的选择都会影响它们的相对成本。已提交的原语基准测试测量的是验证；它并不能说明浏览器的求解延迟。
证明的大小是 $32(1+t(n+1))$ 字节。在深度 13、十六次揭示时，大小为 $32(1+16\times 14)=7200$ 字节。即使构造树这一主要成本保持不变，增加揭示次数也会同时增大证明和验证者的工作量。

练习 3.2. 计算深度 17、16 次揭示时的证明大小。然后把揭示次数加倍。哪一项加倍了，哪一项保持不变？

选择一种构造

有意义的比较对象是资源预算，而不是算法的名声。记录证明者的内存、验证者的最坏情况工作量、证明的字节数，以及所声称的性质需要的假设。内存困难的搜索可能有用，但如果验证者必须重复昂贵的工作，攻击者就多了一个消耗服务器资源的地方。可公开验证的延迟构造所服务的信任模型，与共享密钥的网关不同。
Sibuna 保留了低成本的哈希验证，以及一个基于哈希的顺序工作量选项。后者需要更多的证明字节和更仔细的参数分析。无论哪种选择，都需要在进入昂贵的验证之前拒绝格式错误的证明。

解释这个不变量. 解释认证树与依赖边之间的区别：前者证明某个标签属于一个承诺，后者约束标签的计算方式。

对称认证：挑战与会话

学习目标 解释当签发者和验证者属于同一个信任域时，为什么带密钥的伪随机函数是正确的原语；给出 128 位标签的伪造界；描述无状态挑战记录和已用集合的上界；并读懂密钥派生方案。

签发者即验证者

数字签名让 任何 持有公钥的人都能验证；当验证者不应具备签发能力时，它是正确的工具。Sibuna 的验证者 就是 签发者（或共享同一个种子的集群）。由伪随机函数构建的消息认证码在这个信任域内提供认证。它不提供公开验证：任何持有 MAC 密钥的人也都能签发令牌。基准测试一章比较了已实现的令牌操作。

引理 3（伪造界） 把密钥模式下的 BLAKE3 建模为 PRF，并使用 16 字节标签时，进行 q 次验证查询的对手伪造出有效令牌的概率至多为 $q \cdot 2^{-128} + \varepsilon_{\mathrm{PRF}}$。按每秒一百万次查询计算，即每秒 2^{-108}。

从长远看，还有第二个重要推论：Ed25519 依赖于椭圆曲线群上的离散对数，而 Shor 算法可以攻破它；带密钥哈希只依赖于哈希函数，量子算法对它至多带来平方级加速。对于需要验证者不具备签发能力的部署，Sibuna 保留 Ed25519 作为选项（`--token-scheme ed25519`）。

密钥派生方案

一个 32 字节的主种子（来自 `--secret-file`、`SIBUNA_SECRET` 或一个新生成的随机值）通过带密钥 BLAKE3 和域字符串派生出每个用途的密钥：令牌密钥、挑战密钥、Ed25519 密钥、指纹密钥，以及用于封装要求票据的要求密钥（第 IV 部分）。

实现：`Keys::derive` 从种子派生五个用途密钥。位于 `crates/challenge/src/keys.rs`（crate `sibuna-challenge`）。`Keys` 和 `Seed` 在被丢弃时将其字节清零，它们的 `Debug` 输出也经过脱敏。

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

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

在集群中，`Keys::bind_node` 以挑战密钥为密钥，对 `sibuna/challenge-node/v1` 和节点 ID 计算带密钥 BLAKE3，并用结果替换挑战密钥。这样，挑战必须回到签发它的节点，而会话令牌和要求票据在每个成员上都保持有效。

域分离可以防止不同用途之间的意外复用；主种子一旦泄露，所有派生密钥都会随之泄露。集群只需要就一个种子达成一致。

无状态挑战

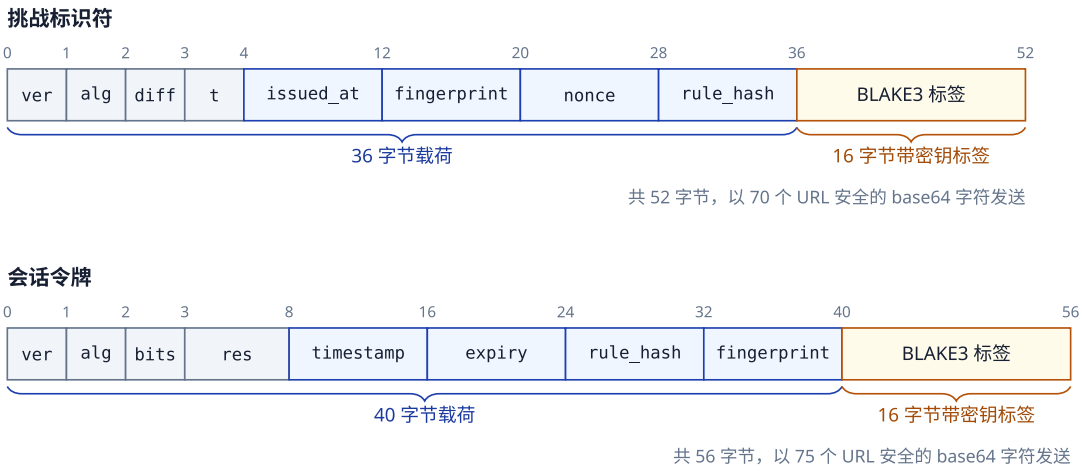


图 4 挑战标识符和会话令牌的线路格式。每条记录上方的数字是字节偏移量。

挑战标识符是一个自认证记录：版本、算法、难度、揭示次数、签发时间、客户端的带密钥指纹、由 PRF 派生的 nonce，以及要求该挑战的策略规则的哈希，后面跟着一个用挑战密钥计算的 16 字节标签。签发挑战不写入任何内容。验证时先解码记录，以常量时间检查标签，对照挑战 TTL 检查时效，对照提交方客户端检查指纹，最后才检查证明。

引理 4（状态只随付出的工作增长） 令 S 为守护进程记住的挑战标签集合。只有在某个标签对应的有效证明被接受之后，该标签才会进入 S，并在挑战 TTL 之后离开。因此，若 r 是该时间段内被接受的不同解答的速率上界，则 $|S| \le r \cdot \mathrm{TTL}$。免费获取挑战不会分配标签。实现还具有固定容量，饱和时可能拒绝有效的解答。这是一个状态上界，而不是对每个对手计算策略的下界。

已用集合由 16 个分片组成，每个分片对 16 字节标签使用 Robin Hood 开放寻址（第 V 部分）。指纹是客户端地址和 User-Agent 的带密钥哈希；输入按长度分隔，使任何两个输入都不会因拼接而碰撞；并且带有密钥，使其他客户端的指纹无法离线计算。

练习 3.3. 令牌载荷包含工作等级、`timestamp`、`expiry`、`rule_hash` 和 `fingerprint`。解释如果去掉每个字段，分别会失去什么防护，以及为什么标签必须覆盖全部五个字段。

解释这个不变量. 一位同事提议把所有令牌都换成 Ed25519，“因为它更强”。解释信任域的论证、成本上的差异，以及后量子方面的考虑。

第 IV 部分

Sibuna 协议

我们按实际实现来规定线路协议：验证中间页的往返过程、无状态挑战记录、两种解答格式、验证顺序、一次性使用的强制执行，以及会话 *Cookie*。

挑战的往返过程

学习目标 学完本章，我们应当能够追踪一个浏览器从第一个请求到获得会话的全过程，说出其中涉及的每个端点和 JSON 字段，并解释为什么挑战要携带受保护路径的策略决策。

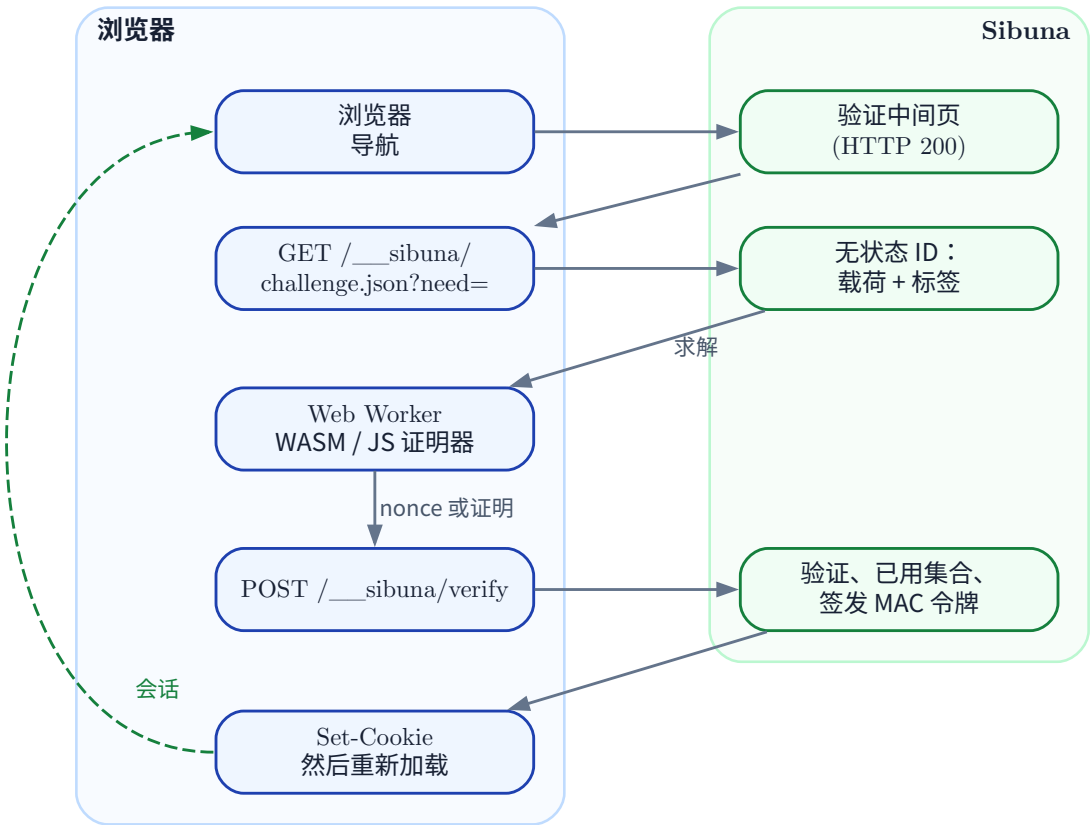


图 5 挑战的往返过程。左侧的步骤在浏览器中运行，右侧的步骤在 Sibuna 中运行。

1. 一次没有有效会话的页面导航到达策略引擎，被分类为 `CHALLENGE`。由于该请求接受 `text/html`，守护进程以 `200` 返回验证中间页，并带上响应头 `X-Sibuna-Status: CHALLENGE`；不接受 HTML

的 API 客户端则收到 401 和一个 JSON 响应体，其中的 challenge URL 可以直接获取。无论哪种情况，响应都携带一张 要求票据：即这个决策所要求的挑战（算法、工作比特、揭示数）和规则的哈希，用该客户端自己的派生密钥计算带密钥 BLAKE3 标签加以封装，在一个挑战有效期内有效。

2. 验证中间页获取 `GET /__sibuna/challenge.json?path=<original URL>&need=<ticket>`。有效的票据决定挑战：工作量由要求该挑战的决策确定，而不是根据上报的 URL 和这次获取请求自身的请求头重新计算，因为这些请求头与页面导航的不同（Accept、Sec-Fetch-*）。没有有效票据时（忽略该 URL 的 API 客户端，或页面打开时间超过了有效期），服务器评估上报的 URL：用读取请求行的同一个函数把它拆分为路径和查询，并把路径转换为准入所用的规范形式（SID 0009）；长度超过 8 KiB 的 URL 以 414 拒绝，绝不截断；没有规范形式的路径以 400 拒绝。票据只是建议性的：准入仍然检查会话的工作等级，因此重放或伪造的票据只会让其持有者多付出工作。
3. 响应是挑战记录：

```
{"id":"AQEND...7O
chars","algorithm":"posw","difficulty":13,"challenges":16,"expires_at":1757241234}
```

对于 hashcash，difficulty 是比特数，challenges 为 0。

4. 一个 Web Worker 用 WebAssembly 模块求解它；如果 WebAssembly 不可用，则使用 JavaScript 证明器（第 VII 部分）。然后它把以下两者之一

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

或

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

提交到 `POST /__sibuna/verify`。

5. 成功时，服务器返回 200 {"status":"ok"} 并带上 Set-Cookie 响应头，页面随后重新加载原来的地址。之后的每个请求都携带这个 Cookie，并在策略评估之前被准入。

为什么路径要随挑战一起传递 验证中间页在受保护的 URL 上提供，因此浏览器知道自己在哪里。发送路径让服务器能够选择 按规则 设定的难度和算法（结账路由可能要求 20 个工作比特的顺序工作量，而博客只要求 16 比特的 Hashcash），并把规则哈希绑定进令牌，再由 X-Sibuna-Rule-Hash 把它报告给上游。

无状态挑战记录

学习目标 阅读挑战编码器，理解每个字段的用途，并了解如何在热路径上不通过熵系统调用来派生每个挑战的 nonce。

实现：`Coordinator::create_challenge_with_spec` 构建 36 字节的载荷，用挑战密钥为它计算标签，并把这 52 字节进行 base64url 编码，得到 70 字符的标识符。位于 `crates/challenge/src/coordinator.rs`（crate `sibuna-challenge`）。

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

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

    Challenge {
        id: encoded::encode(&raw),
        algorithm: effective.algorithm,
        difficulty,
        challenges,
        expires_at: now.saturating_add(u64::from(self.challenge_ttl)),
    }
}
```

标识符是一个 `Encoded<70>`：文本存放在一个长度恰好相等的数组中，因此签发挑战从不分配内存。`ChallengeSpec::parameter` 把工作比特映射为各层自己的单位，即 Hashcash 比特数（最多 64）或 PoSW 深度（`difficulty - 3`，限制在 4 到 24 之间）。
有三个细节值得注意：

- 自适应难度。 协调器用三个原子变量保存挑战签发速率的指数加权移动平均。速率超过每秒 50 次挑战的基线后，速率每翻一倍就增加一个工作比特，最多增加六个。洪泛流量需要付出指数级增长的代价，而平静的流量保持基础成本。
- *nonce* 是 *PRF* 的输出，以挑战密钥为密钥，对一个计数器、时钟和指纹计算得出。它是唯一的（来自计数器）且不可预测的（来自密钥），因此对手无法为尚未签发给它的标识符预先计算解答，热路径也从不为获取熵而调用操作系统。
- 难度包含在标签之内。 客户端无法通过编辑记录来降低难度：在检查任何证明之前，标签校验就会失败。

验证顺序与一次性使用

学习目标 理解为什么低成本的检查要在证明之前执行，为什么只有在证明有效之后才把挑战标记为已用，以及同一个解答的两次并发提交如何解决。

实现：`Coordinator::verify_and_mint` 解码并认证标识符，检查时效和客户端绑定，按记录的层验证证明，把标签记为已用，然后签发会话令牌。位于 `crates/challenge/src/coordinator.rs`（crate `sibuna-challenge`）。

```
pub fn verify_and_mint(
    &self,
    challenge_id: &[u8],
    solution: Solution<'_>,
    client_ip: &[u8],
    user_agent: &[u8],
    now: u64,
) -> Result<Verified, VerifyError> {
    let record = self.decode(challenge_id)?;
    let expires_at = record.issued_at.saturating_add(u64::from(self.challenge_ttl));
    if now > expires_at || record.issued_at > now.saturating_add(CLOCK_SKEW_SECONDS) {
        return Err(VerifyError::ChallengeExpired);
    }
    if record.fingerprint != self.fingerprint(client_ip, user_agent) {
        return Err(VerifyError::FingerprintMismatch);
    }
    check_solution(challenge_id, &record, solution)?;
    self.spent.mark_spent(&record.tag, expires_at, now).map_err(|error| match error {
        SpendError::DoubleSpend => VerifyError::DoubleSpendAttempt,
        SpendError::Full => VerifyError::StoreFull,
    })?;
    let work = WorkLevel {
        algorithm: record.algorithm.to_byte(),
        bits: record.algorithm.work_bits(record.difficulty),
    };
    Ok(Verified {
        token: self.mint_token(now, record.rule_hash, record.fingerprint, work),
        algorithm: record.algorithm,
        difficulty: record.difficulty,
        challenges: record.challenges,
```

```
        ttl_seconds: self.token_ttl,
    })
}
```

这个顺序是有意为之的。标签、时效和绑定都是常量时间的检查，它们在任何哈希计算之前就拒绝垃圾输入。错误的证明不能消耗挑战，因此只有在 check_solution 成功之后才写入已用集合。两个线程同时验证同一个有效解答时，会在 mark_spent 上竞争；分片锁让它们串行执行，恰好一个获胜，另一个收到 VerifyError::DoubleSpendAttempt。条件 issued_at > now + 60（CLOCK_SKEW_SECONDS 为 60）拒绝由时钟超前一分钟以上的节点签发的记录，否则这类记录会延长挑战的寿命。

错误以 400 和 Elm 风格的诊断信息（第 X 部分）呈现给客户端：每个 VerifyError 都转换为 sibuna_core::errors::ErrorKind，其说明就是响应体。端到端测试对响应体中的 DOUBLE SPEND、FINGERPRINT MISMATCH、WRONG SOLUTION TYPE 和 MALFORMED CHALLENGE 进行断言。

会话 Cookie

学习目标 阅读带密钥哈希令牌，理解 Cookie 的各个属性，并解释客户端绑定。

实现：token::mac::mint 和 token::mac::verify 序列化 40 字节的载荷，追加对它计算的带密钥 BLAKE3 的前 16 字节，并用常量时间比较进行验证。位于 crates/challenge/src/token/mac.rs（crate sibuna-challenge）。Coordinator::verify_cookie 用客户端的指纹调用它。

```
pub fn verify(
    key: &[u8; 32],
    text: &[u8],
    now: u64,
    expected_fingerprint: Option<u64>,
) -> Result<Payload, TokenError> {
    if text.len() != TOKEN_LEN {
        return Err(TokenError::InvalidTokenLength);
    }
    let raw: [u8; RAW_LEN] =
        encoded::decode::<RAW_LEN, TOKEN_LEN>(text).ok_or(TokenError::InvalidEncoding)?;
    let (payload, given) = raw.split_at(PAYLOAD_LEN);
    let mut bytes = [0u8; PAYLOAD_LEN];
    bytes.copy_from_slice(payload);
    // 常量时间：提前退出会让伪造者知道标签有几个字节是对的。
    if !ct::eq(&tag(key, &bytes), given) {
        return Err(TokenError::InvalidTokenSignature);
    }
    let payload = Payload::from_bytes(&bytes)?;
    payload.check(now, expected_fingerprint)?;
    Ok(payload)
}
```

标签在解析载荷之前检查：只有在确认载荷真实之后，其他版本的载荷才会以 TokenError::InvalidEncoding 被拒绝，而伪造的载荷永远到不了解析器。

Cookie 以如下形式发出：

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

`HttpOnly` 使页面脚本无法读取它，`SameSite=Lax` 在允许顶层导航的同时阻止跨站重放；当 TLS 在守护进程前端终止时，通过 `--secure-cookie` 加上 `Secure`。由于载荷携带地址和 User-Agent 的带密钥指纹，复制到另一台机器上的 Cookie 会以 `TokenError::TokenBoundAddressMismatch` 失败；端到端测试“Cookie 与客户端身份绑定”正是检验这条路径。

载荷以一个版本字节和持有者付出的 工作等级 开头：即机制（Hashcash 或 PoSW）以及实际求解的工作比特，包括任何随负载自适应增加的部分。只有当会话的等级达到路由的要求时，会话才能免除挑战；路由的要求由规则的难度经过与挑战本身相同的限幅计算得出。等级是有序的，而不是按名称区分：以 24 比特获得的 Cookie 也覆盖所有 16 比特的路由，而 16 比特的 Cookie 出现在 24 比特的路由上时，会落入要求更高的验证中间页，新的 Cookie 将替换旧的。`rule_hash` 仍然是签发该挑战的规则的审计标识；它被报告给上游，但从不用于准入。会话绝不会免除 WAF 检出结果和明确的拒绝。端到端测试“在成本较低的路由上获得的会话不能准入要求更多工作的路由”检验了两个方向。

转发的地址 在入口代理之后，客户端地址通过 `X-Forwarded-For` 到达。只有在使用 `--trust-forwarded` 时（forward-auth 模式下默认隐含），Sibuna 才信任这个请求头。直接暴露在互联网上的反向代理必须关闭它，否则任何客户端都可以自己选择绑定和速率限制桶。

练习 4.1. 画出位于同一个 NAT 地址之后、User-Agent 相同的两个浏览器的交互序列。它们共享哪些检查，不共享哪些检查？位于该 NAT 之后的攻击者需要什么才能复用另一个浏览器的 Cookie？

解释这个不变量. 向运维人员解释为什么守护进程不“记住”未完成的挑战，以及为什么这是一项安全特性，而不是一种偷懒的做法。

第 V 部分

Rust 中的零分配工程

我们通过一个固定的缓冲区，跟随一个连接从接受到响应的全过程，并考察那些让许多连接线程在请求路径上无需堆分配即可共享状态的并发数据结构。在内存安全原本需要专门机制（读者计数、内存区、描述符表、拆除链）的地方，我们展示 *Rust* 的所有权如何取而代之。

连接循环

学习目标 学完本章，我们应当能够解释：一个 64 KiB 的缓冲区如何服务整个 keep-alive 连接；如何在不复制的情况下定位请求头部和请求体；为什么每个连接都有自己的有界线程；代理如何获知源站响应的分帧方式，使被代理的客户端连接可以保持打开；以及这些保证中的哪些由借用检查器为我们证明。

每个连接一个线程，并且有界

接受 worker（`--workers`，默认每个 CPU 一个，最多 64 个）共享监听套接字。每个被接受的连接交给一个拥有 1 MB 栈的独立线程，接受循环随即回到 `accept`。完成连接的线程会停驻并接手下一个连接，因为为每个连接创建并结束一个线程会拖住其他线程（第 VIII 部分，SID 0002）。另一种做法是在接受线程上把连接一直服务到结束，这也是最初的设计：它在单个客户端时测得很好，在六十四个客户端时却很差，因为其中六十个要等待某个 worker 用完它 256 个请求的配额，尾延迟随每个等待的客户端而增长。采用每连接一个线程后，在每种完整产品工作负载中，64 个连接时的 p99 都保持在一毫秒以下，从 305 us 到 857 us 不等（`benchmarks/results/product-latest.md`；第 VIII 部分）。

连接线程的数量受 `--max-connections` 限制（默认 1,024，最多 8,192）。超过上限时，接受循环在新套接字上回答 `503 Service Unavailable` 并关闭它，不创建任何线程，同时把这一事件计入 `sibuna_overloaded_total`。

实现：`accept_loop` 接受连接，对照连接上限检查每个连接，并从任务池中为它启动一个独立线程。位于 `cmd/sibuna/src/server.rs`（crate `sibuna`）。

```
fn accept_loop(server: &Arc<Server>, listener: &TcpListener) {
    while !server.is_stopping() {
        let stream = match listener.accept() {
            Ok((stream, _)) => stream,
            // ... 文件描述符或内存耗尽：退避 1 ms，而不是空转
        };
        let limit = server.max_connections.load(Ordering::Relaxed);
        if server.connections.fetch_add(1, Ordering::Relaxed) >= limit {
            server.connections.fetch_sub(1, Ordering::Relaxed);
```

```
            server.metrics.overloaded.bump();
            let text = "Service Unavailable: connection limit reached";
            reject(server, &stream, text);
            continue;
        }
        set_no_delay(&stream);
        let task_server = Arc::clone(server);
        let launched = server.tasks.launch(move || {
            let _counted = Counted(&task_server);
            crate::connection::serve(&task_server, &stream);
        });
        // ... 启动失败时归还名额，并记一次过载
    }
}
```

`Counted` 是一个守卫，无论线程以返回还是 panic 结束，它的 `Drop` 都会把该连接在上限中占用的名额归还。`503` 路径在关闭自己一侧的套接字之前，会先读完客户端已经发送的内容：关闭一个还有未读字节的套接字会以复位作答，而复位会让 `503` 被丢弃。

线程来自一个 `TaskPool`（`cmd/sibuna/src/tasks.rs`），即一张由可 join 的槽组成的固定表。结束的线程在被 join 之前会保留它的栈，因此一个槽会在被再次占用时 join，并且新任务占用 编号最小 的已结束槽，而不是推进一个游标。游标要绕表一整圈才会再次到达一个已结束的槽，这会让数千个已结束的线程驻留在内存中，内存将随已服务的连接数增长，而不是随打开的连接数增长。第二个守卫 `Finished` 在任务结束时把槽标记为空闲，因此发生 panic 的连接同样会释放它的槽。

内存也以同样的方式有界。一个连接线程在栈上持有 64 KiB 的读缓冲区和 16 KiB 的写缓冲区，并在堆上持有一个 40 KiB 的代理工作空间，后者在连接开始时一次性分配。因此，最坏情况是一个已知的数字，而不是客户端能打开多少套接字的函数。空闲回收器（见下文）关闭停止发送的连接，使慢速客户端无法占住一个线程超过 `--idle-timeout`。

分配器的选择

Rust 1.100 稳定了 `Allocator` API 的核心部分以及 `Vec` 和 `Box` 的分配器参数（发行说明）。我们可以为单个集合选择分配器，而不替换整个进程的分配器。这项 API 无法再减少请求路径的分配：预热后的请求已经不分配内存，`cmd/sibuna/tests/steady_state.rs` 和 `crates/net/tests/steady_state.rs` 对此进行检查。代理为每个连接分配一次三个缓冲区，并重复使用它们。限流器和已用证明集合也只分配一次表；清零分配使未触及的页面不必驻留内存，`crates/store/tests/resident.rs` 对此进行检查。

研究分配器变更时，我们会先测量策略快照的构建和控制台的 JSON 解析。它们的临时集合在请求路径之外分配内存，但集合的分配器不会自动控制其元素的分配。我们的 JSON 树还通过 `Cow<str>` 拥有包含转义的字符串，其拥有形式为 `String`；Rust 1.100 的 `String` 没有分配器参数。`Arc::new_in` 在此版本中仍不稳定（API 文档）。已发布的引擎通过 `Arc` 共享，直到最后一个读取者释放它们，因此重建不能重置旧引擎仍在使用的内存。内存区域分配器必须保留这些生命周期和解析器的输入边界。

我们保留控制台 WebAssembly 模块在 `cmd/console-ui/src/abi.rs` 中现有的全局分配器 `WipeOnFree`。它会擦除释放的内存块，包括重新分配后的旧块。直接为集合选择 `System` 会绕过这项擦除。工作量证明 WebAssembly 模块没有堆分配器，Argon2 也已经使用调用者拥有且可重复使用的工作区。因此，我们保留现有分配器和 Rust 1.94 兼容性，直到测量到的工作负载能够证明变更

合理。候选方案必须改善延迟或驻留内存，同时保留擦除、有界输入处理和不分配内存的请求路径。

每个连接一个缓冲区

请求路径上的每一次动态分配都是拒绝服务的杠杆：经年累月的内存碎片，线程之间对分配器锁的争用，以及攻击者精心构造请求以最大化分配所造成的放大。Sibuna 的规则是绝对的：分类、验证和代理交接都在零堆分配下运行。连接处理器在套接字的整个生命周期中拥有一个 64 KiB 的读缓冲区和一个 16 KiB 的写缓冲区；其他一切都是指向它们的切片。

> **实现：connection::serve** 向空闲回收器登记连接，一次性设置缓冲区、代理工作空间和引擎缓存，然后最多服务 4,096 个请求。位于 cmd/sibuna/src/connection.rs（crate sibuna）。

```
pub fn serve(server: &Server, stream: &TcpStream) {
    let Some(lease) = server.idle.register(stream) else { return };
    // ...
    let mut read_buffer = [0u8; MAX_REQUEST_BYTES]; // 64 KiB
    let mut write_buffer = [0u8; WRITE_BUFFER]; // 16 KiB
    let mut reader = Reader::new(stream, &mut read_buffer[..]);
    let mut peer = PeerText::default();
    peer.format(stream.peer_addr().ok());
    let mut link = Link {
        server,
        stream,
        writer: Writer::new(stream, &mut write_buffer[..]),
        workspace: Workspace::new(),
        cache: EngineCache::new(),
        activity: lease.activity(),
        last: false,
    };
    for served in 0..MAX_REQUESTS_PER_CONNECTION {
        link.last = served + 1 == MAX_REQUESTS_PER_CONNECTION;
        if !serve_one(&mut reader, &mut link, peer.as_bytes()) {
            break;
        }
        lease.touch();
    }
}
```

read_head 不断填充缓冲区，直到出现空行为止，每次只搜索上次搜索之后到达的字节，并以 431 拒绝超过 16 KiB 的请求头部。parse_request（crate sibuna-http）产生一个 Request<'a>，其方法、路径、查询、请求头和 Cookie 都是缓冲区的切片。声明的请求体被填充到缓冲区能容纳的程度，并在头部之后切分出来。然后 reader.consume(head_len + body_len) 推进读取器，使下一个 keep-alive 请求干净地开始；读取器只在需要空间时才把未读字节移到前面。分块请求体在头部之后原地解码（SID 0009）：每一步的分帧信息都从缓冲区中剪除，使解码后的前缀保持连续，便于检查。内部路由以长度定界，并保持连接打开。被代理的请求也会保持连接打开，前提是源站的响应有明确的分帧，且请求体已被完整缓冲；下面几节说明代理如何作出判断。

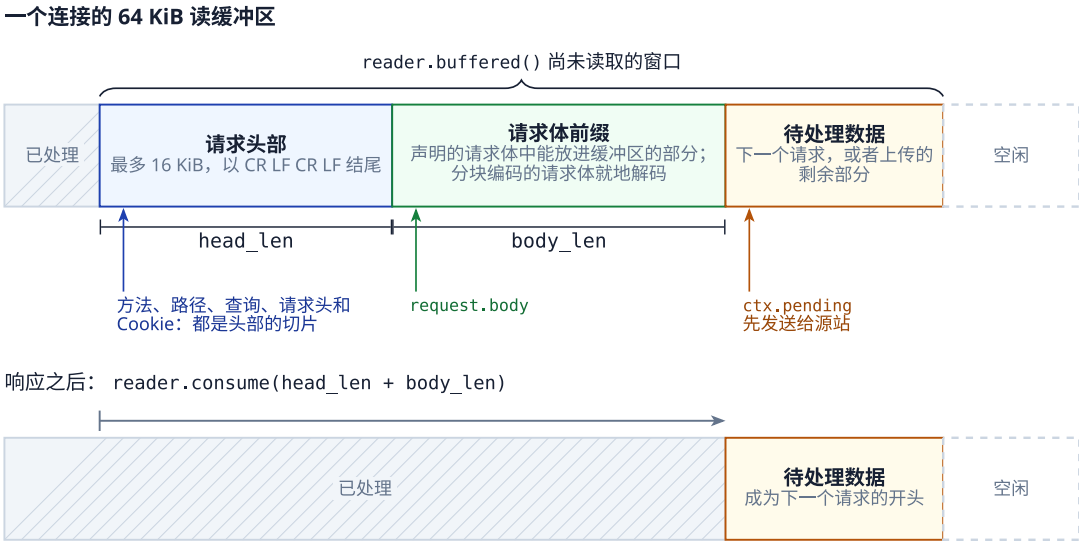


图 6 服务一个请求时连接的读缓冲区。请求的每个字段都是它的一个切片。

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

借用检查器证明了什么 请求切片只在下一次填充缓冲区之前有效。Sibuna 把这条规则表述为类型，而不是注释：请求是 `Request<'a>`，而 `'a` 是对读取器缓冲区的共享借用。填充或消费缓冲区需要 `&mut reader`，而只要从它借出的 `Request` 仍在使用，编译器就会拒绝。在 `serve_one` 中，请求被移动进 `dispatch`，只有在该调用返回之后才能运行 `reader.consume`；如果颠倒两者的顺序，代码就无法编译。

同一条规则解释了两个乍看奇怪的选择。第一，`serve_with_body` 会第二次解析头部。第一次解析必须在缓冲请求体之前结束它的借用，因为缓冲分块请求体会在缓冲区内部移动字节；第二次解析借用的是已经稳定的缓冲区。解析是确定性的，对于一个带 Cookie 的浏览器请求大约需要 133 ns（`benchmarks/results/primitives-latest.md` 中的 `zero_copy_request_and_cookie`，132.5 ns），因此第二次解析成本很低，并且得出相同的结论。第二，代理借用已解析的请求和待处理的字节，不能重新填充连接缓冲区，因此它把更多的客户端字节读入自己的上传缓冲区。由于它无法交还已经读取的内容，只有在请求体已被完整缓冲时，客户端连接才会保持。

代理对请求头部的改写

请求头部不会原样转发。`write_request_head`（`crates/net/src/request.rs`，crate `sibuna-net`）把请求行和每个请求头重新写入源站写入器，同时丢弃逐跳字段（`Connection`、`Keep-Alive`、`Proxy-Connection`、`Transfer-Encoding`、`TE`、`Trailer`、`Upgrade`、代理授权字段）、任何传入的转发或审计字段（`X-Forwarded-For`、`X-Real-IP`、`Forwarded`、`X-Forwarded-Proto`、`X-Forwarded-Host`、`X-Forwarded-Port`、`X-Sibuna-Status`、`X-Sibuna-Rule`）、`Expect`、`Content-Length`，以及请求的 `Connection` 头所指名的每个字段。然后它追加审计字段集合。请求体的分帧由代理生成，从不复制：

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

`Content-Length` 会被重新写出。在连接缓冲区内结束的分块请求体被原地解码，并带着长度发送。更长的请求体以 `Transfer-Encoding: chunked` 声明，并逐次读取、重新分块（SID 0009）：每次读取的数据进入代理的 16 KiB 上传缓冲区，在那里原地解码，然后作为一个规范的块发出，因此源站永远看不到客户端的块大小、扩展或尾部字段。审计字段集合如下：

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

大于缓冲区的请求体以 16 KiB 为单位从客户端中继到源站写入器（relay_body），然后再把响应流式传回。crates/net/src/request.rs 中的单元测试 proxy_head_rewrite_drops_hop_by_hop_headers_and_injects_audit_fields 断言改写会丢弃伪造的 X-Forwarded-For 并注入审计字段。

代理的三个缓冲区，即源站读缓冲区（一个响应头部，16 KiB）、源站写缓冲区（8 KiB）和上传缓冲区（16 KiB），存放在一个 Workspace 中，由连接一次性分配并在每个请求中复用。工作空间使它们的生命周期与连接一致，而不是与单次交换一致，源站读取器和写入器每次只为一次交换借用它们。

中继源站响应

最初的代理把源站的字节一直流式转发到源站关闭连接为止，然后关闭客户端连接：无需解析就能得到正确的分帧，代价是每个被代理的请求都要新建一个 TCP 连接。在负载生成器下，这一代价表现为数以万计处于 TIME_WAIT 状态的套接字，在回环接口上还会耗尽临时端口。现在，代理读取源站的响应头部（最多 16 KiB），并按 RFC 9112 的规则对响应体分类：

> **实现：parse_response_head** 返回状态码、四种分帧方式之一（HEAD、1xx、204 和 304 没有响应体；Content-Length；分块传输编码；或以关闭连接定界），以及源站是否保持连接。位于 crates/net/src/response.rs（crate sibuna-net）。

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

pub fn parse_response_head(
    head: &[u8],
    head_request: bool,
) -> Option<ResponseHead> {
    let status = status_code(head)?;
    let mut fields = FramingFields {
        keep_alive: head.get(7) == Some(&b'1'),
        ..Default::default()
    };
    for line in crlf_lines(head).skip(1) {
        if line.is_empty() {
            break;
        }
        fields.read_line(line)?; // Transfer-Encoding, Content-Length, Connection
    }
    let mut framing = fields.length.map_or(Framing::UntilClose, Framing::Length);
    if head_request || status / 100 == 1 || status == 204 || status == 304 {
        framing = Framing::None;
    } else if fields.encoded {
        framing =
```

```
            if fields.chunked { Framing::Chunked } else { Framing::UntilClose };
    }
    let mut keep_alive = fields.keep_alive;
    if framing == Framing::UntilClose || fields.closing {
        keep_alive = false;
    }
    if fields.nominates_framing {
        return None;
    }
    Some(ResponseHead { status, framing, keep_alive })
}
```

