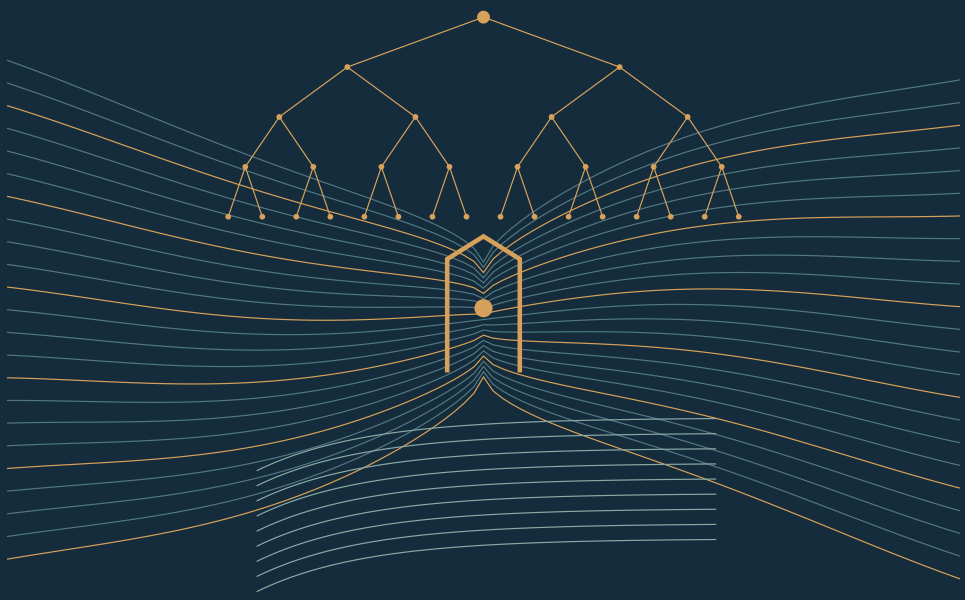
DAS HANDBUCH ZU

SIBUNA

Mathematik und Technik der Webabwehr



ARBEIT · VERTRAUEN · BEGRENZTER ZUSTAND

Vikrant Rathore, mit Unterstützung von Ronak Rathore und Kanak Rathore

Version 1.0.0-beta.3 · Nov. 13, 2026

Inhalt

Vorwort . . . 6

Wie wir dieses Buch lesen . . . 8

- Drei Lesewege . . . 8
- Konventionen . . . 9

Teil I: Die Kosten einer Anfrage . . . 10

- Zulassung ist eine wirtschaftliche Entscheidung . . . 10
- Was ein Proof of Work belegt . . . 11
- Hash-Suche als Zufallsvariable . . . 11
- Sitzungen amortisieren Arbeit . . . 12
- Begrenzter Zustand ist ein zweites Budget . . . 12
- Die Invarianten, die wir mitnehmen . . . 13

Teil II: Stand der Technik und der Entwurfsraum . . . 14

- Woher Sibuna kommt . . . 14
- Drei Ideen in einer Binärdatei . . . 16
- Funktionsvergleich . . . 17

Teil III: Grundlagen aus der kryptographischen Forschung . . . 23

- Hashcash: Die Mathematik von Stufe eins . . . 23
- Nachweis sequenzieller Arbeit: Stufe zwei . . . 24
- Symmetrische Authentifizierung: Challenges und Sitzungen . . . 28

Teil IV: Das Sibuna-Protokoll . . . 31

- Der Challenge-Rundlauf . . . 31
- Der zustandslose Challenge-Datensatz . . . 33
- Reihenfolge der Verifikation und Einmalverwendung . . . 35
- Das Sitzungscookie . . . 36

Teil V: Allokationsfreie Technik in Rust . . . 39

- Die Verbindungsschleife . . . 39
- Nebenläufiger Zustand ohne Allokation . . . 51

Teil VI: Automaten, Tries und die semantische Firewall . . . 62

- Suche nach vielen Mustern in einem Durchlauf . . . 62
- Der Radix-Trie für IPv4 und IPv6 . . . 64
- Die semantische Firewall . . . 65
- Die deklarative Richtlinien-Engine . . . 66

Teil VII: Die Proof-of-Work-Engine im Browser . . . 69

- Eine Quelle, zwei Ziele . . . 69
- Das Worker-Protokoll . . . 73
- Kalibrierung . . . 74
- Die Zwischenseite . . . 74

Teil VIII: Empirische Bewertung . . . 76

- Methodik . . . 76

Latenzen der Primitiven 81
Vergleich der Gesamtprodukte 84
Latenz bei gleicher angebotener Last 86
Sibuna und Anubis 88
Overhead der Konsole 93
Nicht gemessen 94

Teil IX: Betrieb und Einsatz 96
Betriebsformen und Konfiguration 96
Vorschau der Konsole 99
Einsatztopologien 118
Deklarative Richtlinie 125
Persistente Speicherung und Cluster 126
Paketierung 131

Teil X: Nachschlagewerk und Diagnose 135
Endpunkte 135
Metriken 136
Statuscodes 138
Fehlerkatalog 138
Richtlinienschema 140
Speichertabellen 141
Build-Ziele 142

Teil XI: Kurzreferenzkarte 144
Bauen 144
Ausführen 144
Konsole (optionale Vorschau) 145
Die wichtigsten Optionen 145
Endpunkte unter `/__sibuna/` 146
Statuscodes 146
Header 147
Gerüst der Richtliniendatei 147
Speicherung (Edge) in SQL 147
Diagnosen 148
Zahlen zum Merken 148
Routing am Eingangsproxy 149

Ausgewählte Lösungen 150
Teil I: Kosten und Wahrscheinlichkeit 150
Teil II: Stand der Technik 150
Teil III: Kryptographie 150
Teil IV: Protokoll 151
Teil V: Zustand und Reihenfolge 151
Teil VI: Algorithmen 151
Teil VII: Browser-Engine 152
Teil VIII: Auswertung 152

Teil IX: Betrieb . . . 152
Teil X: Referenz . . . 153
Die Untersuchung fortsetzen . . . 153
Glossar . . . 154
Literaturverzeichnis . . . 157
Stichwortverzeichnis . . . 158

Das Sibuna-Handbuch, Version 1.0.0-beta.3
© 2026, Vikrant Rathore und die Mitwirkenden an Sibuna

Mathematik und Technik der Webabwehr.

Vorwort

Eine Anfrage kommt mit einer Methode, einem Pfad, einigen Headern und einem Anspruch auf die Aufmerksamkeit des Servers an. Keines dieser Bytes belegt, dass es sich lohnt, sie zu bedienen. Sibuna stellt eine Entscheidung zwischen die Anfrage und den Ursprungsserver: sie zulassen, sie abweisen oder ihren Absender um nachprüfbare Arbeit bitten. Dieses Buch verfolgt diese Entscheidung von ihrem mathematischen Modell bis zu den Bytes im Speicher. Das System hat zwei Betriebsformen. *Gate* bietet Proof-of-Work-Zulassung und lokale Flutkontrollen. *Shield* ergänzt Anwendungsprüfung und Richtlinienentscheidungen. Beide können Richtlinien, Reputation und Vorfallshistorie persistent speichern, lokal auf einem Knoten oder gemeinsam für einen Cluster; diesen Einsatz nennen wir *Edge*. Eine gültige Sitzung begründet die Zulassung; sie befreit die Anfrage nicht von der Prüfung. Wer dauerhaften Zustand im Cluster teilt, verteilt ihn; aus einem lokalen Ratenzähler wird dadurch kein globales Kontingent.

Die zentrale technische Frage ist nicht, ob ein Hash schnell ist. Sie lautet, ob der gesamte Pfad beschränkt bleibt, wenn ein nicht vertrauenswürdiger Client die Eingabe wählt. Wie viele Bytes kann ein Parser prüfen? Wie viel Speicher kann eine Challenge verbrauchen? Wie viele Verbindungen kann ein Client offen halten? Welche Schreibvorgänge dürfen wiederholt werden? Was bleibt verfügbar, wenn ein Knoten oder die Datenbank verschwindet? Jede Antwort muss sowohl eine Invariante als auch ihre Grenze nennen.

Die Mathematik dient demselben Zweck. Erwartete Kosten sind keine Latenzgarantie. Ein Stichprobenargument für ein festes Commitment ist kein Beweis gegen jeden adaptiven Beweiser. Eine Warteschlange fängt einen Burst ab; einen dauerhaft langsameren Verbraucher kann sie nicht ausgleichen. Wir leiten kleine Modelle her, rechnen Beispiele von Hand durch und fragen dann, wo die Implementierung von ihnen abweicht.

Das Buch ist für drei Arten der Nutzung geschrieben. In der Reihenfolge gelesen, bilden die Teile I bis VII einen Kurs: Jedes Kapitel beginnt mit dem, was wir danach können sollten, rechnet ein Beispiel von Hand durch und endet mit einer Übung und einer Invariante, die wir mit eigenen Worten erklären; ausgewählte Lösungen stehen am Ende des Buchs. Die Teile VIII bis X sind das Nachschlagewerk: Messungen mit ihrer Herkunft, jede Option und jede Topologie, jeder Endpunkt, Status, Fehler und jede Tabelle. Teil XI ist eine zweiseitige Karte für das Terminal. Ein Glossar und ein Literaturverzeichnis schließen das Buch ab.

Teil II stellt Sibuna Funktion für Funktion neben Anubis, SafeLine und die Cloudflare WAF, und Teil VIII präsentiert Sibunas eigene Messungen, von einzelnen Primitiven bis zu Läufen des Gesamtprodukts, und vergleicht Sibuna und Anubis als Gesamtprodukte unter demselben Lastgenerator. Beide Teile sind so geschrieben, dass wir die Aussagen prüfen können: Der Vergleich nennt seine Quellen, und jede Messung nennt die eingecheckte Ergebnisdatei, aus der sie stammt.

Vorausgesetzt werden Grundkenntnisse im Programmieren, Logarithmen und bedingte Wahrscheinlichkeit. Rust wird über Ownership und Datenlayout eingeführt, nicht durch einen Überblick über die Sprache. Die Kapitel lassen sich der Reihe nach lesen: Das Kostenmodell begründet das Protokoll, das Protokoll bestimmt den Zustand, und der Zustand bestimmt den

Entwurf von Nebenläufigkeit und Speicherung. Spätere Kapitel machen aus diesen Invarianten Messungen und Betriebsverfahren.

Dies ist ein Buch über eine Implementierung, keine Behauptung, dass Rechenrätsel automatisierten Datenverkehr beseitigen. Clients können Rechenleistung kaufen, Adressen können geteilt werden, und die Syntax von Anwendungen ist reicher, als ein begrenzter Detektor erfassen kann. Das nützliche Ergebnis ist ein System, dessen Kosten und Grenzen sich untersuchen, testen und ändern lassen, ohne sie hinter einem Slogan zu verbergen.

Ausgabe für Sibuna 1.0.0-beta.3, November 2026

Quellcode, Übungen und reproduzierbare Messungen finden sich im Sibuna-Repository.

Wie wir dieses Buch lesen

Wir beginnen mit einer einzigen Frage: Was ist die billigste sichere Entscheidung, die der Server jetzt treffen kann? Diese Frage begleitet uns durch die Kapitel über Protokoll und Implementierung. Eine billige Operation, die die falsche Anfrage zulässt, ist ein Defekt; eine korrekte Operation mit unbeschränkten Kosten ist eine andere Art von Defekt.

Frage	**Wo die Antwort entwickelt wird**
Was bringt uns ein Rätsel?	Teile I bis III: Kosten, Wahrscheinlichkeit und die Grenzen des Beweises.
Wie schneidet Sibuna im Vergleich ab?	Teil II: Herkunft und die Funktionstabelle; Teil VIII: Messungen.
Was bedeutet eine Sitzung?	Teil IV: Bindungen, Reihenfolge der Prüfung, Ablauf und Wiederholung.
Wo bleibt der Speicher?	Teile V und VI: Puffer, Threads, Tabellen, Automaten und Veröffentlichung.
Wie löst der Browser?	Teil VII: Worker-Nachrichten, Speicher und Kalibrierung.
Woher wissen wir, dass es funktioniert?	Teil VIII: drei Messprogramme und was jedes ausschließt.
Wie betreiben wir es?	Teil IX: Optionen, Topologien, Richtlinien, Speicherung, Clusterbildung, Paketierung.
Was war das für ein Fehler?	Teil X: Endpunkte, Status, Fehler, Schema; Teil XI: die Karte.

Drei Lesewege

- *Den Entwurf lernen.* Die Teile I bis IV der Reihe nach, dann die durchgerechnete Ablaufverfolgung in Teil VI, dann Teil VII. Wir bearbeiten die Übungen und vergleichen erst danach mit den Lösungen am Ende des Buchs. Zu Teil V kehren wir zurück, wenn eine Ablaufverfolgung eine gemeinsame Tabelle oder einen geliehenen Slice erreicht.
- *Einen Einsatz betreiben.* Teil IX, dann Teil XI, mit Teil X zum Nachschlagen. Vor der Wahl einer Betriebsform lesen wir den „Funktionsvergleich“ in Teil II, und vor der Wahl der Worker- und Verbindungsgrenzen den „Vergleich der Gesamtprodukte“ in Teil VIII.
- *Den Code ändern.* Die Teile V und VI, die Quellverweise in jedem Kapitel und die SID-Dokumente, die sie zitieren. Teil VIII erklärt, welches Messprogramm eine Regression in dem erkennt, was wir geändert haben; Teil II nennt die Randbedingungen, die eine Änderung einhalten muss.