解析器拒绝代理不得中继的内容：错误的状态行，不是 token 的字段名，值中的控制字节，相互冲突的长度，重复的或不在最后的 `chunked` 编码，以及指名 `Content-Length` 或 `Transfer-Encoding` 的 `Connection` 字段。它的工作量与头部长度呈线性关系，并且从不 panic。

响应头部被重新写给客户端，源站的 `Connection` 头被替换为 Sibuna 自己的决定，响应体则被精确中继：有长度时用 `copy_exact`，分块编码时逐块处理（大小行、数据、尾部字段），旧式情形用 `copy_remaining`。只有旧式情形会关闭客户端连接。每种中继都建立在同一个复制步骤之上：转发源端已经持有的所有数据，只有当它什么都没有时，才刷新目标端，然后读取一次。因此，一串小的写入会随产生随交付，而大块传输仍然在每次读写中搬运一整个缓冲区。

```
let keep = client_keep_alive && parsed.framing != Framing::UntilClose;
write_client_head(client, head, keep).map_err(client_head_error)?;
origin.consume(head_length);
match parsed.framing {
    Framing::None => {}
    Framing::Length(length) => copy_exact(origin, client, length, progress)?,
    Framing::Chunked => relay_chunked(origin, client, progress)?,
    Framing::UntilClose => copy_remaining(origin, client, progress)?,
}
client.flush().map_err(|_| ProxyError::ClientFailed)?;
Ok(Relayed { client_keep: keep, origin_reusable: parsed.keep_alive })
```

解析一个包含八个字段的源站响应头部需要 322 ns，通过回环接口代理一个响应为 1 KiB 的 keep-alive `GET` 需要 29.8 us（2026 年 10 月测量；`crates/net/src/lib.rs` 记录了这次测量）。

源站连接池

解析分帧还告诉代理 源站 套接字能否再次使用：一个不带 `Connection: close` 的 HTTP/1.1 响应（或带 `keep-alive` 的 HTTP/1.0 响应），如果其响应体已被完整消费，就会让套接字停在一个干净的请求边界上。这样的套接字进入一个容纳 256 个空闲源站连接的固定连接池，下一个被代理的请求从池中取用，而无需重新连接。没有连接池时，每个被代理的请求都会打开一个新的源站连接，每个关闭的连接都会让它的本地端口停留在 `TIME_WAIT`；在负载下，回环接口上的代理很快就会耗尽临时端口，吞吐量随之崩溃。

如果整个连接池只用一把锁，每个被代理的请求都会争用这把锁。因此连接池（`crates/net/src/pool.rs`）是条带化的：16 个条带，每个条带 16 个套接字，每个条带是一个独占一条缓存行的 `Mutex<Vec<Arc<TcpStream>>>`。线程在首次使用时以轮转方式分配到一个主条带，优先向它归还和从它取用；在重新连接之前，它还会查看其他条带，但不会等待已被持有的锁。

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

池中的套接字可能在空闲时已被源站关闭。代理通过两种方式之一发现这一点：写入失败，或者在第一个响应字节之前遇到流结束。两种情况下都还没有任何内容到达客户端，因此代理关闭该套接字，并在一个全新的连接上重试一次，而绝不在另一个池中套接字上重试：经过一段空闲期之后，池中的每个套接字都可能已经失效，而这个重试的第一个版本取的是第二个池中套接字，结果在每段安静期之后都对第一个请求回答 502。只有对请求体已被完整缓冲的安全方法（GET、HEAD、OPTIONS）才允许重试。没有收到回复并不能证明源站没有提交一次修改，而分段中继的请求体也无法再次发送；这两种情况下，客户端都会收到 502。

```
let safe = matches!(request.method, Method::Get | Method::Head | Method::Options);
let buffered = exchange.upload.is_buffered(request.body.len());
let retryable = safe && buffered;
for attempt in 0..2 {
    // 重试总是新建连接：空闲一段时间之后，池中的每个套接字
    // 都可能已经失效，而再取到一个失效的套接字就会让请求失败。
    let pooled = if attempt == 0 { pool.take() } else { None };
    // ... 池中没有连接时新建连接，挂接到活动跟踪，然后交换
    match outcome {
        Ok(relayed) => {
            if relayed.origin_reusable {
                pool.give(origin);
            }
            return Ok(relayed.client_keep && buffered);
        }
        Err(error) => {
            drop(origin);
            let stale = was_pooled
                && retryable
                && matches!(
                    error,
                    ProxyError::UpstreamClosed | ProxyError::UpstreamWriteFailed
                );
            // ... 其他错误立即报告
        }
    }
}
```

单元测试让固定的字节串通过同样的函数中继，并断言除连接头之外输出逐字节相同（crates/net/src/relay/tests.rs）。守护进程的一个端到端测试在同一个客户端套接字上发送两个被代理的

请求，并通过桩源站添加的序号头检查第二个请求复用了池中的源站连接。crates/net/tests/ 中的一个向量测试重放录制的代理场景（一个原始的客户端请求和一个预制的源站响应），并比较源站和客户端收到的每个字节。

> **为什么源站套接字是 Arc** 套接字描述符是一个小整数，内核在它关闭后会立即复用。关闭流程必须能中断正在进行的交换，却绝不能触碰一个在此期间已被另一个连接复用的描述符。一张加锁的活动描述符表可以防范这种情况，代价是每次交换都要加锁。Sibuna 不需要这样的表：源站套接字是一个 Arc<TcpStream>。连接池持有一个句柄，交换持有一个，在交换运行期间，连接的 Activity 还持有一个克隆。描述符在最后一个句柄被丢弃时才关闭，因此只要还有人能访问它，它就不会关闭，也不会被复用。

空闲期限

每个客户端连接都把它的套接字登记到一个 IdleTable（crates/net/src/idle.rs）中，并获得一个 Lease。租约的 Activity 保存连接上最后一次有字节传输的时间。守护进程在服务完每个请求后更新它；代理在每次读取源站以及每上传 16 KiB 时更新它，并附上正在进行的交换所用的源站套接字。回收线程每隔最短超时的四分之一（介于 200 ms 和一秒之间）醒来一次，关闭所有静默时间超过其超时的连接以及附带的源站套接字，使阻塞在任一端的线程看到流结束并释放它的槽。

因此，期限衡量的是静默时间，而不是响应的长度：一个持续传输的大型下载永远不会被切断。上传字节来自不可信的一方，只有每 16 KiB 才刷新一次时间戳，这要求客户端在每个空闲期内至少发送 16 KiB，即最低速率请求超时所提供的慢速请求体防御。

这张表每个槽有一把小锁，没有全表锁。出于上面给出的原因，它保存每个客户端套接字的一个 try_clone，以及每个附带源站套接字的一个 Arc。Lease 在 Drop 中注销它的连接，被回收的槽在此之前一直归该租约所有，因此旧连接永远不会释放一个已被新连接占用的槽。

升级后的连接

在一个经过验证的 101 之后，WebSocket 升级连接会被双向中继。每个方向在自己的线程上运行，使用一个固定的缓冲区：连接自己的线程传输从源站到客户端的字节，一个拥有 128 KiB 栈的作用域线程传输从客户端到源站的字节。一个线程可以借助 poll(2) 同时服务两个方向，但 Rust 标准库没有提供它；使用两个带套接字超时的线程可以让中继不含 unsafe。因此，一个升级后的连接占用两个线程，当所有连接都被升级时，连接线程的数量可以达到 --max-connections 的两倍。

两个方向共享一个空闲期限（--websocket-idle-timeout，默认 300 秒）：任一方向的字节传输都会刷新同一个时间戳。每次套接字等待都带有一个等于剩余时间的超时；超时的等待会重新读取共享时间戳，然后要么继续等待剩余时间，要么结束中继。当一个方向到达流结束时，另一个套接字的发送端被关闭（半关闭）；当一个方向失败时，两个套接字都被关闭，使另一个方向也随之结束。在代理一次升级之前，连接会释放它缓存的策略引擎，使一个可能存活数小时的连接不会让旧引擎一直存活。

停止

Running::stop，以及同样作用的 Running 的 Drop，按固定顺序停止守护进程。它设置停止标志，通过连接监听器唤醒每个接受 worker，join 这些 worker，关闭每个已登记的客户端套接字及其附

带的源站套接字（IdleTable::shutdown），停止源站连接池，join 每个连接线程，join 回收线程，最后关闭池中的套接字。

SIGINT 和 SIGTERM 通过普通的 Rust 代码进入这一流程。在创建任何线程之前，守护进程先屏蔽这两个信号（之后的每个线程都会继承这个掩码），并专门用一个线程执行 sigwait，它记录停止请求并立即唤醒主线程。没有任何代码在信号处理函数中运行，也没有线程轮询标志。信号模块（cmd/sibuna/src/signals.rs）包含守护进程中唯一的 unsafe 代码：即 std 没有封装的少数几个 C 库函数的声明。

> **通过 Drop 释放** 显式的清理调用必须按与设置相反的顺序编写，并且每个提前返回都必须对照它们检查。在 Sibuna 中，每个资源都在 Drop 中释放自己：Daemon 和 Running 停止服务器，Lease 注销它的连接，Counted 归还连接上限中的一个名额，Finished 释放一个任务槽。编译器在离开作用域的每条路径上（包括穿过它展开的 panic）插入相应的释放，因此没有任何路径会遗漏。

我们测得了什么

下表给出 Shield 在一台 Ryzen 9 5950X 上以反向代理模式运行的结果：守护进程绑定到四个 CPU，wrk 以两个线程和 64 个连接运行在另外八个 CPU 上，每次运行五秒，取三次的中位数。各行抄自 benchmarks/results/product-latest.md；完整记录见第 VIII 部分。

表 7 反向代理模式下的 Shield，数据来自 benchmarks/results/product-latest.md

工作负载	状态码	req/s	p50 / p99 (us)	每请求 CPU 时间 (us)
带会话准入，经代理	200	138,965	438 / 857	28.57
被挑战（验证中间页）	200	195,188	162 / 328	13.73
静态资源放行，经代理	200	145,673	415 / 746	27.27
被 WAF 拒绝（带会话）	403	209,507	151 / 306	12.17

wrk 始终保持 64 个请求在途，因此在受 CPU 限制的代理路径上，请求会排队等待四个 CPU，那里的 p99 反映的是调度器的尾部，而不是单个请求的成本。开环生成器以固定速率发送，并从每个请求的预定发送时刻开始计时，它测量的是相同施加负载下的延迟。对于反向代理模式下被准入的请求，每秒 10,000 个请求时 p99 为 65 us，每秒 100,000 个请求时为 100 us；施加每秒 150,000 个请求时，四个 CPU 达到饱和，实际完成每秒 134,344 个请求，队列增长到秒级（benchmarks/results/latency-latest.md）。一个预热过的 keep-alive 连接在处理每个决策时都没有堆分配：cmd/sibuna/tests/steady_state.rs 统计守护进程的分配次数，crates/net/tests/steady_state.rs 在 500 个经代理的 keep-alive 请求中没有发现任何分配。

练习 5.1. 一个客户端上传 200 KB 的数据。追踪哪些字节位于 64 KiB 缓冲区中，哪些由 `relay_body` 经上传缓冲区中继，以及为什么代理必须 先 把头部写给源站，然后才能中继。

无分配的并发状态

学习目标 分析分片锁、Robin Hood 已用集合、GCRA 限制器、带版本号的封禁表、MPSC 事件环，以及借助 `Arc` 和按代次检查的缓存实现的策略引擎读-复制-更新发布。

`sibuna-store` 中的每个存储都在构建时一次性分配其固定容量，之后从不分配内存，也从不阻塞在全局锁上。它们都不需要后台清扫线程：过期状态在原地回收。

锁与分片

Sibuna 各张表中的临界区只有几十条指令，远短于一次内核 futex 往返。表按键的哈希分成 16 个分片，因此两个操作发生争用的概率为 1/16；每个分片独占一条缓存行，因此相邻的锁不会发生伪共享。

这些表使用 `std::sync::Mutex`：无争用时一次比较并交换，持有者运行期间短暂自旋，然后进入 futex 等待。另一种选择是双状态的原子自旋锁，在等待循环中使用自旋提示。SID 0010 要求我们通过测量在两者之间作出选择。自旋锁在每次无争用的操作上大约节省一纳秒，但在争用下会落败：

表 8 Ryzen 9 5950X 上的分片锁吞吐量（2026 年 10 月）

工作负载	`Mutex`	自旋锁
已用集合，16 个线程	10.8 M/s	8.6 M/s
已用集合，64 个线程（多于 CPU 数）	11.2 M/s	6.9 M/s
在六个热点地址上运行 GCRA，16 个线程	44.9 M/s	19.3 M/s
在六个热点地址上运行 GCRA，64 个线程	37.4 M/s	6.5 M/s

边缘服务器运行的连接线程多于 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，因此我们保留互斥锁。中毒的锁会被直接进入，而不是传播错误：没有任何操作会在持有分片时 panic，即使发生了，分片在结构上也仍然有效。

Robin Hood 已用集合

只有被接受的证明才会进入已用集合（第 III 部分）。它有 16 个分片，每个分片是一个开放寻址数组，包含 4,096 个 `Option<Entry>` 槽，其中条目为 `{ tag: [u8; 16], expires_at: u64, dist: u8 }`。Robin Hood 插入（Celis，1986）会挤走任何比新来的键更靠近自己初始位置的占用者，从而使探测长度紧密集中；查找一旦遇到一个比自己更靠近其初始位置的条目，就会停止。

实现：Table::insert 插入一个带过期时间的已用标签，并在途中回收过期的槽。位于 crates/store/src/spent.rs（crate sibuna-store）。

```
fn insert(
    &mut self,
    tag: &Tag,
    expires_at: u64,
    now: u64,
) -> Result<(), SpendError> {
    if !self.can_insert(tag, now) {
        return Err(SpendError::Full);
    }
    let mut carry = Entry { tag: *tag, expires_at, dist: 0 };
    let mut idx = Self::home(tag);
    while carry.dist < MAX_PROBE {
        let slot = &mut self.entries[idx];
        let Some(e) = slot else {
            *slot = Some(carry);
            return Ok(());
        };
        // 只有当新的探测距离不更小时，覆盖已过期的占用者才是安全的，
        // 这样，探测越过此槽的键仍能通过
        // 提前终止检查。
        if e.is_expired(now) && carry.dist >= e.dist {
            *e = carry;
            return Ok(());
        }
        if carry.dist > e.dist {
            std::mem::swap(&mut carry, e);
            // 被挤出的过期条目直接丢弃。
            if carry.is_expired(now) {
                return Ok(());
            }
        }
        idx = (idx + 1) % SHARD_CAPACITY;
        carry.dist += 1;
    }
    Err(SpendError::Full)
}
```

过期槽的规则是其中微妙的部分：只有距离自己初始位置至少同样远的键，才能原地覆盖一个过期条目，否则一个探测会越过该槽的后来键将提前停止并丢失。被挤走的过期条目直接丢弃。不需要清扫线程。插入时先在不改变任何内容的情况下模拟挤占路径（can_insert），确认有界的挤占可以成功且不会丢失任何存活的标签。

不变量 被拒绝的插入永远不会逐出存活的标签。因此，一个已被使用且尚未过期的标签总能被找到，而拒绝（SpendError::Full）可能导致拒绝服务，但永远不会让重放成为可能。

一个槽是 `Option<Entry>`。空槽中没有可读的标签，因此不存在读者可能忘记检查的 `occupied` 标志，而且在两种情况都被处理之前，`match` 无法通过编译。错误类型同样精确：已用集合有两种失败方式（`DoubleSpend` 和 `Full`），`SpendError` 恰好列出这两种。

GCRA：用一个整数实现速率限制

通用信元速率算法（ATM Forum，1996）是漏桶算法的虚拟调度形式。它为每个客户端保存一个 理论到达时间（TAT）。对于每个窗口 W 内的正限额 N，发射间隔为 $T = \max(1, \lceil W/N \rceil)$，突发容限为 $\tau = (N-1)T$：

$$\text{arrival at } t \text{ conforms} \iff \mathrm{TAT} \le t + \tau, \quad \text{then } \mathrm{TAT} \leftarrow \max(\mathrm{TAT}, t) + T.$$

定理（令牌桶上界） 在 t_1 处的第一次合规到达之后，第 k 次合规到达满足 $t_k - t_1 \ge (k-1)T - \tau$。因此，任何长度为 L 的时间段内至多有 $N + \lfloor L/T \rfloor$ 次合规到达：先是 N 次突发，然后每个 T 一次。除了定义的突发之外，不存在固定窗口边界造成的异常，也不需要保存每个请求的历史。

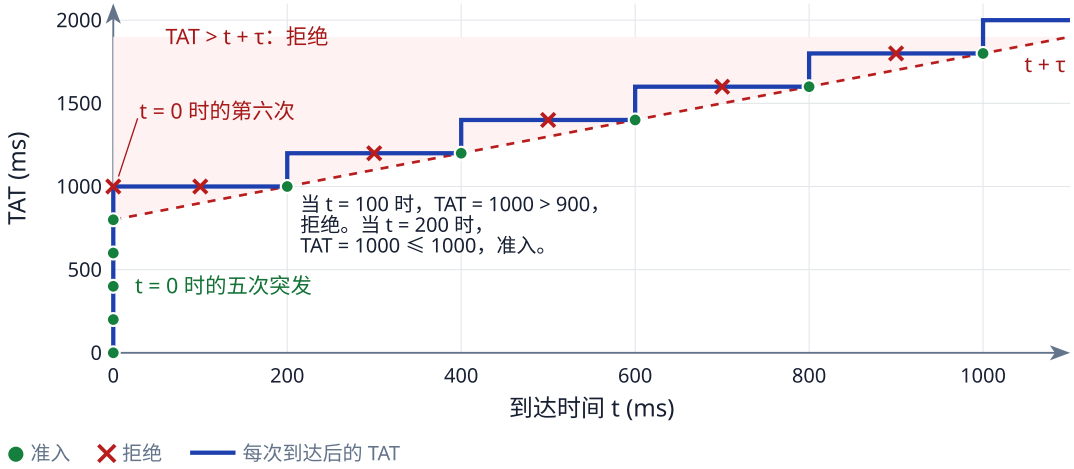


图 7 限额为每秒 5 次时的 GCRA 准入：先是五次突发，然后每 200 ms 一次。到达时若 TAT 位于直线 $t + \tau$ 上或其下方，就会被准入。

实例分析：手工计算每秒五次 取 $N = 5$，$W = 1000$ ms，于是 $T = 200$ ms，$\tau = 800$ ms；让一个一直空闲的客户端在 $t = 0$ 时发送六个请求。第一个请求遇到一个空单元：$\mathrm{TAT} = \max(0, 0) = 0 \le 800$，因此它合规，并且 $\mathrm{TAT} \leftarrow 200$。接下来的四个请求以同样方式合规，使 $\mathrm{TAT} = 1000$。第六个请求发现 $1000 > 0 + 800$，被拒绝，`Retry-After` 为 $\mathrm{TAT} - \tau - t = 200$ ms。在 $t = 199$ 时，请求仍然发现 $1000 > 999$，被拒绝；在 $t = 200$ 时，它发现 $1000 \le 1000$，合规，并使 $\mathrm{TAT} = 1200$。`crates/store/src/limiter.rs` 中的单元测试 `gcra_admits_exactly_the_burst_then_drains_at_the_emission_rate` 正是检查这些数字。

实现：arrive 把一次到达应用到已加锁分片的单元上，并返回决定、重试延迟和剩余预算。位于 crates/store/src/limiter.rs（crate sibuna-store）。

```
fn arrive(
    cells: &mut [Cell; SLOTS_PER_SHARD],
    key: u64,
    now_ms: u64,
    limits: Limits,
) -> Decision {
    let interval = limits.emission_interval();
    let tau = limits.burst_tolerance();
    let Some(cell) = locate(cells, key, now_ms, tau) else {
        return Decision {
            limited: true,
            capacity_exhausted: true,
            retry_after_ms: interval,
            remaining: 0,
        };
    };
    let tat = cell.tat_ms.max(now_ms);
    let horizon = now_ms.saturating_add(tau);
    if tat > horizon {
        return Decision {
            limited: true,
            capacity_exhausted: false,
            retry_after_ms: tat - tau - now_ms,
            remaining: 0,
        };
    }
    cell.tat_ms = tat.saturating_add(interval);
    // ... remaining = min(rate, floor((t + tau - TAT) / T) + 1)
}
```

分片的 check 获取锁，并用已加锁的单元调用 arrive；check 返回时，守卫释放锁。单元为 16 字节（一个键和一个 TAT），分布在 16 个分片中，每个分片 512 个槽，探测窗口为 16 个槽，总计约 128 KiB。TAT 早于 $t - \tau$ 的单元已经排空，会被当场回收。Retry-After 响应头由同样的算术计算得出。探测窗口饱和时，会拒绝新客户端，而不是逐出活跃的配额状态，并且决定中会注明这一点（capacity_exhausted），使调用方不会把饱和造成的拒绝升级为封禁。配置的速率为零时拒绝请求。一个限制器可服务多条规则：单元的键是以规则作用域为种子、对客户端地址计算的 Wyhash，因此同一个客户端在不同作用域中拥有相互独立的预算。

带版本号的封禁表

封禁使用 4,096 个槽，每个槽具有原子的版本、键和过期时间字段，按被封禁地址的带密钥哈希进行开放寻址，探测窗口为 8 个槽。写者通过互斥锁与其他写者串行执行，并在每次更新前后各递增一次版本号；读者不加锁，只接受稳定的偶数版本号，遇到并发修改时重试。

实现：BanTable::ban 和 BanTable::read_slot 写者的前后递增，以及读者获得的一致快照。位于 crates/store/src/ban_table.rs（crate sibuna-store）。

```
pub fn ban(&self, banned: &[u8], until: u64, now: u64) {
    let key = slot_key(banned);
    let _writers = self.lock_writers();
    let victim = &self.slots[self.victim(key, now)];
    // 键和过期时间是一个逻辑值：替换期间让版本号保持为奇数，
    // 使读者无法把旧键与下一个键的过期时间
    // 拼在一起。
    victim.version.fetch_add(1, Ordering::SeqCst);
    victim.until.store(until, Ordering::SeqCst);
    victim.key.store(key, Ordering::SeqCst);
    victim.version.fetch_add(1, Ordering::SeqCst);
}

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

顺序一致性防止把旧的身份与替换后的过期时间混在一起：每次访问都属于同一个全序，这使论证保持简单。读者不使用互斥锁，但可能需要等待写者，因此不是无等待的。这些原子操作有实际的成本，这一成本在 HTTP 测量工具中间接测得，而不是假定为零。探测窗口已满时，覆盖一个已过期的槽，否则覆盖最先过期的槽。`clear` 解除所有封禁，但保留键不动，因为一个被清空的槽可能会让探测链中后面一个发生碰撞的键无法被找到。

MPSC 事件环

当 Shield 形态拒绝一个请求或蜜罐被触发时，事件被复制进一条约 1.3 KB 的固定记录（客户端地址、User-Agent、方法、路径、类别和载荷，各自按容量截断），并推入一个有界的多生产者单消费者环（Vyukov 的带序号设计，512 个槽，`BoundedQueue<IncidentRecord, 512>`）。生产者是连接线程；消费者是存储线程。环满时丢弃最新的记录，而不是阻塞响应，这是一种明确的丢失策略，而不是持久性保证。丢弃计数器使这种丢失可以被观察到。

```
/// 把 `incident` 复制进环形缓冲区；缓冲区已满时丢弃它，并为丢弃计数。
pub(crate) fn record(&self, incident: &Incident<'_>) {
    if self.queue.push(IncidentRecord::from_incident(incident)).is_err() {
        self.metrics.incidents_dropped.bump();
    }
}
```

把一条记录移入待处理批次之后，存储线程会连同准备好的行一起保留它，直到提交得到确认；遇到数据库错误时，它不会悄悄丢弃这条记录，重试时写入的是完全相同的批次。

每个槽带有一个序号，它告诉生产者该槽是否空闲，告诉消费者该槽是否已满，因此双方都不需要加锁。这样的环无法用安全的 Rust 编写：生产者和消费者同时访问同一个数组的不同槽，而证明哪个线程可以访问哪个槽的，只有序号协议，而不是类型系统。因此，模块 `crates/store/src/ring.rs` 把值存放在 `UnsafeCell<MaybeUninit<T>>` 槽中，每个 `unsafe` 块都写明了为其提供依据的协议步骤：

```
if diff == 0 {
    // Relaxed: CAS 只裁定哪个生产者拥有位置 `pos`。
    match self.tail.0.compare_exchange_weak(
        pos, pos.wrapping_add(1), Ordering::Relaxed, Ordering::Relaxed,
    ) {
        Ok(_) => {
            // SAFETY: CAS 把位置 `pos` 分给了本线程，
            // 而 `sequence == pos` 表明该槽为空；其他生产者无法
            // 占用 `pos`，消费者在下面的 Release 存储之前
            // 也不会读取它。
            unsafe { (*slot.value.get()).write(value) };
            // Release: 把写入的值发布给消费者。
            slot.sequence.store(pos.wrapping_add(1), Ordering::Release);
            return Ok(());
        }
        Err(current) => pos = current,
    }
} else if diff < 0 {
    return Err(value);
}
```

这个环的两个约定是用类型而不是注释表达的。环满时，`push` 返回 `Err(value)`，把记录交还给调用方，由调用方决定计数还是丢弃；单纯返回 `false` 会让记录丢失。这个环只有一个消费者，`pop` 接受 `&mut self`，共享代码必须先取得唯一的 `Consumer`，因此第二个消费者要么无法编译，要么在运行时被拒绝。

读-复制-更新式的引擎发布

策略变化时，策略引擎在热路径之外重建。请求绝不能看到构建了一半的引擎，引擎也绝不能在仍有请求读取它时被释放。Sibuna 把每个引擎作为 `Arc<Engine>` 发布到一个 `EngineSlot` 中，每个连接通过一个 `EngineCache` 读取它：

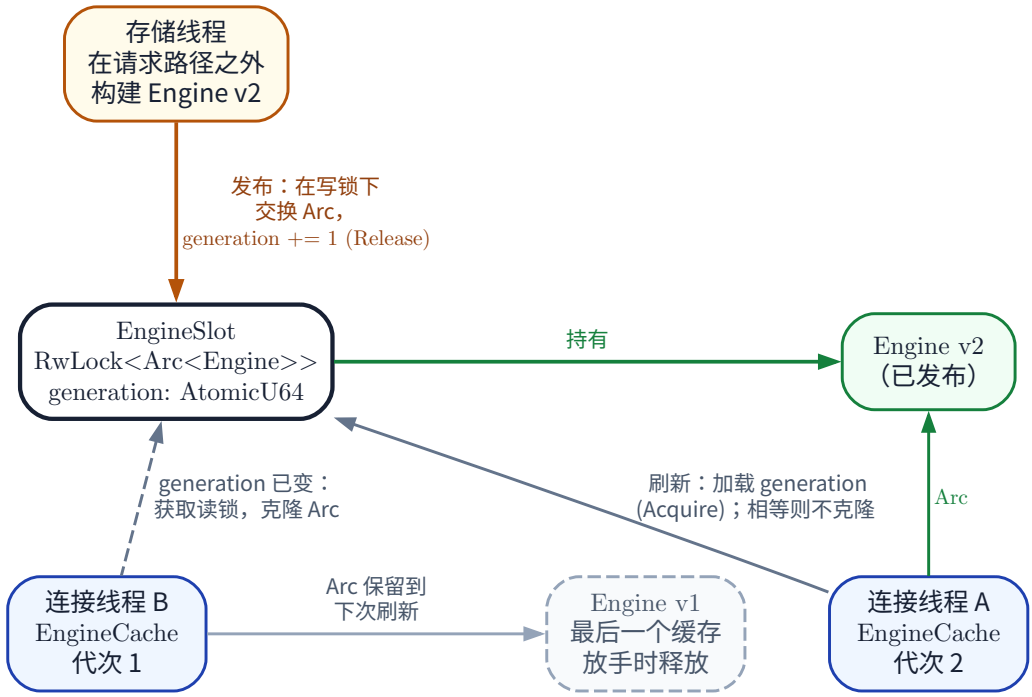


图 8 引擎发布。存储线程发布一个完整的引擎并推进代次；每个连接用一次 acquire 加载检查代次，只有在发生发布之后才克隆已发布的 Arc。旧引擎在最后一个持有它的缓存放手时被释放。

实现：`EngineSlot::publish` 和 `EngineCache::refresh` 完整引擎的发布，以及一个连接对它的视图。位于 `cmd/sibuna/src/slot.rs`（crate `sibuna`）。

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

pub fn publish(&self, engine: Engine) {
    let fresh = Arc::new(engine);
    let previous = {
        let mut current =
            self.current.write().unwrap_or_else(PoisonError::into_inner);
        std::mem::replace(&mut *current, fresh)
    };
    self.generation.fetch_add(1, Ordering::Release);
    // 旧引擎在锁外释放（如果这里是它的最后一个持有者）。
    drop(previous);
}

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

```rust
pub fn refresh(&mut self, slot: &EngineSlot) -> &Engine {
    let generation = slot.generation();
    if self.generation != generation {
        // 若在上面的加载与这次克隆之间发生一次发布，较新的引擎会被
        // 标上较旧的代次；下一次刷新会再次加载，问题随之消失。
        self.generation = generation;
        self.engine = None;
    }
    self.engine.get_or_insert_with(|| slot.load())
}
```

代次在引擎就位之后以 release 顺序存储，并以 acquire 顺序加载：看到新代次的读者也能看到在它之前存入的引擎。注释中提到的竞争是良性的。如果一次发布恰好发生在读者加载代次和克隆之间，读者会把较新的引擎缓存在较旧的代次下，下一次刷新时会发现不匹配并重新加载。读者可能在一个请求中晚一个引擎；但它永远看不到构建了一半的引擎，因为存入槽中的只有完整的引擎。

不变量 请求读取的每个引擎都是完整的，并且只要还有任何请求或连接缓存持有某个引擎，它就不会被释放。前者成立，是因为引擎在 `publish` 存储它之前已经完全构建好；后者成立，是因为每个读者都持有一个 `Arc`，而引擎只会随最后一个 `Arc` 一起被丢弃。

为什么用缓存，而不简单地用锁？SID 0010 要求我们测量 16 个线程下的读取成本，并在 `RwLock<Arc<Engine>>` 和安全 Rust 中的一种无锁方案之间作出选择。通过锁读取要访问锁字和引用计数，这两条缓存行每个核心都会写入。相比之下，代次每次发布只写一次，其余时间只被读取，因此每个核心都能保留自己的副本：

表 9 一次引擎读取的成本，Ryzen 9 5950X，16 个线程循环读取（2026 年 10 月）

读取方式	1 个线程	16 个线程
仅 `RwLock` 读守卫	4.2 ns	752 ns
`RwLock` 读取加 `Arc` 克隆	7.6 ns	833 ns
用缓存的 `Arc` 检查代次	0.8 ns	1.8 ns

在争用下，缓存读取比锁便宜 400 多倍，因此请求路径使用缓存，只在刷新的慢路径上使用 `RwLock`。这项测量是 `cmd/sibuna/src/slot.rs` 中被标记为忽略的测试 `read_cost_under_contention`。请求路径上的每次使用都会刷新，因此请求在每一步看到的都是当时的引擎。引擎本身很小：209,052 字节，约 204 KiB 的紧凑堆表（`benchmarks/results/primitives-latest.json` 中的 `engine_bytes`），可容纳 128 条规则、128 字节的规则名称和 8,192 个字典树节点。

为什么用 `Arc` 而不是读者计数 固定内存的替代方案保留两个引擎槽，每个槽有一个读者计数和自己的内存分配区（arena）。请求通过递增计数并重新检查槽指针来固定一个槽，而且两个原子操作都必须是顺序一致的，才能消除写者在读者加载和递增之间进行交换的竞争。构建者交换指针，然后自旋等待旧槽的读者全部退出，再重置该槽的内存区并在其中

重建。因此，一个在重建期间持有槽的读者会阻塞构建者；除非每个请求都在代理 I/O 开始之前释放它的槽，否则一个慢速源站会拖住下一次发布。

`Arc` 引用计数完全不需要这些。存储线程从每个快照构建一个全新的 `Engine` 并发布它；没有任何复用，因此没有备用槽，没有内存区重置，也没有自旋：`publish` 从不等待读者。旧引擎只是一直存活到最后一个持有者放手为止，慢速源站只能推迟这一时刻，而永远不会推迟下一次发布。判定结果把匹配的规则名称复制进一个 128 字节的缓冲区，以结束一次借用：决策通过连接的缓存借用引擎，而响应路径需要以可变方式访问整个连接上下文。一个可能存活数小时的升级连接，在被代理之前会释放它缓存的引擎。

练习 5.2. 利用 GCRA 定理，计算在 `--rate-limit 100 --rate-window 10` 下，单个客户端在空闲之后的前 3 秒内最多能通过多少个请求。与固定 10 秒窗口计数器以及真正的滑动窗口日志进行比较。

解释这个不变量. 用一个三槽的例子解释，为什么一个过期的 Robin Hood 条目只能被距离至少同样大的键覆盖。

第 VI 部分

自动机、字典树与语义防火墙

我们考察亚微秒级分类背后的算法：带标记的 *Aho-Corasick* 自动机，*IPv4/IPv6* 基数字典树，语义 *WAF* 的字节类分词器，以及带 *WEIGH* 评分的声明式策略引擎。

单遍多模式匹配

学习目标 学完本章，我们应当能够推导 Aho-Corasick 自动机的线性时间保证，解释稠密表布局以及通过字节类实现的大小写折叠，并论证为什么没有采用 SIMD 字面量引擎。

作为稠密 DFA 的 Aho-Corasick

Sibuna 有两个字面量模式集合：大约四十个机器人签名和大约一百个攻击签名。逐个地把一个字段与 N 个模式比较，复杂度为 $O(NM)$；Aho 和 Corasick（1975）的自动机在构建带失败链接的字典树之后，只需以 $O(M)$ 扫描一遍。Sibuna 在构建时把失败链接折叠进转移表，因此扫描时每个输入字节只需一次表读取，不需要针对模式集合进行分支。自动机位于 `sibuna-patterns` crate 中：

```
/// 从 `state`（一个行偏移量）出发，读入 `byte` 后前进一步。
fn step(&self, state: usize, byte: u8) -> usize {
    self.table[state + usize::from(self.classes[usize::from(byte)])] as usize
}

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

状态以它在一张扁平表中的行偏移量存储，因此一步只是一次加法和一次读取。状态的编号使每个匹配状态都排在所有其他状态之后；于是检测匹配只需与 `first_match` 比较一次，内层循环中不必再读取第二张表。