Konventionen

Ein *durchgerechnetes Beispiel* zeigt die Zwischenzustände, nicht nur die Antwort. Eine *Übung* fordert uns auf, eine Annahme zu ändern. Hinweise schlagen einen ersten Schritt vor; ausgewählte Lösungen am Ende des Buchs machen die Begründung überprüfbar. Diagramme unterscheiden den Anfragepfad von Hintergrundarbeit. Ein Quellverweis nennt die Implementierung, die wir untersuchen, wenn Text und Code einander zu widersprechen scheinen. Kästen mit der Überschrift „Implementierung“ nennen eine Funktion und ihre Datei; Kästen mit der Überschrift „Die Invariante erklären“ fordern uns auf, die Idee selbst zu erklären.

In Gleichungen ist b die Hashcash-Schwierigkeit in Bits, n die Tiefe der sequenziellen Arbeit, t die Zahl der Öffnungen, p die Erfolgswahrscheinlichkeit pro Versuch, K die Zahl der Versuche, T das Emissionsintervall eines Ratenbegrenzers, τ seine Bursttoleranz und Q eine Warteschlangenkapazität. Ein Symbol gilt nur in seinem Abschnitt, sofern nicht anders angegeben. Nanosekunden und Anfragen pro Sekunde in Teil VIII sind Messungen aus einem benannten Lauf. Zahlen in durchgerechneten Beispielen sind gewählte Eingaben, keine Benchmark-Aussagen. Statistische Modelle nennen ihre Annahmen, bevor sie Schlüsse ziehen.

TEIL I

Die Kosten einer Anfrage

Bevor wir einen Algorithmus wählen, bestimmen wir die Ressource, die er schützen soll. Dieses Kapitel baut die kleinen Modelle auf, an denen jede spätere Entwurfsentscheidung gemessen wird: die Arbeit des Ursprungsservers, die Rätselarbeit, die Amortisierung durch Sitzungen und den Speicher, den ein nicht vertrauenswürdiger Client den Server belegen lassen kann.

Lernziele Am Ende dieses Kapitels sollten wir die Ungleichung aufschreiben können, die entscheidet, ob sich ein Gate lohnt, angeben, was ein Proof of Work belegt und was nicht, Erwartungswert und Verteilungsende einer Hash-Suche herleiten, erklären, wie Sitzungen Arbeit amortisieren, und die fünf Invarianten nennen können, die der Rest des Buchs aufrechterhält.

Zulassung ist eine wirtschaftliche Entscheidung

Angenommen, ein Ursprungsserver führt für jede zugelassene Anfrage eine Datenbankabfrage aus und rendert eine Seite. Sei c_o diese Arbeit, c_g die Arbeit des Gates pro Anfrage und r die Ankunftsrate. Ohne Gate muss der Ursprungsserver rc_o Arbeitseinheiten pro Sekunde leisten. Ein Gate, das den Anteil a zulässt, ändert diesen Bedarf auf rac_o und wendet selbst rc_g auf. Das Gate ist nur nützlich, wenn die eingesparte Arbeit des Ursprungsservers seine eigenen Kosten und die Kosten übersteigt, die es legitimen Besuchern auferlegt.

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

Dies ist ein Kapazitätsmodell, keine Messung. Es lässt Bandbreite, Verbindungszustand, Speicherung und die Verzögerung beim Client weg. Sein Wert liegt darin, dass jeder fehlende Term sichtbar ist. Ein Cache-Treffer kann c_o klein machen; eine teure Abfrage kann es groß machen. Es gibt kein universelles Kostenverhältnis zwischen Anfragendem und Server.

Durchgerechnetes Beispiel: wann sich ein Gate lohnt Wir wählen $c_o = 100$ Arbeitseinheiten und $c_g = 1$. Wird die Hälfte der Anfragen abgewiesen, wendet das System mit Gate $1 + 0.5 \times 100 = 51$ Einheiten pro Ankunft statt 100 auf. Wird nur eine von tausend abgewiesen, wendet es 100.9 auf: In diesem Modell kostet das Gate mehr, als es einspart. Diese gewählten Zahlen sind keine Zeitmessungen. Sie zeigen, warum die Arbeitslast in jede Leistungsaussage gehört.

Was ein Proof of Work belegt

Ein Client-Rätsel belegt, dass jemand eine Eingabe gefunden hat, die eine öffentliche Prüfregel erfüllt. Es belegt weder Menschsein noch Identität noch gute Absicht. Ein Anfragender kann Rechenleistung mieten, eine gültige Sitzung innerhalb ihrer Lebensdauer wiederverwenden oder Arbeit auf mehrere Maschinen verteilen. Das Rätsel ändert die Kosten der Zulassung; Richtlinien und Prüfung entscheiden weiterhin, was zugelassene Anfragen tun dürfen.
Adressbasierte Limits ergänzen dies. Sie beschränken eine Adresse, aber gemeinsam genutzte Adressen können für viele Menschen stehen, und ein Anfragender kann viele Adressen nutzen. Weder ein Rätsel noch eine Adresse ist ein Identitätsorakel. Sibuna hält deshalb Zulassung, Prüfung und Reputation als getrennte Entscheidungen auseinander.

Hash-Suche als Zufallsvariable

Für einen idealen 256-Bit-Digest ergibt die Forderung nach b führenden Nullbits die Erfolgswahrscheinlichkeit $p = 2^{-b}$ für jeden unabhängigen Versuch. Sei K die Zahl der Versuche bis einschließlich des ersten Erfolgs. Dann gilt

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

Der Erwartungswert folgt aus der Summe über das Verteilungsende:

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

Eine Schwierigkeitsstufe mehr verdoppelt die erwarteten Versuche. Sie verspricht nicht, dass jedes Rätsel doppelt so lange dauert. Manche gelingen beim ersten Versuch; manche dauern weit länger als der Mittelwert. Ein Client, der nur eine Lösung misst, hat vor allem diesen Zufall gemessen.

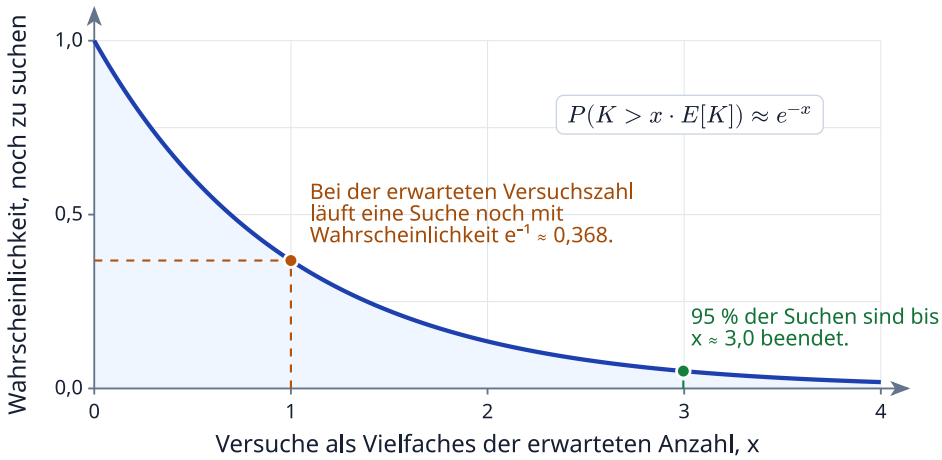


Abb. 1 Überlebenswahrscheinlichkeit der Hash-Suche: die Wahrscheinlichkeit, dass eine Suche nach dem x-Fachen der erwarteten Zahl von Versuchen noch läuft, in der Näherung für große Arbeit $P(K > x/p) \approx e^{-x}$.

Durchgerechnetes Beispiel: Erwartung ist keine Frist Bei $b = 16$ beträgt die erwartete Arbeit 65.536 Hashes. Die Wahrscheinlichkeit, nach so vielen Versuchen noch zu suchen, beträgt etwa $e^{-1} = 0.368$. Die Versuchszahl am 95. Perzentil ist $\lceil \ln(0.05) / \ln(1 - 2^{-16}) \rceil \approx 196327$, fast das Dreifache des Mittelwerts. Eine Benutzeroberfläche sollte diese Streuung tolerieren, statt zum erwarteten Fertigstellungszeitpunkt einen Fehlschlag zu melden.

Sitzungen amortisieren Arbeit

Angenommen, eine Sitzung erlaubt m Anfragen vor ihrem Ablauf, ein Rätsel kostet c_p, die Verifikation kostet c_v und eine Sitzungsprüfung kostet c_s. Ohne erfolglose Versuche betragen die amortisierten Kosten des Gates pro Anfrage $c_v/m + c_s$, während die Rätselkosten des Clients c_p/m betragen. Eine längere Sitzungsdauer hilft Menschen und automatisierten Clients gleichermaßen. Sie zu wählen ist eine Richtlinienentscheidung, keine kryptographische Optimierung.

Übung 1.1. Ein Client erhält 100 Anfragen pro Sitzung und löst ein Rätsel, das eine Million Versuche kostet. Wie groß ist die amortisierte Arbeit pro Anfrage? Was ändert sich, wenn er die Sitzung mit einem zweiten Prozess teilt?
Hinweis: Das Abrechnungsmodell von den tatsächlichen Bindungen des Tokens unterscheiden.

Begrenzter Zustand ist ein zweites Budget

Arbeit vom Anfragepfad wegzuverlagern lässt sie nicht verschwinden. Vorfälle treffen mit der Rate λ ein, der Speicherthread speichert sie mit der mittleren Rate μ, und die Warteschlange fasst Q Datensätze. Für $\lambda > \mu$ ergibt eine Fluidnäherung

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

Eine größere Warteschlange verschafft Zeit; sie schafft keinen Durchsatz. Bündelung amortisiert die Transaktionskosten, aber zurückgehaltene Datensätze brauchen weiterhin Speicher und müssen irgendwann abgearbeitet werden. Unter Überlast verwirft Sibuna neue Vorfallsdatensätze und zählt sie. Es trifft weiterhin Anfrageentscheidungen aus dem Zustand im Speicher. Der Betreiber muss die verlorenen Belege ebenso überwachen wie die HTTP-Erfolgsrate.

Übung 1.2. Eine Warteschlange fasst 512 Datensätze. Die Ankunftsrate beträgt 1.000/s und die Abflussrate 600/s. Wie lange kann eine anfangs leere Warteschlange den Überschuss aufnehmen? Warum ist die Antwort nur eine Näherung?
Hinweis: Die Differenz der Raten nutzen, dann Bursts und gebündelte Commits betrachten.

Die Invarianten, die wir mitnehmen

1. Nicht vertrauenswürdige Anfragen können keinen unbegrenzten Serverzustand erzeugen.
2. Ein gültiges Zulassungstoken umgeht keine Ablehnung durch die Anwendungsprüfung.
3. Ein Puffer bleibt am Leben, solange ein von ihm geliehener Slice existiert.
4. Eine dauerhafte Wiederholung kann einen Vorfall oder seine Wirkung auf die Reputation nicht vervielfachen.
5. Ein veröffentlichter Richtlinienschnappschuss bleibt unveränderlich, bis sein letzter Leser ihn freigibt.

Der Rest des Buchs leitet die Mechanismen her, die diese Aussagen wahr machen, und die Tests, die eine Verletzung aufdecken würden. Ein schneller Pfad ist nur nützlich, solange diese Aussagen wahr bleiben.

Die Invariante erklären. Die Gate-Ungleichung $c_g < (1 - a)c_o$ in Worten formulieren und dann erklären, warum eine gemessene Verifikationslatenz allein einem Betreiber nicht sagen kann, ob sich der Betrieb des Gates lohnt.

TEIL II

Stand der Technik und der Entwurfsraum

Wir ordnen Sibuna unter den Systemen ein, von denen es gelernt hat: Client-Rätsel und sequenzielle Arbeit aus der kryptographischen Literatur, signaturbasierte und semantische Anwendungsfirewalls, Proof-of-Work-Zwischenseiten und gehostete Edge-Netzwerke. Danach vergleichen wir die vier Werkzeuge, die ein Betreiber am ehesten gegeneinander abwägt, Funktion für Funktion, und halten fest, welche Fakten geprüft wurden.

Woher Sibuna kommt

Lernziele Am Ende dieses Kapitels sollten wir die Herkunftslinie der Client-Rätsel von 1992 bis zu den Konstruktionen für sequenzielle Arbeit von 2018 nachzeichnen, die drei Architekturideen nennen, die Sibuna verbindet, Sibuna, Anubis, SafeLine und die Cloudflare WAF in einem Entwurfsraum einordnen und angeben können, welche Vergleichsaussagen verifiziert und welche nur aus der Dokumentation übernommen wurden.

Client-Rätsel und sequenzielle Arbeit

Dwork und Naor schlugen in „Pricing via Processing“ (CRYPTO 1992) vor, dass ein Dienst eine mäßig schwere, leicht prüfbare Berechnung verlangt, bevor er Arbeit für einen Anfragenden leistet. Backs Hashcash (1997, ausgearbeitet 2002) machte das Rätsel zu einer partiellen Hash-Umkehrung, deren Schwierigkeit eine einzige ganze Zahl ist, und die Client-Rätsel von Juels und Brainard (NDSS 1999) wandten die Idee auf Verbindungsfluten an, bei denen sich der Server nichts über ein ungelöstes Rätsel merken darf. Alle drei teilen die Form, die Sibuna beibehält: eine vom Server gewählte Aussage, eine Lösung mit einstellbaren Kosten und eine Verifikation, die um Größenordnungen billiger ist als das Lösen.

Hash-Suche ist trivial parallelisierbar, daher wird ein Anfragender mit mehr Kernen früher fertig. Mahmoody, Moran und Vadhan (CRYPTO 2013) definierten Nachweise *sequenzieller* Arbeit, bei denen der Beweiser unabhängig von seiner Kernzahl eine lange Kette abhängiger Schritte ausführen muss, und Cohen und Pietrzak (EUROCRYPT 2018) gaben die einfache Hash-Graph-Konstruktion an, deren Verifizierer nur $O(t \log N)$ Arbeit braucht. Blocki, Lee und Zhou (ITC 2021) bewiesen die Korrektheit derselben Konstruktion gegen Quanten-Beweiser im Quanten-Random-Oracle-Modell. Teil III leitet Sibunas Stufe zwei aus dieser Forschungslinie her; SID 0006 hält die Annahmen fest.

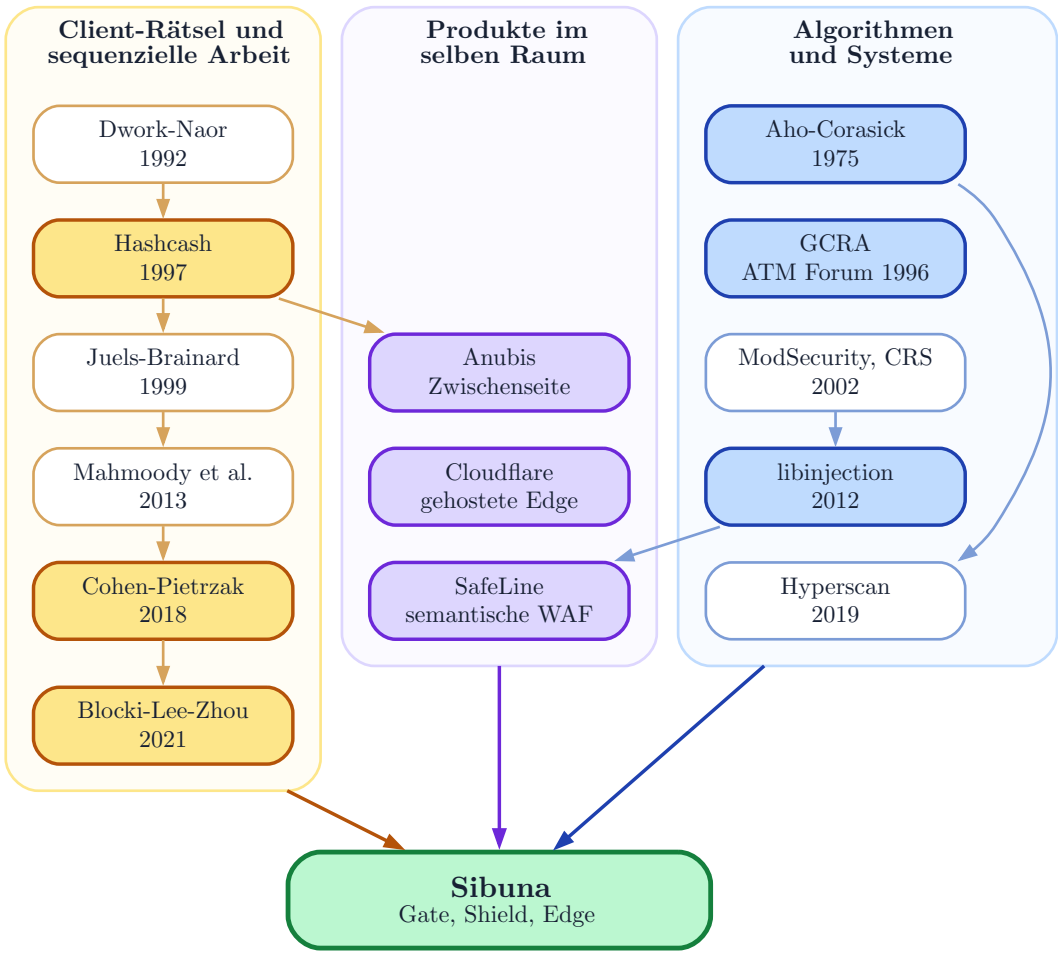


Abb. 2 Herkunftslinie der Ideen, die Sibuna verbindet. Die linke Spalte ist die Kryptographie der Client-Rätsel und der sequenziellen Arbeit, die rechte Spalte die Algorithmen und Systeme, auf denen Sibuna aufbaut, und die mittlere Spalte die Produkte im selben Raum; die Zeit läuft nach unten. Dünne Pfeile bedeuten direkte Abstammung, und die gefüllten Kästen sind die Ideen, die Sibuna direkt übernimmt. Ein früherer Speicherentwurf stützte sich auf Multi-Paxos (Lamport 1998; SID 0005); heute speichert Sibuna seinen Zustand in SQLite oder PostgreSQL (SID 0011).

Anwendungsfirewalls

Die erste Generation von Web Application Firewalls verglich Anfragebytes mit regulären Ausdrücken. ModSecurity (2002) und das OWASP Core Rule Set machten diesen Ansatz offen und weit verbreitet, und sie machten auch seine Kosten bekannt: Hunderte Backtracking-Muster pro Anfrage, von Hand abgestimmte Anomaliewerte und eine lange Reihe von Fehlalarmen bei gewöhnlichen Eingaben. libinjection (Hanson, 2012) ersetzte Musterlisten durch einen Tokenizer, der fragt, ob sich eine Zeichenkette als SQL *parsen* lässt, und senkte so Kosten und Fehlalarme. Chaitins SafeLine überträgt diese semantische Idee auf ein vollständiges Produkt, und Hyperscan (Wang et al., NSDI 2019) zeigte, wie weit sich vektorisierte Literalsuche bei langen Eingaben treiben lässt. Teil VI erklärt, warum Sibuna für die kurzen Felder, die es

prüft, einen dichten Automaten plus kleine strukturelle Tokenizer gewählt hat, statt regulärer Ausdrücke oder einer SIMD-Engine.

Zwischenseiten und Edges

Die Proof-of-Work-Zwischenseite vor einer Website wurde 2025 verbreitet, als der Datenverkehr von Scrapern für das Training von Sprachmodellen den Betrieb gewöhnlicher Ursprungsserver unbezahlbar machte. Anubis (Techaro) ist die bekannteste offene Implementierung: ein Reverse Proxy in Go, der ein JavaScript-Rätsel ausliefert, ein Sitzungstoken signiert und den Datenverkehr ansonsten durchreicht. Sibunas Betriebsform Gate löst dasselbe Problem mit einer zustandslosen Challenge, einem symmetrischen Token und einer zweiten Arbeitsstufe; der folgende Vergleich hält fest, wo sich die beiden unterscheiden.

Am anderen Ende der Skala verteilen gehostete Edge-Netze wie Cloudflare Prüfung, Ratenbegrenzung und Bot-Bewertung auf Hunderte von Points of Presence mit einer globalen Steuerungsebene. Sibunas Edge-Einsatz übernimmt diese Form (eine gemeinsame Richtlinien- und Reputationsebene, die lokale, unveränderliche Schnappschüsse speist), bleibt aber selbst gehostet. Deshalb ist die Speicherschicht eine Datenbank, die der Betreiber selbst betreibt (eine SQLite-Datei auf einem Knoten, eine gemeinsame PostgreSQL-Datenbank für einen Cluster), gelesen von einem Speicherthread abseits des Anfragepfads, und kein Dienstaufruf.

Drei Ideen in einer Binärdatei

Sibuna verbindet eine Arbeits-Challenge im Browser für die Zulassung, eine begrenzte Prüfung der Anwendungsnutzlast und eine gemeinsame Steuerungsebene, die lokale Richtlinienschnappschüsse veröffentlicht. Gate und Shield sind die beiden Betriebsformen des Produkts; Speicherung und verteilter Einsatz sind für beide optional.

Eigenschaft	Gate	Shield
Zulassung	Native Verifikation von Hashcash oder sequenzieller Arbeit	Ebenso
Sitzung	BLAKE3 mit Schlüssel; optional Ed25519	Ebenso
Lokale Kontrollen	Regeln, CIDRs, GCRA, Sperren	Ebenso
Prüfung	Deaktiviert	Begrenzte Signaturen und strukturelle Tokenizer
Zustand	Optional persistente Richtlinien und Reputation, im Cluster gemeinsam genutzt	Zusätzlich Prüfforensik

Der Browser-Löser wird aus demselben Quellcode für Nachweise kompiliert wie der native Verifizierer. Dichte Literalsuche vermeidet Backtracking und hält die Sucharbeit linear in der Eingabelänge. Unveränderliche Richtlinienschnappschüsse entkoppeln die Anfragebearbeitung von SQL und von der Verfügbarkeit der Datenbank. Keine dieser Entscheidungen begründet

eine allgemeine algorithmische Optimalität: Kurze Header, lange Bodys, Mustervielfalt, Kernzahl und Cachegröße verschieben die Abwägungen.

Übernommene Entwurfsbeschränkungen

1. Keine Heap-Allokation auf dem Anfragepfad; jede Tabelle hat eine feste Kapazität und ein festgelegtes Verhalten bei Sättigung.
2. Das Ausstellen einer Challenge schreibt nichts; nur eine bezahlte Lösung belegt Speicher.
3. Eine Sitzung lässt zu; sie befreit eine Anfrage nie von der Prüfung oder von einer expliziten Ablehnung.
4. Jede genannte Zahl ist entweder von einem eingecheckten Messprogramm gemessen oder als Modell gekennzeichnet.
5. Kryptographische Entscheidungen zitieren eine Veröffentlichung und einen Beweis und nennen ihre Haltung gegenüber Quantenrechnern.

Funktionsvergleich

Die vier folgenden Tabellen vergleichen Sibuna mit den drei Werkzeugen, die am häufigsten neben ihm genannt werden. Die Angaben zu den anderen Produkten haben wir im September 2026 ihrer öffentlichen Dokumentation und ihren Release-Artefakten entnommen: Anubis 1.27.0 (MIT-Lizenz, Go), SafeLine Community Edition 9.x (Verwaltungscode unter GPL-3.0 um geschlossene Detektor-Images, Docker Compose) und die Entwicklerdokumentation der Cloudflare WAF. Anubis lief außerdem auf dem Benchmark-Host, SafeLine und Cloudflare nicht (Teil VIII erklärt, warum). Ein Strich bedeutet, dass das Produkt die Funktion nicht anbietet; ein Tarifname bedeutet, dass es die Funktion nur in diesem Tarif gibt. Jede Messung von Sibuna nennt die eingecheckte Ergebnisdatei, aus der sie stammt. Die Zeile zum TLS-Fingerabdruck haben wir im Oktober 2026 geprüft, anhand des Changelogs von Anubis (1.21.0 und 1.26.0), der Release Notes von SafeLine (8.10.0 und 9.2.0) und der Cloudflare-Seite „JA3/JA4 fingerprint“.