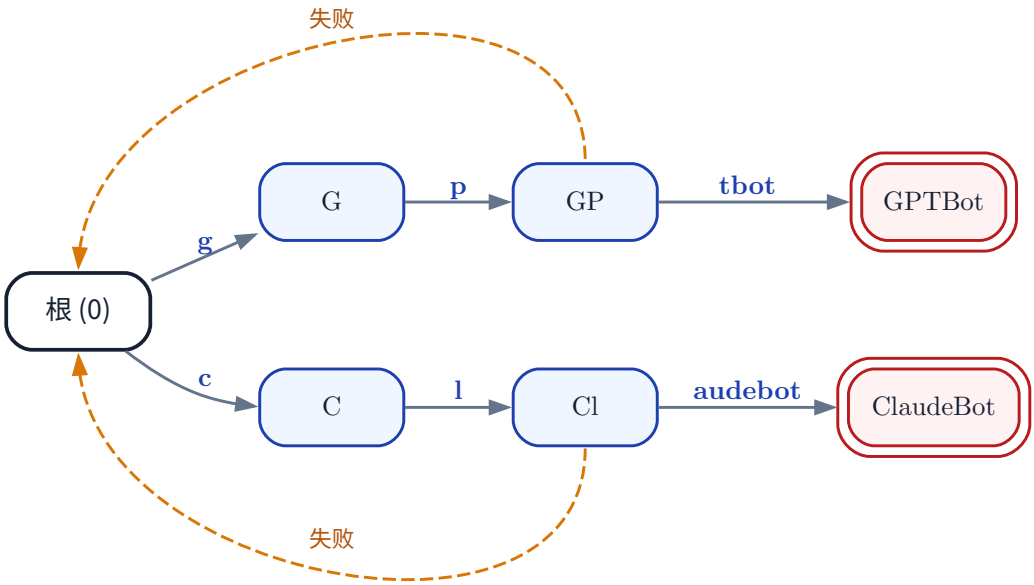


图 9 两个签名的字典树状态和失败边

自动机的状态预算是一个运行时选项。策略引擎最多预留一个根加上所有签名长度之和，这是字典树状态数的一个安全上界。这样既避免了以往过大的固定表，又为每个字节类保留了稠密的转移。每个模式带有一个小的类别标记（0 到 63，因此一组标记就是一个 64 位字）。大小写折叠使用一张随自动机一起构建的 256 项字节类表：一个字母的大小写形式共用一个类，所有不出现在任何模式中的字节共用一个类，因此表的一行的宽度等于不同模式字节的数量加一，而不是 256。确切的引擎大小，以及与顺序子串搜索的比较，记录在第 VIII 部分中。后者使用相同的模式和“任一匹配”语义。

正则表达式、SIMD 引擎与最终选择

为签名阶段权衡过三类方案：

- 回溯式正则表达式（回溯规则集的风格）每个请求要运行数百个模式，每次匹配都要分配内存；它们是最慢的选项，也最容易产生误报。
- *SIMD* 字面量引擎（Hyperscan、Vectorscan）用基于 shuffle 的半字节掩码（“Teddy”算法）对 16 到 32 字节的数据块进行预过滤，再用自动机验证候选。它们擅长处理带有大型字面量集合的长输入。代价是一个数兆字节的 C++ 依赖、一个编译期的模式数据库、不支持 WebAssembly 目标，而且对于一百字节左右的输入没有优势：扫描整个字段的时间还不到一次缓存未命中。
- 稠密 *Aho-Corasick* 加小型结构分词器，即 libinjection 一脉的做法，在一张固定表中提供精确的线性时间行为，并且不需要可执行文件之外的任何东西。

Sibuna 的字段很短，因此第三类方案在内存和速度上都胜出；第 VIII 部分记录了 8 KiB 请求体上约每字节 1.56 ns 的成本（语义扫描耗时 12,784.3 ns，见 `benchmarks/results/primitives-latest.md` 中的 `8kb_body_semantic_scan`）。这是 SIMD 预过滤唯一可能值得的工作负载，SID 0006 把它列为一个待定事项。

IPv4 和 IPv6 的基数字典树

> **学习目标** 在 128 位字典树中追踪一次最长前缀查找，解释 IPv4 的根捷径，并了解这棵字典树如何兼作由集群供数的信誉存储。

一棵基于 128 位键的二叉字典树同时服务两个地址族：IPv6 前缀按原样插入，IPv4 前缀被映射到 `::ffff:0:0/96` 中，因此一个 `/24` 会成为深度为 120 的路径。节点存放在一个以 `u32` 为索引的扁平 `Vec` 中，索引零既表示根也表示“无子节点”，因此一次查找最多是 128 次相互依赖的读取，没有指针追逐。这棵字典树就是 `sibuna-cidr` crate 中的 `PrefixTrie`。
如果每次 IPv4 查找都走完映射前缀的 96 层，那么在到达 IPv4 地址的第一个比特之前，就要付出 96 次相互依赖的读取。因此，字典树在初始化时记录深度 96 处的节点，并从那里开始 IPv4 查找：

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

分类一个 IPv4 地址耗时 32.0 ns，一个 IPv6 地址耗时 79.3 ns（`benchmarks/results/primitives-latest.md` 中的 `ipv4_cidr_classification` 和 `ipv6_cidr_classification`）。引擎会查询字典树两次：在规则表之前查询终止性的 `allow`/`deny` 判定，使集群范围的封禁优先于每条规则；在规则表之后查询 `challenge` 判定。

语义防火墙

> **学习目标** 理解字节类预扫描、带标记的签名自动机、带恒真式检测的 SQL 分词器、HTML 和 shell 结构检查、规范化，以及误报纪律：真实的浏览器请求必须原封不动地通过。

关于误报的一课

WAF 的第一个版本把一个扁平的子串列表与每个请求头进行匹配。列表中包含 `/*`，而每个浏览器都会发送 `Accept: text/html,...,*/*;q=0.8`。于是每个真实的浏览器都被分类为 SQL 注入，并收到 `403`。这一教训塑造了设计：检测器只能针对 结构 触发；测试套件现在会让一个完整的 Chrome 请求（包括一个含有 `select`、`--` 和撇号的评论正文）通过引擎，并断言它被判定为干净。

单遍收集字节类

每个被检查的字段只扫描一遍，收集其中出现了哪些字节类：引号、`=`、`<`、shell 分隔符、`%`、NUL、规范化触发字节和连续空格。每个检测器都查询这些字节类，而不是重新扫描；只有出现了触发字节时，才运行规范化后的第二遍。扫描对每个字节查一次 256 项的表，代码来自 `sibuna-policy` 的 WAF 模块。四个累加器使 OR 链保持简短；只有当没有任何单个字节已经要求规范化时，才搜索两种字节对（`/*` 和两个相邻的空格）：

```
pub(crate) fn scan(text: &[u8]) -> Classes {
    let (chunks, remainder) = text.as_chunks::<8>();
    // 四个累加器让 OR 链保持很短，使各次加载可以背靠背地发出。
    let mut lanes = [0u16; 4];
    for chunk in chunks {
        lanes[0] |= TABLE[usize::from(chunk[0])] | TABLE[usize::from(chunk[4])];
        lanes[1] |= TABLE[usize::from(chunk[1])] | TABLE[usize::from(chunk[5])];
        lanes[2] |= TABLE[usize::from(chunk[2])] | TABLE[usize::from(chunk[6])];
        lanes[3] |= TABLE[usize::from(chunk[3])] | TABLE[usize::from(chunk[7])];
    }
    let mut bits = lanes[0] | lanes[1] | lanes[2] | lanes[3];
    for &byte in remainder {
        bits |= TABLE[usize::from(byte)];
    }
    // 只有当还没有任何单个字节要求规范化时，这些字节对才有意义。
    if bits & CANONICAL == 0
        && ((bits & SPACE != 0 && has_pair(text, b' ', b' '))
            || (bits & (SLASH | STAR) == SLASH | STAR && has_pair(text, b'/', b'*')))
    {
        bits |= CANONICAL;
    }
    Classes { bits }
}
```

检测器

- 签名。所有类别的所有强签名都位于同一个带标记的自动机中；命中即为终止性的 `deny`，名称为 `waf:sqli`、`waf:xss`、`waf:path-traversal` 或 `waf:rce`。
- *SQL* 注入。一个单遍的单词扫描器在关键字表中查找每个字母数字单词（最多七个字节），遇到 `or` / `and` 时检查是否存在 `literal = literal` 形式的恒真式；`id=1 or 1=1` 是终止性的。否则必须出现引号字节，并且证据得分至少为 4，计分项为：引号（1），最多两个关键字（2），注释标记（1），一个 `=`（1）。`admin'--` 本身就是终止性的。“it's a group order from the shop” 得 3 分，可以通过。
- 跨站脚本。要求出现 `<`；当一个可以承载脚本的标签与处于属性位置的 `on<event>=` 属性同时出现时触发。`onboarding=1` 不算。
- 命令注入。 要求出现分隔符；当 `;`、`|`、`&&`、`$(`、反引号或换行后面跟着一个已知的命令名时触发。`status ok; done | next` 可以通过。
- 路径遍历。 签名（`../`、`..%2f`、`/etc/passwd`、`id_rsa` 等），外加 NUL 和 `%00`。

语法中合法地包含引号和星号的结构性请求头（`Accept*`、`Content-Type`、`Sec-*`、`If-*` 等）会被跳过；路径、查询、User-Agent、其他请求头以及请求体的前 8 KB 会被检查。含有百分号转义、加号、注释起始符或可折叠空白的字段，会被规范化到一个 8 KB 的栈缓冲区中再检查一次，因此 `1%27/**/UnIoN/**/SeLeCt` 和 `%252e%252e/` 都会归结为原始签名。

成本

检查成本取决于提交给引擎的字段和字节，而不仅仅是请求的数量。基准测试一章分别报告了 Gate、Shield 和 8 KB 请求体的情形。8 KB 的上界是一个检查限制，并不意味着上传内容的其余部分是安全的。使用 forward-auth 的入口代理必须传递相关的请求信息；被省略的字节无法由这个进程检查。

声明式策略引擎

> **学习目标** 阅读规则模型、评估顺序、带阈值的 WEIGH 评分，以及 JSON 策略语法。

一条规则是若干可选条件的合取（路径模式、User-Agent 模式、最多四个请求头、最多八个 IPv4/IPv6 CIDR 块、一个 JA4 指纹或前缀），带有一个动作 `ALLOW`、`DENY`、`CHALLENGE` 或 `WEIGH`，可选的按规则设定的难度（以工作比特计）和算法，以及一个有符号的 `weight`。评估顺序如下：

1. 语义 WAF（仅 Shield）：违规即为终止性的 `deny`。
2. 信誉字典树：`deny` 或 `allow` 前缀是终止性的。
3. 按顺序评估规则：第一个终止性匹配直接返回；`WEIGH` 匹配累加分数。
4. 分数：总分为负时 `allow`；总分 ≥ 40 时 `deny`；总分 ≥ 10 时 `challenge`，超过阈值的部分每 5 分增加一个工作比特，最多增加六个。
5. 静态绕过路径，然后是字典树的 `challenge` 判定，然后是机器人自动机。
6. 默认动作，除非策略文件另有规定，否则为 `challenge`。

内置规则放行 `/.well-known/*`、`/favicon.ico`、`/robots.txt` 和 `/__sibuna/*`，拒绝 `CF-Worker` 客户端和 `Amazonbot`，并对每个 `Mozilla` User-Agent 发起挑战：浏览器字符串可以随意伪造，准入它会让网关形同虚设。

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

逐步追踪：持有有效会话的编码攻击

考虑一个持有有效会话的客户端发出的 `GET /search?q=%2527%2520OR%25201%253D1`。首先，HTTP 解析器在不复制字节的情况下分离路径和查询。接着执行本地限制，然后由 Shield 检查请求。百分号把该字段标记为需要规范化。第一遍解码得到 `%27%20OR%201%3D1`；第二遍得到 `' OR 1=1`。此时分词器看到一个引号、关键字 `OR` 和相等的字面量操作数。拒绝是终止性的策略结果。有效会话不会把它变成放行：会话检查只用于解决挑战决定。存储钩子复制一条有界的事件记录；SQL 稍后才执行。当源站连接空闲时，响应路径不必持有引擎快照。

> **练习 6.1.** 把载荷换成 `O'Reilly`，再换成 `order=1`。为什么单独其中任何一个都不应被认定为 SQL 注入？举出一个只做子串匹配的检测器可能产生的误报例子。

练习 6.2. 写出为 LDAP 注入（例如 `)(|(` 这样的模式）增加一个检测器所需的字节类表项，并解释它可以复用哪些现有的类比特。

解释这个不变量. 向审阅者解释为什么 `Accept: */*` 曾是一个误报，新的 SQL 检测器要求什么样的结构性质，以及自动机和分词器如何分工。

第 VII 部分

浏览器端的工作量证明引擎

我们考察由服务器自己的验证器 *crate* 编译而成的 *WebAssembly* 模块、*Web Worker* 协议、*JavaScript* 后备证明器，以及验证中间页。

一份源码，两个目标

> **学习目标** 学完本章，我们应当能够解释浏览器模块如何由服务器自己的 `sibuna-pow` crate 构建，说出它导出的函数，并描述顺序工作量证明器在一个标签页中所需的内存。

构建方式

浏览器求解器编译的正是服务器用来验证的那些源码。`cmd/sibuna-pow-wasm` 是一个类型为 `cdylib` 的 `no_std` 库 crate，为 `wasm32-unknown-unknown` 构建，它仅有的依赖是工作区 crate `sibuna-pow` 和 `sibuna-crypto`：

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

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

# Cargo.toml（工作区根目录）：模块按体积优化构建
[profile.release.package.sibuna-pow-wasm]
opt-level = "z"
debug = false
strip = true
```

守护进程内嵌了这个模块。它的构建脚本（`cmd/sibuna/build.rs`）运行一次嵌套的 `cargo rustc -p sibuna-pow-wasm --target wasm32-unknown-unknown --profile wasm`，输出到工作区的目标目录（`target/wasm32-unknown-unknown/wasm/`）中。嵌套构建只锁定 `wasm` profile 的目录，因此从不等待外层构建。最终链接时加上 `--compress-relocations`。脚本检查结果是否以 WebAssembly 魔数字节开头，然后把它复制到构建输出目录。对于没有 WebAssembly 目标的构建，`SIBUNA_POW_WASM=<path>` 可以改为嵌入一个预先构建的模块。然后，守护进程把这些字节包含进它的可执行文件：

```
/// 浏览器求解器，由 `build.rs` 从 `cmd/sibuna-pow-wasm` 构建。
pub const WASM_SOLVER: &[u8] =
    include_bytes!(concat!(env!("OUT_DIR"), "/sibuna-pow.wasm"));
/// 求解器所用的 Web Worker。
pub const WORKER_JS: &str = include_str!("../../../web/worker.js");
```

不存在会发生漂移的第二份实现。守护进程在 `/__sibuna/wasm/sibuna-pow.wasm` 提供这个模块，并带有一小时的缓存头（`Cache-Control: public, max-age=3600`）。用上面的 profile 构建时，包含两层的模块大小为 14,744 字节（`benchmarks/results/primitives-latest.json` 中的 `wasm_bytes`）。

导出

模块导出六个函数，worker 脚本按名称调用它们：

导出函数	用途
`sibuna_get_buffer_ptr()` / `sibuna_get_buffer_len()`	256 字节的输入缓冲区，worker 把挑战字符串写入其中
`sibuna_solve_step(ptr, len, bits, start, max_steps)`	Hashcash：从 `start` 开始搜索 `max_steps` 个 nonce；返回找到的 nonce 或全 1 的哨兵值
`sibuna_solve_sha256(ptr, len, bits)`	从 nonce 0 开始一次性批量求解 Hashcash（步数为 `u32::MAX` 的 `sibuna_solve_step`）
`sibuna_posw_solve(len, depth, challenges)`	顺序工作量：对缓冲区运行证明器，并返回证明长度，参数无效时返回 0
`sibuna_posw_proof_ptr()`	证明器工作空间中证明字节的起始位置

证明器的工作空间（保留的顶部各层、兄弟标签栈和证明）是一块约 92 KiB 的静态区域；没有链接任何分配器，模块也没有任何导入。`Workspace::new` 是一个 `const fn`，因此这块静态区域只占用清零的内存，而不是模块中已初始化的数据。

> **实现：`sibuna_solve_step`** Hashcash 的导出函数：从线性内存借用挑战，并运行共享 nonce 搜索的一个批次。位于 `cmd/sibuna-pow-wasm/src/lib.rs`（crate `sibuna-pow-wasm`）。

```rust
#[unsafe(no_mangle)]
pub unsafe extern "C" fn sibuna_solve_step(
    challenge_ptr: *const u8,
    challenge_len: usize,
    difficulty_bits: u32,
    start_nonce: u64,
    max_steps: u32,
) -> u64 {
    let challenge: &[u8] = if challenge_len == 0 {
        &[]
    } else {
        // SAFETY: 调用者保证 `challenge_ptr` 处有 `challenge_len` 个可读字节，
        // 并且它们在调用期间保持不变。
        unsafe { core::slice::from_raw_parts(challenge_ptr, challenge_len) }
    };
    Search::new(challenge)
        .find(difficulty_bits, start_nonce, u64::from(max_steps))
        .unwrap_or(EXHAUSTED)
}
```

这些导出函数是该 crate 的外部函数边界，也是它的 unsafe 代码唯一所在的地方。输入缓冲区和证明器工作空间是位于 ExportCell 中的静态变量；这个类型是对 UnsafeCell 的封装，被声明为 Sync，并写明了理由：没有共享内存的 WebAssembly 实例只运行一个线程，而每个导出函数都在返回之前创建、使用并丢弃它的引用，因此任何时刻至多只有一个指向单元的引用是活跃的。在宿主机上，测试直接调用导出函数，此时由一个仅用于测试的互斥锁让它们串行执行。panic 会触发陷阱（unreachable 指令），而不是格式化一条消息，这使模块不含格式化代码和导入；从构造上说 panic 是不可达的，因为 sibuna-pow 的函数都是全函数，从不 panic。

前缀预哈希

Hashcash 的内层循环只吸收一次挑战和冒号，然后为每个 nonce 克隆 SHA-256 状态，因此每个候选只需一次压缩，而不是两次。搜索由 sibuna_pow::hashcash::Search 实现，浏览器模块和原生参考求解器共用它：

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

pub fn digest(&self, nonce: u64) -> Digest {
    let mut buf = [0u8; NONCE_DIGITS_MAX];
    let mut hasher = self.prefix.clone(); // 按值复制已吸收的前缀
    hasher.update(format_decimal(nonce, &mut buf));
    hasher.finalize()
}

pub fn find(&self, bits: u32, start: u64, steps: u64) -> Option<u64> {
    let mut nonce = start;
    for _ in 0..steps {
        if meets_bits(&self.digest(nonce), bits) {
            return Some(nonce);
        }
        nonce = nonce.wrapping_add(1);
    }
    None
}
```

哈希器状态是一个约 110 字节的普通值，因此克隆的成本很低。顺序工作量证明器使用同样的技巧：陈述 χ 在 base_hasher 中只吸收一次，然后为每个节点克隆；验证器现在也是如此：它从一个已经吸收过陈述的状态派生每个被挑战的叶子，这使验证在得到相同摘要的同时少做 t 次压缩。

在服务器上，压缩函数使用 x86-64 SHA 扩展或 ARMv8 SHA-2 指令，在运行时检测。浏览器中没有这样的指令，求解器几乎把所有时间都花在可移植的压缩函数上（crates/crypto/src/sha256/portable.rs）。这个函数是为 WebAssembly 编写的：64 轮全部展开，消息调度保存在一个 16 字的窗口中，从第 16 轮起，每一轮都覆盖它不再需要的那个字，八个工作变量在轮与轮之间重命名，而不是移位：

```
fn compress_block(state: &mut State, block: &Block) {
```

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

/// 返回消息调度字 W(i)；从第 16 轮起就地计算它。
#[inline(always)]
fn schedule(w: &mut [u32; 16], i: usize) -> u32 {
    if i >= 16 {
        let w15 = w[(i + 1) % 16];
        let w2 = w[(i + 14) % 16];
        let s0 = w15.rotate_right(7) ^ w15.rotate_right(18) ^ (w15 >> 3);
        let s1 = w2.rotate_right(17) ^ w2.rotate_right(19) ^ (w2 >> 10);
        w[i % 16] = w[i % 16]
            .wrapping_add(s0)
            .wrapping_add(w[(i + 9) % 16])
            .wrapping_add(s1);
    }
    w[i % 16]
}
```

每个 `eight_rounds!` 是变量名的一次完整轮换：宏在每一轮把它们轮转一个位置后传入，因此一轮只写入其中两个变量，从不移动另外六个。由于轮次索引是字面量，每个调度位置都是常量，整个分组变成直线代码，其值始终保存在 WebAssembly 局部变量中。这种布局在 V8 中比使用 64 字调度和未展开循环的写法快约 20%，在原生环境中快约 14%（在 Rust 实现上测量，2026 年 10 月）。

Worker 协议

> **学习目标** 阅读消息协议，理解后备层级，并了解 WebAssembly 与 JavaScript 之间的有符号整数边界所造成的哨兵值缺陷。

页面向 `web/worker.js` 发送一条消息，并接收进度、后备和结果消息：

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

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

```
{ type: 'solved', challenge: spec.id, solution: { proof: '<base64url>' } } // posw
{ type: 'error', message: '...' }
```

worker 先尝试 WebAssembly；当 `WebAssembly` 不存在、被策略阻止或实例化失败时，回退到纯 JavaScript。JavaScript 证明器逐个标签地镜像 `sibuna-pow` 的 Rust 证明器，产生 逐字节相同 的输出。测试为两层都固定了 nonce 和证明，SID 0010 要求求解器在无头浏览器中，无论使用 WebAssembly 还是 JavaScript 后备，都能通过。JavaScript 证明器慢得多，但它保证每个浏览器都能通过。

有符号的哨兵值 WebAssembly 把 64 位整数作为 有符号 的 `BigInt` 返回给 JavaScript。Hashcash 导出函数用全 1（`EXHAUSTED`，即 `u64::MAX`）表示“批次已耗尽”，而 JavaScript 看到的是 `-1n`，而不是 `18446744073709551615n`。最初的 worker 与无符号值进行比较，因此循环在第一个批次之后就带着一个错误的 nonce 退出了。修复只需一次调用 `BigInt.asUintN(64, result)`，worker 保留了它：

```
// WebAssembly 的 i64 结果以有符号 BigInt 的形式返回，因此 u64 的
//  “已耗尽”哨兵值（全 1）读回时是 -1n。
const found = BigInt.asUintN(64,
    ex.sibuna_solve_step(ptr, encoded.length, bits, nonce, maxSteps));
if (found !== 18446744073709551615n) return found.toString();
```

这个教训具有普遍性：每个跨越边界的 u64 都必须规范化。Rust 无法在 JavaScript 一侧强制这一点；导出函数的类型写的是 `u64`，而转换发生在引擎中。

校准

学习目标 把难度设置与浏览器引擎中和原生硬件上的实际耗时联系起来。

不要根据一台笔记本电脑的平均求解时间来选择难度。Hashcash 有几何分布的尾部；浏览器中的计时包括调度、温度状态、启动和编译。部署时的校准应当记录浏览器版本、设备、算法、参数，以及多次重复求解的分布。把模块的获取和编译与稳态哈希分开。

校准方案详解 固定 b，预热模块，并在每类目标设备上求解至少 100 道独立签发的谜题。记录耗时的中位数和高百分位数、失败次数以及证明大小。在 $b+1$ 下重复。期望尝试次数会翻倍；单个样本则未必。按照慢速设备的预算选择设置，然后测试完整的获取、求解和验证流程。

仓库中的原生原语测量结果，不能作为对手机或浏览器求解时间作出承诺的证据。WASM 模块可以执行相同的算术，但启动、执行和内存成本都不同：在原生环境中，服务器用 SHA-NI 计算哈希，而浏览器运行的是上面那个可移植的压缩函数。worker 的消息构成了一个边界，可以在这里测量这些成本，而无需在服务器的请求处理器中加入计时器。

验证中间页

这个页面（`web/challenge.html`，内嵌在守护进程中）是一个单独的 HTML 文件，包含一张卡片、一个不确定进度条和一行状态文字。它为自己的位置请求挑战（`?path=`），启动 worker，用 `fetch` 提交解答，成功后重新加载；`fetch` 调用会以带抖动的指数退避重试，验证返回 `4xx` 时会显示其诊断标题。页面的 `Content-Security-Policy` 通过 SHA-256 哈希只允许一个脚本，即求解器代码块，该哈希由守护进程从内嵌页面计算一次得出。一个位于屏幕外的蜜罐链接（`/__sibuna/honeypot`）对人和执行脚本的浏览器都不可见；盲目跟随每个 `href` 的爬虫会让自己被封禁。

练习 7.1. 假设 WebAssembly 证明器在一台笔记本电脑上以默认难度求解一个挑战需要 20 ms，而 JavaScript 顺序工作量证明器慢 50 倍。估算这台笔记本电脑上以及一部慢四倍的手机上的后备求解时间，并为报告使用了后备方案的客户端提出一条策略（提示：worker 会发送一条页面可以转发的 `fallback` 消息）。

解释这个不变量. 解释为什么用服务器的验证器源码编译浏览器模块，比让两份实现相互对照测试提供了更强的保证。

第 VIII 部分

实证评估

我们给出由已提交的测量工具记录的 *Sibuna* 测量结果：原语延迟；在外部负载生成器下运行的完整产品，并统计 *CPU* 和内存；开环生成器在相同施加负载下测得的延迟；以及与 *Anubis*（一种广泛部署的工作量证明网关）的比较。一次运行八个实时仪表板的控制台测试补全了这份记录。本部分的每个数字都来自一个已提交的结果文件（每张表都注明了该文件及其生成时间），或者来自一份设计记录。功能性通过与性能测量回答的是不同的问题，在一台主机上测得的结果也不能作为另一台主机上的排名。

方法

> **学习目标** 学完本章，我们应当能够复现本部分中的每个数字，说明每套测量工具包含什么、不包含什么，解释离散范围列的含义，并区分实测数字与记录的或公开发布的数字。

这些测量工具是同一个 Rust 工具 `tools/bench`（可执行文件 `sibuna-bench`）的若干命令，通过 `cargo xtask bench` 运行。另一个工具记录控制台的运行。

表 11 测量工具及其结果文件，均位于 `benchmarks/results/` 下

测量工具	测量内容	结果文件
`cargo xtask bench primitives`	十六个原语工作负载，外加针对哈希速率限制器路径的两行，在一个 CPU 上运行：七个批次，给出每次操作的中位数及最小-最大离散范围	`primitives-latest.json`, `primitives-latest.md`
`cargo xtask bench product`	在 `wrk` 下运行的完整守护进程：Gate 和 Shield 在 forward-auth 和反向代理模式下的吞吐量、延迟百分位、每请求 CPU 时间和峰值常驻内存	`product-latest.json`, `product-latest.md`
`cargo xtask bench latency`	开环生成器在固定施加速率下测得的已准入请求的延迟百分位	`latency-latest.json`, `latency-latest.md`
`cargo xtask bench product --anubis`	把 Anubis 作为另一个产品加入的完整产品工作负载，以 `caddy respond` 为源站	`comparison-latest.json`, `comparison-latest.md`
`cargo xtask bench admission --anubis`	在 forward-auth 模式下，由两个 keep-alive 客户端测得的 Sibuna 和 Anubis 每秒准入操作数	`admission-latest.json`, `admission-latest.md`

测量工具	测量内容	结果文件
sibuna-browser dashboards（tools/browser）	一次功能性的控制台运行：生成器发送流量时，在无头 Chromium 中运行八个实时仪表板	console-8-dashboards-2026-10-15.md

.md 文件是摘要。.json 文件保存每次重复、机器、修订版本，以及每个被测可执行文件的路径、大小和 SHA-256。Git 保存了历史，因此该目录只保留最新的文件。有些文件是由较早版本的测量工具记录的，其中还包含同一次运行中测得的另一个守护进程构建的数据行；本部分报告 Sibuna 的数据行。

原语

每个工作负载在一次不计时的预热（最多 1,000 次操作）之后，运行七个计时批次。可变状态（已用集合、速率限制器）在每个批次之前、在计时之外重建。计时器包围整个批次，从不包围单次操作：每次操作读取一次时钟的成本，会与它要测量的不到 100 ns 的工作一样大。因此不报告每次操作的百分位，离散范围是跨批次的。我们报告每次操作的批次中位数，旁边列出最快和最慢的批次。

测试套件在一个绑定到单个 CPU 的子进程（sibuna-bench primitives-run）中运行。在共享主机上，繁忙的 SMT 兄弟线程或较低的时钟频率会拖慢整轮测试，因此工具把套件运行三次（--rounds），并为每一行保留中位数最低的那一轮。每一轮内部的统计方法同上。

完整产品

对于每种模式（forward auth、反向代理）和每种形态（Gate、Shield），测量工具以 --algorithm hashcash --difficulty 8 --rate-limit 100000000 --idle-timeout 5 和四个 worker（其 CPU 集合中每个 CPU 一个）在内存模式下启动守护进程。它像浏览器那样，用 sibuna-pow 求解守护进程的 Hashcash 挑战来获得会话。然后，它用 wrk 通过 keep-alive 连接驱动四种工作负载：2 个线程，64 个连接，5 秒，重复 3 次。

- admitted：带有效会话 Cookie 的 GET /private。在反向代理模式下，请求到达源站，其应答被中继回来；在 forward-auth 模式下，由守护进程自己回答 200。
- challenged：同样的请求，但没有 Cookie，并使用浏览器 User-Agent。守护进程提供挑战：反向代理模式下是带 HTML 的 200，forward-auth 模式下是 401。
- allowed_static：来自 curl/8.0 的 GET /robots.txt，两种形态都会在不挑战的情况下准入它。
- attack：一个有效会话加上 q=' OR 1=1--。只有 Shield 拒绝它（403）。Gate 准入它，并在反向代理模式下把它中继到源站。

源站是一个桩程序（sibuna-bench origin）：只使用标准库，每个连接一个线程，keep-alive，返回一个 41 字节的页面。每请求 CPU 时间是守护进程在整个运行期间来自 /proc/<pid>/stat 的用户态加内核态时间（以 10 ms 为一个时钟滴答），除以 wrk 统计的请求数；不包括源站的 CPU。峰值内存是 /proc/<pid>/status 中的 VmHWM，每次运行之前通过 /proc/<pid>/clear_refs 重置，使每次运行报告各自的峰值。每个数字都是各次重复的中位数，最小值和最大值保存在 JSON 文件中与之并列。

wrk 是一个闭环生成器：它始终保持 64 个请求在途。因此，越快的守护进程收到的请求越多，而 wrk 的百分位遗漏了它在等待慢应答期间没有发送的请求（协同遗漏）。它的 p99 反映的是守护进程在自身饱和时的表现，而不是在给定负载下的延迟。延迟测量工具回答的是后一个问题。

相同负载下的延迟

开环生成器（`sibuna-bench generate`）为每个连接运行一个阻塞线程。64 个连接中的每一个都按固定的时间表发送带会话的 `GET /private`：一个连接的第 k 个请求在 $t_0 + o + kT$ 时刻到期，其中 o 是该连接的偏移量，T 是它的间隔，与之前的应答无关。当应答在下一个请求到期之后才到达时，下一个请求立即发出，其延迟从它的到期时刻开始计算，与 `wrk2` 的做法相同。当连接在到期时刻处于空闲状态时，延迟从实际发送时刻开始计算，因此生成器自身睡眠过头的时间不会算到守护进程头上。

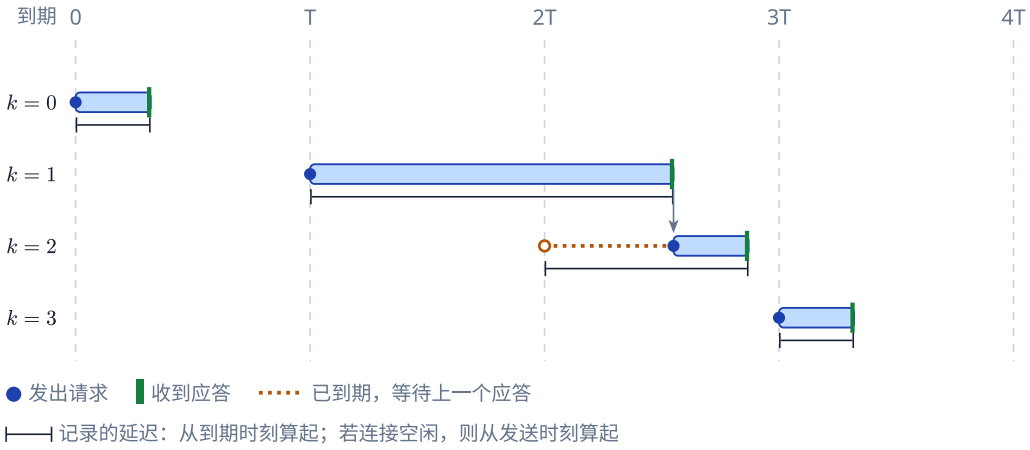


图 10 开环生成器的一个连接。请求 1 在请求 2 到期之后才得到应答，因此请求 2 立即发出，其延迟从 2T 开始计算；请求 3 到期时连接处于空闲状态，从它的发送时刻开始计算。

测量工具运行 Shield 形态和已准入工作负载，施加五种速率：每秒 10,000、25,000、50,000、100,000 和 150,000 个请求，每种速率预热 2 秒，测量 10 秒，运行 3 次。测量工具可以在一次运行中测量两个守护进程构建：此时它把两者启动在同一个 CPU 集合上，并在每种速率和每次重复之间交替运行，空闲的那个不占用 CPU，从而让机器的缓慢漂移对两者产生相同的影响。记录的这次运行就是这样做的。百分位按最近秩法在一次运行的所有被测请求上计算；每个数字是三次运行的中位数，旁边列出 p99 的最小-最大范围。

CPU 集合

CPU 集合比任何其他设置都更重要。守护进程运行在 CPU 0 到 3 上（`--cpus`；`--workers` 等于该集合的大小）。`wrk`、延迟生成器和源站运行在 CPU 8 到 15 上（`--load-cpus`）。原语测试套件运行在 CPU 4 上（`--cpu`）。每个进程在启动之前用 `taskset -c` 绑定，因此它创建的每个线程都继承该集合。准入测量工具为产品使用 0 和 1 两个 CPU，为其客户端使用 CPU 8 到 15。



图 11 AMD Ryzen 9 5950X 上的 CPU 集合。CPU 0 到 7 是第一个 CCD 的八个核心，8 到 15 是第二个 CCD 的核心，16 到 31 是它们的 SMT 兄弟线程。守护进程和负载生成器从不共享核心。

在其他机器上，我们选择不共享核心的集合（`lscpu -e` 显示拓扑）。

机器与条件

这些文件记录于 `paxos-zig`：一台 AMD Ryzen 9 5950X 16 核处理器（32 个逻辑 CPU），Linux 7.0.0-28-generic x86_64，`powersave` 调频策略并启用 boost，rustc 1.99.0。原语、产品和延迟文件记录于 2026 年 10 月 15 日，与 Anubis 的比较记录于 2026 年 10 月 16 日。守护进程是 `cargo build --release -p sibuna`，即编译了 SQLite 存储和控制台的默认构建（SID 0010）；产品和延迟文件注明的可执行文件大小为 49,113,856 字节。每个文件都记录了其修订版本（原语为 `efce14d`，产品和延迟运行为 `249d913`），并带有 `dirty: true`；后者的工作树中已经包含 SID 0002 所记录的连接线程复用，SID 0002 引用 `latency-latest.md` 说明其效果。比较和准入文件注明修订版本 `54e4ec5` 并带有 `dirty: true`，因为工作树中包含 `tools/bench` 里新增的测量工具代码；它们的守护进程是该修订版本的 release 构建（37,450,232 字节）。比较运行之前的一分钟平均负载为 3.53，准入运行结束时为 0.56。

文件记录的一分钟平均负载为：原语运行时 2.46，延迟运行之前 1.30、之后 52.31，产品运行之前 52.31、之后 33.39。较高的数字来自延迟生成器的 64 个线程，产品运行紧接其后。

测量范围 只给出本地测量结果。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，没有真实浏览器。守护进程在内存模式下运行，不带存储。计时器包围批次；状态重置和预热不计时。测量工具不统计分配次数（文件中写着 “not instrumented; API/source audit only”）；无分配的请求路径改由第 V 部分的稳态测试检查。没有任何测量工具链接进守护进程或在请求路径上添加钩子：产品和延迟测量工具运行未经修改的 release 可执行文件，并以其 SHA-256 标识。在这台主机上的排名不能作为另一台主机上的排名。

如何复现

我们需要 Linux、taskset（util-linux），以及 PATH 中的 wrk。

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

cargo xtask bench 先用 cargo build --release -p sibuna 构建守护进程，然后以 release 模式运行 sibuna-bench。工具在它自己的可执行文件旁边寻找守护进程；--rust <path> 可以覆盖这一位置。cargo xtask bench help 列出所有选项，其中包括 --cpus、--load-cpus、--cpu、--rounds、--repetitions、--seconds、--rates 和 --warmup。下面关于 Anubis 的一节给出了记录那次比较所用的命令。

原语延迟

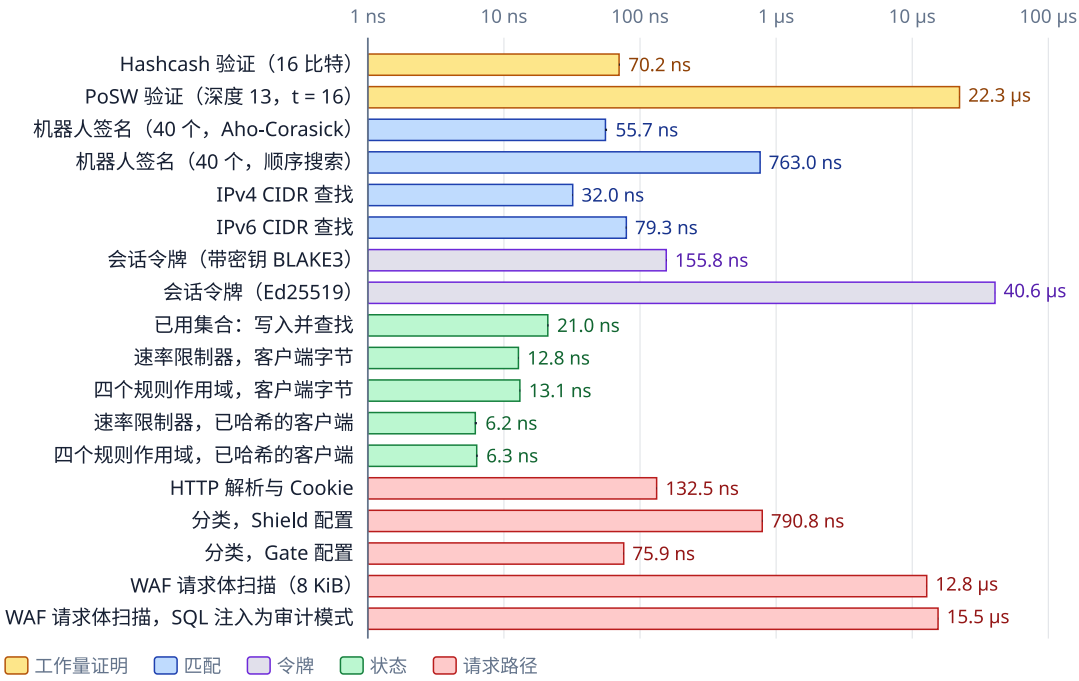
下表给出原语运行的每一行。JSON 文件按子系统和工作负载命名每一行（例如 pow_verify 和 hashcash_16_bits）。

表 12 在一个 Ryzen 9 5950X 核心上测得的原语耗时，单位为每次操作纳秒（benchmarks/results/primitives-latest.md，生成于 2026-10-15T22:34:59Z）

工作负载	ns/op	最小-最大
Hashcash 验证（16 比特）	70.2	69.9-71.0
PoSW 验证（深度 13，t = 16）	22272.4	22257.0-22316.4
机器人签名（40 个，Aho-Corasick）	55.7	55.3-57.2
机器人签名（40 个，顺序搜索）	763.0	762.2-763.5
IPv4 CIDR 查找	32.0	31.8-32.2
IPv6 CIDR 查找	79.3	78.5-79.6
会话令牌（带密钥 BLAKE3）	155.8	155.3-156.4
会话令牌（Ed25519）	40587.5	40544.6-40631.1
已用集合（Robin Hood），写入并查找	21.0	20.8-21.3
速率限制器（GCRA），客户端字节	12.8	12.7-12.8
速率限制器，四个规则作用域，客户端字节	13.1	13.0-13.2
速率限制器（GCRA），已哈希的客户端	6.2	6.1-6.3
速率限制器，四个规则作用域，已哈希的客户端	6.3	6.2-6.4
HTTP 解析与 Cookie	132.5	132.3-133.2
分类，Shield 配置	790.8	787.2-793.1
分类，Gate 配置	75.9	75.6-76.1

工作负载	ns/op	最小-最大
WAF 请求体扫描（8 KiB）	12784.3	12771.5-12809.2
WAF 请求体扫描（8 KiB），SQL 注入为审计模式	15485.4	15460.9-15530.1

离散范围很窄：任何一行中的任何批次与该行中位数的偏差都不超过 2.8%。图中以对数刻度展示同样的各行，使 6 ns 和 40 us 的行能够并排阅读。



每根条形是 AMD Ryzen 9 5950X 16-Core Processor 单个核心上 7 个批次的中位数，测于 2026-10-15；细线从最快的批次延伸到最慢的批次。对数刻度。

图 12 以对数刻度绘制的原语耗时，数据取自 `benchmarks/results/primitives-latest.json`。

工作量证明验证

Hashcash 验证对一个挑战、一个分隔符和一个十进制 nonce 做哈希：70.2 ns。顺序工作量验证器检查图的揭示：22,272.4 ns。由于输入长度不同，它的哈希调用次数和 SHA-256 压缩次数与 Hashcash 不同。两行都受 SHA-256 的限制，`sibuna-crypto` 在 CPU 支持时使用 SHA-NI 指令，在运行时检测（SID 0010）。

匹配

稠密 Aho-Corasick 自动机和顺序子串搜索使用相同的模式和“任一匹配”语义。在同样的 40 个签名上，自动机比顺序搜索快 13.7 倍（55.7 ns 对 763.0 ns）。它的表使用字节类，而不是每个状态 256 列（SID 0006，F 部分），这使机器人签名表保持在 42,244 字节（见下面的占用表）。CIDR 分类器耗时 32.0 ns（IPv4）和 79.3 ns（IPv6）。

令牌

BLAKE3 MAC 令牌的验证耗时 155.8 ns。`sibuna-crypto` 以可移植的方式实现 BLAKE3，没有 SIMD 路径；SIMD 路径在大输入上占优，但 Sibuna 用 BLAKE3 处理的输入都不到 100 字节，在这种输入上可移植代码已经很快。Ed25519 紧凑令牌的验证耗时 40,587.5 ns：验证器使用宽度为 5 和 8 的 NAF 数字，并保留解码后的公钥。一次 Ed25519 验证的成本约等于 260 次 MAC 验证；MAC 是 Sibuna 的默认令牌方案。

状态

已用集合这一行插入一个新标签并再次查找它：21.0 ns。这一行中的表有 16 个分片，每个分片 4,096 个槽（65,536 个标签；守护进程的默认值是 1,048,576），并在每个批次之前、在计时之外原地清空。因此每个批次探测的都是常驻内存。测试套件的早期版本为每个批次构建一张守护进程默认大小、全新清零的表；它测到的是内核在首次写入时映射表页面的成本，而运行中的守护进程只付出一次这种成本，所以这不是我们报告的那一行。

速率限制器的几行需要仔细解读。客户端字节的两行（`gcra_check` 和 `gcra_four_rule_scopes`）调用字节 API：每次检查以字节形式接收客户端地址，先用一个取自操作系统随机源的密钥计算它的带密钥 SipHash-1-3。密钥使任何人都无法计算出与受害者共享探测窗口的地址（见 SID 0010 和 `sibuna-store` 的文档）。这两行耗时 12.8 ns 和 13.1 ns。

守护进程在请求路径上不使用字节 API。它每个请求只对客户端做一次哈希（`ClientHasher`），并把哈希值传给它查询的每张表。已哈希客户端的两行（`gcra_check_hashed` 和 `gcra_four_rule_scopes_hashed`）测量的就是这条路径，哈希值在循环之前计算：每次检查 6.2 ns 和 6.3 ns。字节行与已哈希行中位数之间 6.6 ns 的差距，大约就是在这些输入上计算带密钥哈希的代价，守护进程每个请求只付一次，而不是每张表付一次。

HTTP 解析器

解析器这一行解析一个 400 字节的浏览器请求并查找一个 Cookie：132.5 ns。该 crate 自己的计时测试结果与之一致（`crates/http`：在同一台机器上约 135 到 180 ns）。解析器是零拷贝的；每个字段都是连接缓冲区的一个切片。

策略引擎

对一个浏览器请求进行完整分类耗时 790.8 ns，Gate 配置耗时 75.9 ns。这几行没有填入请求中已解析的客户端地址，因此引擎在评估过程中解析客户端地址文本。守护进程在评估之前、每个请求只解析一次地址，因此这几行包含了一小部分守护进程在引擎之外付出的成本。引擎的堆表总计 209,052 字节（SID 0002）。

WAF 检查

对 8 KiB 请求体的语义扫描耗时 12,784.3 ns，即每字节 $12784.3/8192 \approx 1.56$ ns。第二行扫描同一个请求体，但 SQL 注入类别为审计模式，其他类别为强制模式：15,485.4 ns。当某个类别为审计而非强制时，每个字段按类别分别检查（SID 0004，“检查模式”）。引擎用签名自动机扫描字段一次，得到出现了签名的类别集合，然后只在其中搜索需要的类别，因此审计一个类别比强制所有类别大约多花五分之一，而不是每个类别各扫描一次。

完整产品比较

> **学习目标** 解读守护进程作为完整进程的吞吐量、延迟、CPU 时间和内存，比较它的形态和模式，并指出在哪些行中是守护进程设定了上限，在哪些行中是负载生成器设定了上限。

表中给出每种形态和工作负载下三次五秒 `wrk` 运行的中位数。每个工作负载旁边的状态码是测量工具期望并实际收到的应答。

Forward-auth 模式

入口代理询问守护进程是否准入一个请求；没有源站，也没有请求体中继，因此这是对准入成本最纯粹的测量。

表 13 Forward-auth 模式，Gate 和 Shield；延迟单位为微秒（`benchmarks/results/product-latest.md`，生成于 2026-10-15T23:15:24Z）

形态	工作负载（状态码）	req/s	p50	p90	p99	CPU us/请求	峰值 RSS KiB
Gate	`admitted` (200)	211,345	151	175	305	11.84	15912
Gate	`challenged` (401)	198,315	160	180	326	13.15	16176
Gate	`allowed_static` (200)	211,053	152	175	305	11.73	16176
Gate	`attack` (200)	211,618	151	176	305	11.84	16176
Shield	`admitted` (200)	208,101	152	178	308	12.87	15928
Shield	`challenged` (401)	197,532	161	183	327	13.77	16192
Shield	`allowed_static` (200)	211,636	151	175	305	11.78	16192
Shield	`attack` (403)	210,549	152	175	307	12.32	16192

在 forward-auth 模式下，守护进程不是瓶颈。它的忙碌核心数（CPU 时间除以实际时间，即 JSON 文件中的 `cores_busy_median`）保持在四个守护进程 CPU 中的 2.47 到 2.72 之间，每一行都在每秒 197,532 到 211,636 个请求之间。这表明上限来自负载生成器。在这个上限附近，行与行之间几个百分点的差异会随运行而变化：仅 Gate 的 `admitted` 这一行，三次运行就从每秒 205,151 到 211,513 个请求不等。在这种模式下更有用的数字是每请求 CPU 时间：11.73 到 13.77 us，Shield 比 Gate 略多，`challenged` 各行最多，因为它们要渲染挑战应答。

反向代理模式

守护进程位于桩源站之前，中继已准入的请求和响应。因此，经代理的各行包含源站自身的成本和一次额外的回环跳转。

表 14 反向代理模式，Gate 和 Shield；延迟单位为微秒（`benchmarks/results/product-latest.md`，生成于 2026-10-15T23:15:24Z）

形态	工作负载（状态码）	req/s	p50	p90	p99	CPU us/请求	峰值 RSS KiB
Gate	admitted (200)	142,498	427	548	768	27.68	15896
Gate	challenged (200)	199,465	160	181	324	12.98	16160
Gate	allowed_static (200)	147,303	411	541	759	26.88	16324
Gate	attack (200)	143,900	422	555	745	27.67	16348
Shield	admitted (200)	138,965	438	571	857	28.57	16064
Shield	challenged (200)	195,188	162	185	328	13.73	16328
Shield	allowed_static (200)	145,673	415	551	746	27.27	16328
Shield	attack (403)	209,507	151	177	306	12.17	16328

这里经代理的各行（admitted、allowed_static，以及由 Gate 中继的 Gate attack 行）受 CPU 限制：忙碌核心数为四个中的 3.96 到 3.98。它们每秒处理 138,965 到 147,303 个请求，每请求 CPU 时间为 26.88 到 28.57 us，约为守护进程自己作答时的两倍。challenged 各行和 Shield 的 attack 行由守护进程自己作答，看起来与 forward-auth 的各行相似。在饱和状态下，wrk 的 p99 是守护进程以自身速率运行时调度器的尾部，因此我们改为解读下一章的尾延迟。

在三个反向代理行中，wrk 统计到了不应出现的 2xx 和 3xx 以外的响应：三次运行合计，2,162,579 个请求中有 3 个（Gate attack），2,082,640 个中有 1 个（Shield admitted），2,204,358 个中有 1 个（Shield allowed_static）。文件没有记录这些响应的状态码。带守护进程指标复现这次运行后发现，它们是计入 sibuna_overloaded_total 的 503 应答：wrk 从一个地址打开它的 64 个连接，恰好等于默认的 --max-connections-per-ip；当一个连接在第 4,096 个请求之后被关闭时，wrk 在旧连接归还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（SID 0002，“缺陷修复”）；这些表格的数据早于该修复。

资源占用

表 15 守护进程的大小和常驻内存（benchmarks/results/product-latest.json，生成于 2026-10-15T23:15:24Z；primitives-latest.json，生成于 2026-10-15T22:34:59Z）

指标	数值
守护进程可执行文件	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
空闲常驻内存，原语运行	6,052 KiB
策略引擎表	209,052 字节（堆）
机器人签名表	42,244 字节（堆）
WAF 表	164,908 字节（堆）

指标	数值
浏览器求解器模块（WebAssembly）	14,744 字节

范围取自四次产品运行（两种模式、两种形态），峰值则取自它们的十六个工作负载。表的大小是引擎报告的堆大小。记录没有细分可执行文件的大小构成。

相同施加负载下的延迟

> **学习目标** 解读固定施加负载下的延迟分布，结合离散范围解读尾部百分位，并跟随一个尾部缺陷从测量到修复的全过程。

两张表都来自 `cargo xtask bench latency` 的同一次运行。错误列统计生成器的连接、I/O 和分帧错误。

表 16 Forward-auth 模式，Shield，固定施加速率下的已准入请求；延迟单位为微秒（`benchmarks/results/latency-latest.md`，生成于 2026-10-15T23:07:14Z）

施加 req/s	实际 req/s	p50	p90	p99（最小-最大）	p99.9	最大值	错误	CPU us/请求
10,000	10,000	30	32	37 (36-39)	58	355	0	16.92
25,000	25,000	28	31	34 (32-34)	45	1035	0	15.83
50,000	50,000	25	27	30 (30-30)	40	3029	0	13.83
100,000	99,998	24	26	41 (40-42)	62	2358	0	13.48
150,000	149,996	25	36	51 (51-52)	74	1753	0	14.16

表 17 反向代理模式，Shield，固定施加速率下的已准入请求；延迟单位为微秒（`benchmarks/results/latency-latest.md`，生成于 2026-10-15T23:07:14Z）

施加 req/s	实际 req/s	p50	p90	p99（最小-最大）	p99.9	最大值	错误	CPU us/请求
10,000	10,000	56	61	65 (65-66)	76	555	0	32.25
25,000	25,000	51	54	58 (58-60)	72	1105	0	28.63
50,000	50,000	47	50	60 (58-66)	74	3066	0	27.45
100,000	99,996	55	75	100 (100-104)	150	2239	0	28.42
150,000	134,344	583934	154416	01906273 (1631234-2218760)	1967134	1989964	4	29.22

在 forward-auth 模式下，每种速率的 p50 都保持在 24 到 30 us 之间。直到每秒 150,000 个请求，p99 都不超过 51 us，p99.9 不超过 74 us。守护进程每请求消耗 13.48 到 16.92 us 的 CPU 时间。

在反向代理模式下，直到每秒 100,000 个请求，p50 为 47 到 56 us；在每秒 100,000 个请求时，p99 为 100 us，p99.9 为 150 us。在每秒 150,000 个请求时，守护进程达到饱和：它每秒完成 134,344 个请求，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结果。这一速率接近上一章中闭环测得的代理吞吐量（Shield `admitted` 为每秒 138,965 个请求）。该行的四个错误发生在过载期间。

在饱和之前，p99 的离散范围很窄；最宽的是每秒 50,000 个请求时反向代理的 p99，为 58 到 66 us。最大值列是一次运行中多达 150 万个请求里的一个。在共享主机上，我们把它解读为主机最糟糕的时刻，而不是守护进程的性质。

测量带来的改变

外部负载改变了 Sibuna 的设计。守护进程在独立的线程上服务每个连接（SID 0002, SID 0010），它最初的几次延迟测量发现了一个 `wrk` 没有显示出来的尾部。在 64 个连接、每个连接由守护进程在 4,096 个请求后关闭的情况下，每秒 100,000 个已准入请求时的第 99.9 百分位为 0.3 到 2 ms。

SID 0002 和 `cmd/sibuna/src/tasks.rs` 的文档中记录的诊断结论是线程更替。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守护进程的其他线程。一个实验就解决了这个问题：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。修复方法是复用连接线程：完成连接的线程停驻并接手下一个连接，因此持续不断的新连接不会创建线程。停驻 10 秒的线程会结束，并由下一次启动或下一个完成任务的线程 join，因此突发流量不会让它的栈一直驻留。连接发生 panic 的线程会结束，而不是停驻。与之前一样，同时服务的连接最多为 `--max-connections` 个。

上面的表格来自修复之后的运行：每秒 100,000 个请求时，forward-auth 模式的 p99.9 为 62 us，反向代理模式为 150 us。修复之前的运行没有保留为结果文件，因此它们的数字只出现在 SID 0002 和本节中。

实例分析：为什么卡顿出现在 p99.9 在 64 个连接上每秒 100,000 个请求时，每个连接每秒承载 $100000/64 = 1562.5$ 个请求。守护进程在 4,096 个请求之后关闭一个 keep-alive 连接（`cmd/sibuna/src/connection.rs` 中的 `MAX_REQUESTS_PER_CONNECTION`），因此一个连接持续 $4096/1562.5 \approx 2.62$ 秒，生成器每秒打开约 $64/2.62 \approx 24.4$ 个新连接。在每连接一个线程的模型下，这相当于每秒约 24 次线程启动和 24 次线程退出。

在这一速率下运行 10 秒，会测量 1,000,000 个请求。超过其 p99.9 的有 1,000 个，即每秒 100 个；超过其 p99 的每秒有 1,000 个。如果每次更替使其他线程上约 $100/24.4 \approx 4$ 个在途请求延迟零点几毫秒，那么仅这些被延迟的请求就能填满最高的 0.1%，而最高的 1% 仍然由普通的排队构成。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。这就是为什么要看到这类缺陷，需要 p99.9 以及对每个请求计时。

Sibuna 与 Anubis

> **学习目标** 在相同负载下把 Sibuna 的数字与一个成熟产品的数字并列解读，并指出哪些差异来自产品本身，哪些来自测试设置。

Techaro 的 Anubis 是一种广泛部署的工作量证明网关，与 Sibuna 的 Gate 一样位于网站之前。我们在同一台主机、同样的 CPU 上，用同样的工作负载，把 Anubis v1.27.0 的官方发布可执行文件（`anubis-1.27.0-linux-amd64`，40,001,698 字节）与 Sibuna 并列测量。两个结果文件由以下命令记录：

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

Anubis 的启动方式与本项目早期的 Python 驱动程序相同：`--bind`、`--metrics-bind`、`--target=`（forward-auth 模式下为空，反向代理模式下为源站）、`--difficulty 2`、`--slog-level ERROR`、`--cookie-secure=false`，以及一个对每个 `Mozilla` User-Agent 发起挑战、并停用 DNSBL 和蜜罐的策略。`GOMAXPROCS` 等于 Sibuna 的 worker 数量，两个产品的每个线程都绑定到同一个 CPU 集合。Hashcash 工作量是对等的：Sibuna 为 8 个零比特，Anubis 为 2 个十六进制零位，期望都是 256 次哈希。测量工具像浏览器那样获取每个产品的会话。对于 Anubis，它获取挑战页面，从 `<script id="anubis_challenge">` 中读取挑战，找到使 `SHA-256(randomData || nonce)` 以 `00` 开头的 nonce，并连同页面的验证 Cookie 一起把它传给 `pass-challenge`。在 forward-auth 模式下，`admitted` 和 `challenged` 工作负载访问 Anubis 的检查端点（`/.within.website/x/cmd/anubis/api/check`）。