Tabelle 3 Funktionsvergleich, Teil 1: Einsatz

Funktion	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
Läuft als	Eine Binärdatei; Reverse Proxy oder Forward-Auth; optional eine SQLite-Datei oder eine gemeinsame PostgreSQL-Datenbank	Eine Go-Binärdatei; Reverse Proxy oder Forward-Auth	Sieben Docker-Container (tengine, detector, mgt, luigi, fvm, chaos, PostgreSQL)	Gehostetes Netzwerk; DNS zeigt auf Cloudflare
Sprache, Laufzeit	Rust, kein Garbage Collector; ohne	Go, mit Garbage Collection	Proxy und Detektor in C/C+	Proprietäre Edge-Software

Funktion	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
	Speicherung kein externes Crate		+, Verwaltung in Go, PostgreSQL	
Lizenz	LGPL-3.0-or-later	MIT	GPL-3.0-Verwaltungscode; geschlossene Detektor-Images	Proprietärer Dienst
Ressourcenbedarf auf dem Host	35,7 MiB Binärdatei (`benchmarks/results/comparison-latest.json`); 5,9 MiB RSS im Leerlauf (`benchmarks/results/primitives-latest.json`); höchstens 16,0 MiB Spitzen-RSS unter Last (`benchmarks/results/product-latest.md`)	38,1 MiB Binärdatei; 21 MiB RSS im Leerlauf; Linux-Testaufbau, Konsole aus	Mindestens 1 CPU-Kern, 1 GB RAM, 5 GB Festplatte, Docker 20.10+	Nichts vor Ort

Tabelle 4 Funktionsvergleich, Teil 2: Zulassung und Sitzungen

Funktion	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
Proof-of-Work-Zulassung	Hashcash (auf Bitebene) und sequenzielle Arbeit nach Cohen-Pietrzak, pro Regel gewählt; WebAssembly mit JavaScript-Fallback	SHA-256-Hashcash in Hex-Nibbles (`fast`, `slow`); Challenges ohne Arbeit `metarefresh` und `preact`	JavaScript-Anti-Bot-Challenge und CAPTCHA; nicht an Arbeit gebunden	Managed Challenge, JS Challenge, Turnstile; der Bot Fight Mode stellt eine Rechen-Challenge
Challenge-Zustand auf dem Server	Keiner bis zur Lösung; gelöste Tags in einer festen Robin-Hood-Tabelle	Speicher-Backend: Arbeitsspeicher, bbolt, Valkey oder S3	Vom Stack verwaltet	Von Cloudflare verwaltet

Funktion	**Sibuna**	**Anubis 1.27**	**SafeLine CE 9**	**Cloudflare WAF**
Sitzungstoken	Tag aus BLAKE3 mit Schlüssel, 75 Zeichen langes Cookie, an die bezahlte Arbeit gebunden; Ed25519 optional	JSON Web Token, mit Ed25519 signiert; HS512 optional	Vom Stack ausgestelltes Cookie	Cookie `cf_clearance`
Post-Quanten-Sicherheit des Standardtokens	Symmetrisch: Es gilt nur Grovers quadratische Beschleunigung	Ed25519: durch Shors Algorithmus gebrochen	Nicht dokumentiert	Nicht dokumentiert
Richtliniensprache	JSON-Regeln: Pfad, User-Agent, Header, IPv4/IPv6-CIDR, JA4-Fingerabdruck; ALLOW, DENY, CHALLENGE, WEIGH mit Schwellen; dynamische Regeln aus der Datenbank	YAML/JSON-Bots: Regex auf UA, Pfad, Header; CIDR; CEL-Ausdrücke; WEIGH mit Schwellen	In der Konsole definierte eigene Regeln, ACLs, IP-Gruppen	Ausdrücke der Rules Language; *5 / 20 / 100 / 1000 eigene Regeln je nach Tarif*
TLS-Fingerabdruck (JA4)	Regeln vergleichen einen JA4-Fingerabdruck exakt oder per Präfix; ein vertrauenswürdiger Eingangsproxy berechnet ihn und leitet ihn in einem Header weiter (`--ja4-header`); Sibuna berechnet ihn nicht	Nicht dokumentiert; Richtlinien können sich auf den HTTP-Fingerabdruck JA4H beziehen, den Anubis berechnet und als Header `X-Http-Fingerprint-JA4H` hinzufügt	Bei der eigenen TLS-Terminierung berechnet; in den Angriffsdetails angezeigt; eigene Regeln vergleichen ihn (gleich, ungleich); eine eingebaute Regel lehnt Fingerabdrücke ab, die die eigenen JA4-Bedrohungsdaten als bösartig führen	JA3- und JA4-Fingerabdrücke in eigenen WAF-Regeln und in der Analyse *(Enterprise mit Bot Management)*

Tabelle 5 Funktionsvergleich, Teil 3: Prüfung und Zustand

Funktion	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
Prüfung auf SQLi, XSS, RCE, Path Traversal	Shield: markierter Automat plus strukturelle Tokenizer, erste 8 KB des Bodys	-	Semantische Analyse-Engine über OWASP-Kategorien, außerdem SSRF, XXE, CRLF	Managed Rulesets *(ab Pro)*; OWASP Core Ruleset; Attack Score *(ab Business)*
Ratenbegrenzung	GCRA pro Client, lokal auf einem Knoten	- (die Systemlast steht CEL-Regeln zur Verfügung)	Pro IP, Pfad, Sitzung	*1 / 2 / 5 / 100 Regeln je nach Tarif*; unterhalb von Enterprise nur IP-Schlüssel
Sperren und Reputation	Honeypot-Sperren; Reputations-Trie, geladen aus der im Cluster gemeinsam genutzten Reputationstabelle	DNSBL; ASN und GeoIP über Thoth *(kostenpflichtig, geschlossen)*	IP-Gruppen und Sperrlisten; Threat Intelligence *(Pro)*	IP-Listen; verwaltete Listen und Bot-Score *(Enterprise)*
Forensik	SQLite oder PostgreSQL: Volltextsuche über Vorfälle, Kampagnen-Clustering über Vektoren	Nur Prometheus-Metriken	PostgreSQL-Angriffsprotokoll mit Konsole	Security Events; im Free-Tarif stichprobenartig
Mehrere Knoten	Eine PostgreSQL-Datenbank, die alle Knoten für Richtlinien und Reputation teilen; gemeinsamer Seed; Challenge-Verifikation am ausstellenden Knoten	Gemeinsamer Signaturschlüssel; gemeinsamer Valkey-Speicher	Ein Stack pro Host	Globales Anycast

Tabelle 6 Funktionsvergleich, Teil 4: Betrieb

Funktion	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
Metriken	Prometheus unter `/__sibuna/metrics`	Prometheus auf einem separaten Port	Konsolen-Dashboards	Dashboard und Analysen
TLS-Terminierung	- (Eingangsproxy)	- (Eingangsproxy)	**Ja**	**Ja**
Verwaltungskonsole	Opt-in; SID 0007 im Status Committed	-	**Ja**	**Ja**
Auf dem Benchmark-Host gemessen	**Ja** (Teil VIII)	**Ja**	- (nur Docker)	- (gehostet)

Drei Unterschiede tragen bei einer Auswahlentscheidung das meiste Gewicht.

- *Wohin die Arbeit geht.* Anubis und Sibuna Gate lassen den *Client* zahlen, bevor der Ursprungsserver irgendetwas tut. SafeLine und Cloudflare prüfen zuerst und stellen gezielt Challenges; ihre Grundhaltung ist zulassen und filtern. Sibuna Shield tut beides, in dieser Reihenfolge: Die Prüfung kann eine Anfrage ablehnen, die bereits eine gültige Sitzung besitzt.
- *Was sich der Server merkt.* Anubis hält den Challenge-Zustand in einem Speicher (Arbeitsspeicher, bbolt, Valkey oder S3) und signiert ein JSON Web Token mit Ed25519; Sibuna stellt einen selbstauthentifizierenden Datensatz aus und merkt sich nur gelöste Tags in einer festen Tabelle. SafeLine hält Angriffsprotokolle in PostgreSQL; Cloudflare hält alles, mit stichprobenartiger Sichtbarkeit im Free-Tarif.
- *Was ein Betreiber betreiben muss.* Sibuna ist eine Binärdatei mit einer optionalen SQLite-Datei oder einer gemeinsamen PostgreSQL-Datenbank für einen Cluster; Anubis ist eine Go-Binärdatei; SafeLine besteht aus sieben Containern hinter einem Docker-Daemon auf einem Linux-Host mit mindestens einem Kern, einem Gigabyte Arbeitsspeicher und fünf Gigabyte Festplatte; Cloudflare ist eine DNS-Änderung und ein Abonnement.

Aktuelle Grenzen des Produkts Sibuna überlässt die TLS-Terminierung am Eingang dem Proxy der jeweiligen Installation, bewertet Bots nicht mit einem trainierten Modell und veröffentlicht keine kostenpflichtigen Signaturen. Es wertet den JA4-Fingerabdruck aus, den ein vertrauenswürdiger Eingangsproxy berechnet; selbst berechnen kann es ihn nicht (SID 0014, Teil IX). Seine optionale Konsole bietet authentifizierte Dashboards, Anreicherung mit Länderdaten, Bearbeitung von Richtlinien, Untersuchung und Clusterverwaltung. Länderabfrage und Speicherarbeit laufen außerhalb der Anfrageklassifikation. SID 0007 hat den Status Committed, und die Konsole besteht ihre Browser-Workflows (SID 0010). Ihr formales Abnahmekriterium für die Leistungsisolation, mit acht aktiven Dashboards unter Last, ist noch nicht gemessen; Teil VIII hält fest, was gemessen ist.

Übung 2.1. Erklären, warum es Protokolldrift verringert, eine einzige Verifizierer-Quelle zu nativem Code und zu WebAssembly zu kompilieren. Welche Eigenschaften erfordern dennoch unabhängige Tests oder eine kryptographische Begutachtung?

Übung 2.2. Nur anhand der Vergleichstabellen den kleinsten Einsatz aufschreiben, der einer statischen Website (a) ein Proof-of-Work-Gate, (b) eine Prüfung auf SQL-Injection und (c) eine clusterweite Sperre bietet. Welche Zeilen haben jede Wahl erzwungen? Hinweis: Bei den Zeilen „Läuft als“ und „Mehrere Knoten“ beginnen.

Die Invariante erklären. Beschreiben, wie Gate, Shield und der optionale gemeinsame Speicher Zulassung, Prüfung und Richtlinienverteilung aufteilen. Dann eine Zeile der Vergleichstabellen nennen, die wir erneut verifizieren würden, bevor wir sie einem Kunden zitieren, und sagen, wie wir sie verifizieren würden.

TEIL III

Grundlagen aus der kryptographischen Forschung

Wir leiten die Kostenmodelle hinter Sibunas zwei Proof-of-Work-Stufen her, stellen den Nachweis sequenzieller Arbeit nach Cohen-Pietrzak mit seinem Sicherheitsargument vor und begründen die Authentifizierung von Challenges und Sitzungen mit schlüsselbasierten Hashes. SID 0006 hält die Annahmen und Quellen fest.

Hashcash: Die Mathematik von Stufe eins

Lernziele Am Ende dieses Kapitels sollten wir die erwartete Arbeit, die Varianz und die Wahrscheinlichkeit am Verteilungsende eines Hashcash-Rätsels auf Bitebene angeben, erklären, warum der Prüfaufwand des Servers den Vorteil des Angreifers begrenzt, und die beiden Grenzen beschreiben können, die eine zweite Stufe begründen.

Definition und Kosten

Sibunas Rätsel der Stufe eins für die Challenge-Zeichenkette C und die Schwierigkeit b Bits lautet: Gesucht ist ein $N \in \mathbb{N}$, sodass SHA-256 $(C \parallel ":" \parallel \operatorname{dec}(N))$ b führende Nullbits hat. Die dezimale Darstellung der Nonce ist Absicht; sie hält das Übertragungsformat zwischen dem WebAssembly-Löser, dem JavaScript-Fallback und dem nativen Verifizierer identisch, ohne einen Schritt binärer Kodierung.

Die Einheit der Schwierigkeit ist ein erwarteter *Hash-Versuch*, kein Aufruf der Kompressionsfunktion. Eine Eingabe kann mehrere SHA-256-Blöcke belegen. Eine Vorberechnung des Präfixes kann zudem die Arbeit pro Nonce verändern. Für die Übertragung maßgeblich bleibt das Prädikat über den Digest; Implementierungen müssen sich auf die Aussage und die Kodierung der Nonce einigen, nicht auf geschätzte CPU-Kosten.

Für unabhängige ideale Digests ist die Versuchszahl X geometrisch verteilt mit $p = 2^{-b}$. Ihr Median ist $\lceil \ln(1/2) / \ln(1-p) \rceil$, für kleine p ungefähr $2^b \ln 2$, und ihre Varianz ist $(1-p)/p^2$. Der Verifizierer prüft einen einzigen Kandidaten-Digest, statt die Suche auszuführen. Parsen, Authentifizierung der Challenge, Prüfung des Fingerabdrucks, Eintrag in den Verbrauchszustand und die Antwort sind zusätzliche Arbeit. Eine Zeitmessung einer Primitive lässt sich nicht in die Flutkapazität eines Servers umrechnen.

Zwei Grenzen der Suche

Die erste ist die Varianz: Eine lange Suche bedeutet nicht unbedingt einen defekten Worker. Die zweite ist die Parallelität: Unabhängige Nonce-Versuche können gleichzeitig laufen. Mehr Kerne verkürzen die verstrichene Suchzeit, auch wenn die erwartete Gesamtzahl der Versuche gleich

bleibt. Der Hardwaredurchsatz ist deshalb für die Zulassungsrichtlinie von Belang. Weder eine behauptete Hashrate noch ein einzelner Benchmark kann den Vorteil jedes Anfragenden begrenzen.

Übung 3.0. Mit dem geometrischen Verteilungsende die Wahrscheinlichkeit abschätzen, mehr als 4.6×2^b Versuche zu benötigen. Warum bleibt die erwartete Versuchszahl unverändert, wenn der Nonce-Raum auf mehrere Worker aufgeteilt wird?
Hinweis: Die Gesamtzahl der Versuche zählen, nicht die verstrichenen Sekunden.

Nachweis sequenzieller Arbeit: Stufe zwei

Lernziele Verstehen, wie die Beschriftung nach Cohen-Pietrzak funktioniert, warum sie sequenzielle Berechnung erzwingt und was der Verifizierer prüft; die bedingte Stichprobenschranke $(1-\alpha)^t$ und ihre Grenzen, das Post-Quanten-Ergebnis, die Speicherschranke des Beweisers und die Kalibrierungsdaten hinter der Zuordnung der Arbeitsbits kennen.

Warum sequenzielle Arbeit

Eine Konstruktion für sequenzielle Arbeit führt Abhängigkeiten zwischen Berechnungen ein: Ein späteres Label benötigt einen früher erzeugten Wert. Damit geht es um eine andere Größe als die Gesamtarbeit, nämlich um die Tiefe der Berechnung. Unabhängige Rätsel können weiterhin parallel laufen, und die Hash-Latenz der Hardware spielt weiterhin eine Rolle.
Sibuna folgt der hashbasierten Konstruktion, die Cohen und Pietrzak beschrieben haben. Ihr Graph hat bei Tiefe n genau $2^{n+1}-1$ Labels, aber eine feste Zahl von Labels ist keine deterministische Browserlatenz: Labels können mehrere Kompressionsblöcke umfassen, Öffnungen können Teilbäume neu berechnen, und das Scheduling fügt Schwankungen hinzu. Sicherheitsaussagen aus Analysen im Orakelmodell dürfen nicht zu einer uneingeschränkten Behauptung über jede konkrete Implementierung oder jeden Gegner gemacht werden.

Die Konstruktion

Sei n die Tiefe, und betrachten wir den vollständigen Binärbaum, dessen Knoten durch Binärzeichenketten der Länge höchstens n benannt sind (Wurzel ε, Kinder $v0$ und $v1$, Blätter der Länge n). Jeder Knoten v trägt ein 32-Byte-Label $\ell_v = H(\chi, v, \text{parents}\ (v))$, wobei χ die Challenge-Aussage ist und H SHA-256 mit einmal absorbiertem χ (derselbe Präfix-Trick wie beim Hashcash-Löser):

- Ein innerer Knoten hasht seine beiden Kinder: $\ell_v = H(\chi, v, \ell_{v0}, \ell_{v1})$;
- ein Blatt $u = u_1 \ldots u_n$ hasht die *linken Geschwister seiner Vorfahren*, wo immer der Pfad von der Wurzel nach rechts abgebogen ist: $\ell_u = H(\chi, u, \{\ell_{u_1 \ldots u_{i-1} 0} : u_i = 1\})$.

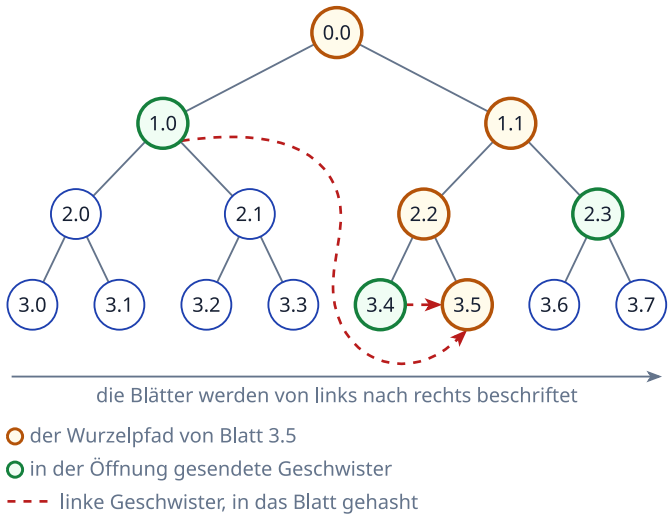


Abb. 3 Der Baum der Tiefe 3: Eine Öffnung von Blatt 3.5 sendet das Blattlabel und die drei Geschwisterlabels; die gestrichelten Kanten sind die Abhängigkeiten von linken Geschwistern, die die Beschriftung sequenziell machen.

Die Kanten zu linken Geschwistern machen die Berechnung sequenziell: Blatt u kann erst beschriftet werden, wenn jeder Teilbaum links davon fertig ist. Damit ist eine Postorder-Tiefensuche erzwungen, und das Wurzellabel $\varphi = \ell_\varepsilon$ hängt der Reihe nach von allen $2^{n+1} - 1$ Labels ab.

Öffnungen. Nach der Berechnung von φ leitet der Beweiser per Fiat-Shamir t Blätter ab, $\gamma_i = H(\chi \parallel \varphi \parallel i) \bmod 2^n$, und sendet für jedes das Blattlabel und die n Geschwisterlabels entlang seines Pfads zur Wurzel. Der Verifizierer berechnet das Blatt aus den Geschwistern, die seine Eltern sind, neu, hasht den Pfad aufwärts und prüft, dass er bei φ ankommt. Die Kosten des Verifizierers betragen $O(tn)$ Labelberechnungen mit begrenztem Arbeitsbereich. Eine Labelberechnung kann mehrere Hash-Blöcke umfassen; Labels zu zählen heißt nicht, Kompressionsaufrufe zu zählen.

Implementierung: `posw::verify_opening` Berechnet ein geöffnetes Blatt aus seinen Geschwisterlabels neu und läuft den Pfad zur Wurzel hinauf. In `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` läuft von Tiefe n bis Tiefe 1; umgekehrt passt es zu den Tiefen
1..=n.
    let mut hasher = node_hasher(base, n, gamma);
    for (d, sibling) in (1..=n).zip(siblings.iter().rev()) {
```

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

Stichproben, Korrektheit und ihre Grenzen

Betrachten wir ein festes Commitment mit einem unabhängig erkennbaren fehlerhaften Anteil α. Zieht jede Öffnung gleichverteilt und unabhängig, so beträgt die Wahrscheinlichkeit, in t Öffnungen jede fehlerhafte Stelle zu verfehlen, $(1-\alpha)^t$. Diese elementare Rechnung erklärt, warum mehr Öffnungen helfen. Für sich genommen ist sie kein Sicherheitsbeweis für ein Protokoll sequenzieller Arbeit.

Durchgerechnetes Beispiel: eine bedingte Stichprobenschranke Mit einem festen fehlerhaften Anteil $\alpha = 0.1$ und $t = 16$ unabhängigen Öffnungen beträgt die Wahrscheinlichkeit, jede fehlerhafte Stelle zu verfehlen, $0.9^{16} \approx 0.185$. Mit 32 Öffnungen sind es etwa 0,034. Die Voraussetzung legt das Commitment fest, bevor die Stichproben gewählt werden. Kann ein Beweiser viele Wurzeln durchprobieren, wählen, welches Commitment er offenlegt, oder Fehlschläge korrelieren, so berücksichtigt diese Rechnung diese Strategie nicht.

Die zitierte Konstruktion für sequenzielle Arbeit hat eine Sicherheitsanalyse in einem Orakelmodell. Sibunas Domänentrennung, Parameterzuordnung, Kürzung und Implementierung müssen dennoch gegen die Annahmen der Konstruktion geprüft werden. Fiat-Shamir-Stichproben machen Öffnungen aus einem Commitment reproduzierbar; Reproduzierbarkeit allein beweist keine Resistenz gegen Durchprobieren. Wir behaupten nicht bedingungslos, dass Betrug nie billiger ist.

Übung 3.1. Ein Angreifer probiert g unabhängige Commitments, die jeweils mit Wahrscheinlichkeit q akzeptiert werden. Die Wahrscheinlichkeit herleiten, dass mindestens eines akzeptiert wird. Welche Kosten fehlen in diesem Ausdruck?
Hinweis: Zuerst die Wahrscheinlichkeit berechnen, dass alle g scheitern.

Speicher des Beweisers

Ein naiver Beweiser speichert jedes Label ($2^{n+1} - 1$ Stück). Sibunas Beweiser hält im ersten Durchlauf einen Stack der Labels der linken Geschwister des aktuellen Pfads ($n + 1$ Labels) und behält nur die obersten $m = \min(n, 10)$ Ebenen ($2^{m+1} - 1$ Labels, höchstens 64 KB). Um ein Blatt zu öffnen, berechnet er den Teilbaum des Vorfahren des Blatts auf Tiefe m neu, $2^{n-m+1} - 1$ Hashes, und füllt den Stack aus den behaltenen Ebenen vor. Die Gesamtkosten des Beweisers betragen $2^{n+1} - 1 + t\,(2^{n-m+1} - 1)$ Hashes, und der gesamte Arbeitsbereich einschließlich des ausgegebenen Nachweises liegt unter 100 KB: Ein Browser-Tab und ein nativer Test teilen denselben Typ `Workspace` (`sibuna_pow::posw::Workspace`).

Parameter sind keine Zeitangaben

Sibuna bildet Arbeitsbits mit $n = b - 3$ auf die Tiefe ab, begrenzt auf das unterstützte Intervall $[4, 24]$. Dies ist eine Konvention für die Arbeitsskala, keine Garantie gleicher Echtzeit zwischen den Algorithmen. Ein Blatt kann mehrere Elternlabels hashen, während Hashcash wiederholt eine Nonce testet. Compiler, Browser, Prozessor und Parameterwahl beeinflussen alle ihre relativen Kosten. Der eingecheckte Primitive-Benchmark misst die Verifikation; er belegt keine Lösungslatenz im Browser.
Die Nachweisgröße beträgt $32(1 + t(n + 1))$ Bytes. Bei Tiefe 13 mit sechzehn Öffnungen sind das $32(1 + 16 \times 14) = 7200$ Bytes. Mehr Öffnungen vergrößern sowohl den Nachweis als auch die Arbeit des Verifizierers, selbst wenn die dominierenden Kosten für den Aufbau des Baums gleich bleiben.

Übung 3.2. Die Nachweisgröße bei Tiefe 17 mit 16 Öffnungen berechnen. Dann die Zahl der Öffnungen verdoppeln. Welcher Term verdoppelt sich, und welcher bleibt fest?

Eine Konstruktion wählen

Der maßgebliche Vergleich ist ein Ressourcenbudget, nicht der Ruf eines Algorithmus. Wir halten den Speicher des Beweisers, die Arbeit des Verifizierers im schlechtesten Fall, die Nachweisbytes und die Annahmen fest, die für die behauptete Eigenschaft nötig sind. Speicherharte Suche kann nützlich sein, aber ein Verifizierer, der teure Arbeit wiederholen muss, gibt einem Angreifer eine weitere Stelle, an der er die Ressourcen des Servers verbrauchen kann. Öffentlich verifizierbare Verzögerungskonstruktionen dienen einem anderen Vertrauensmodell als ein Gate mit gemeinsamem Geheimnis.
Sibuna behält die billige Hash-Verifikation und eine hashbasierte Option für sequenzielle Arbeit. Letztere erfordert mehr Nachweisbytes und eine sorgfältigere Parameteranalyse. Keine der beiden Entscheidungen erspart es, fehlerhaft geformte Nachweise abzuweisen, bevor die teure Verifikation beginnt.

Die Invariante erklären. Den Unterschied erklären zwischen einem Authentifizierungsbaum, der beweist, dass ein Label zu einem Commitment gehört, und Abhängigkeitskanten, die einschränken, wie Labels berechnet werden.

Symmetrische Authentifizierung: Challenges und Sitzungen

Lernziele Erklären, warum eine Pseudozufallsfunktion mit Schlüssel die richtige Primitive ist, wenn Aussteller und Verifizierer derselben Vertrauensdomäne angehören; die Fälschungsschranke für ein 128-Bit-Tag angeben; den zustandslosen Challenge-Datensatz und die Schranke der Verbrauchsmenge beschreiben; den Schlüsselplan lesen.

Aussteller gleich Verifizierer

Eine digitale Signatur lässt *jeden*, der den öffentlichen Schlüssel besitzt, verifizieren; das ist das richtige Werkzeug, wenn Verifizierer selbst keine Tokens ausstellen dürfen. Sibunas Verifizierer *ist* der Aussteller (oder ein Cluster, der einen Seed teilt). Ein Message Authentication Code, gebaut aus einer Pseudozufallsfunktion, liefert Authentifizierung innerhalb dieser Vertrauensdomäne. Öffentliche Verifikation liefert er nicht: Wer den MAC-Schlüssel besitzt, kann auch Tokens ausstellen. Das Benchmark-Kapitel vergleicht die implementierten Token-Operationen.

Lemma 3 (Fälschungsschranke) Wird BLAKE3 im Schlüsselmodus als PRF modelliert und ein 16-Byte-Tag verwendet, so fälscht ein Gegner mit q Verifikationsanfragen ein gültiges Token mit Wahrscheinlichkeit höchstens $q \cdot 2^{-128} + \varepsilon_{\mathrm{PRF}}$. Bei einer Million Anfragen pro Sekunde sind das 2^{-108} pro Sekunde.

Eine zweite Folge ist auf lange Sicht wichtig: Ed25519 beruht auf diskreten Logarithmen in einer Gruppe elliptischer Kurven, die Shors Algorithmus bricht; ein schlüsselbasierter Hash beruht nur auf der Hashfunktion, gegen die Quantenalgorithmen höchstens eine quadratische Beschleunigung bringen. Sibuna behält Ed25519 als Option (`--token-scheme ed25519`) für Einsätze, die Verifizierer brauchen, die keine Tokens ausstellen können.

Der Schlüsselplan

Aus einem 32-Byte-Master-Seed (aus `--secret-file`, `SIBUNA_SECRET` oder einem frischen Zufallswert) werden mit BLAKE3 mit Schlüssel und einer Domänenzeichenkette alle Zweckschlüssel abgeleitet: die Schlüssel für Token, Challenge, Ed25519 und Fingerabdruck sowie der Anforderungsschlüssel, der Anforderungstickets versiegelt (Teil IV).

Implementierung: `Keys::derive` Leitet die fünf Zweckschlüssel aus dem Seed ab. In `crates/challenge/src/keys.rs` (Crate `sibuna-challenge`). `Keys` und `Seed` überschreiben ihre Bytes beim Freigeben mit Nullen, und ihre `Debug`-Ausgabe ist geschwärzt.