完整产品工作负载

四种工作负载、负载和统计方法都与完整产品测量工具相同，只有一处改动：源站是绑定到负载 CPU 上的 `caddy respond`，与早期驱动程序的做法一致。每次运行之前，测量工具都会等待处于 `TIME_WAIT` 状态的套接字少于 3,000 个，因为 Anubis 的反向代理运行会留下数以万计这样的套接字。

表 18 Forward-auth 模式：三次五秒 `wrk` 运行的中位数（`benchmarks/results/comparison-latest.md`，生成于 2026-10-16T00:55:19Z）。

工作负载	指标	Sibuna Gate	Sibuna Shield	Anubis 1.27.0
`admitted`	req/s（状态码）	210,682 (200)	208,635 (200)	36,509 (200)
	p99 us	305	308	7,477
	CPU us/请求	12.09	13.21	106.30
	峰值 RSS KiB	16,228	16,316	32,624
`challenged`	req/s（状态码）	198,251 (401)	193,087 (401)	113,170 (401)
	p99 us	326	342	3,781
	CPU us/请求	13.49	15.02	32.96

工作负载	指标	Sibuna Gate	Sibuna Shield	Anubis 1.27.0
	峰值 RSS KiB	16,240	16,328	32,696
allowed_static	req/s（状态码）	212,569 (200)	212,444 (200)	89,904 (200)
	p99 us	304	304	4,547
	CPU us/请求	11.73	11.95	41.63
	峰值 RSS KiB	16,240	16,328	33,396
attack	req/s（状态码）	212,086 (200)	211,347 (403)	36,853 (200)
	p99 us	305	306	7,283
	CPU us/请求	12.07	12.56	104.88
	峰值 RSS KiB	16,240	16,328	33,764

表 19 反向代理模式，以 caddy respond 为源站：三次五秒 wrk 运行的中位数（benchmarks/results/comparison-latest.md，生成于 2026-10-16T00:55:19Z）。

工作负载	指标	Sibuna Gate	Sibuna Shield	Anubis 1.27.0
admitted	req/s（状态码）	106,376 (200)	103,298 (200)	19,398 (200)
	p99 us	3,331	3,321	11,062
	CPU us/请求	27.02	27.92	190.40
	峰值 RSS KiB	16,528	16,444	39,980
challenged	req/s（状态码）	198,402 (200)	197,327 (200)	28,842 (200)
	p99 us	325	328	9,546
	CPU us/请求	13.05	13.78	137.73
	峰值 RSS KiB	16,544	16,464	516,872
allowed_static	req/s（状态码）	105,427 (200)	106,819 (200)	28,862 (200)
	p99 us	3,320	3,316	30,124
	CPU us/请求	26.24	26.52	135.41
	峰值 RSS KiB	16,544	16,464	616,548
attack	req/s（状态码）	105,616 (200)	210,483 (403)	23,399 (200)
	p99 us	3,327	306	14,124
	CPU us/请求	26.98	12.28	169.04
	峰值 RSS KiB	16,544	16,464	599,516

在 forward-auth 模式下，Sibuna 每秒处理 193,087 到 212,569 个请求，p99 为 304 到 342 us；Anubis 每秒处理 36,509（admitted）到 113,170（challenged）个请求，p99 为 3,781 到 7,477 us。Sibuna 的忙碌核心数保持在四个中的 2.49 到 2.90 之间，因此上限由负载生成器设定；Anubis 为 3.73 到 3.88，因此 Anubis 受 CPU 限制。最清楚的数字是每请求 CPU 时间：一个已准入请求在 Sibuna Gate 上耗费 12.09 us，在 Anubis 上耗费 106.30 us。这一差异的一部分很可能来自会话令牌。Anubis 默认用 Ed25519 签名会话，而 Sibuna 检查带密钥 BLAKE3 MAC（原语表中为 155.8 ns）；下面的准入测量中，HS512 令牌使 Anubis 的会话检查提高了一半以上，这支持了上述解读。

在反向代理模式下，Sibuna 经代理的各行每秒处理 103,298 到 106,819 个请求，忙碌核心数为 2.77 到 2.98。这低于 product-latest.json 中桩源站下的 138,965 到 147,303，而守护进程并未受 CPU 限制，因此上限位于守护进程之外：源站 caddy respond 与 wrk 共享负载 CPU。它的尾部体现在 2.1 到 2.4 ms 的 p90 上。Anubis 每秒处理 19,398 到 28,862 个请求，忙碌核心数为 3.68 到 3.97，每请求 CPU 时间为 135 到 190 us。它的常驻内存从 admitted 工作负载中的 39,980 KiB 增长到 challenged 中的 516,872 KiB 和 allowed_static 中的 616,548 KiB，工作负载结束后仍占用 589,276 KiB，而空闲时为 21,552 KiB。增长始于 challenged，其中每个请求都会收到一个带有自己 ID 的新挑战；这暗示它为每个签发的挑战保存了一条记录，但测量工具没有证实原因。在两种模式的每一行中，Sibuna 的峰值都保持在 16,228 到 16,544 KiB 之间。

还有几行也不是同类比较：

- 在 forward-auth 模式下，Anubis 自己以 200 回答 /robots.txt 和 attack 路径，不经过源站；Sibuna 则像回答该模式下的任何请求一样回答它们。
- Anubis 没有请求检查，因此它会准入 attack 工作负载中的 SQL 注入。Shield 在付出更多工作之后以 403 拒绝它；Gate 则像 Anubis 一样准入它。
- 两个产品签名会话的方式不同（Anubis 默认使用 Ed25519，Sibuna 使用带密钥 BLAKE3 MAC），渲染的挑战页面也不同。

准入操作

准入测量工具在 forward-auth 模式下，为构成准入的四种操作计时，每个产品使用两个 CPU（两个 Sibuna worker，GOMAXPROCS=2）。负载 CPU 上的两个客户端线程各持有一个 keep-alive 连接，共同完成每个批次：200 次操作，验证则为 64 个证明，都在批次开始之前准备好。我们报告七个批次的中位数。

- 会话检查：带有效会话访问检查端点（200）。
- 未认证检查：同样的请求，但不带 Cookie（401）。
- 挑战引导：浏览器为开始一个挑战而获取的内容。对 Sibuna 而言是两个请求，即验证中间页和挑战记录；对 Anubis 而言是一个请求，即内嵌了挑战的页面。
- 证明验证：每次操作一个新证明。Sibuna 的证明挑战以 25 ms 的间隔签发，低于其自适应难度开始增加比特的每秒 50 次，因此对等的工作量保持不变。Sibuna 以 --challenge-rate-limit 100000000 运行，因此它的限制器会被检查，但永远不会耗尽。

Anubis 运行了两次，一次使用默认的 Ed25519 令牌，一次使用 --hs512-secret。

表 20 每秒准入操作数，两个客户端，产品使用两个 CPU（benchmarks/results/admission-latest.md，生成于 2026-10-16T00:55:44Z）

产品（令牌）	会话检查	未认证检查	挑战引导	证明验证	运行后 RSS，KiB
Sibuna（带密钥 BLAKE3）	88,104	78,997	42,664	57,924	7,792
Anubis 1.27.0（Ed25519）	10,951	28,619	7,893	12,668	33,400
Anubis 1.27.0（HS512）	16,976	27,785	7,952	19,261	29,840

七个批次的范围记录在文件中。Sibuna 的范围在其中位数的 11% 以内；Anubis 的范围更宽（使用 Ed25519 令牌的未认证检查从 25,394 到 32,741 不等），而且一个 200 次操作的批次只持续 2 到 25 ms，因此我们只把这些数字解读到大约十分之一的精度。较慢客户端的每次操作时间给出了往返时间：Sibuna 会话检查为 22.7 us，使用 Ed25519 令牌的 Anubis 检查为 182.5 us，使用 HS512 时为 117.7 us。

在上面任何一行中，Anubis 处理的请求都没有比 Sibuna 多，使用的 CPU 或内存也没有比 Sibuna 少。这只是关于这台主机上这些工作负载的陈述，仅此而已。

> **如何解读 Anubis 的各行** Anubis 是一个能力出众、部署广泛的产品，这些数据行并不是在说它慢。它运行在带垃圾回收的运行时上，默认用 Ed25519 签名会话，使验证无需共享密钥，提供多种挑战方式和丰富的策略语言，并把挑战状态保存在一个存储中；这些设计选择带来的好处，这套测量工具并不测量。这些数据行显示的是 2026 年 10 月，在一台主机、一种负载形态下，一个请求在每个产品上的成本。Sibuna 是新产品，这些是它与一个成熟产品并列的首批测量结果。

控制台开销

SID 0007 规定了控制台的隔离约定。启用控制台最多只能使数据平面的吞吐量降低 1%，使 p99 延迟增加 10%，这需要在一个由配置（编译时排除、编译进来但停用、空闲、八个活动仪表板）和工作负载组成的矩阵上测量，采用多次交错运行并报告不确定度。不确定或噪声过大的运行不能证明达标，原语基线和冒烟测试也不能代替这道门槛。

SID 0010 记录了一次测量：在控制台编译进来但停用、仅启用存储、或启用但空闲的情况下，16 个产品工作负载与不含控制台构建的守护进程相差都在约 1% 以内，处于噪声范围内。该记录没有为此注明结果文件。负载下运行八个活动仪表板的配置尚未测量，因此 SID 0007 的正式门槛仍未完成。

八仪表板运行检验的是功能，而不是开销。`sibuna-browser dashboards` 在一个 SQLite 节点上启动了一个带控制台的 release 守护进程，配置四个 worker，以及一个放行 `/static/*`、`/about`、`/missing/*` 和 `/broken/*`、对其他一切发起挑战的策略；一个源站测试桩对 `/missing/*` 回答 404，对 `/broken/*` 回答 500。生成器以每秒 200 个请求的速率发送 30 秒，轮流使用十个路径（其中一个是 SQL 注入）、三个 User-Agent 和 500 个转发地址。每个仪表板运行在自己的 Chromium 中，使用自己的会话：仪表板 1 到 3 是管理员会话，仪表板 4 到 8 是用第一个仪表板生成的代码打开的大屏显示，因为控制台对每个地址和每个账户每分钟只允许五次登录。

表 21 一个节点上的八个流量仪表板（benchmarks/results/console-8-dashboards-2026-10-15.md，记录于 2026-10-15，Chromium 155.0.8059.39）。结果：PASS

指标	数值
数据平面请求	30.0 秒内 6,000 个（每秒 200 个，4 个线程，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）
启动时守护进程的 VmRSS	36,152 KiB
打开 8 个仪表板时守护进程的 VmRSS	38,028 KiB
流量结束后守护进程的 VmRSS	41,496 KiB（VmHWM 41,496 KiB）

在流量期间每两秒读取一次各张指标卡片，并在流量结束后 3 秒再读取一次。每个仪表板最终都显示相同的总数（Requests 6,000、Admitted 2,400、Challenged 3,000、Policy denied 600、Origin 4xx 600、Origin 5xx 600），收到 134 到 139 个 stats 增量，保持 WebSocket 打开，并且没有报告任何异常、控制台错误或 Content-Security-Policy 违规。八个浏览器订阅者远低于控制台 64 个的上限。这次运行的守护进程带有 SQLite 存储和控制台，不同于产品测量工具中的内存型守护进程，因此它的常驻内存不能与它们比较。

未测量的内容

表 22 本部分没有测量的内容，以及用来代替测量的依据

对象	这里为什么没有测量	代替的依据
SafeLine 社区版 9.x	只以 Docker Compose 组件栈的形式发布；主机上没有 Docker 守护进程，而检测器镜像也不是可以在那里编译的开源构建	公开发布的、由厂商报告的事实（2026 年 9 月核查）：检出率 71.65%（平衡）和 76.17%（严格），误报率 0.07%（平衡）和 0.22%（严格）；最低主机要求为带 SSSE3 的 Linux x86_64 或 arm64、1 个 CPU 核心、1 GB 内存、5 GB 磁盘、Docker 20.10.14+ 和 Compose 2.0+
Cloudflare WAF	这是一项托管服务；回环测量测到的将是网络路径，而不是 WAF	公开发布的套餐配额（2026 年 9 月核查）：自定义规则 Free 5、Pro 20、Business 100、Enterprise 1000；速率限制规则

对象	这里为什么没有测量	代替的依据
		Free 1、Pro 2、Business 5、Enterprise 100
负载下的集群	目前还没有测量工具在同一个 PostgreSQL 数据库上驱动多个节点	SID 0011 规定的协议性质（一个共享种子，挑战与签发者绑定）及其三节点功能测试（标准 2）；没有测量
请求路径上的存储	在每套测量工具中，两个守护进程都在内存模式下运行	SID 0011：一个存储线程负责所有 SQL，请求路径从不等待它
TLS、广域网、真实浏览器	测量工具使用回环接口和 HTTP；每种情形的会话由 `sibuna-pow` 求解一次	没有；部署需要测量自己的路径
浏览器求解器的求解时间	测量工具用原生求解器在 8 比特下求解	上面给出的模块大小；第 VII 部分解释了为什么原生耗时不能代表浏览器求解时间
分配次数	测量工具没有对分配器进行插桩	第 V 部分的稳态测试
负载下运行八个活动仪表板时的控制台隔离	SID 0007 的正式门槛测试尚未进行	SID 0010 的数字（控制台空闲时在约 1% 以内）以及上面的功能性运行

这张表中的一行并不是关于该产品性能的结论。SafeLine 自己的数字关注的是检测质量，而不是吞吐量；Cloudflare 公布的是套餐配额，而不是每请求成本。任何拥有 Docker 主机的人都可以在同样的 `wrk` 工作负载下运行 SafeLine。

练习 8.1. 在另一台机器上运行 `cargo xtask bench primitives`，把 Hashcash 验证和令牌验证这两行与参考主机进行比较。两者中哪一个更依赖硬件哈希指令，为什么？

练习 8.2. 在 forward-auth 表中，计算某一行的忙碌核心数：它的每请求 CPU 微秒数乘以它的每秒请求数，再除以一百万。守护进程是否受 CPU 限制？然后把它的每请求 CPU 微秒数与第一张表中的分类成本比较。为什么每请求的 CPU 成本高出这么多？哪些组成部分造成了这一差异？
提示： 列出从 `accept` 到响应被刷新之间、原语测试套件没有计时的所有内容。

解释这个不变量. 本部分的每个数字都是多次重复的中位数，旁边列出其离散范围。解释批次中位数与每请求 p99.9 之间的区别，为什么原语表报告前者而延迟表报告后者，以及为什么只有后者才能揭示上面描述的线程更替卡顿。

第 IX 部分

运维与部署

我们介绍两种形态、每个命令行选项、*forward-auth* 配置方案、策略文件、*SQLite* 或 *PostgreSQL* 中的持久存储、基于一个共享数据库的集群，以及打包。

形态与配置

> **学习目标** 学完本章，我们应当能够把 Sibuna 作为反向代理或 forward-auth 验证器运行，选择形态，设置工作量证明的层和难度，并管理主密钥。

选择形态

- *Gate*（`--gate`，别名 `--no-waf`）：工作量证明准入、会话、声明式规则、信誉、GCRA 限制、封禁。
- *Shield*（默认，`--shield`，别名 `--waf`）：Gate 加上语义防火墙。
- 分布式部署（两种形态都可选用）：`--data-dir` 把策略、信誉和取证数据保存在本地 SQLite 文件中；`--database-url` 加上 `--cluster-node` 把多个节点连接到同一个 PostgreSQL 数据库（SID 0011）。

命令行参考

下表列出 `sibuna --help` 的每个选项。范围是解析器强制执行的范围。

表 23 守护进程选项

选项	默认值	含义
`--port, -p`	`8080`	监听端口，1 到 65535
`--host, -h`	`0.0.0.0`	监听地址，一个 IP 地址字面量
`--upstream-host`	`127.0.0.1`	源站主机（反向代理模式）
`--upstream-port, -u`	`3000`	源站端口
`--mode, -m`	`reverse_proxy`	`reverse_proxy` 或 `forward_auth`（也可写作 `proxy` 和 `auth`），只能给出一次
`--workers, -w`	每个 CPU 一个	共享监听套接字的接受线程，0（每个 CPU 一个）到 64；之后每个被接受的连接都有自己的线程
`--max-connections`	`1024`	同时服务的连接数，1 到 8192；超出的连接被回答 `503`

选项	默认值	含义
`--max-connections-per-ip`	64	一个套接字对端地址可以持有的连接数，0（不限）到 8192；IPv6 对端按其 `--ipv6-prefix` 网络计数，转发头不起作用；超出的连接被回答 `503`
`--idle-timeout`	15	最长静默时间（秒），0 到 86400。请求头部必须在此时间内到达。源站响应每次读取都会刷新它，因此只要有字节流动，慢速流就能存活，而静默的源站会在两个套接字上都被切断。经代理的上传必须在每个周期内传送 16 KiB（防御慢速请求体攻击的最低速率）。客户端套接字带有同样长度的写超时，因此一个在这么长时间内不接收任何字节的客户端会失去连接
`--websocket-idle-timeout`	300	升级连接的空闲期限（秒），0 到 86400
`--trust-forwarded`	关闭；forward-auth 模式下开启	信任对端发来的 `X-Forwarded-For` / `X-Real-IP`
`--no-trust-forwarded`	关闭	即使在 forward-auth 模式下也从不信任它们
`--trusted-hops`	1	信任转发身份时，以从右数第 N 个 `X-Forwarded-For` 条目作为客户端，N 为该选项的值，取 1 到 8
`--ja4-header`	关闭	受信任的入口代理用来发送客户端 JA4 TLS 指纹的请求头，例如 `cf-ja4`；1 到 64 个 token 字符，不能是 `X-Forwarded-*`、`X-Real-IP`、`X-Sibuna-*` 或 `Forwarded`。只有信任转发身份时才读取它（见“来自入口代理的 JA4 指纹”）
`--ipv6-prefix`	64	速率限制、封禁和按客户端连接上限所作用的 IPv6 前缀长度，32 到 128
`--algorithm, -a`	posw	`posw` 或 `hashcash`
`--difficulty, -d`	16	工作比特，1 到 64：Hashcash 的零比特数，或 PoSW 深度加三
`--posw-challenges`	16	每个顺序工作量证明的揭示数，1 到 32
`--token-scheme`	mac	`mac`（带密钥 BLAKE3）或 `ed25519`
`--token-ttl`	86400	会话有效期，秒（1 到 31,536,000）
`--challenge-ttl`	300	挑战有效期，秒（1 到 86,400）

选项	默认值	含义
`--secret-file, -s`	随机	64 个十六进制字符或 32 个原始字节；也接受 `SIBUNA_SECRET`（64 个十六进制字符）
`--cookie-name`	`__sibuna_token`	会话 Cookie 名称，1 到 64 个 token 字符
`--secure-cookie`	关闭	添加 `Secure` 属性
`--gate` / `--shield`	shield	形态选择
`--rate-limit`	`100`	每个客户端每个窗口内的请求数（GCRA 突发量）
`--rate-window`	`10`	窗口长度（秒），1 到 86400
`--challenge-rate-limit`	`30`	每个客户端每个窗口内的挑战签发和验证次数，与请求预算分开计算
`--ban-seconds`	`3600`	蜜罐封禁时长，0 到 31,536,000
`--policy-file, -P`	无	声明式 JSON 策略
`--rate-table-cells`	`262144`	每张速率限制器表的单元数，每个 16 字节，4,096 到 16,777,216，向上取整为 2 的幂
`--ban-table-slots`	`65536`	封禁表槽数，每个 24 字节，1,024 到 16,777,216，向上取整为 2 的幂
`--spent-tags`	`1048576`	为防重放而记住的已解挑战数，每个 24 字节，65,536 到 16,777,216
`--data-dir, -D`	无	单节点：持久化到 SQLite 文件 `<path>/sibuna.db`
`--database-url`	无	集群：共享的 PostgreSQL 数据库（`postgres://` 或 `postgresql://`）
`--database-url-file`	无	从一个仅所有者可读的文件中读取数据库 URL
`--cluster-node`	无	本节点的 ID；绑定挑战密钥和事件标识符的高位；使用数据库 URL 时必需
`--storage-poll-ms`	`500`	存储节拍间隔，1 到 60,000 ms
`--verbose, -v`	关闭	详细日志
`--help` / `--version`		打印帮助（编译了控制台时，控制台选项列在最前）或版本号，然后退出

`--data-dir` 与数据库 URL 互斥，`--database-url` 与 `--database-url-file` 也互斥。没有 `--cluster-node` 的数据库 URL 会被拒绝。Zaxonlite 的集群选项 `--cluster-listen`、`--cluster-peer`、`--cluster-secret-file` 以及 `--cluster-tls-cert`、`--cluster-tls-key` 和 `--cluster-tls-ca` 已被移除；给出其中任何一个都会以 `RemovedOption` 终止启动，并附带一条指向 SID 0011 的提示。控制台的选项列在下一章。

主密钥 没有 `--secret-file` 或 `SIBUNA_SECRET` 时，守护进程会生成一个随机种子，启动横幅打印 “random (set –secret-file)”。此时令牌和挑战会随进程一起失效，而节点持有不同种子的集群会相互拒绝对方的 Cookie。用 `head -c 32 /dev/urandom | xxd -p -c 64 > /etc/sibuna/secret` 生成一个，并设置权限为 0600。

控制台预览

默认的 `console` 特性会把控制台编译进守护进程，但只有使用 `--console` 时才会启动。它需要存储：没有 `--data-dir` 或 `--database-url` 的 `--console` 会以 `CONSOLE001`（`StorageRequired`）被拒绝。使用 `cargo build --release -p sibuna --no-default-features --features sqlite` 可以把它排除在编译之外；只用 `--no-default-features` 的构建既没有存储也没有控制台。在守护进程停止时于本地完成初始化：

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

`sibuna console init-admin` 是同一个命令。打开 `http://127.0.0.1:19446/console/` 并更换生成的临时密码；该密码必须在首次登录时修改，一小时后过期。登录后的导航提供动画国家地球、请求和挑战统计、事件调查、策略和检查编辑器、用户、限定范围的 API 令牌、审计以及服务节点控制。认证界面不加载任何地球几何数据或遥测数据。界面是一个由守护进程的构建脚本从 `cmd/console-ui`（crate `sibuna-console-ui`）构建的 WebAssembly 模块；它和已提交的样式表随普通构建一起发布。

表 24 控制台选项

选项	含义
`--console <ip:port>`	在这个监听地址上启动控制台（IPv6 放在方括号中，端口非零）；需要 `--data-dir` 或 `--database-url`
`--console-key-file <path>`	64 个十六进制字符，仅所有者可访问；保护存储的第二因素密钥和 webhook 密钥
`--console-origin <https-origin>`	代理之后的规范 HTTPS origin
`--console-behind-proxy`	每个请求都经由一个终止 TLS 的代理送达
`--console-trusted-proxy <cidr>`	一个其连接可以携带转发字段的网络；可重复，最多 16 个
`--console-advertise <origin>`	其他控制台为本控制台显示的链接
`--console-location <latitude,longitude>`	本节点在地球上声明的位置
`--console-probe <node-id>=<http://ip:port>`	一个需要健康检查的对等数据平面监听地址；可重复，最多八个

选项	含义
`--console-peer <node-id>=<https://origin>`	一个管理对等节点；可重复，最多八个
`--console-peer-key-file <path>`	对等密钥：仅所有者可访问，64 个十六进制字符，独立于控制台密钥
`--console-peer-ca-file <path>`	可选的对等节点 PEM 信任锚，替代系统根证书
`--console-query-steps <n>`	每次聚合读取的 SQLite 步数，100,000 到 50,000,000，默认 4,000,000
`--console-capture-heads`	为每个事件存储脱敏后的请求头部和源站响应头部；默认关闭
`--console-capture-header <name>`	在存储的头部中保留这个头字段的值；可重复，最多 16 个；需要 `--console-capture-heads`
`--console-cookie-secure`	在没有代理的情况下为控制台 Cookie 加上 `Secure` 属性（解析器接受，但不在 `--help` 中列出）

没有 `--console` 的控制台选项、重复的选项或格式错误的值，会以 `CONSOLE002` 和该选项的错误名称终止启动。违反控制台规则的配置（例如没有受信任代理却监听非回环地址）会以 `CONSOLE001` 终止启动。

控制台导览

下面的截图来自一个 release 构建，运行在一个带 SQLite 存储的回环节点上，策略是在默认挑战之上加四条 `ALLOW` 规则，截图前运行了大约四分钟的合成流量：来自文档地址段的地址、SQL 注入和跨站脚本探测、三次蜜罐命中，以及在无头 Chromium 中解出的七个挑战。因此，每个数字都只是示意。我们用 `cargo xtask browser console --viewport-only` 截取它们，该命令在 1440 x 960 像素的 Chromium 中遍历控制台，并检查每个页面。每个页面回答其标题中提出的一个问题，并在同样的位置显示产品、服务节点、当前页面和返回路径。

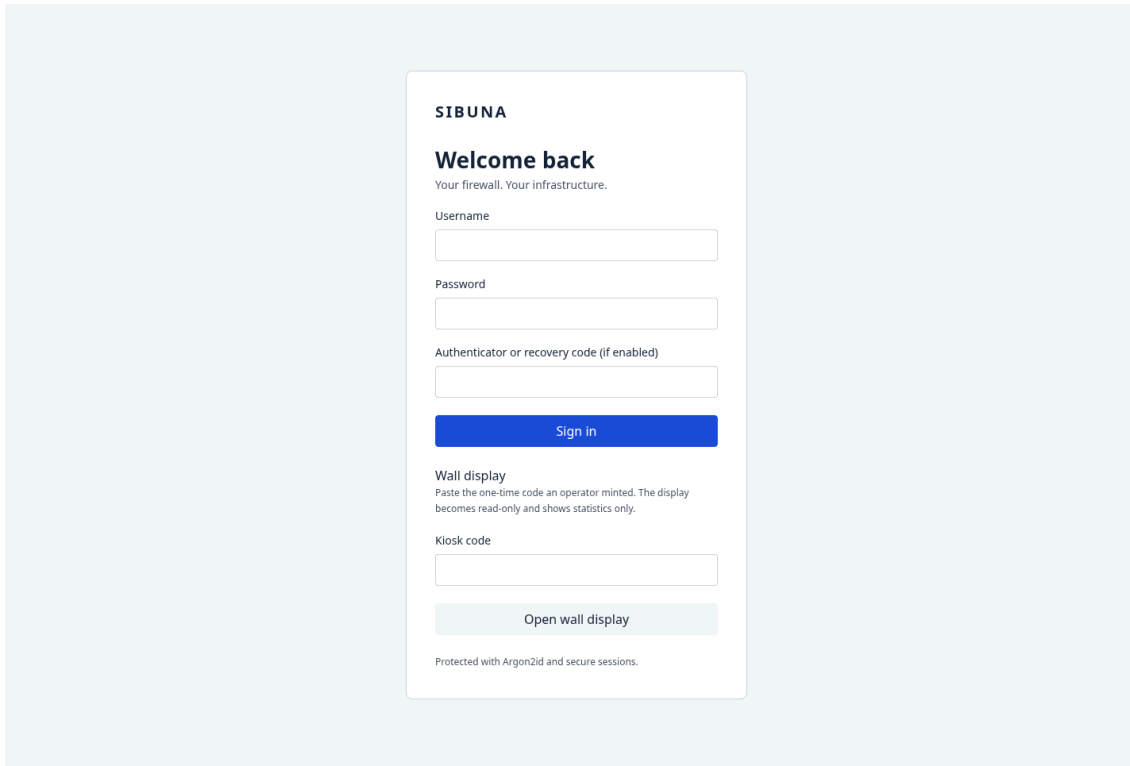


图 13 登录。公开界面不加载任何地球几何数据、GeoIP 数据或遥测数据；第二个表单用大屏显示代码换取一个只读的统计会话。

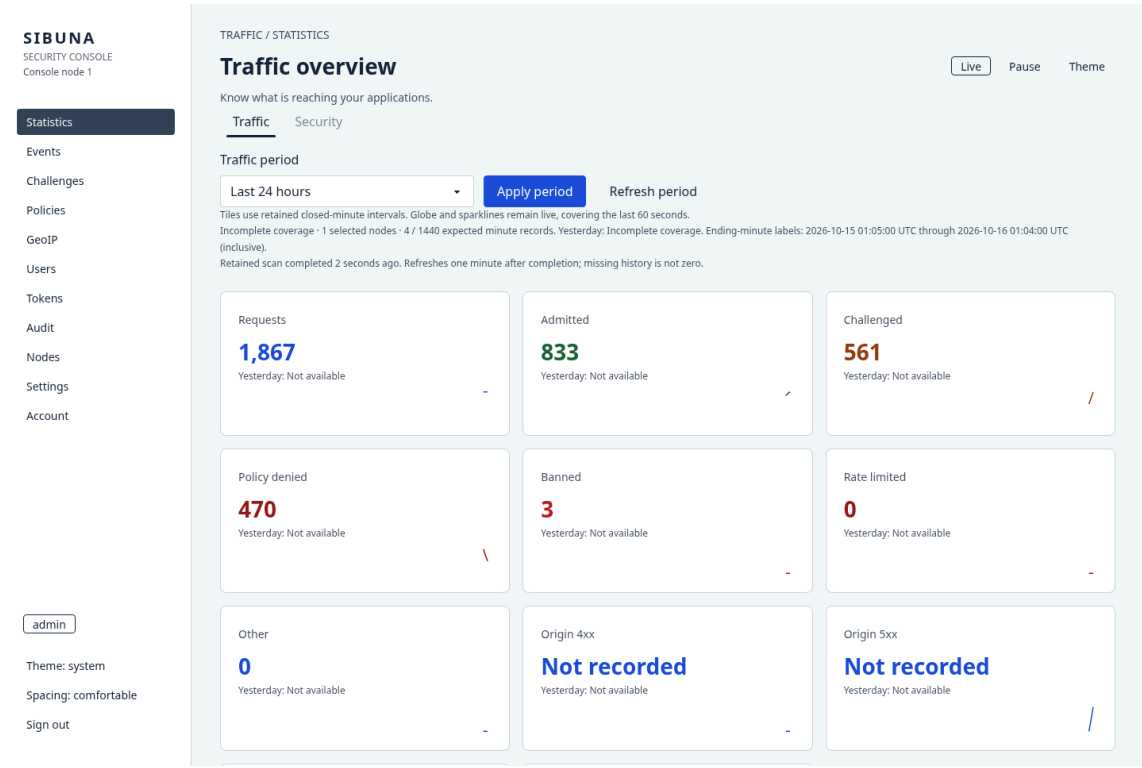


图 14 统计，流量，最近 24 小时。时间段控件下方的说明给出覆盖情况：预期的 1,440 条分钟记录中只存在四条，因此每个 “Yesterday” 都显示 “Not available”。指标卡片显示保留的已结束分钟计数，并附有实时的 60 秒迷你走势图；决定的颜色（准入为绿色，挑战为琥珀色，拒绝为红色）在每个面板中都相同。对于保留的时间段，源站 4xx 和 5xx 显示 “Not recorded”，因为分钟记录没有保存赋予源站计数意义的代理模式；实时的本次启动总数会显示它们。可见指标卡片下方还有两张显示当前水平的指标卡片（活跃封禁条目和健康节点）、时间线，以及若干抽样面板，它们对请求路径和来源主机排名，并统计客户端操作系统、浏览器和响应状态码。

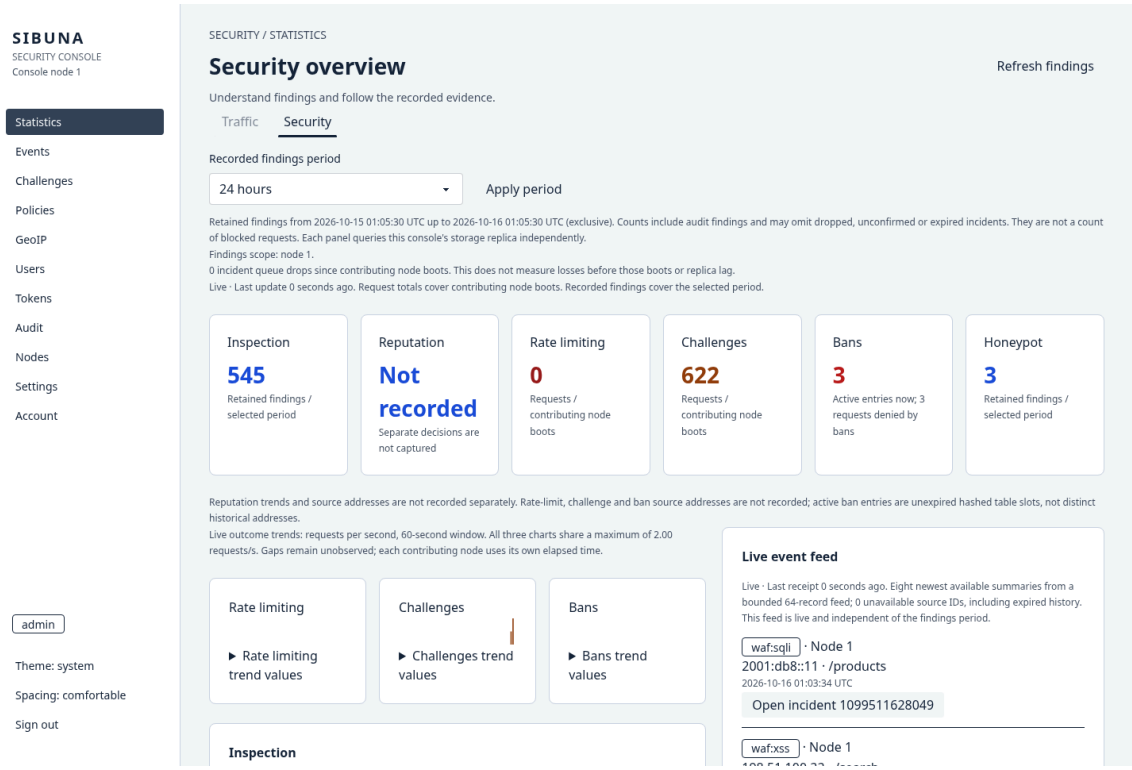


图 15 统计，安全。所选时间段内每个模块一个指标卡片，实时结果趋势，以及带有最新检出结果的实时事件流（这里是 `waf:sqli` 和 `waf:xss`）。未记录的值会如实说明，而不是显示为零：信誉指标卡片显示 “Not recorded”。页面再往下是按六十个等宽时间桶统计的检出结果、攻击类别和被攻击的路径。

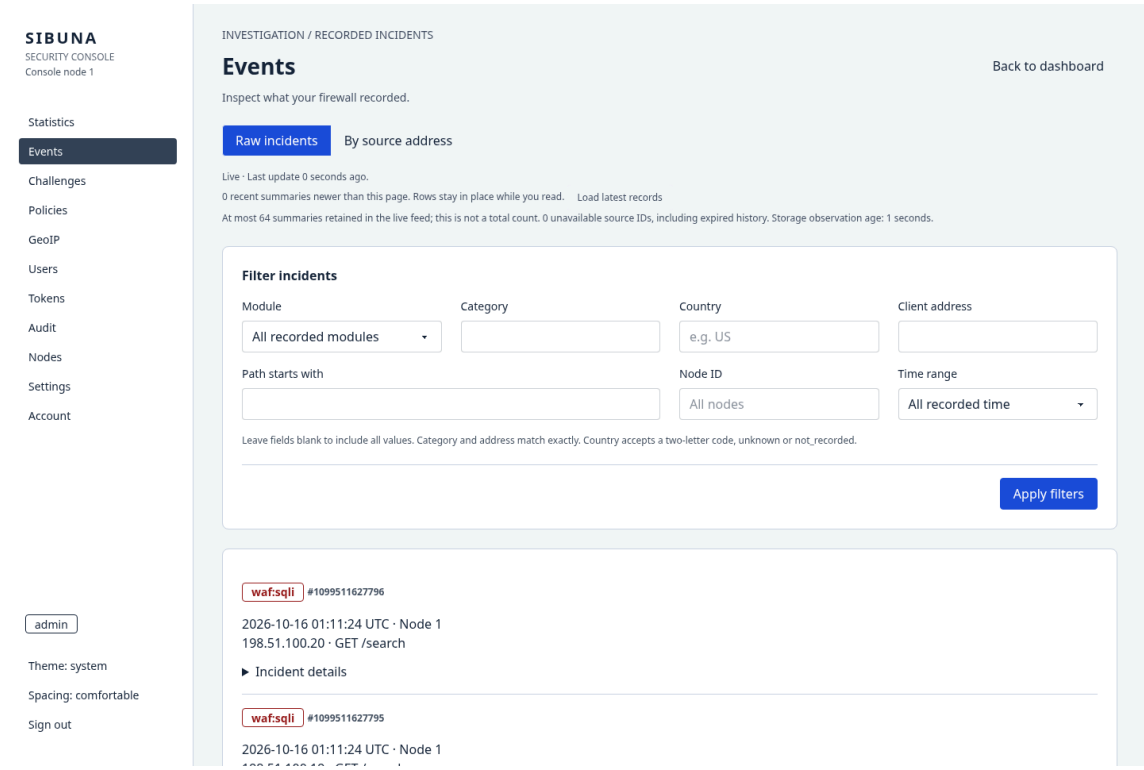


图 16 事件。已记录事件上方的筛选表单，最新的排在最前。每个事件都可以就地展开并显示其证据：所选的本地响应、字节长度、候选攻击活动，以及对匹配规则、评分项和 JA4 指纹明确标注的 “not recorded” 条目。使用 `--console-capture-heads` 时，脱敏的请求头部（对于经反向代理、处于审计模式的准入，还有源站响应头部）可以按需展开，以 UTF-8 或 Latin-1 渲染，并注明响应情况（已捕获、本地响应、forward-auth 下未观测、源站不可用），附带一条 “复制为 cURL” 命令；只保留列出的头字段的值，`--console-capture-header <name>` 可以向该列表中添加。否则头部显示为未记录。拒绝和放行动作会打开 IP 组表单，并预先填好地址。

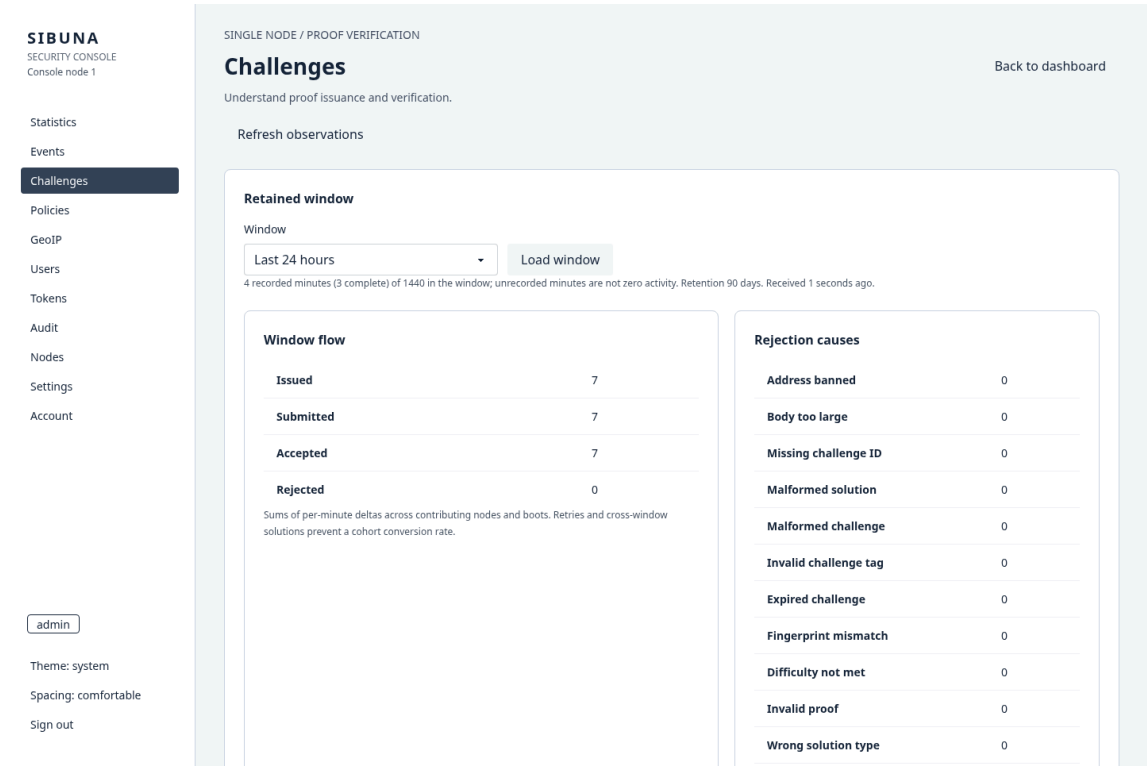


图 17 挑战。一个保留的时间窗口（一小时、一天或七天）汇总持久化的每分钟记录，并说明有多少分钟被记录且完整；这里签发、提交并接受了七个挑战，各项拒绝原因都为零。页面再往下是本次启动以来的实时总数、配置的参数和最近签发的参数、按算法和参数分组的已接受求解时间直方图、自适应难度的变化，以及带有拒绝和放行快捷操作的按地址记录，保留七天。

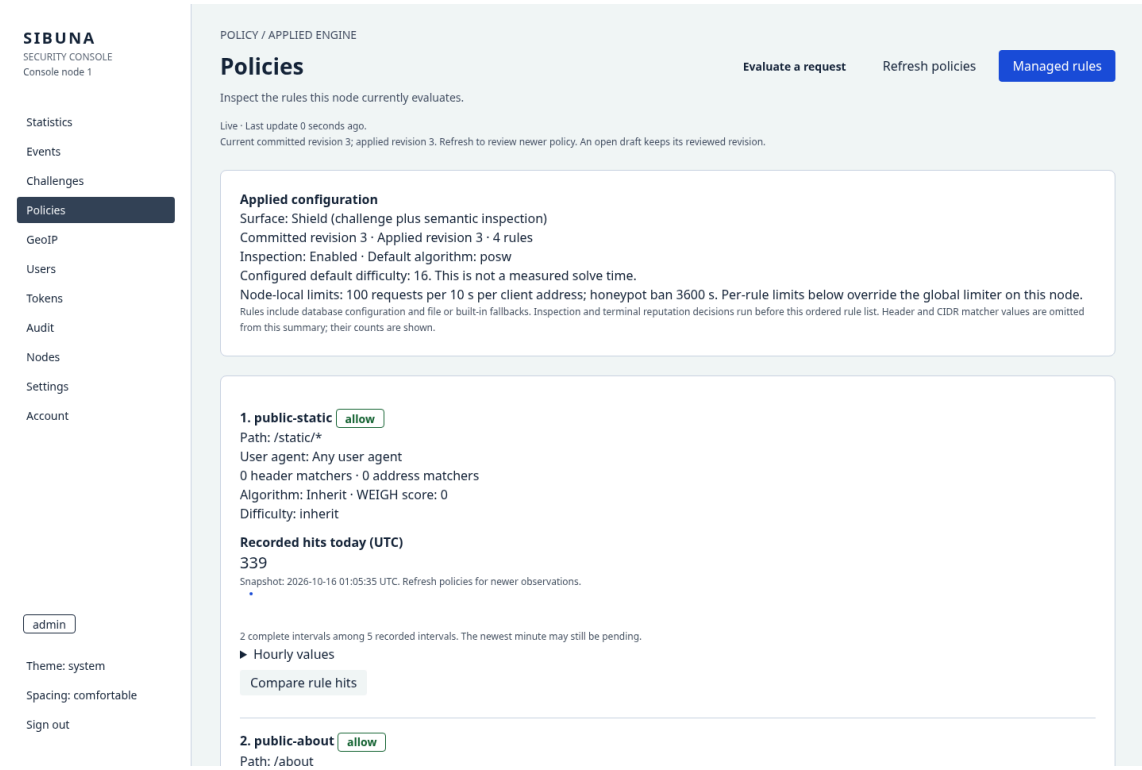


图 18 策略。已应用的引擎列出它的形态、修订版本、默认算法和难度，以及它的节点本地限制，然后按评估顺序列出规则、它们今天的命中次数以及这些计数的覆盖情况。页面再往下是检查模式矩阵、请求测试器和 IP 组。*Managed rules* 打开编辑器；保存修改时会先显示被修改字段的审阅，确认后会验证整个候选并创建一个新的修订版本。

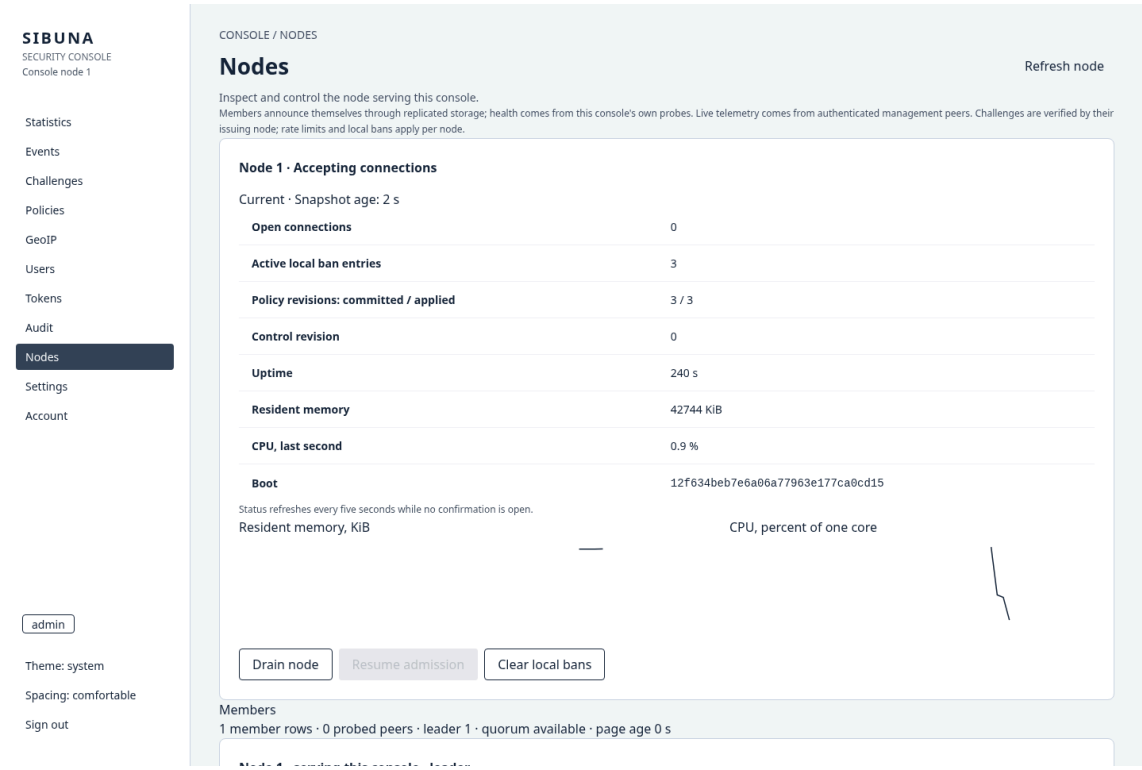


图 19 节点。服务节点的状态和命令，以及最近一秒的常驻内存和 CPU 及其六十秒迷你走势图；下方是每个已通告的成员，包括它已应用的策略修订版本与已提交修订版本的对照、它的管理地址和探测结果（SID 0007, SID 0013）。

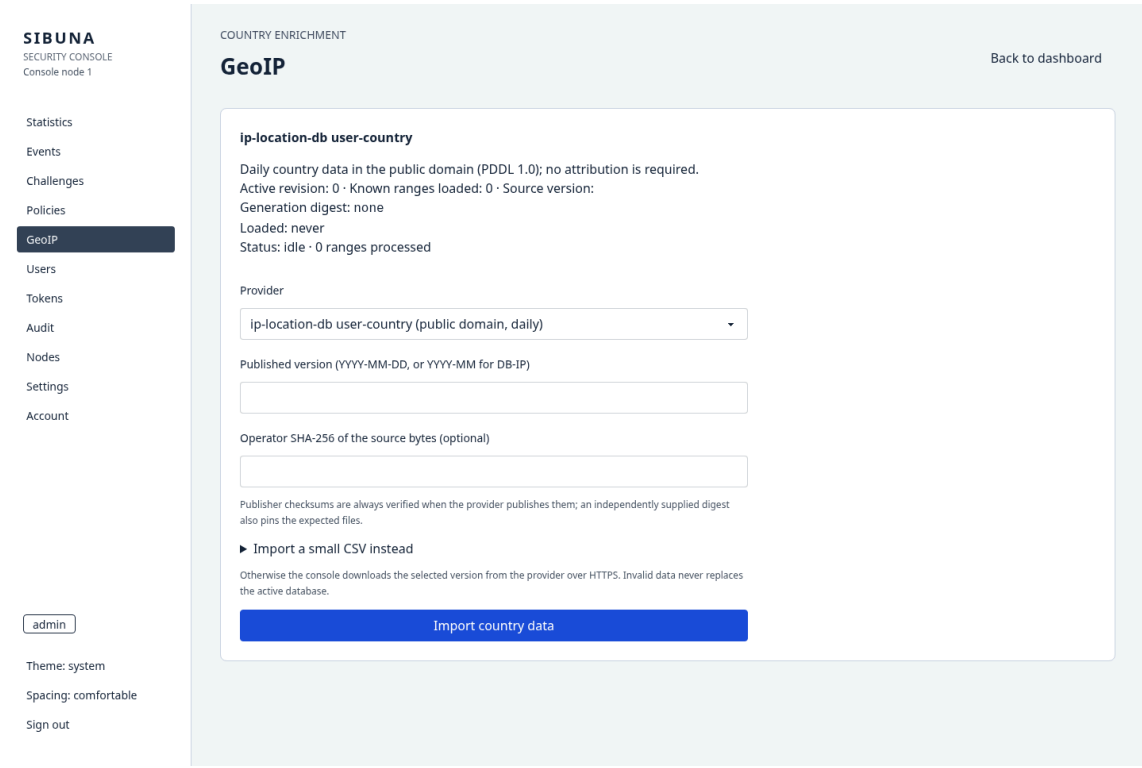


图 20 GeoIP。提供方、许可证、当前生效数据代的摘要、加载时间和导入表单；这里尚未加载任何数据代。失败的导入永远不会替换当前生效的数据代。

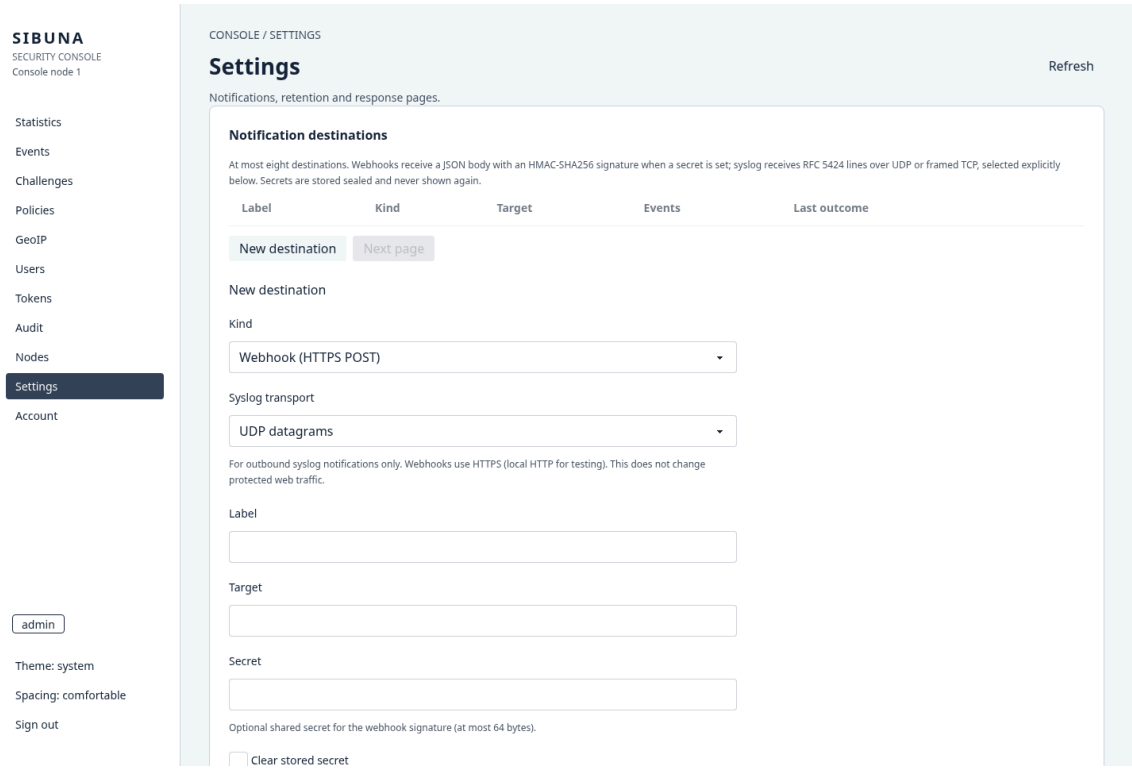


图 21 设置，此时还没有通知目标，页面上是新建通知目标的表单。页面再往下是拒绝激增阈值、按时间窗口逐一明确确认的保留期设置、响应页面模板和“关于”。

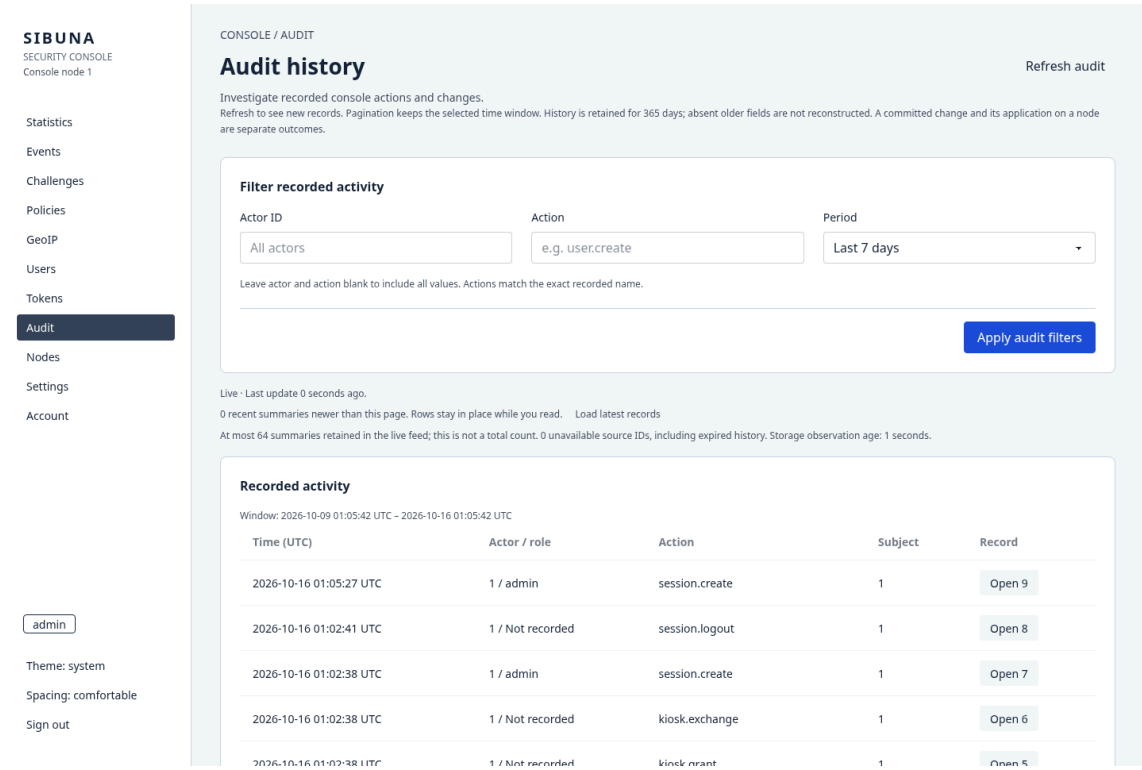


图 22 审计。只追加的历史记录，包含执行者及其角色、动作、对象，以及一个显示脱敏前后摘要的详情视图；被拒绝的登录也会被记录。每条管理变更和认证记录都会记下出示凭据的客户端地址；系统自行写入的记录将其显示为未记录。策略记录提供 *Revert this change*，它在确认后把该修订之前记录的文档恢复为一个新的、经过审计的修订版本；如果规则集已经发生了后续变化，则会被拒绝。

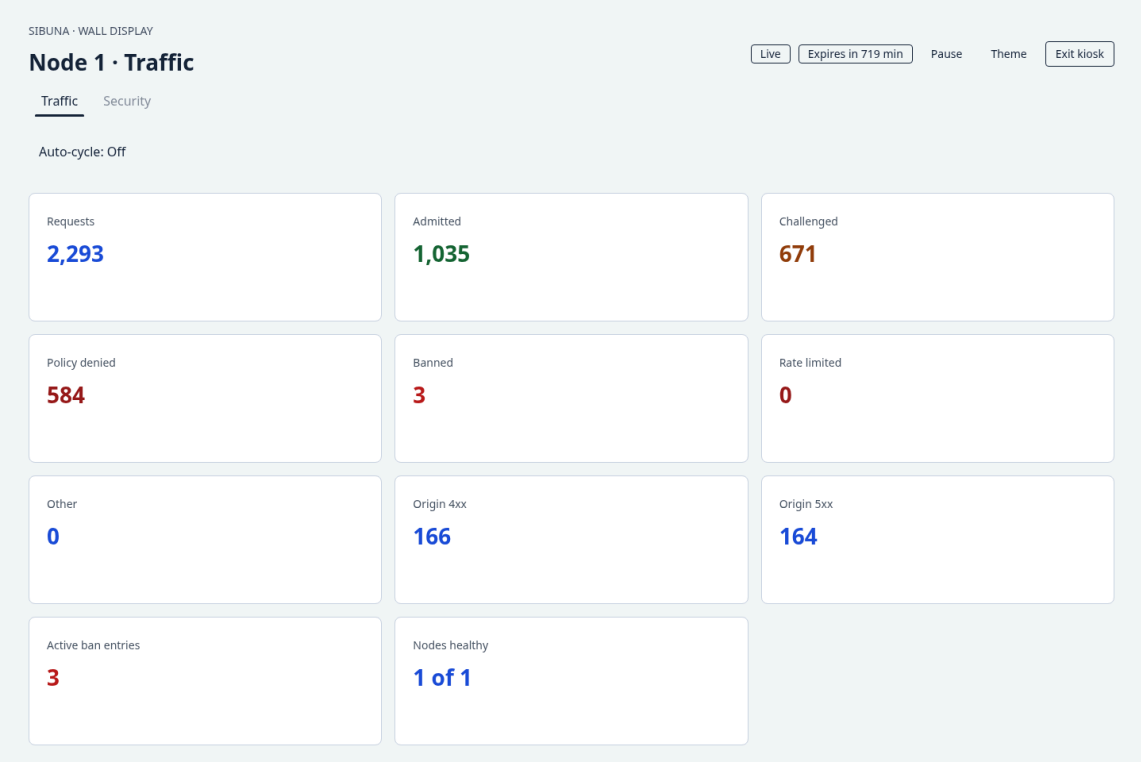


图 23 大屏显示。大屏会话只显示流量和安全视图，没有导航，并注明会话何时过期。

可选的 `--console-location <latitude,longitude>` 声明本节点的位置，例如新加坡的部署为 `1.3521,103.8198`。地球最初以该点为中心；旋转之后，Center Sibuna 会回到该点。各国的活动沿着朝向节点的动画大圆弧显示，背面半球和平面地图上的日期变更线会被裁剪。国家位置是具有代表性的中心点，箭头表示观测到的六十秒窗口，而不是单个连接。未设置的服务器位置明确保持为未知。遥测中断期间，配置的标记仍然可见；过时的流量不会产生动画。

策略预览会构建一个包含已配置文件规则的私有候选。保存时会比较预期的修订版本，并要求确认字段对比。历史回退会与当前规则比较，并创建一个新的修订版本。已提交的修订版本与本地已应用的修订版本是不同的。审计详情保留有界的决策变化以及实际生效的执行角色。匹配器的值会被脱敏，缺少上下文的旧记录明确保持为缺失。排空、恢复和清除本地封禁需要先预览命令。当节点上另一条命令正在运行时到达的命令会以 `503 CONSOLENODE` 被拒绝，而不是排队。丢失响应后，先检查持久化的回执再重试；仅凭已提交的意图并不能说明运行时效果已经完成。

远程运维时，请使用 HTTPS 代理，并配置 `--console-origin`、`--console-behind-proxy`、明确的 `--console-trusted-proxy` CIDR 和 `--console-key-file`。密钥文件包含 64 个十六进制字符，权限仅限所有者，用于保护存储的第二因素密钥；重启时要保留它，并与挑战种子分开保存。对于非回环访问，配置的 HTTPS origin 和代理允许列表是必需的。Sibuna 在控制台监听地址上同样不终止入站 TLS；控制台只在其出站连接中通过 rustls 使用 TLS（SID 0012）。

原生 CLI 通过相同的授权和存储约定访问运行中的控制台。它从私有文件中读取凭据，而不是从命令行参数中读取：