```
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())
}
```

In einem Cluster ersetzt `Keys::bind_node` den Challenge-Schlüssel durch BLAKE3 mit diesem Schlüssel über `sibuna/challenge-node/v1` und die Knoten-ID. So muss eine Challenge zu dem Knoten zurückkehren, der sie ausgestellt hat, während Sitzungstokens und Anforderungstickets auf jedem Mitglied gültig bleiben.
Domänentrennung verhindert die versehentliche Wiederverwendung über Zwecke hinweg; die Kompromittierung des Master-Seeds kompromittiert jeden abgeleiteten Schlüssel. Ein Cluster muss sich nur auf einen Seed einigen.

Zustandslose Challenges

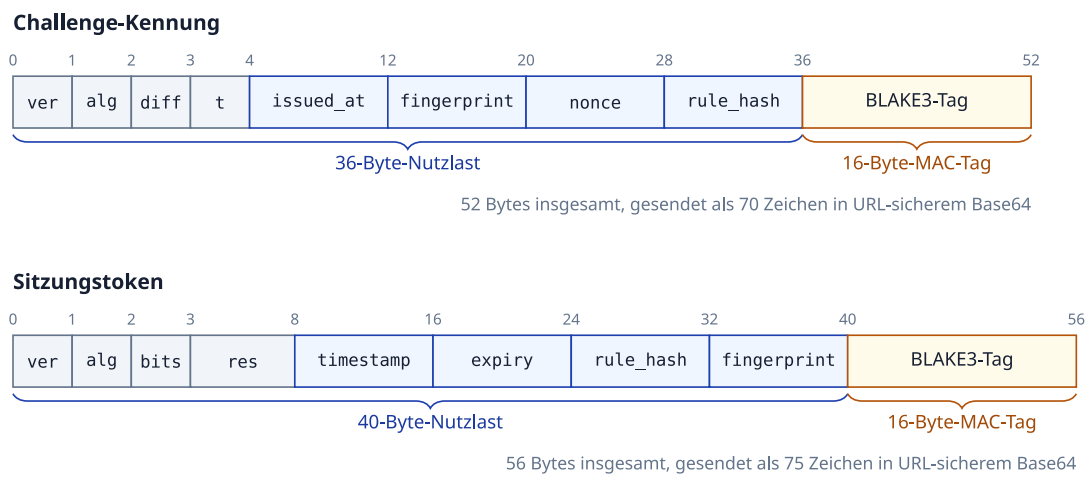


Abb. 4 Übertragungsformate der Challenge-Kennung und des Sitzungstokens. Die Zahlen über jedem Datensatz sind Byte-Offsets.

Eine Challenge-Kennung ist ein selbstauthentifizierender Datensatz: Version, Algorithmus, Schwierigkeit, Zahl der Öffnungen, Ausgabezeit, der schlüsselbasierte Fingerabdruck des

Clients, eine per PRF abgeleitete Nonce und der Hash der Richtlinienregel, die die Challenge verlangt hat, gefolgt von einem 16-Byte-Tag unter dem Challenge-Schlüssel. Das Ausstellen schreibt nichts. Die Verifikation dekodiert den Datensatz, prüft das Tag in konstanter Zeit, prüft das Alter gegen die Challenge-TTL, prüft den Fingerabdruck gegen den einreichenden Client und untersucht erst dann den Nachweis.

Lemma 4 (Zustand wächst nur mit bezahlter Arbeit) Sei S die Menge der Challenge-Tags, die der Daemon sich merkt. Ein Tag gelangt erst nach Annahme eines gültigen Nachweises in S und verlässt die Menge nach Ablauf der Challenge-TTL. Daher gilt $\mid S \mid \leq r \cdot \mathrm{TTL}$, wenn r die Rate angenommener verschiedener Lösungen in diesem Intervall begrenzt. Beim kostenlosen Abruf einer Challenge wird kein Tag angelegt. Die Implementierung hat zudem eine feste Kapazität und kann bei Sättigung eine gültige Lösung abweisen. Dies ist eine Schranke für den Zustand, keine untere Schranke für die Rechenstrategie jedes Gegners.

Die Verbrauchsmenge besteht aus 16 Shards mit Robin-Hood-Open-Addressing über die 16-Byte-Tags (Teil V). Der Fingerabdruck ist ein schlüsselbasierter Hash von Clientadresse und User-Agent, längengetrennt, sodass keine zwei Eingaben durch Verkettung kollidieren, und mit Schlüssel, sodass Fingerabdrücke anderer Clients nicht offline berechnet werden können.

Übung 3.3. Eine Token-Nutzlast trägt eine Arbeitsstufe, `timestamp`, `expiry`, `rule_hash` und `fingerprint`. Erklären, was jedes Feld verhindert, wenn es entfernt würde, und warum das Tag alle fünf abdecken muss.

Die Invariante erklären. Ein Kollege schlägt vor, alle Tokens auf Ed25519 umzustellen, „weil es stärker ist“. Das Argument der Vertrauensdomäne, den Kostenunterschied und die Post-Quanten-Überlegung erklären.

TEIL IV

Das Sibuna-Protokoll

Wir spezifizieren das Übertragungsprotokoll so, wie es implementiert ist: den Rundlauf über die Zwischenseite, den zustandslosen Challenge-Datensatz, beide Lösungsformate, die Reihenfolge der Verifikation, die Durchsetzung der Einmalverwendung und das Sitzungscookie.

Der Challenge-Rundlauf

Lernziele Am Ende dieses Kapitels sollten wir einen Browser von seiner ersten Anfrage bis zu einer ausgestellten Sitzung verfolgen, jeden beteiligten Endpunkt und jedes JSON-Feld benennen und erklären können, warum die Challenge die Richtlinienentscheidung für den geschützten Pfad mit sich trägt.

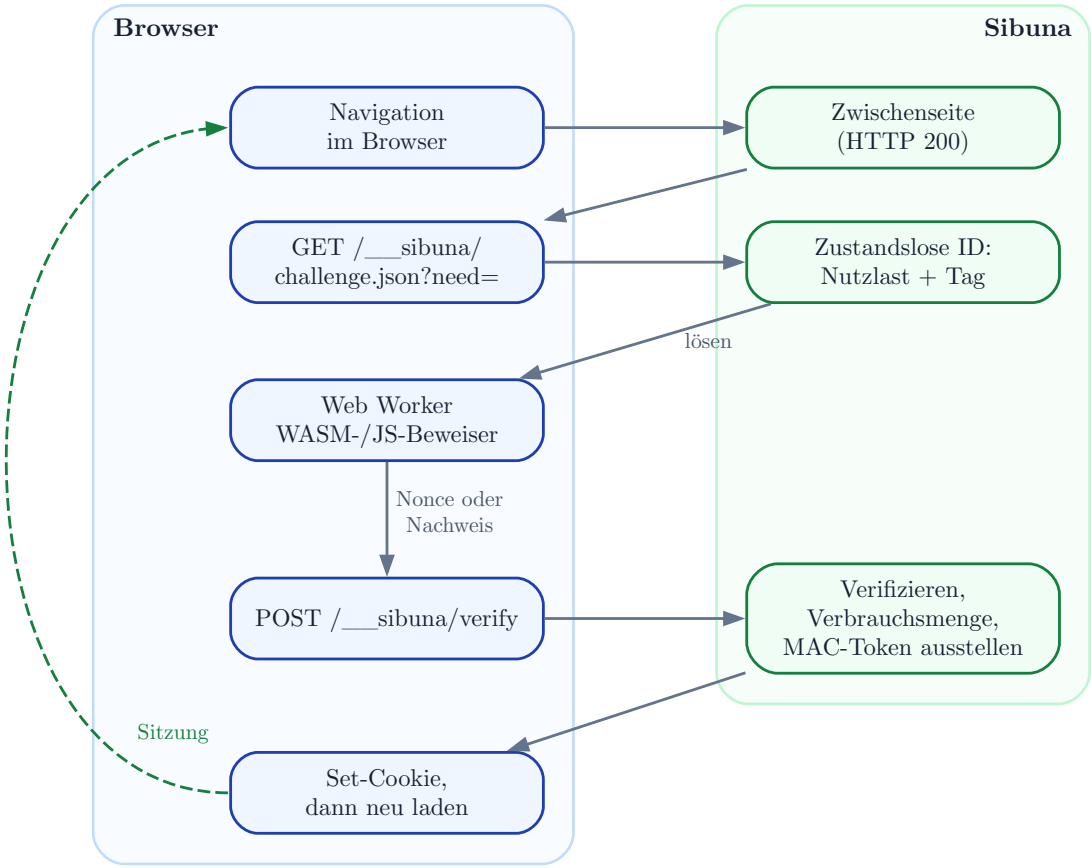


Abb. 5 Der Challenge-Rundlauf. Die Schritte links laufen im Browser, die Schritte rechts in Sibuna.

1. Eine Navigation ohne gültige Sitzung erreicht die Richtlinien-Engine und wird als `CHALLENGE` eingestuft. Weil die Anfrage `text/html` akzeptiert, antwortet der Daemon mit `200`, der Zwischenseite und dem Header `X-Sibuna-Status: CHALLENGE`; ein API-Client, der kein HTML akzeptiert, erhält `401` mit einem JSON-Body, dessen URL im Feld `challenge` direkt abrufbar ist. In beiden Fällen trägt die Antwort ein *Anforderungsticket*: die Challenge, die diese Entscheidung verlangt (Algorithmus, Arbeitsbits, Öffnungen), und den Hash der Regel, für diesen Client versiegelt mit einem Tag aus BLAKE3 mit Schlüssel unter einem eigenen abgeleiteten Schlüssel und gültig für eine Challenge-Lebensdauer.
2. Die Zwischenseite ruft `GET /__sibuna/challenge.json?path=<original URL>&need=<ticket>` ab. Ein gültiges Ticket bestimmt die Challenge: Die Arbeit ist durch die Entscheidung festgelegt, die sie verlangt hat, und wird nicht aus einer gemeldeten URL und den eigenen Headern des Abrufs neu berechnet, die sich von denen der Navigation unterscheiden (`Accept`, `Sec-Fetch-*`). Ohne gültiges Ticket (ein API-Client, der die URL ignoriert, eine Seite, die über die Lebensdauer hinaus offen blieb) wertet der Server die gemeldete URL aus. Er zerlegt sie mit derselben Funktion, die die Anfragezeile liest, in Pfad und Query und bringt den Pfad in die Normalform, die die Zulassung verwendet (SID 0009); eine URL, die länger als 8 KiB ist, wird mit `414` abgewiesen, nie gekürzt, und ein Pfad ohne

Normalform mit `400`. Das Ticket ist nur ein Hinweis: Die Zulassung prüft weiterhin die Arbeitsstufe der Sitzung, sodass ein wiederholtes oder gefälschtes Ticket seinen Inhaber nur Arbeit kosten kann.

3. Die Antwort ist der Challenge-Datensatz:

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

   Für `hashcash` ist `difficulty` die Bitzahl und `challenges` ist `0`.

4. Ein Web Worker löst sie mit dem WebAssembly-Modul oder, wenn WebAssembly nicht verfügbar ist, mit dem JavaScript-Beweiser (Teil VII) und sendet entweder

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

   oder

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

   an `POST /__sibuna/verify`.

5. Bei Erfolg antwortet der Server mit `200 {"status":"ok"}` und einem `Set-Cookie`-Header, und die Seite lädt ihre ursprüngliche Adresse neu. Jede spätere Anfrage trägt das Cookie und wird vor der Richtlinienauswertung zugelassen.

Warum der Pfad mit der Challenge reist Die Zwischenseite wird unter der geschützten URL ausgeliefert, also weiß der Browser, wo er ist. Den Pfad mitzusenden lässt den Server die Schwierigkeit und den Algorithmus *pro Regel* wählen (eine Checkout-Route kann 20 Arbeitsbits sequenzieller Arbeit verlangen, während ein Blog 16 Bits Hashcash verlangt) und bindet den Regelhash in das Token, von wo `X-Sibuna-Rule-Hash` ihn an den Ursprungsserver meldet.

Der zustandslose Challenge-Datensatz

Lernziele Den Challenge-Encoder lesen, den Zweck jedes Felds verstehen und sehen, wie die Nonce pro Challenge ohne Systemaufruf für Entropie im Hot Path abgeleitet wird.

Implementierung: `Coordinator::create_challenge_with_spec` Baut die 36-Byte-Nutzlast, versieht sie mit einem Tag unter dem Challenge-Schlüssel und kodiert die 52 Bytes mit base64url in eine 70 Zeichen lange Kennung. In `crates/challenge/src/coordinator.rs` (Crate `sibuna-challenge`).