```
sibuna console geoip update --version 2026-09-09 \
```

```
    --origin http://127.0.0.1:19446 --username admin \
    --password-file ./admin-password
```

请使用仅所有者可读的密码文件，先完成密码设置，并在需要时为认证器代码或恢复代码加上 `--factor-file`；对于账户和 GeoIP 操作，`--token-file` 可以代替用户名和密码。客户端使用 HTTP/1.1，除字面量回环地址外都要求 HTTPS，从不跟随重定向，并把每个请求限制在 20 秒以内、每个回复限制在 16 KiB 以内。`geoip status` 读取当前生效的数据代。更新时通过 HTTPS 下载所选提供方发布的文件（默认是公有领域的 `user-country` 数据集，或用 `--provider dbip --version YYYY-MM` 选择 DB-IP Lite），校验发布者的校验和（或 `--checksum` 给出的校验和），通过 `sibuna-geoip` 验证有界的地址范围，持久化新的数据代并在本地激活。失败的更新会保留之前的数据代。CLI 超时（`--timeout`，默认 1200 秒）只是停止等待；它不会取消已提交的存储工作。DB-IP Lite 要求 CC BY 4.0 署名，只在其数据生效期间显示。使用 `SIBUNA_GEOIP_DATA=<snapshot>` 的构建会嵌入一个经过验证的快照（由 `geoip-snapshot` 工具写出），直到第一次持久导入为止；不带控制台的构建会以 `GEOIP001` 拒绝它。未知地址、样本丢失和过时数据都保持可见；导入国家数据不会产生流量，也不会补充已经过期的样本。

集群成员在启动时、每分钟一次，以及已应用的策略或控制修订版本变化后的两秒内，通过共享数据库的成员表（`console_nodes`）通告自己，并出现在每个控制台的节点页面上。没有领导者选举：SQLite 节点报告角色 `single` 并以自己为领导者，PostgreSQL 节点报告角色 `unknown` 且没有领导者，因为 PostgreSQL 的故障切换对 Sibuna 不可见（SID 0007）。为需要由本控制台对其数据平面监听地址进行健康检查的对等节点配置 `--console-probe <node-id>=<http://ip:port>`，并用 `--console-advertise <origin>` 配置其他控制台显示的链接。直接的实时遥测使用 `--console-peer <node-id>=<https://origin>`（可重复，最多八个）和专用的 `--console-peer-key-file`。对等节点需要 `--console-behind-proxy`。两端必须相互列出对方，并使用受信任的 HTTPS 入口。对等密钥是一个仅所有者可访问、包含 64 个十六进制字符的文件，与控制台密钥和挑战种子分别配置。对等节点之间通过一个经过认证的 TLS WebSocket 通信，使用带 `ring` 提供方的 rustls（SID 0012）。`--console-peer-ca-file` 可选地为私有管理 PKI 提供 PEM 信任锚；否则客户端使用系统根证书。主机名验证始终生效且无法关闭：每个管理 origin 的主机（DNS 名称或 IP 地址字面量）必须出现在对等节点的证书中。被拒绝的证书会把该对等节点标记为 `rejected`，并写入一条 `peer.rejected` 审计记录。轮换密钥和更换 CA 文件需要协调重启；对等节点的证书可以在不重启的情况下续期。节点 API 及其订阅会报告接收时延、时钟偏差、启动变化和采样丢失。缺失的观测保持为不可用，断开连接的值保持为过时；收到的统计数据永远不会变成另一个节点自己的贡献。仪表板的 *Live traffic scope* 选择一个节点，或合并已配置的各节点。*Node coverage and locations* 标出缺失、过时和存在时钟偏差的来源、观测时间和配置的目标位置。国家排名显示由被省略的来源行带来的不确定性。箭头保留其接收节点，缺失的节点位置不会被凭空捏造。只要任何一个参与的来源缺少一个连续的时间间隔，合并后的速率就保持为未观测。选择一个节点可以查看它保留的秒级数据和当前分钟的路径排名。远程查询使用经过认证的对等连接；查询结果必须能放进一条 4 KiB 的 WebSocket 消息，因此当一页放不下时，提供服务的控制台会把该页的行数减半。缺失的对等节点保持为不可用，历史游标不能跨越节点重启。分钟历史保留它自己的节点选择和按节点的行。导航栏和移动端页头标识的是提供服务的控制台节点，与所选的流量来源无关。本地命令仍然按节点执行。

流量指标卡片默认显示所选节点最近 24 小时保留的已结束分钟记录。时间段控件还提供一小时、七天或九十天，以及实时的本次启动总数。与昨天的偏差需要每个所选节点都有完整匹配的覆盖。地球和迷你走势图仍然描述它们各自独立的实时 60 秒窗口。保留数据的扫描在完成

一分钟后刷新，在被替换之前保持之前的值及其时效可见；更换来源或时间段会丢弃旧的范围。缺失的历史永远不会变成零。

Compare retained traffic 把两个已结束的分钟窗口并排显示：昨天的同一窗口、前一个时间段，或另一个节点。时长和结束偏移控件会冻结两个 UTC 范围。每次操作每侧最多读取十六个紧凑页面，每页 96 条存储记录。这可以在一个批次中覆盖一个不跨越重启的 24 小时窗口；Continue 会为更长的扫描保持相同的边界。完整覆盖、不完整覆盖和缺失历史都会明确标出。速率百分比要求在相等的 UTC 窗口中有完整且不重叠的时间间隔，并按观测到的毫秒数归一化；参考值为零而出现新流量时显示 “New”。主流量时间段保持独立。分钟格式不记录历史的代理模式，因此源站响应的比较保持为不可用，而不是把 forward-auth 下的零解读为观测到的响应。

Compare retained path rankings 选择两个已结束的窗口或节点。节点 0 包括所有保留的节点，包括已退役的成员。一次操作最多读取十六个不可变的归档，检查它们的标识和校验和，并在选择显示行之前合并所有计数器。归档在 60 秒的迟到样本窗口之后封存，因此最新的已结束分钟可能仍在等待中。Continue 保持原来的窗口和游标。边界描述的是抽样的路径前缀，而不是精确的请求总数。页面会显示部分扫描、截断、保留期边界和报告的队列丢失；缺失的归档不能证明流量为零或覆盖完整。

安全视图默认显示最近 24 小时，并把所选时间段分成六十个等宽的时间桶。它的聚合在存储线程上运行，并受一个工作量上界的约束。在 SQLite 上，这个上界是 `--console-query-steps` 个虚拟机步数（100,000 到 50,000,000，默认四百万）。它与 `--console` 一起设置；无效或重复的值会被拒绝。在 PostgreSQL 上，这个上界是在读取事务中设置的 2,000 ms 语句超时（SID 0013）；修改 `--console-query-steps` 不会改变它。耗尽上界的查询会提示缩小时间段。步数限制的是查询工作量，而不是经过的时间。各图表共享同一刻度；展开趋势值可以看到 UTC 区间的起点和精确的分组计数。这些都是保留的检出结果，因此缺失的事件覆盖不能被解读为没有攻击。

大屏显示使用大屏会话。在 Account 下，运维人员为显示屏命名，然后选择 Create display code。显示屏需要在十分钟内把这个一次性代码粘贴到登录表单中；它的会话是只读的，仅限于统计，并在十二小时内过期。账户页面在导航离开或被隐藏时会清除显示的代码。隐藏不会撤销一个尚未使用的授权；兑换期限仍然有效。代码绝不应出现在 URL 中。流量和安全视图共用显示屏：安全视图显示聚合的模块趋势和请求结果，不显示事件地址或载荷证据。自动轮播是可选的，初始关闭，在减少动态效果、数据过时或 Pause 时暂停。

通知目标（针对拒绝激增、封禁、不可达成员和领导者变更的签名 webhook 和 syslog）位于 Settings 下；webhook 密钥需要 `--console-key-file`。领导者变更这一类型仍然保留定义，以便已存储的目标保持其含义，但没有任何节点会触发它，因为没有领导者选举。每个目标有自己的冷却时间和三次尝试的重试预算。投递审计记录显示结果；已完成的队列历史限制为七天和 4,096 个事件。webhook 包含一个稳定的 `Idempotency-Key`，使接收方可以在网络完成情况不确定时抑制重复的效果。webhook 通过 HTTPS 发送，只有发往字面量回环地址时才使用普通 HTTP，回复体最多读取 64 KiB。syslog 的 UDP/TCP 选择只适用于出站通知，与受保护的 Web 流量无关。手动测试会在审计中记录意图和完成情况，并刷新目标的结果；未确认的审计完成情况会在运维人员重试之前明确显示。

Settings 还控制保留期：分钟历史为 1 到 90 天，排名为 1 到 7 天，事件为 1 到 30 天，审计为 1 到 365 天。上限值即默认值；排名保持其 512 MiB 的配额。保存保留期的值需要确认，因为清理会以有界批次永久删除较旧的记录。之后再增大这个值也不会恢复已删除的历史。过时的修改会被拒绝，成功的修改会出现在审计中。

响应页面（挑战、拒绝、限速、封禁、过载）可以在 Settings 下以带固定占位符的有界 HTML 形式编辑；草稿在一个沙箱标签页中预览，保存的页面从下一个策略快照开始提供。

策略页面上的策略工作流程：对托管规则重新排序，针对保留的检查检出结果重放草稿，管理 IP 组和固定到当前 GeoIP 数据代的国家封锁，以及导出或以原子方式导入托管规则集（也可以用 `sibuna console policies export` 和 `sibuna console policies import --file <set.json>`）。一次导入是一个请求，携带 1 到 128 个文档，每个最多 4 KiB，总计最多 256 KiB；对于超过约 96 KiB 的规则集，页面说明会指出命令行导入方式。已应用的规则显示今天记录的命中次数、按小时的迷你走势图和一张无障碍表格。一次命中是指一次成功的声明式匹配器评估：匹配的 WEIGH 规则和第一个终止性匹配都会计数；更早的检查或信誉决定可能使评估不会发生。私有测试不会递增这些计数器。*Compare rule hits* 为一个已记录的节点冻结两个已结束的时间段。修订历史提供 *Compare hits around this edit*，它排除修改所在的那一分钟，并使用不超过所选时长的相等可用时间段。可选的已应用修订版本筛选器可以把无关的数据代排除在比较之外。启动、切换、缺失的写入和保留期都会留下可见的覆盖缺口；百分比要求两侧都有完整覆盖。规则观测以及按小时/按天的汇总遵循分钟保留期设置。在一次修改前后观测到的变化只是一种比较，并不能证明是这次修改导致了流量变化。

导入较新的 GeoIP 数据代不会自动刷新已有的、由国家派生的信誉行。先预览国家动作，比较新增、保留和移除的前缀；在应用之前翻阅审阅过的差异。替换会移除该国家所拥有的过时行，并保留独立管理的前缀。数据代或策略修订版本发生变化后需要重新预览，相互重叠的独立修改会被拒绝。事件页面包含保留的 WAF 检出结果和蜜罐事件；它不是完整的访问日志。加载了 GeoIP 时，存储线程会在事件批次的事务中记录每个事件的国家和数据代。这是持久化时的归属，而不是重建出的请求时刻位置。之后的导入不会改变已记录的映射。国家筛选器接受一个大写的两字母代码，`unknown` 表示不在已加载数据代中的地址，`not_recorded` 表示没有映射数据的事件。来源分组会明确报告混合的国家或覆盖情况。地球的 *View events* 动作保持所选节点，并打开该国家最近一小时内保留的事件。统计页面的 *Security* 视图把实时的限速、挑战和封禁速率，与一小时、一天、七天或三十天内保留的检查和蜜罐检出结果结合起来。类别、来源和路径链接以相同的节点和固定的时间边界打开常规的事件工作流程。应用事件筛选器会开始一个新的时间段。检出结果包括审计记录，与被阻止请求的总数不同；缺失的信誉和规则命中归属显示为未记录。事件和审计筛选器让标签与其控件保持在一起，把 Apply 动作单独对齐，并让列宽适应可用的内容宽度。一个经过认证的 WebSocket 在导航过程中始终存活，承载统计、事件摘要、节点状态、策略修订版本、挑战和审计摘要。新的事件和审计记录在 *Load latest records* 之后等待，使表格在阅读时保持不动。策略更新会显示当前已提交和已应用的修订版本，而不会替换打开的草稿。流量计数器实时更新；所选的非默认挑战计时分区需要显式刷新。历史查询、详情读取和变更操作仍然是 HTTP 请求。

页面片段可以加入书签；后退和前进会重新打开需要认证的页面。侧边栏还会在本浏览器中记住主题和间距设置，默认主题为 System。SID 0007 的状态为 Committed。控制台在无头 Chromium 中通过了黄金样例和浏览器工作流测试（`cargo xtask browser console`，SID 0010 标准 5）。它的端到端测试套件是 `cmd/sibuna/tests/console_*.rs`，其中的 PostgreSQL 集群场景在设置了 `SIBUNA_TEST_DATABASE_URL` 时运行。SID 0010 记录：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。SID 0007 的正式隔离门槛，即在负载下运行八个活动仪表板，尚未测量（第 VIII 部分）。完整的 WebAssembly 应用有 768 KiB 的未压缩上限，守护进程的构建脚本在超出时使构建失败；另有 640 KiB 的审查阈值。在 1.0.0-beta.3 中，该模块大小为 724,330 字节（gzip 压缩后 272,568 字节），超过了审查阈值，SID 0007 已将其记录待审。这些项目限制为控制台的工作流留出了空间；它们不能代替对浏览器加载速度和响应速度的测量。模块的线性内存按需增长，声明的上限为 16 MiB。

部署拓扑

学习目标 在 Nginx 或 Caddy 之后部署自主运行的反向代理和 forward-auth 验证器。

反向代理

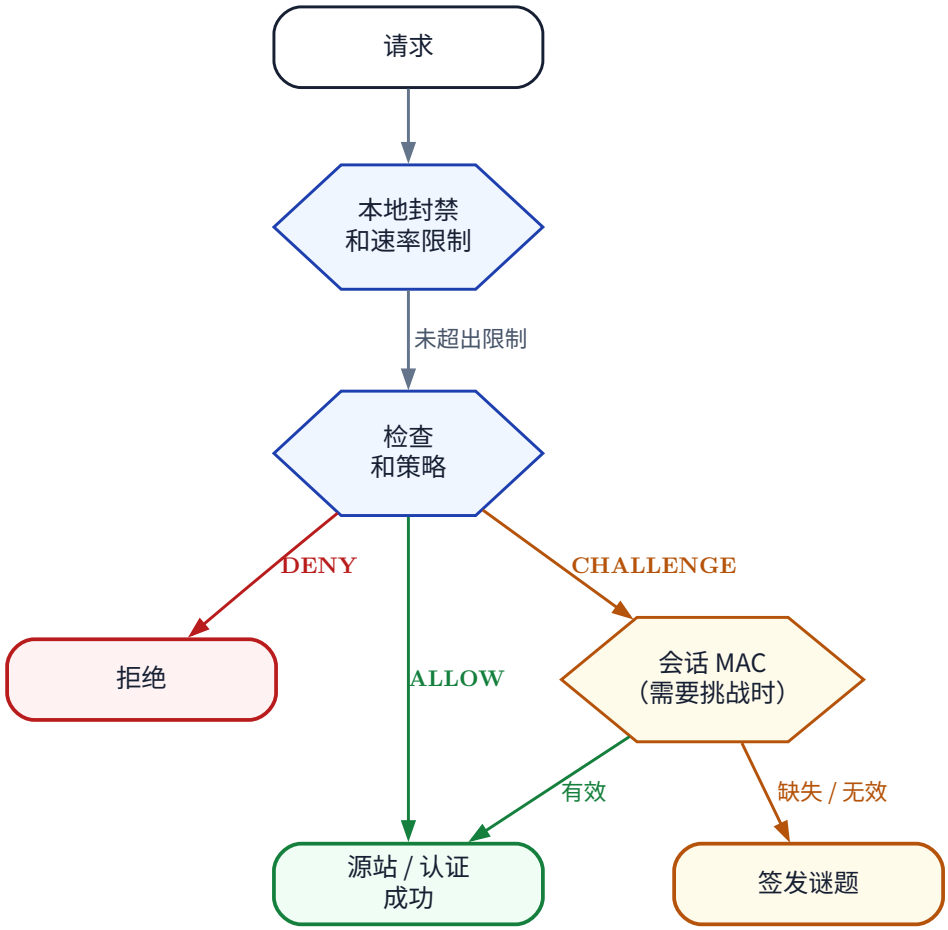


图 24 Shield 形态下的请求路由

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

被准入的请求带着 X-Forwarded-For、X-Forwarded-Proto、X-Real-IP、X-Sibuna-Status 和 X-Sibuna-Rule 头到达源站。逐跳头、Sibuna 自己写入的字段以及其他转发字段（Forwarded、X-Forwarded-Host、X-Forwarded-Port）都会被剥离，客户端用下划线拼写这些名称（X_Forwarded_For）时也是如此。X-Real-IP 是判定请求所依据的客户端。X-Forwarded-For 只是套接字对端，从不包含客户端写入的条目；信任转发身份时，它是收到的链后面再加上套接字对端。因此，源站必须从 X-Real-IP 或 X-Forwarded-

For 的右侧读取客户端，绝不能读取其最左边的条目。源站收到的路径是策略所判定的规范形式（SID 0009）；没有规范形式的路径会被回答 400。

HTTP/1.1 WebSocket 升级在源站的握手被接受之前，要先通过准入和策略检查。子协议和扩展的协商保持端到端。两个线程中继字节，每个方向一个，各自使用一个固定的缓冲区，包括预取的字节和半关闭；因此，一个升级后的连接占用两个线程，当所有连接都被升级时，连接线程的数量可以达到 --max-connections 的两倍。升级后的套接字仍然受普通连接配额和关闭登记表的约束，但使用 --websocket-idle-timeout（默认 300 秒），与 HTTP 空闲期限相互独立。任一方向的流量都会刷新这个上界，但如果一个对端在整个周期内不接收任何字节，无论另一个方向多么繁忙，中继都会结束。握手之后的帧不是 WAF 检查的输入。HTTPS/WSS 需要在 Sibuna 私有的 HTTP/1.1 监听地址之前放置一个终止 TLS 的入口代理；Sibuna 自己不终止浏览器的 TLS。

入口代理可以与浏览器协商 HTTP/2，并以 HTTP/1.1 转发给 Sibuna。Sibuna 对原生 HTTP/2 的支持推迟到以后的更新；这种安排不能为原生 gRPC 之类的应用提供端到端的 HTTP/2 语义。

带 Content-Length 的请求体以流方式通过固定缓冲区，字节和 MIME 头都保持不变。multipart 上传保留边界、重复的字段名、文件名和各部分的类型。WAF 最多检查请求体的前 8 KiB：multipart 元数据和非文件字段仍然作为文本检查的输入，而文件载荷是不透明的。filename 为空的部分是普通字段。属于已识别的二进制顶层 MIME 类型（SVG 以外的图像、音频、视频、PDF、ZIP、gzip、7z、protobuf 和 octet-stream）的请求体，只有当其前缀看起来也是二进制时才被视为不透明，即每十六个字节中至少有一个是制表符、换行、垂直制表、换页和回车以外的控制字节；否则按文本检查（SID 0004）。JSON、XML、SVG、URL 编码表单和未知类型保持文本检查。含糊或格式错误的 MIME 元数据会回退到文本检查。multipart 解析在同一前缀内限制为 32 个部分、每部分 2 KiB 的头部，并接受分隔符之后符合 RFC 2046 的传输填充。

这是上传兼容性，而不是文件验证或恶意软件扫描。后端必须自己强制执行可接受的媒体类型，而不是信任客户端声明的 Content-Type。检查前缀之后的字段（包括大文件上传之后的字段）不会被检查。压缩的载荷不会为检查而解压。上传仍然要经过准入、路径、查询和请求头检查，并受连接期限的约束。分块请求体在检查之前于连接缓冲区内解码（SID 0009）。在缓冲区内结束的请求体以 Content-Length 到达后端。更长的请求体由 Sibuna 重新分块，每次读取一个块，因此后端永远看不到客户端的块大小、扩展或尾部字段。包含单独的 CR 或 LF、大小两侧有空白、扩展格式错误或超过 4 KiB 的块行会以 400 被拒绝。超过 16 KiB 的尾部字段段也是如此。使用其他编码的 Transfer-Encoding 会收到 501。无法解析分块请求的后端，对于约 44 KiB 以下的请求体仍然会收到 Content-Length。承载过请求体的源站连接在交换结束后关闭，永远不会回到连接池。等待 100-continue 的客户端会在发送请求体之前从本地收到它；不支持的期望会收到 417。Expect 头在转发给后端之前被消费掉。

入口代理之后的 forward-auth

在 --mode forward_auth 下，守护进程对入口代理的子请求回答 200（外加审计头 X-Sibuna-Status、X-Sibuna-Rule 和 X-Sibuna-Rule-Hash），需要挑战时回答 401，拒绝时回答 403，被限速时回答 429。入口代理必须转发客户端地址和原始 URL；forward-auth 模式默认信任它们。请把这个监听地址绑定在私有网络上，使只有入口代理能够连接。X-Forwarded-Uri（Caddy）或 X-Original-URI（Nginx）以及 X-Forwarded-Method 为策略评估还原应用请求；原始路径会像请求行中的路径一样被转换为规范形式。重复或相互冲突的原始 URL 字段，以及不是 origin-form 的原始 URI，会以 400 被

拒绝。守护进程的内部路由始终使用实际的请求 URI。forward auth 不检查入口代理省略的请求体。

```
map $http_upgrade $sibuna_connection_upgrade {
    default upgrade;
    '' close;
}
server {
    listen 443 ssl;
    location / {
        auth_request /__sibuna_auth;
        auth_request_set $sibuna_auth_status $upstream_status;
        auth_request_set $sibuna_retry_after $upstream_http_retry_after;
        auth_request_set $sibuna_status $upstream_http_x_sibuna_status;
        auth_request_set $sibuna_rule $upstream_http_x_sibuna_rule;
        auth_request_set $sibuna_rule_hash $upstream_http_x_sibuna_rule_hash;
        error_page 401 = @sibuna_challenge;
        error_page 500 = @sibuna_auth_error;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $sibuna_connection_upgrade;
        proxy_set_header Host $host;
        proxy_set_header X-Sibuna-Status $sibuna_status;
        proxy_set_header X-Sibuna-Rule $sibuna_rule;
        proxy_set_header X-Sibuna-Rule-Hash $sibuna_rule_hash;
        proxy_read_timeout 300s;
        proxy_pass http://127.0.0.1:3000;
    }
    location = /__sibuna_auth {
        internal;
        proxy_pass http://127.0.0.1:8080/;
        proxy_pass_request_body off;
        proxy_set_header Content-Length "";
        proxy_set_header X-Original-URI $request_uri;
        proxy_set_header X-Forwarded-Method $request_method;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-Uri "";
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header User-Agent $http_user_agent;
        proxy_set_header Cookie $http_cookie;
    }
    location @sibuna_challenge {
        rewrite ^ /__sibuna/challenge break;
        proxy_pass http://127.0.0.1:8080;
        proxy_pass_request_body off;
        proxy_set_header Content-Length "";
        proxy_set_header X-Original-URI $request_uri;
        proxy_set_header X-Forwarded-Method $request_method;
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
    location @sibuna_auth_error {
        add_header Retry-After $sibuna_retry_after always;
        if ($sibuna_auth_status = 429) { return 429; }
        return 503;
    }
    location /__sibuna/ {
```

```
        proxy_pass http://127.0.0.1:8080;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
}
```

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

在两种配置方案中，`/__sibuna/*` 命名空间（验证中间页、挑战、验证、求解器资源）都必须直接到达守护进程。nginx 的错误页面会传递原始 URI 和方法，使验证中间页携带授权所决定的要求；没有它们时页面仍然可用，签发会回退为评估浏览器上报的 URL。Caddy 自己提供原始 URI 和方法的元数据。互斥的 `handle` 块确保挑战和验证路由不会进入 forward-auth 预检查。对于应用的 WebSocket，Nginx 需要上面所示的显式 Upgrade/Connection 头。Nginx 的错误处理器渲染内部挑战路由时，不会进行第二次授权决定，也不会消耗上传的请求体。它的 auth 模块只直接接受 2xx、401 和 403，因此示例把限速导致的认证错误转换回带 Retry-After 的 429；其他授权失败以 503 保持关闭。Nginx 提供它自己的 403 响应体，而 Caddy 转发 Sibuna 的拒绝页面。两种配置方案都会用实际的授权响应替换传入的 Sibuna 审计头。请在入口代理上配置常规的 TLS 证书以及应用的上传和期限限制；示例只展示路由逻辑。仓库目前还没有针对真实运行的入口代理测试这些配置方案；它们所依赖的转发元数据规则由 `sibuna-http` 中的单元测试覆盖。

来自入口代理的 JA4 指纹

JA4 指纹概括了客户端如何建立 TLS 连接。它根据客户端的第一条握手消息，即 ClientHello，计算得出。客户端的 TLS 库在发送任何 HTTP 字节之前就确定了这条消息，因此客户端无法在自己的 HTTP 请求内部改变它的指纹。浏览器、库和机器人的指纹各不相同。这使 JA4 成为比 `User-Agent` 值更强的信号，因为后者任何客户端都可以随意填写（SID 0014）。JA4 由 FoxIO 定义。一个指纹有 36 个字符，分为三部分，用 _ 连接：

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

a  协议（t 为 TCP，q 为 QUIC，d 为 DTLS），TLS 版本（13 即 TLS 1.3），
   带 SNI 为 d、不带为 i，密码套件数（两位数字），
   扩展数（两位数字），ALPN 代码（h2 即 HTTP/2）
b  排序后的密码套件的哈希，12 位小写十六进制数
```

```
c   排序后的扩展和签名算法的哈希，
    12 位小写十六进制数
```

只有终止 TLS 的组件才能看到 ClientHello，因此只有它能计算 JA4。Sibuna 不终止 TLS：它在入口代理之后提供普通的 HTTP/1.1 服务，由入口代理与浏览器使用 TLS、HTTP/2 和 HTTP/3 通信。已经有几种入口代理能够计算 JA4。因此，我们让入口代理计算指纹并通过一个请求头发送，Sibuna 从该请求头中读取它。

`--ja4-header <name>` 指定这个请求头。它默认关闭，也没有默认名称；我们总是写出入口代理所用的请求头名称。Sibuna 以小写形式保存该名称，匹配时不区分大小写。它拒绝自己写入或剥离的字段名（`X-Forwarded-*`、`X-Real-IP`、`X-Sibuna-*` 和 `Forwarded`，也包括用 `_` 代替 `-` 的写法）。设置该选项后，启动横幅会打印一行，例如 `JA4: from cf-ja4 (trusted)`。

只有信任转发字段时才会读取该请求头。这与决定是否信任 `X-Forwarded-For` 的条件相同：`--trust-forwarded`，或者没有 `--no-trust-forwarded` 的 forward-auth 模式。来自不受信任对端的该请求头会被忽略。在信任的情况下，当恰好存在一个该名称的字段且其值符合 JA4 格式时，请求才带有指纹。格式错误的值或重复的字段会使请求没有指纹。两个计数器观测这一点：`sibuna_ja4_present_total` 统计带有指纹的请求，`sibuna_ja4_invalid_total` 统计请求头格式错误或重复的请求。来自不受信任对端的请求不会增加任何一个计数器。

在反向代理模式下，只有当 Sibuna 信任该请求头时，源站才能看到它。不信任时，Sibuna 会从发往源站的请求中删除所有该名称的字段，客户端用 `_` 代替 `-` 拼写名称时也是如此。因此，客户端无法以该名称向源站递交伪造的指纹。信任时，该字段以其原有的拼写原样通过，而使用 `_` 的别名仍会被删除。没有 `--ja4-header` 时，该名称的字段只是一个普通字段。

> **入口代理必须掌控该请求头** 指纹的可信程度取决于写入它的入口代理。Sibuna 无法区分入口代理计算出的值和客户端发送的值。因此，在每一条到达 Sibuna 的路由上，入口代理都必须覆盖或删除客户端提供的该请求头的任何副本。与 `X-Forwarded-For` 一样，Sibuna 的监听端口必须只接受来自入口代理的连接。

Cloudflare。 托管转换 “Add bot protection headers” 会在发往源站的请求中加入 `cf-ja4`，与 `cf-bot-score`、`cf-verified-bot` 和 `cf-ja3-hash` 并列。它需要带有 Bot Management 的 Enterprise 套餐。Cloudflare 与 Sibuna 之间的入口代理必须原样传递该字段。我们这样启动 Sibuna：

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

Sibuna 前面的入口代理必须只接受来自 Cloudflare 地址的连接；否则客户端可以绕过 Cloudflare，发送它自己的 `cf-ja4`。

nginx。 FoxIO 的 `ja4-nginx-module` 在 nginx 内部计算 JA4，并将其作为 `$http_ssl_ja4` 提供。它需要 nginx 1.26 或更高版本，并打上该模块的 ClientHello 捕获补丁，使用 `--with-http_ssl_module` 构建。`proxy_set_header` 会替换客户端发来的同名字段，因此每个到达 Sibuna 的 location 都自己设置该请求头：

```
# 反向代理：Sibuna 以 --ja4-header x-ja4 --trust-forwarded 运行
location / {
    proxy_set_header X-JA4 $http_ssl_ja4;
```

```
    proxy_set_header X-Forwarded-For $remote_addr;
    proxy_pass http://127.0.0.1:8080;
}

# Forward auth: Sibuna 以 --mode forward_auth --ja4-header x-ja4 运行
location = /__sibuna_auth {
    internal;
    proxy_pass http://127.0.0.1:8080/;
    proxy_set_header X-JA4 $http_ssl_ja4;
    # ... 上文 forward auth 配置中的其余字段
}
```

在 forward-auth 配置方案中，同样的 `proxy_set_header X-JA4 $http_ssl_ja4;` 这一行也应放在 `@sibuna_challenge` 和 `location /__sibuna/` 中。省略它的 location 会把客户端自己的 `X-JA4` 传给 Sibuna，而 Sibuna 会信任它。

Apache Traffic Server。 `ja4_fingerprint` 插件通过 `plugin.config` 中的 `ja4_fingerprint.so` 这一行启用。它没有选项。它会在每个 TLS 连接的请求中加入一个带有指纹的 `ja4` 字段，以及一个带有代理名称的 `x-ja4-via` 字段。我们用 `--ja4-header ja4 --trust-forwarded` 启动 Sibuna。该插件的文档没有说明它是否会替换客户端发送的 `ja4` 字段。在信任该值之前，我们必须针对自己使用的版本确认这一点，或者在插件加入自己的字段之前删除客户端的字段。

规则用 `ja4` 字段匹配指纹（第 X 部分“策略模式”）。该值要么是一个完整的指纹，此时必须与之相等；要么是一个后跟 `*` 的前缀，此时指纹必须以它开头。单独的 `"*"` 匹配任何指纹。没有指纹的请求永远不会匹配 `ja4` 规则。加载器会以 `INVALID POLICY JA4` 拒绝任何指纹都无法匹配的模式，例如大写的 `T13D*`。匹配只比较字节，不分配任何内存。

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

第一条规则精确拒绝一个指纹；该值只是一个示例，并不是某个已知的扫描器。接下来的两条规则不直接作出决定，而是为请求的分数增加权重：一个没有发送服务器名称的 TLS 1.3 客户端，以及一个通过 `User-Agent` 识别出的无头浏览器。如果后面没有规则先作出决定，同时表现出这两个信号的请求得分为 45，达到默认的拒绝阈值 40，而只表现出其中一个信号的请求会被挑战。最后一条规则按前缀挑战在结账路径上通过 TCP 协商 TLS 1.2 的每个客户端。它选择挑战而不是拒绝，因为许多合法的旧客户端仍在使用 TLS 1.2。规则应该像这样把 JA4 与其他信号结合起来。JA4 值指明的是一个 TLS 库及其配置，而不是一个人或一台设备。许多客户端共用同一个库，而专门的库可以模仿浏览器的 ClientHello。JA4 提高了自动化的成本；它不能证明身份。出于同样的原因，Sibuna 的封禁和信誉始终以地址为键，从不以指纹为键。

事件（WAF 拒绝、蜜罐命中和审计检出结果）在其请求带有指纹时会记录该指纹，保存在 `security_incidents.ja4` 列中。控制台在事件证据中以“JA4 fingerprint”显示它，内容为该值或“Not recorded”。事件页面的筛选器接受指纹或前缀。托管规则编辑器有一个 JA4 字段，保存在 `policies.ja4_pattern` 中；策略测试表单也接受指纹。

声明式策略

> **学习目标** 编写一个包含规则、权重、阈值、IP 规则和 WAF 开关的策略文件。

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

加载器采用失败即关闭（fail closed）的策略：一个未知的键、一个拼错的动作、一个格式错误的地址或一个超出范围的数字都会使整个文件被拒绝，诊断信息会指出规则和字段，守护进程不会启动。带有未知选项或超出范围的值的命令行也以同样方式被拒绝。存在 `rules` 时，它替换内置的规则表；`ip_rules` 为信誉字典树提供数据，可以扩展到数千个前缀；`waf: false` 从文件中选择 Gate 形态。模式中还有两个键（第 X 部分）：`inspection` 把每个 WAF 类别设置为 `disabled`、`audit` 或 `enforce`，规则的 `limits` 对象为它设定自己的配额。规则的 `ja4` 字段匹配受信任的入口代理转发的 JA4 指纹，可以精确匹配，也可以按后跟 `*` 的前缀匹配；它需要 `--ja4-header`（见“来自入口代理的 JA4 指纹”）。路径模式匹配规范形式的请求路径（SID 0003），因此模式必须以这种形式书写。

持久存储与集群

> **学习目标** 启用 SQLite 或 PostgreSQL 存储，用 SQL 添加一条动态策略和一次封禁，查询事件取证数据，理解攻击活动聚类，并在同一个 PostgreSQL 数据库上运行多个节点。

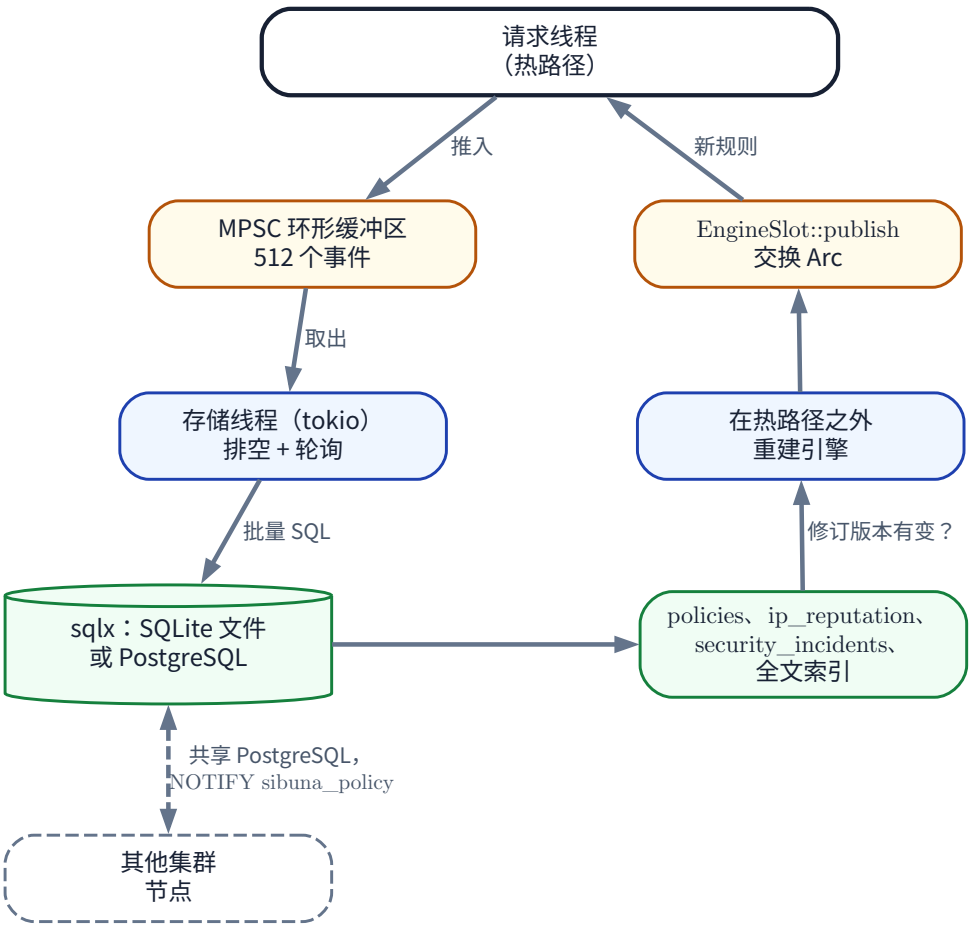


图 25 存储架构：worker 从不阻塞在数据库上

`--data-dir /var/lib/sibuna` 打开 SQLite 文件 `/var/lib/sibuna/sibuna.db`（WAL 模式，`synchronous=NORMAL`，设置忙等待超时），并启动存储线程。`--database-url`（或 `--database-url-file`）则改为连接一个 PostgreSQL 数据库。存储线程拥有一个单线程的 tokio 运行时和一个小型连接池；它是唯一执行 SQL 的地方。在启动时、每当策略修订版本变化时（对 `policies`、`ip_reputation` 或 `policy_inspection` 的写入），以及信誉条目过期时，该线程都会从策略文件和数据库重建引擎并发布它；请求从不等待 SQL。一次重建会在加载前后各读取一次修订版本，如果期间有写者提交，就丢弃这次的候选。

模式

表 25 核心表

表	用途与键
`policy_revision`	策略修订版本，一行；三张策略表上的触发器会推进它。
`policies`	有序的动态规则，以 `id` 为键。可为空的 `ja4_pattern` 保存规则的 JA4 条件，最多 37 字节。

表	用途与键
policy_inspection	每个类别存储的 WAF 模式，一行。
ip_reputation	分数、命中次数和过期时间，以地址或前缀为键。
incident_receipts	每个节点的持久事件回执（其最后一个已提交批次之后的下一个序号），以节点为键。
security_incidents	一条有界的事件记录及其 64 维嵌入向量（256 字节），以 node_id << 40 | sequence 为键。可为空的 ja4 保存请求的 JA4 指纹，恰好 36 字节。
incidents_fts	在 SQLite 上：一张基于路径和载荷规范词元的 FTS5 表。在 PostgreSQL 上，同样的词元存放在 security_incidents 的 tsvector 列 search 中，并带有 GIN 索引。
sibuna_schema	每次已应用的迁移，以组件和版本为键。

核心模式位于 crates/storage/src/schema.rs，每种方言各写一份。启用控制台时，组件 console 会增加 33 张名称以 console_ 开头的表（SID 0013；crates/console-storage/src/schema/）。迁移按组件编号并编译进可执行文件。核心迁移 2 增加这两个 JA4 列；在它之前创建的数据库会在启动时迁移。所有待执行的迁移在一个事务中运行，如果数据库中某个组件的版本比节点所知道的更新，节点会拒绝该数据库。打印出来的简略模式不是升级脚本。SQLite 的表是 STRICT 的，CHECK 约束与 Rust 类型相对应，因此数据库会拒绝守护进程无法表示的行。枚举值以小写存储：直接写入 'DENY' 会被拒绝。

日常运维

在 sibuna.db 上使用 sqlite3 shell（WAL 模式允许在守护进程运行时读取），或在集群数据库上使用 psql：

```
-- 添加一条动态规则；每个节点在下一个节拍重建引擎。
INSERT INTO policies (id, name, priority, path_pattern, action, difficulty, algorithm,
  header_matchers, cidr_matchers, weight, enabled, created_at, updated_at)
VALUES ('p-checkout', 'protect-checkout', 10, '/checkout/*', 'challenge', 20, 'posw',
  '{"X-Api": "v2"}', '["10.0.0.0/8"]', 0, 1, unixepoch(), unixepoch());

-- 在整个集群内封禁一个地址一天。
INSERT INTO ip_reputation (ip_or_cidr, reputation_score, banned_until,
  trigger_rule, hits, last_seen)
VALUES ('198.51.100.7', -100, unixepoch() + 86400, 'analyst', 1, unixepoch());

-- 取证：对已记录的载荷做全文搜索。
SELECT s.id, s.client_ip, s.violation_category, s.path, s.campaign_id
FROM incidents_fts f JOIN security_incidents s ON s.id = f.rowid
WHERE incidents_fts MATCH 'union' ORDER BY rank LIMIT 20;
```

这些语句是为 SQLite 编写的。在 PostgreSQL 上，enabled 为 TRUE，时间用 CAST(extract(epoch FROM now()) AS BIGINT) 代替 unixepoch()，搜索则在 security_incidents 上写作 WHERE search @@ CAST('union' AS tsquery)。path 和 offending_payload 保存原始请求字节（BLOB 或 BYTEA）。违反约束的直接写入会被数

据库拒绝，永远不会到达节点。在 PostgreSQL 下，修订版本触发器还会发出 `NOTIFY sibuna_policy`，使其他节点立即执行一次节拍；轮询仍然是最终保证。

分数不高于 −50 的在字典树中成为 `deny` 前缀，分数不低于 50 的成为 `allow`，`banned_until` 限定封禁的期限。蜜罐命中会自动插入一条 −100 的记录，因此在共享 PostgreSQL 数据库时，一个节点上触发的陷阱会在所有节点上封禁该地址。

事务回执使重试变得安全

存储线程最多收集 32 条待处理记录，并构建一个 SQL 事务。该事务插入事件、更新全文索引，并应用蜜罐带来的信誉变化。每个签发者（节点）都有一个单调递增的回执，即它在 `incident_receipts` 中的那一行。每条修改数据的语句都受该回执的保护；事务最后才推进回执，并且从不回退。

> **实例分析：确认消失了** 假设签发者 2 准备了序号 101 到 132，最终回执为 133。数据库提交了所有记录和回执 133，然后连接在确认到达之前失败。存储线程保留准备好的批次（标识符、嵌入向量和攻击活动）并重试。回执已经等于 133，因此受保护的写入什么都不做：事件和蜜罐命中次数都不会加倍。如果原来的事务回滚了，旧的回执仍然存在，重试会把所有写入施加一次。

提交失败时，待处理批次保留在内存中，并递增 `incident_write_failures`。节拍仍然会尝试轮询策略。512 条记录的队列满时计入 `incidents_dropped`；它从不为等待磁盘而阻塞 HTTP 响应。待处理批次额外提供 32 条记录的空间。进程在提交之前死亡可能丢失排队的数据：这是一条有界的异步取证路径，而不是在入队时就保证持久的消息队列。

练习 9.1. 把回执更新移到数据事务之外。构造一个使信誉效果重复的崩溃时序，再构造一个使事件丢失的崩溃时序。

攻击活动聚类

每个事件的载荷都通过对字节三元组（数字折叠、大小写折叠）使用哈希技巧，嵌入为一个 64 维单位向量，并以 256 字节存储在 `security_incidents.embedding` 中。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），并在每次节拍时从数据库刷新。插入之前，它在其中找出最近的一个（包括同一批次中较早的记录）；如果余弦距离不超过 0.35，新事件就加入该事件的 `campaign_id`，否则以自己的标识符为名开始一个新的攻击活动。早于这个窗口的事件不参与比较。相似性只是一种启发式：回归示例会把相关的 SQL 载荷分到一组，但这并不保证每一对攻击都属于同一个攻击活动。这里没有运行任何模型，也没有训练任何东西。

集群

集群是共享同一个 PostgreSQL 数据库的一组节点。使用 `postgres-tls` 特性构建（`cargo build --release -p sibuna --features postgres-tls`；它隐含 `postgres`），并用数据库和各自的节点 ID 启动每个成员：

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

```
    --secret-file /etc/sibuna/secret
```

每个成员都必须使用相同的主密钥，以及一个唯一的、取值 1 到 8,388,607 的 `--cluster-node`。URL 文件保存一个 `postgres://` URL，并且必须只有其所有者可读；启动横幅从不打印 URL。写入发往 PostgreSQL 主库。每个节点的存储线程看到已提交的修订版本变化（通过 `NOTIFY` 或轮询）后重建自己的引擎。

节点通过网络访问共享数据库，因此我们对这条连接进行加密和验证。我们为服务器颁发一张由自己的证书颁发机构签名的证书，并在 URL 中指定该机构：`postgres://sibuna:...@db.internal:5432/sibuna?sslmode=verify-full&sslrootcert=/etc/sibuna/db-ca.pem`。在服务器上，我们只通过 `pg_hba.conf` 中的 `hostssl` 行接受 TCP 连接。

启动时，守护进程会打印一行关于数据库连接的信息。使用 `verify-full` 时，它说明证书和主机名都已验证。使用任何更弱的模式时，它会打印一条警告和一条提示。没有 `sslmode` 时，模式为 `prefer`：只有当服务器提供 TLS 时连接才会加密，并且不验证服务器。不带 `postgres-tls` 构建的守护进程会拒绝 `require`、`verify-ca` 和 `verify-full`，并指出缺少的特性；它从不回退为明文。被拒绝的证书会立即终止启动；提示会要求我们检查 `sslrootcert`，以及 URL 中的主机是否是服务器证书中的名称。`sslrootcert` 把我们的证书颁发机构添加到系统信任存储中；它不会移除系统的证书颁发机构（SID 0011，SID 0012）。

Sibuna 不复制自己的数据库，也不选举领导者。复制、故障切换和备份都由 PostgreSQL 负责（流复制、Patroni 或托管服务）。控制台的节点视图中没有领导者：SQLite 节点显示角色 `single`，PostgreSQL 节点显示 `unknown`，`quorum` 只表示该节点最近一次数据库往返成功了。如果数据库不可达，最后发布的引擎继续提供服务，事件在有界环中排队，环满后被丢弃并计数，存储线程以退避方式重试；节点在重新连接之前保持它最后的策略。Zaxonlite 的数据目录（SID 0005）不会被读取；策略通过 `sibuna console policies export` 和 `sibuna console policies import` 迁移，取证历史不会迁移。

集群中的挑战路由

挑战密钥与 `--cluster-node` 绑定；请让挑战的签发和验证发生在同一个成员上。令牌使用共享种子，在其他成员上也有效。本地速率配额不会变成全局配额，进程重启会清空已用集合。在一个本地 PostgreSQL 上运行三个节点时，直接写入的策略行会在一个节拍内到达全部三个节点，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（`crates/storage/tests/multi_node.rs`，SID 0011 标准 2；设置 `SIBUNA_TEST_DATABASE_URL` 时运行）。

打包

> **学习目标** 安装并校验发行版软件包，把 Sibuna 作为服务运行、在容器中运行或在 Kubernetes 下运行，并自己构建可执行文件。

GitHub 上的每个发行版都为每个平台提供相同的可执行文件，它们从发行标签以默认特性构建：SQLite 存储和控制台。PostgreSQL 集群需要我们自己构建（见下文）。

平台	发行版文件
Linux x86-64, ARM64	`sibuna-linux-amd64.tar.gz` 和 `sibuna-linux-arm64.tar.gz` 包含一个静态 musl 可执行文件，可在任何 Linux 发行版上运行。`sibuna_<version>-1_amd64.deb`、`sibuna_<version>-1_arm64.deb`、`sibuna-<version>-1.x86_64.rpm` 和 `sibuna-<version>-1.aarch64.rpm` 会连同一个 systemd 服务一起安装它；`sibuna-bin-<version>-1-x86_64.pkg.tar.zst` 在 Arch Linux 上做同样的事。
macOS arm64, x86-64	`sibuna-macos-arm64.tar.gz`、`sibuna-macos-amd64.tar.gz`，适用于 macOS 12 或更高版本。
Windows x86-64	`sibuna-windows-amd64.zip` 包含 `sibuna.exe`，适用于 Windows 10 或 Server 2016 及更高版本。
FreeBSD 15.1, OpenBSD 7.9 (amd64)	`sibuna-<version>-freebsd-15.1-amd64.pkg` 和 `sibuna-<version>.tgz`。
Homebrew、Kubernetes、容器	`sibuna.rb`、`helm-sibuna-<version>.tgz`，以及适用于 amd64 和 arm64 的镜像 `ghcr.io/insanai/sibuna:<version>`，其摘要记录在 `IMAGE-DIGEST.txt` 中。
源码	`sibuna-<version>-source.tar.gz`。

每个压缩包和软件包都包含 `LICENSE`、`NOTICE`、`LICENSES/`（其中有 `THIRD-PARTY-RUST.txt`，即可执行文件中的 crate）、`README.md`、`SECURITY.md` 和 `sibuna.build.json`。这份构建清单记录版本、提交、Rust 目标、编译器版本、特性，以及可执行文件和它的两个 WebAssembly 模块的 SHA-256。发布工作流在发布任何内容之前，会在每个可执行文件自己的平台上启动它，并在 Debian、Fedora、Arch Linux、FreeBSD 和 OpenBSD 上安装、重新安装和移除这些软件包。

这些文件没有签名。安装之前，我们用同一发行版中的 `SHA256SUMS` 校验下载的文件：

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

在 macOS 上，通过浏览器下载的文件带有隔离属性，它会阻止未签名的可执行文件运行；校验之后，`xattr -d com.apple.quarantine sibuna` 可以移除该属性。

systemd 服务

Debian、RPM 和 Arch 软件包会安装 `/usr/bin/sibuna`、单元 `sibuna.service`、账户 `sibuna`（通过 `systemd-sysusers` 创建）、辅助脚本 `/usr/lib/sibuna/seed.sh` 和设置文件 `/etc/sibuna/service.env`。安装不会启动任何东西。我们在设置文件中设置源站（一个 IP 地址字面量），然后启动服务：

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

除非 SIBUNA_HOST 和 SIBUNA_PORT 另有设置，服务监听 127.0.0.1:8080。SIBUNA_OPTIONS 传递更多选项，以空格分隔，例如 --mode forward_auth 或 --console 127.0.0.1:19446。每次启动之前，如果准入种子 /var/lib/sibuna/admission.seed 不存在，seed.sh 就会创建它（64 个十六进制字符，权限 0600，属主为 sibuna）；它从不替换已有的种子，因为新种子会终结所有会话。守护进程把它的 SQLite 数据库保存在 /var/lib/sibuna 中，这是一个权限为 0700 的目录。该单元运行时不带任何 capabilities，系统只读（ProtectSystem=strict），/tmp 和设备是私有的，并且不能获得新权限。
升级会保留设置文件、种子和状态，并且不会重启服务；我们在备份之后再重启它。卸载会停止并禁用服务，但保留账户、种子、设置和状态；我们在让主机退役时自己删除它们。
FreeBSD 和 OpenBSD 软件包只安装 /usr/local/bin/sibuna 及其文档（使用 pkg add；对于不是来自 OpenBSD 镜像站点的软件包，使用 pkg_add -D unsigned）。它们不创建服务、账户或状态：我们以一个非特权用户运行守护进程，并为它配置私有的种子和数据目录。

Homebrew

sibuna.rb 安装适合本机的发行版压缩包：macOS 或 Linux，arm64 或 x86-64。它不从源码构建，因为 Homebrew 的 Rust 没有 wasm32-unknown-unknown 的标准库，而构建两个嵌入的 WebAssembly 模块需要它；压缩包中是发布工作流已经检查过的可执行文件。Homebrew 只从 tap 安装 formula，因此我们把这个 formula 放进一个本地 tap：

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

这个 formula 不启动任何服务。它的 caveats 说明了如何创建种子；它的测试在同一个数据目录上两次启动守护进程。

容器和 Kubernetes

镜像 ghcr.io/insanai/sibuna:<version> 在一个 distroless 基础镜像上包含这个静态可执行文件：CA 证书、用户 65532，没有 shell，也没有包管理器。入口点是 sibuna，因此参数就是守护进程的选项。发布工作流以只读方式运行它，不带任何 capabilities，以只读方式挂载种子，并为状态挂载一个卷，然后把它重启一次：

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

Helm chart helm-sibuna-<version>.tgz 运行该镜像的一个 Pod。它需要一个由我们创建的 Secret（secret.existingSecret，键为 admission.seed），因此种子从不出现在 values 或模板中。Pod 以 UID 65532 运行，根文件系统只读，符合 Restricted Pod 安全标准。单个副本加上 Recreate 升级方式，使 ReadWriteOnce 卷上始终只有一个写入者，Sibuna 把它的 SQLite 状态保存在该卷的 data/ 中；卸载时会保留卷声明（claim）。探针调用 /__sibuna/health。Service 的类型是 ClusterIP，chart 不创建 ingress，也不创建控制台，它的 NetworkPolicy 会拒绝所有流量，直到 networkPolicy.ingressFrom 和 networkPolicy.egressTo 放行客户端和源站。upstream.host 必须是一

个 IP 地址字面量，extraArgs 传递更多守护进程选项。工作流在一个强制执行 Restricted 标准的 kind 集群中安装、升级和卸载该 chart。

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

从源码构建

我们使用 Rust 1.100 构建和测试；最低支持版本仍为 Rust 1.94。
我们使用 `rustup toolchain install 1.100.0 --profile minimal --target wasm32-unknown-unknown` 安装发行工具链，并在以下命令中使用 `cargo +1.100.0` 选择它。

- `cargo build --release -p sibuna` 以默认特性（SQLite 存储和控制台）生成 `target/release/sibuna`。sqlx 自带 SQLite，ring（rustls）在构建时需要 C 编译器；可执行文件动态链接系统 C 运行时。构建脚本还会构建 WebAssembly 求解器和控制台界面，这需要 `rustup target add wasm32-unknown-unknown`；`SIBUNA_POW_WASM=<path>` 和 `SIBUNA_CONSOLE_WASM=<path>` 可以改为嵌入预先构建的模块。
- `cargo build --release -p sibuna --no-default-features` 生成完全不依赖外部 crate 的内存型防火墙；此时 `--data-dir` 和 `--database-url` 会在启动时以 `StorageDisabled` 被拒绝，`sibuna console` 则以 `CONSOLEBUILD` 被拒绝。
- `cargo build --release -p sibuna --features postgres-tls` 增加 PostgreSQL 集群支持，并对数据库连接使用经过验证的 TLS。它不需要 OpenSSL。
- 发行版中的静态 Linux 可执行文件来自 `cargo build --release --locked -p sibuna --target x86_64-unknown-linux-musl`（或 `aarch64-unknown-linux-musl`），并以 `musl-gcc` 作为 C 编译器（`CC_x86_64_unknown_linux_musl=musl-gcc`）。

release profile 使用 fat LTO 和单个代码生成单元。工作流只构建一次这两个 WebAssembly 模块，并把它们传给每个平台，因此同一发行版的所有可执行文件都嵌入逐字节相同的模块。依赖变化之后，`cargo xtask licenses` 会重新生成 `LICENSES/THIRD-PARTY-RUST.txt`。
在 Windows 上，守护进程在收到 Ctrl+C、Ctrl+Break 和控制台窗口关闭事件时停止；它不作为 Windows 服务运行。Windows 文件没有权限位，因此 Sibuna 无法拒绝其他人可能读取的密钥文件、凭据文件或数据库 URL 文件：我们用 ACL 限制这些文件的访问。控制台的内存和 CPU 仪表读取自 `/proc`，在其他系统上不可用。

练习 9.2. 编写策略文件和所需的 `ip_reputation` 行，使预发布网络（`10.20.0.0/16`）绕过挑战，`/admin/*` 要求 20 个工作比特的顺序工作量，并以每 10 秒 30 个请求的速率准入一个由 `X-Partner-Key` 标识的合作伙伴爬虫。
提示：守护进程的选项设定了每个请求首先要满足的按客户端限制；规则可以在 `limits` 中携带自己的配额。

解释这个不变量. 用存储线程和引擎槽来向运维人员解释：为什么向 `policies` 添加一行无需重启就能生效，而且从不需要任何请求等待。

第 X 部分

案头参考与诊断

端点、指标、错误目录、策略模式和存储表，集中在一处。

端点

学习目标 了解 /__sibuna/ 下的每个路由及其返回内容。

表 27 保留路由

端点	方法	功能
/__sibuna/challenge	GET	验证中间页
/__sibuna/challenge.json?need=&path=	GET	签发一个无状态挑战：若被挑战的响应中带有要求票据，则由票据决定；否则按规范形式评估上报的 URL（超过 8 KiB 返回 414，路径没有规范形式时返回 400）。返回 {id, algorithm, difficulty, challenges, expires_at}
/__sibuna/verify	POST	接受 {"challenge_id", "nonce"} 或 {"challenge_id", "proof"}；返回带 Set-Cookie 的 200、带诊断信息的 400，或在请求体大于连接缓冲区时返回 413
/__sibuna/wasm/sibuna-pow.wasm	GET	求解器模块，14,744 字节（benchmarks/results/primitives-latest.json 中的 wasm_bytes），可缓存一小时
/__sibuna/worker.js	GET	包含 WASM 和 JavaScript 证明器的 Web Worker，可缓存一小时
/__sibuna/honeypot	GET	将调用方封禁 --ban-seconds 并记录一条事件；返回 403。只有顶层请求才会触发封禁（没有 Sec-Fetch-Dest 或其值为 document，且 Sec-Fetch-Site 不是跨站）；其他请求返回 404 并计数
/__sibuna/health	GET	JSON 格式的存活状态，包含引擎名称、版本、代理模式和证明算法
/__sibuna/metrics	GET	Prometheus 文本格式（版本 0.0.4）

/__sibuna/ 下的任何其他路径都返回 404。签发和验证消耗单独的 --challenge-rate-limit 预算，被封禁的地址在每个路由上都以 403 被拒绝。挑战响应带有 X-Sibuna-Status: CHALLENGE；被准入的请

求向上游携带 X-Sibuna-Status: PASS 和 X-Sibuna-Rule（在 forward-auth 回复中还有十六进制的 X-Sibuna-Rule-Hash）。不接受 HTML 的客户端收到 401 和 {"error":"challenge_required","challenge":"/__sibuna/challenge.json?need=<ticket>"}。
单独的、需显式启用的管理监听器提供 /console/。先用 sibuna init-admin 在本地初始化管理员，然后在首次登录时更换临时密码。/console/api/stats 和 /console/ws 需要经过认证的会话（包括大屏会话）。WebSocket 以有界的快照、增量和缺口恢复，多路传输统计、事件、节点、策略、挑战和审计数据。管理对等节点连接到 /console/peer，每个 API 路由都位于 /console/api/ 之下。forward-auth 模式下不观测源站错误计数。第 IX 部分记录了控制台的工作流程、HTTPS 部署和 SID 0007 当前的限制。

指标

计数器以 Prometheus 文本格式暴露为 sibuna_<name>_total，顺序如下：

表 28 计数器

计数器	递增时机
incidents_persisted, incident_batches	存储线程已确认提交的记录和事务
incidents_dropped, incident_write_failures	因环已满而被拒绝的入队操作，以及失败的提交尝试
requests	一个解析后的请求到达分派阶段（包括保留路由）
allowed, denied, challenged	作出了一个策略决定（准入、拒绝、签发或重新签发挑战）
challenges_issued	/__sibuna/challenge.json 签发了一条挑战记录
solutions_accepted, solutions_rejected	/__sibuna/verify 接受或拒绝了一个证明
rate_limited	GCRA 拒绝了一个请求（429）：请求限制、挑战预算或规则配额
banned	一个被封禁的地址被拒绝，或蜜罐封禁了一个地址
bans_issued	记录了一次封禁（蜜罐或规则配额触发的封禁）
proxied, upstream_errors	一个请求被中继到源站，或源站失败（502）
parse_errors	拒绝了格式错误的头部、重复的 Host、没有规范形式的目标（400），或不支持的传输编码（501）
overloaded	超出 --max-connections 或 --max-connections-per-ip 的连接被回答 503
bans_refused	一次封禁未被记录，因为其键可能占用的每个槽都持有一个仍然有效的封禁
forwarded_invalid	一个受信任的转发客户端地址无法解析；该请求按套接字对端判定

计数器	递增时机
honeypot_refused	一次蜜罐命中没有封禁任何人，因为请求它的是页面而不是人
rate_evictions	一次限制器到达逐出了另一个活跃客户端的单元
ja4_present	在设置 --ja4-header 并信任转发的情况下，请求恰好带有一个该名称的请求头，其中包含一个 JA4 指纹
ja4_invalid	在设置 --ja4-header 并信任转发的情况下，该请求头格式错误或重复；该请求在没有指纹的情况下被判定

重试可能递增 incident_write_failures 而不丢失记录，因为待处理批次会被保留。应监控一段时间内的增量；仅凭总数并不能反映队列深度。

状态码

表 29 数据平面的状态码

状态码	时机
200	已准入（forward-auth）、验证中间页（需要挑战的 HTML 导航）、内部路由
302	当前协议不使用；验证返回 200，页面重新加载
400	格式错误的请求、重复的 Host 字段、没有规范形式的目标、无效的转发元数据、格式错误的请求体，或带有 Elm 风格诊断信息的被拒解答
401	对不接受 HTML 的客户端或在 forward-auth 模式下要求挑战
403	策略或 WAF 拒绝、被封禁的地址、蜜罐
404	/__sibuna/ 下的未知路径，或不是顶层请求的蜜罐命中
413	解答的请求体大于 64 KB 的连接缓冲区
414	上报的挑战 URL 超过 8 KiB
417	不支持的请求期望；100-continue 在本地处理
429	超出 GCRA 限制；Retry-After 以秒为单位
431	请求头部超过 16 KB
501	Transfer-Encoding 使用了 chunked 以外的编码
502	源站不可达，或其响应头部格式错误或大于 16 KB
503	达到连接限制（--max-connections 或 --max-connections-per-ip），或节点正在排空；回复后关闭套接字

错误目录

对于守护进程不认识的任何选项、超出文档所述范围的任何值，以及无效、缺失或重复的模式选择，`INVALID COMMAND LINE` 会终止启动。错误块会指出选项、给定的值、期望的范围和错误（`UnknownOption`、`MissingValue`、`InvalidValue`、`InvalidMode`、`MissingMode`、`DuplicateMode`；Zaxonlite 集群选项对应 `RemovedOption`，两个互斥的存储选项对应 `ConflictingOptions`，没有 `--cluster-node` 的数据库 URL 对应 `MissingOption`），并以一条提示结尾。数据库 URL 显示为 `[redacted]`。请只提供一次 `--mode reverse_proxy` 或 `--mode forward_auth`；`-m` 是等价的短选项。被拒绝的策略文件会打印一个以其错误命名标题的错误块（例如 `INVALID POLICY ACTION`），其中包含文件、错误名称、规则、字段和值。任何指纹都无法匹配的 `ja4` 模式会打印 `INVALID POLICY JA4`（`InvalidJa4`）；带有这种模式的控制台管理文档会以同一个错误名称被拒绝。控制台选项单独检查：`CONSOLE002` 指出格式错误或重复的控制台选项，`CONSOLE001` 指出违反规则的控制台配置（没有存储时的 `StorageRequired`）。