```rust
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)),
    }
}
```

Die Kennung ist ein `Encoded<70>`: Der Text liegt in einem Array von genau seiner Länge, sodass das Ausstellen einer Challenge nie allokiert. `ChallengeSpec::parameter` bildet Arbeitsbits auf die eigene Einheit der Stufe ab, Hashcash-Bits (höchstens 64) oder PoSW-Tiefe (`difficulty - 3`, begrenzt auf 4 bis 24).

Drei Details verdienen Aufmerksamkeit:

- *Adaptive Schwierigkeit.* Der Koordinator führt einen exponentiell gewichteten gleitenden Mittelwert der Challenge-Ausgaberate in drei atomaren Variablen. Oberhalb einer Grundrate von 50 Challenges pro Sekunde fügt jede Verdopplung der Rate ein Arbeitsbit hinzu, höchstens sechs. Eine Flut zahlt exponentiell mehr, während ruhiger Datenverkehr die Grundkosten behält.
- *Die Nonce ist eine PRF-Ausgabe*, mit dem Challenge-Schlüssel über einen Zähler, die Uhr und den Fingerabdruck berechnet. Sie ist eindeutig (Zähler) und unvorhersagbar

(Schlüssel), sodass ein Gegner keine Lösungen für Kennungen vorberechnen kann, die ihm nicht ausgestellt wurden, und der Hot Path nie das Betriebssystem nach Entropie fragt.
- *Die Schwierigkeit reist im Tag mit.* Ein Client kann die Schwierigkeit nicht senken, indem er den Datensatz bearbeitet: Die Prüfung des Tags würde scheitern, bevor irgendein Nachweis untersucht wird.

Reihenfolge der Verifikation und Einmalverwendung

Lernziele Verstehen, warum die billigen Prüfungen vor dem Nachweis laufen, warum die Challenge erst als verbraucht markiert wird, nachdem der Nachweis gültig ist, und wie zwei gleichzeitige Einreichungen derselben Lösung aufgelöst werden.

Implementierung: `Coordinator::verify_and_mint` Dekodiert und authentifiziert die Kennung, prüft Alter und Clientbindung, verifiziert den Nachweis für die verzeichnete Stufe, verzeichnet das Tag als verbraucht und stellt das Sitzungstoken aus. In `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,
    })
}
```

Die Reihenfolge ist Absicht. Tag, Alter und Bindung sind Prüfungen in konstanter Zeit, die unbrauchbare Eingaben vor jedem Hashen abweisen. Ein falscher Nachweis darf die Challenge nicht verbrauchen, daher wird die Verbrauchsmenge erst beschrieben, nachdem `check_solution` erfolgreich war. Zwei Threads, die beide dieselbe gültige Lösung verifizieren, liefern sich bei `mark_spent` einen Wettlauf; die Shard-Sperre serialisiert sie, genau einer gewinnt, und der andere erhält `VerifyError::DoubleSpendAttempt`. Die Klausel `issued_at > now + 60` (`CLOCK_SKEW_SECONDS` ist 60) weist Datensätze ab, die ein Knoten ausgestellt hat, dessen Uhr um mehr als eine Minute vorgeht; sonst würde sich die Lebensdauer einer Challenge verlängern. Fehler erreichen den Client als `400` mit einer Diagnose im Elm-Stil (Teil X): Jeder `VerifyError` wird in `sibuna_core::errors::ErrorKind` umgewandelt, dessen Erklärung der Antwort-Body ist. Die End-to-End-Tests prüfen den Antwort-Body auf `DOUBLE SPEND`, `FINGERPRINT MISMATCH`, `WRONG SOLUTION TYPE` und `MALFORMED CHALLENGE`.

Das Sitzungscookie

Lernziele Das Token mit schlüsselbasiertem Hash lesen, die Cookie-Attribute verstehen und die Clientbindung erklären.

Implementierung: `token::mac::mint` und `token::mac::verify` Serialisiert die 40-Byte-Nutzlast, hängt die ersten 16 Bytes von BLAKE3 mit Schlüssel über sie an und verifiziert mit einem Vergleich in konstanter Zeit. In `crates/challenge/src/token/mac.rs` (Crate `sibuna-challenge`). `Coordinator::verify_cookie` ruft es mit dem Fingerabdruck des Clients auf.

```
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);
```

```
    // Konstante Zeit: Ein früher Abbruch verriete einem Fälscher korrekte Tag-
Bytes.
    if !ct::eq(&tag(key, &bytes), given) {
        return Err(TokenError::InvalidTokenSignature);
    }
    let payload = Payload::from_bytes(&bytes)?;
    payload.check(now, expected_fingerprint)?;
    Ok(payload)
}
```

Das Tag wird geprüft, bevor die Nutzlast geparst wird: Eine Nutzlast einer anderen Version wird erst dann mit `TokenError::InvalidEncoding` abgewiesen, wenn feststeht, dass sie authentisch ist, und eine gefälschte erreicht den Parser nie.
Das Cookie wird ausgegeben als

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

`HttpOnly` hält es von Seitenskripten fern, `SameSite=Lax` verhindert websiteübergreifende Wiederholung und erlaubt dennoch Navigation auf oberster Ebene, und `Secure` kommt mit `--secure-cookie` hinzu, wenn TLS vor dem Daemon terminiert. Weil die Nutzlast den schlüsselbasierten Fingerabdruck von Adresse und User-Agent trägt, scheitert ein auf eine andere Maschine kopiertes Cookie mit `TokenError::TokenBoundAddressMismatch`; der End-to-End-Test „the cookie is bound to the client identity“ prüft genau diesen Pfad.
Die Nutzlast beginnt mit einem Versionsbyte und der *Arbeitsstufe*, die der Inhaber bezahlt hat: dem Verfahren (Hashcash oder PoSW) und den tatsächlich gelösten Arbeitsbits, einschließlich einer etwaigen lastabhängigen Erhöhung. Eine Sitzung erfüllt eine Challenge nur, wenn ihre Stufe erreicht, was die Route verlangt; das wird aus der Schwierigkeit der Regel mit derselben Begrenzung berechnet, die die Challenge selbst anwendet. Stufen sind geordnet, nicht benannt: Ein bei 24 Bits erworbenes Cookie deckt auch jede 16-Bit-Route ab, während ein 16-Bit-Cookie an einer 24-Bit-Route zur stärkeren Zwischenseite durchfällt und das neue Cookie das alte ersetzt. `rule_hash` bleibt die Audit-Kennung der Regel, die die Challenge ausgestellt hat; er wird an den Ursprungsserver gemeldet, nie für die Zulassung verwendet. WAF-Befunde und explizite Ablehnungen werden durch eine Sitzung nie aufgehoben. Der End-to-End-Test „a session earned on a cheaper route does not admit a route that demands more work“ prüft beide Richtungen.

> **Weitergeleitete Adressen** Hinter einem Eingangsproxy kommt die Clientadresse in `X-Forwarded-For` an. Sibuna vertraut diesem Header nur mit `--trust-forwarded` (im Forward-Auth-Modus impliziert). Ein Reverse Proxy, der direkt dem Internet ausgesetzt ist, muss die Option ausgeschaltet lassen, sonst könnte jeder Client seine eigene Bindung und seinen eigenen Ratenlimit-Bucket wählen.

Übung 4.1. Den Ablauf für zwei Browser hinter einer NAT-Adresse mit identischen User-Agents skizzieren. Welche Prüfungen teilen sie, welche nicht, und was bräuchte ein Angreifer hinter diesem NAT, um das Cookie des anderen Browsers wiederzuverwenden?

Die Invariante erklären. Einem Betreiber erklären, warum sich der Daemon offene Challenges nicht „merkt“ und warum das ein Sicherheitsmerkmal ist und keine Abkürzung.

TEIL V

Allokationsfreie Technik in Rust

Wir verfolgen eine Verbindung vom Annehmen bis zur Antwort durch einen einzigen festen Puffer und untersuchen die nebenläufigen Datenstrukturen, mit denen viele Verbindungsthreads Zustand teilen, ohne auf dem Anfragepfad Heap-Speicher zu allokieren. Wo Speichersicherheit sonst eigene Mechanismen bräuchte (Leserzähler, Arenen, Deskriptortabellen, Abbauketten), zeigen wir, wie Rusts Ownership sie stattdessen liefert.

Die Verbindungsschleife

Lernziele Am Ende dieses Kapitels sollten wir erklären können, wie ein einziger 64-KiB-Puffer eine ganze Keep-alive-Verbindung bedient, wie Kopf und Body ohne Kopieren gefunden werden, warum jede Verbindung einen eigenen begrenzten Thread erhält, wie der Proxy die Rahmung des Ursprungsservers erfährt, damit eine weitergeleitete Clientverbindung offen bleiben kann, und welche dieser Garantien der Borrow-Checker für uns beweist.

Ein Thread pro Verbindung, begrenzt

Accept-Worker (`--workers`, standardmäßig einer pro CPU, höchstens 64) teilen sich den lauschenden Socket. Jede angenommene Verbindung wird einem eigenen Thread mit einem Stack von einem Megabyte übergeben, und die Accept-Schleife kehrt sofort zu `accept` zurück. Ein Thread, der seine Verbindung beendet hat, wartet geparkt und übernimmt die nächste, weil das Erzeugen und Beenden eines Threads für jede Verbindung die anderen Threads ins Stocken brachte (Teil VIII, SID 0002). Die Alternative, eine Verbindung auf dem Accept-Thread bis zum Ende zu bedienen, war der erste Entwurf: Mit einem einzelnen Client maß er gut und mit vierundsechzig schlecht, weil sechzig von ihnen warteten, bis ein Worker sein Kontingent von 256 Anfragen abgearbeitet hatte, und die Latenz am Verteilungsende mit jedem wartenden Client wuchs. Mit einem Thread pro Verbindung bleibt das p99 bei 64 Verbindungen in jeder Arbeitslast des Gesamtprodukts unter einer Millisekunde, zwischen 305 µs und 857 µs (`benchmarks/results/product-latest.md`; Teil VIII).
Die Zahl der Verbindungsthreads ist durch `--max-connections` begrenzt (standardmäßig 1.024, höchstens 8.192). Jenseits der Grenze antwortet die Accept-Schleife auf dem neuen Socket mit `503 Service Unavailable` und schließt ihn, ohne etwas zu starten, und zählt das Ereignis in `sibuna_overloaded_total`.