实现：`ErrorKind::explain` 把每个领域错误映射为一条标题分隔线、一段说明和一条 `Hint:`。位于 `crates/core/src/errors.rs`（crate `sibuna-core`）。

表 30 挑战与会话错误

错误	原因	提示
`MalformedChallenge`	标识符不是格式正确的挑战记录	获取一个新挑战，并原样提交
`InvalidChallengeTag`	标签认证失败；不是由本集群签发，或已被编辑	挑战无法伪造；请重新请求挑战
`ChallengeExpired`	超过挑战 TTL，或签发时间在未来	请求一个新挑战
`FingerprintMismatch`	从不同的地址或 User-Agent 提交	从获取挑战的那个客户端提交
`DifficultyNotMet`	Hashcash nonce 缺少所需的零比特	继续搜索 nonce
`InvalidProof`	顺序工作量证明无法揭示已承诺的标签	按签发的深度和揭示数完整运行证明器
`WrongSolutionType`	为 PoSW 挑战提交了 nonce，或为 Hashcash 提交了证明	让解答字段与算法相符
`DoubleSpendAttempt`	挑战已被使用	挑战只能使用一次
`StoreFull`	已用集合已满	降低挑战 TTL，或调大 `--spent-tags`
`InvalidTokenSignature`	Cookie 的标签或签名校验失败	通过验证中间页重新认证
`TokenExpired`	Cookie 已过期	重新认证

错误	原因	提示
TokenBoundAddressMismatch	Cookie 由不同的客户端身份出示	Cookie 不能共享

控制台 API 以 {"error":"<CODE>","hint":"<sentence>"} 报告失败；其中的 hint 就是界面显示的文字（crates/console/src/http/reply.rs）。

表 31 与运维人员相关的控制台失败代码

状态码	状态码	含义
CONSOLE002	400	请求格式错误、过大，或其形式不被允许
CONSOLE003	503 / 429	存储任务队列已满（32 个任务），或密码验证器繁忙
CONSOLEQUORUM	503	数据库未在请求的 10 秒等待时间内应答；结果未知，数据平面继续使用最后应用的策略
CONSOLE004	503	服务无法应答：此构建中不可用、正在停止，或发生内部故障
CONSOLENODE	400 / 409 / 503	节点命令被拒绝：它指向另一个节点（400），启动或控制修订版本已过时（409），或者它在另一条命令运行时到达，或某次完成无法被记录（503）；重试之前请检查操作回执
CONSOLEMUTATION	429	一个会话在一分钟内发起了超过六十次管理变更

策略模式

```
{
  "default_action": "ALLOW" | "DENY" | "CHALLENGE",
  "waf": true | false,
  "inspection": { "path_traversal" | "sqli" | "xss" | "rce": "disabled" | "audit" | "enforce" },
  "thresholds": { "challenge_at": int, "deny_at": int, "bits_step": int },
  "ip_rules": { "<cidr>": "ALLOW" | "DENY" | "CHALLENGE", ... },
  "rules": [                                                  // 最多 128 条
    {
      "name": "<kebab-case>",                                 // 1 到 128 个可打印字节
      "path" | "path_regex": "<pattern>",                     // 最多 512 字节
      "user_agent" | "user_agent_regex": "<pattern>",
      "headers" | "headers_regex": { "<Header>": "<pattern>" }, // 最多 4 个
      "remote_addresses" | "cidrs": ["<cidr>", ...],          // 最多 8 个，IPv4 或 IPv6
      "action": "ALLOW" | "DENY" | "CHALLENGE" | "WEIGH",
      "weight": int,                                          // 仅用于 WEIGH，-1000..1000
      "challenge": { "difficulty": <work bits>, "algorithm": "hashcash" | "posw" },
      "limits": { "rate": int, "window_seconds": int, "ban_seconds": int },
      "ja4": "<fingerprint>" | "<prefix>*"                    // 36 字节，或前缀加 *
    }
  ]
```

```
}
```

challenge_at 和 deny_at 的取值范围为 -1000 到 1000，并且 deny_at 不小于 challenge_at；bits_step 的取值范围为 1 到 1000。challenge 块只属于 CHALLENGE 规则，工作比特为 1 到 64。limits 为规则设定自己的配额：rate（1 到 1,000,000）和 window_seconds（1 到 86,400）是必需的，ban_seconds（最多 86,400）是可选的，WEIGH 规则不能带有 limits。ja4 是一个完整的 JA4 指纹，此时必须与之相等，或者是一个后跟 * 的前缀，此时指纹必须以它开头；单独的 * 匹配任何指纹，没有指纹的请求不匹配任何 ja4 规则。任何指纹都无法匹配的模式会以 InvalidJa4 被拒绝（第 IX 部分“来自入口代理的 JA4 指纹”）。ip_rules 的信誉字典树最多容纳 8,192 个节点。每条规则对每个条件只能使用一种写法。

模式语法：.* 或 * 匹配任何内容；^...$ 锚定一个精确路径；末尾的 *、/* 或 .* 表示前缀；以 / 开头的模式是精确路径；其他情况是不区分大小写的子串。路径模式匹配规范形式的路径（段之间只有一个斜杠，没有点段，非保留字符不转义，其他转义使用大写十六进制数字），因此包含 ;、%2F、%5C 或 \ 的模式不会匹配任何内容，并且路径匹配区分大小写（SID 0003）。

存储表

policy_revision、policies（含可为空、最多 37 字节的 ja4_pattern 条件）、policy_inspection、ip_reputation、incident_receipts、security_incidents（含 256 字节的嵌入向量和可为空的 36 字节 ja4 指纹）、全文索引（SQLite 上为 incidents_fts，即基于规范词元的 FTS5；PostgreSQL 上为带 GIN 索引的 tsvector 列 search）以及 sibuna_schema；控制台另外增加 33 张 console_* 表。存储的动作、算法和模式名称均为小写。事件 ID 为 node_id << 40 | sequence，无需协调即可在集群内保持唯一；节点 ID 的范围为 1 到 8,388,607。

构建目标

表 32 构建与仓库命令

命令	结果
cargo build --release -p sibuna	带 SQLite 存储和控制台的守护进程（默认特性）
cargo build --release -p sibuna --no-default-features	不带存储和控制台、也不依赖任何外部 crate 的内存型守护进程
cargo build --release -p sibuna --features postgres-tls	还能通过经验证的 TLS 连接共享 PostgreSQL 数据库的守护进程（只启用 postgres 特性时不使用 TLS 连接）
cargo test	单元测试、针对运行中守护进程的集成测试、存储和控制台测试（设置 SIBUNA_TEST_DATABASE_URL 时包括 PostgreSQL 测试）
cargo clippy --all-targets, cargo fmt --check	lint 和格式化（max_width = 99）

命令	结果
`cargo xtask check`	70 行函数、1,408 行文件和 99/108 列的行长度限制；不允许制表符
`cargo xtask browser`	在无头 Chromium 中进行的检查（`sibuna-browser`）
`cargo xtask bench <primitives|product|latency>`	release 模式的基准测试；结果位于 `benchmarks/results/<command>-latest.json`
`cargo build -p sibuna-pow-wasm --target wasm32-unknown-unknown --profile wasm`	只构建浏览器模块（控制台界面用 `-p sibuna-console-ui`）
`cargo run -p sibuna-geoip --bin geoip-snapshot -- ...`	校验国家数据，并写出一个可嵌入的 GeoIP 快照
`guidedog build` / `guidedog dsc`	本书 / SID 记录

守护进程的构建脚本（`cmd/sibuna/build.rs`）用 `wasm` profile 把两个 WebAssembly 模块构建到工作区的目标目录中，并在控制台界面超过 768 KiB 时使构建失败。对于没有 `wasm32-unknown-unknown` 目标的构建，可以用 `SIBUNA_POW_WASM` 和 `SIBUNA_CONSOLE_WASM` 指定预先构建的模块。`SIBUNA_GEOIP_DATA` 指定要嵌入的 GeoIP 快照；它在构建时解码，不带控制台的构建会拒绝它（`GEOIP001`）。

练习 10.1. 一个客户端在求解过程中从 Wi-Fi 切换到蜂窝网络后，收到标题为 `CLIENT FINGERPRINT MISMATCH` 的 `400`。根据令牌和挑战的格式解释原因，并提出对验证中间页最小的修改，使其能够平稳恢复。

解释这个不变量. 不看书，列出位于 Sibuna 之前的反向代理必须路由到守护进程而不是源站的端点，并说明每个端点为什么需要。

第 XI 部分

速查卡

放在终端旁边：构建、运行、配置、路由、观测和诊断。完整的表格见第 X 部分；这张卡只列出运维人员要输入的内容。

构建

```
cargo build --release -p sibuna                          # 守护进程 + SQLite + 控制台
cargo build --release -p sibuna --no-default-features    # 纯内存，不依赖外部 crate
cargo build --release -p sibuna --features postgres-tls  # 共享 PostgreSQL 的集群
rustup target add wasm32-unknown-unknown   # 或设置 SIBUNA_POW_WASM / SIBUNA_CONSOLE_WASM
cargo test            # 单元、实时守护进程、存储和控制台测试
cargo clippy --all-targets && cargo fmt --check
cargo xtask check     # 70 行 / 1,408 行 / 99 列检查
cargo xtask browser   # 无头 Chromium 检查
cargo xtask bench product   # 在 wrk 下测完整产品 -> benchmarks/results/
guidedog build        # 本书（guidedog dsc：SID 记录）
```

运行

```
# Shield（默认）：PoW + WAF，代理到 :3000
sibuna -p 8080 -u 3000 -s /etc/sibuna/secret
# Gate：仅 PoW
sibuna --gate -p 8080 -u 3000 -s /etc/sibuna/secret
# 位于 Nginx/Caddy 之后的 forward auth（信任 X-Forwarded-For）
sibuna -m forward_auth --host 127.0.0.1 -p 8080 -s /etc/sibuna/secret
# Edge：持久策略、信誉和取证数据存放在 <dir>/sibuna.db
sibuna -D /var/lib/sibuna -s /etc/sibuna/secret
# 来自受信任入口代理的 JA4（Cloudflare 的请求头；nginx 模块：x-ja4；ATS：ja4）
sibuna --ja4-header cf-ja4 --trust-forwarded -p 8080 -u 3000 -s /etc/sibuna/secret
# 集群成员：共享 PostgreSQL，节点 ID 唯一
sibuna --database-url-file /etc/sibuna/database-url --cluster-node 1 -s /etc/sibuna/secret
head -c 32 /dev/urandom | xxd -p -c 64 > /etc/sibuna/secret
```

控制台（可选启用的预览功能）

```
# 在守护进程停止时初始化，然后启动控制台
sibuna init-admin admin --data-dir ./data
sibuna --data-dir ./data --console 127.0.0.1:19446
```

打开 `http://127.0.0.1:19446/console/` 并更换临时密码。加上 `--console-location 1.3521,103.8198` 可以把节点标注在地球视图上。远程访问需要 HTTPS 代理、明确指定的 origin 和受信任代理的

CIDR；请按照第 IX 部分操作。SID 0007 的状态为 Committed。控制台在无头 Chromium 中通过了它的浏览器工作流测试（SID 0010，标准 5）；它的正式性能隔离门槛仍未完成（第 VIII 部分）。

最先需要关注的选项

选项	默认值	含义
`-d, --difficulty`	16	工作比特（PoSW 深度 = 比特数 - 3）
`-a, --algorithm`	posw	`posw` 或 `hashcash`
`--posw-challenges`	16	每个证明的揭示数
`--token-ttl`	86400	会话秒数
`--challenge-ttl`	300	挑战秒数
`--token-scheme`	mac	`mac` 或 `ed25519`
`--rate-limit / --rate-window`	100 / 10	每个窗口的 GCRA 突发量
`--ban-seconds`	3600	蜜罐封禁
`-w, --workers`	CPU 数	接受线程
`--max-connections`	1024	超出后返回 `503`
`--max-connections-per-ip`	64	按对端地址（IPv6：按 `--ipv6-prefix`），超出后返回 `503`
`--trusted-hops`	1	客户端 = 从右数第 n 个 `X-Forwarded-For` 条目
`--ja4-header`	关闭	包含入口代理所给 JA4 指纹的请求头；仅在受信任时读取
`--idle-timeout`	15	回收秒数
`-P, --policy-file`	无	JSON 策略
`--trust-forwarded`	关闭	forward-auth 模式下开启
`--secure-cookie`	关闭	加上 `Secure`

`/__sibuna/` 下的端点

路由	应答
`challenge`	验证中间页 HTML
`challenge.json?need=&path=`	`{id, algorithm, difficulty, challenges, expires_at}`

路由	应答
`verify`（POST）	`{challenge_id, nonce | proof}` -> `200` + `Set-Cookie`，否则 `400`
`wasm/sibuna-pow.wasm`	15,181 字节的求解器
`worker.js`	包含 WASM 和 JS 证明器的 worker
`honeypot`	封禁调用方，`403`；子资源或跨站获取返回 `404`
`health`	`{"status":"ok",...}`
`metrics`	Prometheus 文本

把它们全部路由到守护进程，绝不路由到源站。

状态码

状态码	时机
200	已准入（forward-auth）、验证中间页、内部路由
400	请求或目标格式错误，解答被拒（响应体为诊断信息）
401	需要挑战：非 HTML 客户端或 forward-auth 模式
403	策略或 WAF 拒绝、封禁、蜜罐
404	未知的 `/__sibuna/` 路由，被拒绝的蜜罐命中
413	解答请求体超过 64 KB 缓冲区
414	上报的挑战 URL 超过 8 KiB
417	不支持的请求期望
429	GCRA 限制，`Retry-After` 秒数
431	头部超过 16 KB
501	chunked 以外的传输编码
502	源站不可达或响应格式错误
503	达到连接限制，或节点正在排空

请求头与响应头

- 发往上游（代理）：`X-Forwarded-For`（套接字对端；受信任时追加在收到的链之后）、`X-Forwarded-Proto`、`X-Real-IP`（被判定的客户端）、`X-Sibuna-Status: PASS`、`X-Sibuna-Rule: <rule>`；逐跳字段和其他转发字段会被丢弃，使用下划线拼写的也一样。
- *forward-auth* 回复：`X-Sibuna-Status`、`X-Sibuna-Rule`、`X-Sibuna-Rule-Hash`。
- 挑战回复：`X-Sibuna-Status: CHALLENGE`。

- *Cookie*：`__sibuna_token=<75 chars, 139 with ed25519>; Path=/; Max-Age=<ttl>; HttpOnly; SameSite=Lax[; Secure]`

策略文件骨架

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

JA4：完整的指纹，或者前缀加 `*`；没有指纹则不匹配。模式：`*` 或 `.*` 匹配任意内容；`^...$` 精确匹配；末尾 `*` 为前缀；以 `/` 开头为精确路径；其他为不区分大小写的子串。路径按规范形式匹配。顺序：WAF -> 字典树 deny/allow -> 规则 -> 分数 -> 绕过 -> 字典树 challenge -> 机器人 -> 默认。

用 SQL 操作存储（Edge）

```
INSERT INTO policies (id, name, priority, path_pattern, action, difficulty, algorithm,
  header_matchers, cidr_matchers, weight, enabled, created_at, updated_at)
  VALUES ('p1', 'protect', 10, '/checkout/*', 'challenge', 20, 'posw', '{}', '[]', 0, 1,
  unixepoch(), unixepoch());
INSERT INTO ip_reputation (ip_or_cidr, reputation_score, banned_until, trigger_rule, hits,
last_seen)
  VALUES ('198.51.100.7', -100, unixepoch() + 86400, 'analyst', 1, unixepoch());
SELECT s.client_ip, s.violation_category, s.path, s.campaign_id
  FROM incidents_fts f JOIN security_incidents s ON s.id = f.rowid
  WHERE incidents_fts MATCH 'union' ORDER BY rank LIMIT 20;
```

SQLite（`sqlite3 <dir>/sibuna.db`）；动作名为小写。分数 ≤ -50 -> deny 前缀；≥ 50 -> allow。集群：每个节点使用相同的密钥，一个 PostgreSQL 数据库，`--features postgres-tls`，`--database-url-file <file> --cluster-node N`，其中 N 唯一，取值 1 到 8,388,607；没有成员列表、领导者或集群 TLS（SID 0011）。

诊断

标题	含义
`MALFORMED CHALLENGE`	ID 被编辑或截断；获取一个新的
`INVALID CHALLENGE TAG`	不是由这个种子或节点签发的

标题	含义
CHALLENGE EXPIRED	超过 TTL，或时钟偏差 > 60 s
CLIENT FINGERPRINT MISMATCH	地址或 User-Agent 发生了变化
DIFFICULTY NOT MET	nonce 缺少零比特
INVALID PROOF OF SEQUENTIAL WORK	揭示无法到达根
WRONG SOLUTION TYPE	为 PoSW 提交了 nonce，或为 Hashcash 提交了证明
DOUBLE SPEND PREVENTED	挑战已被使用
STORE FULL	已用集合饱和；降低 TTL 或调大 `--spent-tags`
NETWORK BINDING MISMATCH	Cookie 被复制到另一个客户端
INVALID COMMAND LINE	未知选项、错误的值，或已移除的 Zaxonlite 选项

需要告警的指标：`denied`、`rate_limited`、`overloaded`、`upstream_errors`、`incidents_dropped`、`incident_write_failures`、`bans_refused`、`forwarded_invalid`（均为 `sibuna_*_total`）。

需要记住的数字

挑战 ID 70 字符（36 字节载荷 + 16 字节标签）；令牌 75 字符（40 + 16），使用 Ed25519 时为 139；证明字节数 $32(1 + t(n + 1))$；PoSW 深度 = 比特数 - 3，范围 4 到 24；请求缓冲区 64 KB；头部上限 16 KB；检查请求体 8 KB；已用集合 1,048,576 个标签（`--spent-tags`）；速率单元每张表 262,144 个（`--rate-table-cells`）；封禁 65,536 个槽（`--ban-table-slots`）；每个客户端地址 64 个连接；事件环 512；批次 32；攻击活动距离 0.35，基于最近的 16,384 个嵌入向量；keep-alive 每个连接 4,096 个请求；源站连接池 256 个套接字；存储任务队列 32。

入口路由

使用第 IX 部分“入口代理之后的 forward-auth”中的 Nginx 或 Caddy 配置方案。直接路由 `/__sibuna/*`；用原始 URI、方法和受信任的客户端地址对应用请求进行授权。用授权结果替换传入的 Sibuna 审计头。Nginx 需要显式处理挑战、429 和认证服务不可用的情况。

`reverse_proxy` 承载已准入的 HTTP/1.1 请求体和 WebSocket；`forward_auth` 只授予准入，由入口代理承载它们。被省略的认证请求体和源站响应无法被 Sibuna 检查或计数。TLS 在入口代理处终止；原生 HTTP/2 推迟实现。上传使用 Content-Length 或分块编码；检查最多只看到 8 KiB 的前缀，文件字节保持不透明。

部分练习解答

这些解答用来核对论证，而不是代替练习。如果结果不同，我们先比较假设：时间区间的端点、什么被计为工作，以及某次写入是否已经提交，都是常见的分歧来源。

第 I 部分：成本与概率

1.1. 一百万次尝试分摊到 100 个请求，每个请求 10,000 次尝试。只要总请求数仍是 100，共享会话就不会改变这个算式。实际系统必须判断第二个进程能否满足指纹和路由绑定；这些条件不在分摊等式之内。

1.2. 超出的速率为 $1000 - 600 = 400$ 条记录/秒。在流体模型中，一个容量为 512 条记录的空队列能争取到 $512/400 = 1.28$ 秒。真实的生产者以突发方式到达，消费者按批次提交，因此第一次丢弃的确切时间取决于到达时间和提交时间。

第 II 部分：前人工作

2.1. 同一份源码编译两次，不会在陈述、nonce 编码、树的形状或揭示的布局上与自己不一致；要发生漂移，除非编译器有缺陷。但这并不能说明构造是否可靠（需要密码学审查），浏览器运行时是否正确执行了它（WebAssembly 与 JavaScript 证明器之间的逐字节比较覆盖了这一点），或者参数是否经过校准（需要测量）。

2.2. (a) Sibuna Gate 或 Anubis，一个可执行文件；(b) Sibuna Shield、SafeLine 或 Cloudflare（Pro 套餐及以上）；(c) 使用共享 PostgreSQL 数据库（`--database-url`）的 Sibuna Edge，使用共享 Valkey 存储的 Anubis（用于状态，而不是封禁），或 Cloudflare。在不需要 Docker 主机或订阅的情况下同时满足这三项的唯一单一产品，是各节点共享一个 PostgreSQL 数据库（`--database-url` 和 `--cluster-node`）的 Sibuna Shield；起决定作用的是“运行形式”“检查”和“多节点”这几行。

第 III 部分：密码学

3.0. 尾部估计为 $e^{-4.6} \approx 0.010$。划分 nonce 空间改变的是完成尝试的速率，而不是每次独立尝试的成功概率。重复的尝试或协调工作，可能使真实的并行实现效率更低。

3.1. 如果每个承诺成功的概率为 q，那么全部 g 个都失败的概率为 $(1-q)^g$，至少一个成功的概率为 $1-(1-q)^g$。这个公式没有计入构造承诺的成本、查询预算、尝试之间的依赖，以及找到具有特定接受概率的承诺所需的工作。

3.2. 在深度 17、16 次揭示时，证明有 $32(1 + 16 \times 18) = 9248$ 字节。32 次揭示时有 $32(1 + 32 \times 18) = 18464$ 字节。揭示部分的字节数翻倍；32 字节的根仍然只发送一次。

3.3. 如果没有工作等级，在低成本路由上获得的会话就会准入发往要求更多工作的高成本路由的请求；没有 `timestamp`，令牌可能被签发在“未来”，从而比它的策略活得更久；没有 `expiry`，它将永不失效；没有 `rule_hash`，上游就无法得知是哪条规则准入了该客户端；没有 `fingerprint`，复制到另一台机器上的 Cookie 也会被接受。标签必须覆盖全部五个字段，因为任何留在标签之外的字段都可以被随意修改，而验证者无从察觉。

第 IV 部分：协议

4.1. 两个浏览器共享地址和 User-Agent，因此共享指纹、速率限制单元和封禁槽。它们不共享挑战（各自获取带有自己 nonce 的记录）、已用集合条目或 Cookie。要复用另一个浏览器的 Cookie，位于该 NAT 之后的攻击者需要从另一台机器上读取它：此时指纹会匹配，这就是为什么在共享地址上，`HttpOnly` 和 `SameSite` 比绑定更重要。

第 V 部分：状态与顺序

5.1. 头部和请求体的第一部分（64 KB 中头部之后能容纳的部分）位于连接缓冲区中，并通过 `write_all(request.body)` 发送。其余的字节从不完整地进入缓冲区：`relay_body` 以 16 KB 为单位从客户端读取器读取它们，并把每一块写给源站。头部必须先发送，因为源站在请求体之前需要 `Content-Length`，而且审计头也是头部的一部分。
5.2. 发射间隔为 100 ms，突发容限为 9,900 ms。一个空闲的客户端可以立即发送 100 个请求，然后每 100 ms 发送一个。算上 3,000 ms 这个端点，上界为 130。固定的 10 秒计数器会在第一个窗口中允许 100 个请求，并在窗口滚动的那一刻再允许 100 个；滑动日志在任意 10 秒内恰好允许 100 个，但必须为每个客户端保存 100 个时间戳。

第 VI 部分：算法

6.1. 名字中的撇号，以及一个名为 `order` 的普通参数，都不能构成 SQL 表达式。子串检测器可能把其中任何一个误认为语法。结构检测器会寻找额外的证据，包括运算符和字面量之间的关系。它仍然是有界的检查，而不是针对每种应用语言的解析器。
6.2. LDAP 过滤器由括号、`|`、`&`、`!` 和 `=` 构成。检测器需要一个用于 `(` 和 `)` 的字节类（新增），并可以复用现有的 `=` 类以及用于 `|` 和 `&` 的 shell 分隔符类。它应当在 `)(` 或 `(|(` 这样的 序列 上触发，而不是在单个括号上触发，原因与 SQL 检测器要求引号或恒真式相同。

第 VII 部分：浏览器引擎

7.1. 在默认的 16 个工作比特下（深度 13，16 次揭示），假设在 WebAssembly 中需要 20 ms，那么在笔记本电脑上用 JavaScript 就需要 $50 \times 20 = 1000$ ms，即一秒，在较慢的手机上则需要四秒。这些数字只是假设；在制定策略之前，我们要测量自己客户的设备。一条合理的策略是：让页面转发 `fallback` 消息，在服务器端将其记录为一个指标，并只为匹配已知会阻止 WebAssembly 的客户端的规则降低难度，而不是对所有人降低。

第 VIII 部分：评估

8.2. 对于 Gate 的 `admitted` 行，四个核心中有 $11.84 \times 211,345/10^6 \approx 2.50$ 个处于忙碌状态，因此守护进程不受 CPU 限制，是负载生成器决定了上限。每请求 11.84 us 的 CPU 时间远高于亚微秒级的分类成本，因为产品测量的一行还包含内核的套接字读写、解析器、会话标签检查、指标递增、响应格式化、写入器刷新，以及调度器在负载生成器和连接线程之间的交接。原语测试套件对这些都不计时。

第 IX 部分：运维

9.1. 如果数据在单独的回执更新之前提交，两者之间的崩溃会留下旧的回执，重试可能再次施加该效果。如果回执先提交，在写入数据之前崩溃，可能导致重试跳过从未存在过的数据。一个原子事务可以消除这两个缺口。丢失事务的确认时，仍然需要保留的重试。

9.2. `ip_rules: {"10.20.0.0/16": "ALLOW"}`（或一条分数为 100 的 `ip_reputation` 行）准入预发布环境；规则 `{"name": "admin", "path": "/admin/*", "action": "CHALLENGE", "challenge": {"difficulty": 20, "algorithm": "posw"}}` 要求付出工作；一条匹配 `headers: {"X-Partner-Key": ".*"}`、动作为 `ALLOW` 并带有 `"limits": {"rate": 30, "window_seconds": 10}` 的规则以每 10 秒 30 个请求的速率准入合作伙伴。规则的配额在它匹配的每个请求上检查，检查发生在守护进程的按客户端限制（`--rate-limit`、`--rate-window`）之后，因此后者必须至少同样宽松。

第 X 部分：参考

10.1. 挑战记录携带获取它的地址的带密钥指纹；解答从一个新地址到达，因此 `verify_and_mint` 在查看证明之前就返回 `FingerprintMismatch`。最小的恢复方式在验证中间页中：收到标题为 `CLIENT FINGERPRINT MISMATCH` 的 `400` 时，获取一个新的挑战并重新求解，而不是显示错误，因为已经完成的工作无法转移。

继续探究

在一个可丢弃的测试中，每次只改变一个不变量。去掉快照的重新检查，省略事件回执，或者把速率间隔向下取整。在运行测试之前，先预测失败的时间表。一个有用的回归测试不只是断言正常路径的输出：它要证明被省略的条件为什么是必要的。

术语表

准入（Admission）
决定把请求转发给源站。在 Sibuna 中，准入由有效会话、`ALLOW` 规则、静态绕过路径或信誉 `allow` 前缀作出，并且绝不推翻 WAF 的拒绝。

Aho-Corasick 自动机
一种有限自动机，只需遍历输入一次，就能找到固定模式集合中的任意模式。Sibuna 把失败链接折叠进稠密转移表，使每个字节只需一次表读取。

封禁表
4,096 个无锁槽，保存被封禁的客户端地址及其过期时间；由蜜罐和集群共享的信誉写入，每个请求都会读取。

攻击活动（Campaign）
一组安全事件，它们的载荷嵌入向量彼此之间的余弦距离不超过 0.35；这是一种启发式分组，并不能证明来源相同。

挑战记录
客户端必须求解的 70 字符无状态标识符：版本、算法、难度、揭示数、签发时间、客户端指纹、PRF nonce、规则哈希，以及 16 字节的带密钥 BLAKE3 标签。签发挑战记录不会在服务器上写入任何内容。

难度，工作比特
同时缩放两层证明的那个整数：对 Hashcash 而言是前导零比特数，对顺序工作量而言是树深度加三。

Edge
使用 `--data-dir`（单个节点上的一个 SQLite 文件），或者使用 `--database-url` 和 `--cluster-node`（由集群共享的一个 PostgreSQL 数据库）的 Sibuna 部署，为任一形态提供持久策略、共享信誉和事件取证。

引擎槽
已发布的策略引擎所在的位置，形式为一个引用计数指针。worker 在分类期间持有一个引用；存储线程在请求路径之外构建新引擎，并通过交换指针发布它，旧引擎在最后一个读者释放它时被回收。

指纹
客户端地址和 User-Agent 的带密钥哈希，绑定进挑战和令牌中，使两者都无法从另一个客户端重放。

Forward auth
一种部署模式：入口代理询问 Sibuna 是准入请求（`200`），还是挑战、拒绝或限制它（`401`、`403`、`429`），并自行代理源站。

Gate
具备工作量证明准入、会话、规则、信誉、GCRA 限制和封禁，但没有应用检查的形态（`--gate`）。

GCRA
通用信元速率算法（Generic Cell Rate Algorithm）：每个客户端保存一个理论到达时间，允许 N 个请求的突发，此后每个发射间隔允许一个请求，没有窗口边界。

Hashcash
第一层：寻找一个 nonce，使挑战、一个冒号和十进制 nonce 拼接后的 SHA-256 具有 b 个前导零比特。期望尝试 2^b 次，验证只需一次哈希。

蜜罐
一个指向 `/__sibuna/honeypot` 的不可见链接；访问它的客户端会被封禁，并记录一条信誉为 −100 的事件。

空闲回收器
一个线程，关闭静默时间超过 `--idle-timeout` 的连接，使慢速客户端无法占住连接线程。

验证中间页
代替受保护页面提供的嵌入式 HTML 页面；它获取挑战，在 Web Worker 中求解，提交解答，然后重新加载。

JA4
由 FoxIO 定义的 36 字符指纹，描述客户端如何建立 TLS 连接：协议、TLS 版本、服务器名称、密码套件和扩展的数量、ALPN，以及排序后的密码套件和扩展的哈希。Sibuna 不计算它；它读取受信任的入口代理在 `--ja4-header` 所指定的请求头中转发的值，规则用 `ja4` 字段匹配它（SID 0014）。它与绑定进挑战和令牌中的客户端指纹无关。

带密钥 BLAKE3
Sibuna 的密钥派生、挑战标签、会话标签、指纹和挑战 nonce 背后的伪随机函数。

MAC 令牌
默认的会话 Cookie：40 字节的载荷（版本、工作等级、时间戳、过期时间、规则哈希、指纹）加上 16 字节的带密钥 BLAKE3 标签，以常量时间验证。

工作等级
会话持有者实际求解的机制和工作比特，保存在令牌中；只有当会话的等级达到路由的要求时，该路由才准入这个会话。

揭示（Opening）
在顺序工作量证明中，一个叶子标签加上它通往根的路径上的兄弟标签；验证者重新计算这条路径，并与承诺比较。

顺序工作量证明（PoSW）
第二层：Cohen-Pietrzak 哈希图，其中每个标签都依赖于之前的所有标签，无论证明者有多少个核心，都必须完成 $2^{n+1} - 1$ 次顺序哈希。验证需要 $t(n + 1)$ 次哈希。

信誉字典树
128 位基数字典树，保存来自策略文件和 `ip_reputation` 表的 `allow`、`deny` 和 `challenge` 前缀。

Robin Hood 已用集合
固定容量的开放寻址表，保存已求解的挑战标签，防止同一个有效解答被提交两次。

规则哈希
要求挑战的规则名称的 64 位哈希，携带在挑战、令牌和 `X-Sibuna-Rule-Hash` 请求头中。

Shield
默认的形态：Gate 加上语义防火墙（`--shield`）。

SID
Sibuna 讨论（Sibuna Discussion）记录：位于 `docs/sid/records` 下的论文式设计文档，记录系统某一部分背后的假设、证明和测量。

形态（Surface）
两种产品形态之一，即 Gate 或 Shield；存储和集群是两者都可选用的选项。

WEIGH
: 一种规则动作，它不直接作出决定，而是给请求的分数加上一个有符号的权重；总分达到或超过阈值时，以额外的工作比特发起挑战或拒绝请求。

Zaxonlite
: SID 0005 中的嵌入式 SQLite 复制引擎（Multi-Paxos），在 SID 0011 中被 SQLite 和 PostgreSQL 存储取代；它的集群选项现在会以 `RemovedOption` 终止启动。

参考文献

- Aho, A. V., and Corasick, M. J. “Efficient string matching: an aid to bibliographic search.” *Communications of the ACM* 18(6), 1975.
- ATM Forum. *Traffic Management Specification Version 4.0*, af-tm-0056.000, 1996（通用信元速率算法）。
- Back, A. “Hashcash: a denial of service counter-measure.” Technical report, 2002.
- Blocki, J., Lee, S., and Zhou, S. “On the security of proofs of sequential work in a post-quantum world.” *Information-Theoretic Cryptography (ITC)*, 2021.
- Celis, P. *Robin Hood Hashing*. PhD thesis, University of Waterloo, 1986.
- Cohen, B., and Pietrzak, K. “Simple proofs of sequential work.” *EUROCRYPT*, 2018.
- Dwork, C., and Naor, M. “Pricing via processing or combatting junk mail.” *CRYPTO*, 1992.
- Fiat, A., and Shamir, A. “How to prove yourself: practical solutions to identification and signature problems.” *CRYPTO*, 1986.
- Fielding, R., Nottingham, M., and Reschke, J. *HTTP Semantics*, RFC 9110, and *HTTP/1.1*, RFC 9112. IETF, 2022.
- FoxIO. *JA4+ Network Fingerprinting*. https://github.com/FoxIO-LLC/ja4, 2023（JA4 TLS 客户端指纹）。
- Hanson, N. “libinjection: SQL injection detection by tokenization.” Black Hat USA, 2012.
- Juels, A., and Brainard, J. “Client puzzles: a cryptographic countermeasure against connection depletion attacks.” *NDSS*, 1999.
- Lamport, L. “The part-time parliament.” *ACM Transactions on Computer Systems* 16(2), 1998; and “Paxos made simple.” *ACM SIGACT News* 32(4), 2001.
- Mahmoody, M., Moran, T., and Vadhan, S. “Publicly verifiable proofs of sequential work.” *Innovations in Theoretical Computer Science (ITCS)*, 2013.
- O’Connor, J., Aumasson, J.-P., Neves, S., and Wilcox-O’Hearn, Z. *BLAKE3: one function, fast everywhere*. Specification, 2020.
- Vyukov, D. "Bounded MPMC queue." 1024cores.net, 2011（事件队列所用的带序号环形缓冲区）。
- Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J., and Zhu, H. “Hyperscan: a fast multi-pattern regex matcher for modern CPUs.” *NSDI*, 2019.
- Sibuna 讨论 0001 至 0014，`docs/sid/records`，2026：流程、基础架构、声明式策略引擎、语义检查、Zaxonlite 存储架构、数学基础、控制台管理界面、AI 机器人流量监控、分块请求体、Rust 移植、通过 sqlx 实现的存储、通过 rustls 为控制台提供的出站 TLS、控制台存储，以及来自入口代理的受信任 JA4 指纹。

索引

A
Aho-Corasick 自动机, **128**

E
Edge, **128**

F
Forward auth, **128**

G
Gate, **128**
GCRA, **128**

H
Hashcash, **129**

J
JA4, **129**

M
MAC 令牌, **129**

R
Robin Hood 已用集合, **129**

S
Shield, **129**
SID, **129**

W
WEIGH, **130**

Z
Zaxonlite, **130**

信
信誉字典树, **129**

准
准入（Admission）, **128**

封
封禁表, **128**

工
工作等级, **129**

带
带密钥 BLAKE3, **129**

引
引擎槽, **128**

形
形态（Surface）, **129**

指
指纹, **128**

挑
挑战记录, **128**

揭
揭示（Opening）, **129**

攻
攻击活动（Campaign）, **128**

空
空闲回收器, **129**

蜜
蜜罐, **129**

规
规则哈希, **129**

难
难度，工作比特, **128**

顺
顺序工作量证明（PoSW）, **129**

验
验证中间页, **129**