Implementierung: `accept_loop` Nimmt Verbindungen an, prüft jede gegen die Verbindungsgrenze und startet sie auf einem eigenen Thread aus dem Task-Pool. In `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,
            // ... Deskriptoren oder Speicher erschöpft: 1 ms warten statt spinnen
        };
        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);
        });
        // ... scheitert der Start: Platz zurückgeben und Überlastung zählen
    }
}
```

`Counted` ist ein Guard, dessen `Drop` den Platz der Verbindung innerhalb der Grenze zurückgibt, wie auch immer der Thread endet, durch Rückkehr oder durch Panic. Der `503`-Pfad liest zuerst ab, was der Client bereits gesendet hat, bevor er seine Seite des Sockets schließt: Ein Socket, der über ungelesenen Bytes geschlossen wird, antwortet mit einem Reset, und der Reset würde die `503` verwerfen.

Threads stammen aus einem `TaskPool` (`cmd/sibuna/src/tasks.rs`), einer festen Tabelle joinbarer Slots. Ein beendeter Thread behält seinen Stack, bis er gejoint wird. Daher wird ein Slot gejoint, wenn er erneut belegt wird, und eine neue Aufgabe nimmt den *niedrigsten* beendeten Slot, statt einen Cursor weiterzuschieben. Ein Cursor würde einen beendeten Slot erst nach einer vollen Runde durch die Tabelle wieder erreichen, sodass Tausende beendeter Threads resident blieben, und der Speicher würde mit den bedienten Verbindungen wachsen statt mit den offenen. Ein zweiter Guard, `Finished`, markiert den Slot als frei, wenn die Aufgabe endet, sodass auch eine Verbindung mit Panic ihren Slot freigibt.

Der Speicher ist auf dieselbe Weise begrenzt. Ein Verbindungsthread hält seinen 64-KiB-Lesepuffer und seinen 16-KiB-Schreibpuffer auf seinem Stack und einen 40-KiB-Proxy-Arbeitsbereich auf dem Heap, der einmal beim Start der Verbindung allokiert wird. Der schlechteste Fall ist daher eine bekannte Zahl und keine Funktion davon, wie viele Sockets ein Client öffnen kann. Der Idle-Reaper (siehe unten) schließt Verbindungen, die nichts mehr senden, sodass ein langsamer Client einen Thread nicht über `--idle-timeout` hinaus festhalten kann.

Wahl des Allokators

Rust 1.100 stabilisiert den Kern der `Allocator`-API und die Allokatorparameter von `Vec` und `Box` (Versionshinweise). Wir können den Allokator einer Sammlung wählen, ohne den Allokator des Prozesses zu ersetzen. Auf dem Anfragepfad sparen wir dadurch keine Allokationen: Anfragen auf warmen Verbindungen allokieren bereits nichts, wie `cmd/sibuna/tests/steady_state.rs` und `crates/net/tests/steady_state.rs` prüfen. Der Proxy allokiert seine drei Puffer einmal pro Verbindung und verwendet sie wieder. Auch der Begrenzer und die Menge verbrauchter Nachweise allokieren ihre Tabellen nur einmal; die mit Nullen angeforderten Allokationen lassen unberührte Seiten nichtresident bleiben, wie `crates/store/tests/resident.rs` prüft.
Bei einer Untersuchung eines anderen Allokators würden wir zuerst den Aufbau von Regelwerks-Snapshots und das JSON-Parsen der Konsole messen. Ihre temporären Sammlungen allokieren außerhalb des Anfragepfads, doch der Allokator einer Sammlung bestimmt nicht automatisch die Allokationen ihrer Elemente. Unser JSON-Baum besitzt außerdem Zeichenketten mit Escape-Sequenzen über `Cow<str>`, dessen besitzende Form `String` ist; `String` hat in Rust 1.100 keinen Allokatorparameter. `Arc::new_in` bleibt in dieser Version instabil (API-Dokumentation). Veröffentlichte Engines bleiben über `Arc` geteilt, bis ihr letzter Leser sie freigibt. Ein Neuaufbau darf daher keinen Speicher zurücksetzen, den eine ältere Engine noch verwendet. Eine Arena müsste diese Lebensdauern und die Eingabegrenzen des Parsers erhalten.
Wir behalten den globalen Allokator `WipeOnFree` des WebAssembly-Moduls der Konsole in `cmd/console-ui/src/abi.rs` bei. Er löscht freigegebene Blöcke, einschließlich des alten Blocks nach einer Reallokation. Würden wir `System` direkt für eine Sammlung wählen, würde dieses Löschen umgangen. Das WebAssembly-Modul für Arbeitsnachweise hat keinen Heap-Allokator, und Argon2 verwendet bereits einen wiederverwendeten Arbeitsbereich, der dem Aufrufer gehört. Wir behalten daher die bisherigen Allokatoren und die Kompatibilität mit Rust 1.94 bei, bis eine gemessene Last eine Änderung rechtfertigt. Ein Kandidat muss Latenz oder residenten Speicher verbessern und dabei das Löschen, die begrenzte Eingabeverarbeitung und den allokationsfreien Anfragepfad erhalten.

Ein Puffer pro Verbindung

Jede dynamische Allokation auf einem Anfragepfad ist ein Hebel für Denial of Service: Fragmentierung über Tage, Konkurrenz um die Sperre des Allokators zwischen Threads und Verstärkung durch Angreifer, die Anfragen so gestalten, dass sie die Allokation maximieren. Sibunas Regel ist absolut: Klassifikation, Verifikation und Übergabe an den Proxy laufen ohne jede Heap-Allokation. Der Verbindungshandler besitzt für die Lebensdauer des Sockets einen 64-KiB-Lesepuffer und einen 16-KiB-Schreibpuffer; alles andere ist ein Slice in diese Puffer.

Implementierung: `connection::serve` Meldet die Verbindung beim Idle-Reaper an, richtet die Puffer, den Proxy-Arbeitsbereich und den Engine-Cache einmal ein und bedient bis zu 4.096 Anfragen. In `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` füllt den Puffer, bis die Leerzeile erscheint, durchsucht dabei nur die seit der letzten Suche eingetroffenen Bytes und weist Köpfe über 16 KiB mit `431` ab. `parse_request` (Crate `sibuna-http`) erzeugt einen `Request<'a>`, dessen Methode, Pfad, Query, Header und Cookies Slices des Puffers sind. Der angegebene Body wird gefüllt, soweit er passt, und hinter dem Kopf als Slice genommen. Dann rückt `reader.consume(head_len + body_len)` den Leser vor, sodass die nächste Keep-alive-Anfrage sauber beginnt; der Leser schiebt die ungelesenen Bytes nur dann nach vorn, wenn er Platz braucht. Ein Chunked-Body wird an Ort und Stelle hinter dem Kopf dekodiert (SID 0009): Die Rahmung jedes Schritts wird aus dem Puffer herausgeschnitten, sodass das dekodierte Präfix für die Prüfung zusammenhängend bleibt. Interne Routen sind längenbegrenzt und halten die Verbindung offen. Weitergeleitete Anfragen halten sie ebenfalls offen, sofern die Antwort des Ursprungsservers gerahmt ist und der Body vollständig gepuffert wurde; die nächsten Abschnitte zeigen, wie der Proxy entscheidet.

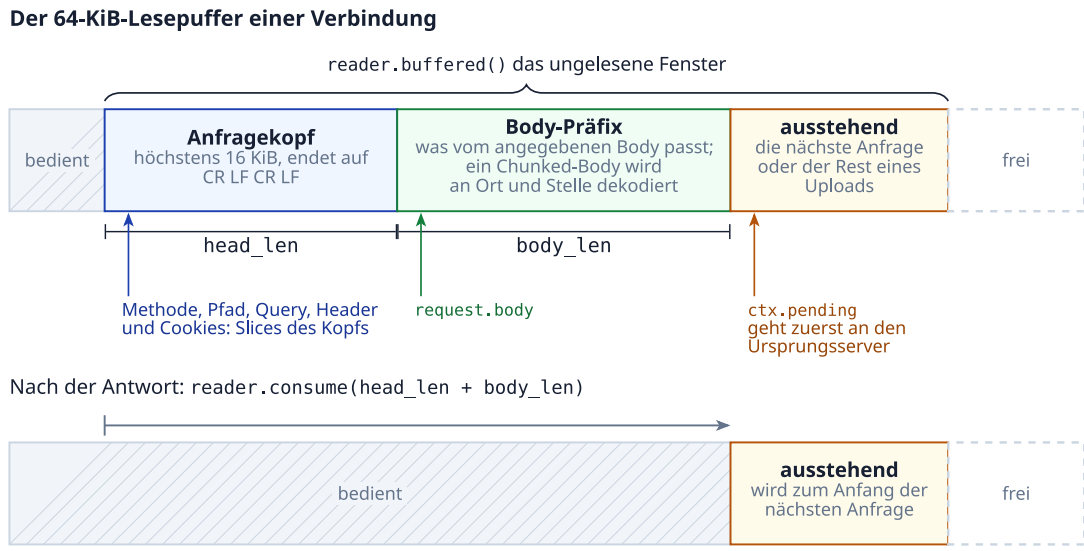


Abb. 6 Der Lesepuffer der Verbindung, während eine Anfrage bedient wird. Jedes Feld der Anfrage ist ein Slice davon.

```
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)
}
```

Was der Borrow-Checker beweist Die Slices der Anfrage bleiben nur bis zum nächsten Füllen des Puffers gültig. Sibuna formuliert diese Regel als Typ, nicht als Kommentar: Die Anfrage ist ein `Request<'a>`, und `'a` ist eine geteilte Ausleihe des Leserpuffers. Den Puffer zu füllen oder zu verbrauchen erfordert `&mut reader`, was der Compiler verweigert, solange ein davon geliehener `Request` noch verwendet wird. In `serve_one` wird die Anfrage in `dispatch` verschoben, und erst nachdem dieser Aufruf zurückgekehrt ist, darf `reader.consume` laufen; in umgekehrter Reihenfolge geschrieben, kompiliert der Code nicht.
Dieselbe Regel erklärt zwei Entscheidungen, die zunächst seltsam aussehen. Erstens parst `serve_with_body` den Kopf ein zweites Mal. Der erste Parse muss seine Ausleihe beenden, bevor der Body gepuffert wird, weil das Puffern eines Chunked-Bodys Bytes innerhalb des Puffers verschiebt; der zweite Parse leiht den zur Ruhe gekommenen Puffer. Das Parsen ist deterministisch und dauert für eine Browseranfrage mit Cookie etwa 133 ns (`zero_copy_request_and_cookie`, 132,5 ns, in `benchmarks/results/primitives-latest.md`), daher kostet der zweite Parse wenig und liefert dasselbe Urteil. Zweitens leiht der Proxy die geparste Anfrage und die ausstehenden Bytes und darf den Verbindungspuffer nicht neu füllen; deshalb liest er weitere Client-Bytes in einen eigenen Upload-Puffer. Weil er Gelesenes nicht zurückgeben kann, wird die Clientverbindung nur dann behalten, wenn der Body vollständig gepuffert wurde.

Das Umschreiben des Kopfs im Proxy

Der Kopf wird nicht wörtlich weitergeleitet. `write_request_head` (`crates/net/src/request.rs`, Crate `sibuna-net`) gibt die Anfragezeile und jeden Header neu in den Writer zum Ursprungsserver aus und lässt dabei weg: Hop-by-Hop-Felder (`Connection`, `Keep-Alive`, `Proxy-Connection`, `Transfer-Encoding`, `TE`, `Trailer`, `Upgrade`, die Felder der Proxy-Autorisierung), jedes eingehende Weiterleitungs- oder Auditfeld (`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` sowie jedes Feld, das der `Connection`-Header der Anfrage benennt. Danach hängt er die Auditfelder an. Die Rahmung des Bodys wird erzeugt, nie kopiert:

```
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")?,
}
```

Ein `Content-Length` wird neu ausgegeben. Ein Chunked-Body, der innerhalb des Verbindungspuffers endete, wird an Ort und Stelle dekodiert und mit seiner Länge gesendet. Ein längerer wird als `Transfer-Encoding: chunked` angekündigt und Lesevorgang für Lesevorgang neu in Chunks zerlegt (SID 0009): Jeder Lesevorgang landet im 16-KiB-Upload-Puffer des Proxys, wird dort an Ort und Stelle dekodiert und geht als ein kanonischer Chunk hinaus, sodass der Ursprungsserver die Chunk-Größen, Erweiterungen und Trailer des Clients nie sieht. Die Auditfelder:

```
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 | ...
```

Bodys, die größer als der Puffer sind, werden in Stücken von 16 KiB vom Client zum Writer des Ursprungsservers weitergereicht (`relay_body`), bevor die Antwort zurückgestreamt wird. Der Unit-Test `proxy_head_rewrite_drops_hop_by_hop_headers_and_injects_audit_fields` in `crates/net/src/request.rs` stellt sicher, dass das Umschreiben ein gefälschtes `X-Forwarded-For` entfernt und die Auditfelder einfügt.
Die drei Puffer des Proxys, der Lesepuffer zum Ursprungsserver (ein Antwortkopf, 16 KiB), der Schreibpuffer zum Ursprungsserver (8 KiB) und der Upload-Puffer (16 KiB), liegen in einem `Workspace`, den die Verbindung einmal allokiert und für jede Anfrage wiederverwendet. Der Arbeitsbereich gibt ihnen die Lebensdauer der Verbindung, nicht die des einzelnen Austauschs, und Leser und Writer zum Ursprungsserver leihen sie jeweils für einen Austausch.

Die Antwort des Ursprungsservers weiterreichen

Der erste Proxy streamte die Bytes des Ursprungsservers, bis dieser schloss, und schloss dann den Client: korrekte Rahmung ohne Parsen, um den Preis einer neuen TCP-Verbindung pro weitergeleiteter Anfrage. Unter einem Lastgenerator zeigte sich dieser Preis als Zehntausende Sockets in `TIME_WAIT` und, auf Loopback, als erschöpfte ephemere Ports. Der Proxy liest jetzt den Kopf des Ursprungsservers (höchstens 16 KiB) und klassifiziert den Body nach den Regeln von RFC 9112:

> **Implementierung: `parse_response_head`** Gibt den Status zurück, eine von vier Rahmungen (kein Body bei HEAD, 1xx, 204 und 304; ein `Content-Length`; Chunked-Transfer-Kodierung; oder durch Schließen begrenzt) und ob der Ursprungsserver die Verbindung offen hält. In `crates/net/src/response.rs` (Crate `sibuna-net`).

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

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

Der Parser weist zurück, was der Proxy nicht weiterreichen darf: eine fehlerhafte Statuszeile, einen Feldnamen, der kein Token ist, ein Steuerbyte in einem Wert, widersprüchliche Längen, eine wiederholte oder nicht abschließende `chunked`-Kodierung und ein `Connection`-Feld, das `Content-Length` oder `Transfer-Encoding` benennt. Seine Arbeit ist linear in der Größe des Kopfs, und er gerät nie in Panic.

Der Kopf wird an den Client neu ausgegeben, wobei die `Connection`-Header des Ursprungsservers durch Sibunas eigene Entscheidung ersetzt werden, und der Body wird exakt weitergereicht: `copy_exact` bei einer Länge, Chunk für Chunk (Größenzeile, Daten, Trailer) bei Chunked-Kodierung, `copy_remaining` im Altfall. Nur der Altfall schließt den Client. Jedes Weiterreichen baut auf einem einzigen Kopierschritt auf: weitergeben, was die Quelle bereits hält, und erst wenn sie nichts hält, das Ziel leeren und dann einmal lesen. Ein Strom kleiner Schreibvorgänge wird daher so ausgeliefert, wie er entsteht, während eine Massenübertragung weiterhin einen vollen Puffer pro Lese- und Schreibvorgang bewegt.

```
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 })
```

Das Parsen eines Ursprungskopfs mit acht Feldern dauert 322 ns, und ein Keep-alive-`GET` mit einer 1-KiB-Antwort, über Loopback weitergeleitet, dauert 29,8 µs (gemessen im Oktober 2026; `crates/net/src/lib.rs` dokumentiert die Messung).

Der Pool zum Ursprungsserver

Das Parsen der Rahmung sagt dem Proxy auch, ob der Socket zum *Ursprungsserver* wiederverwendet werden kann: Eine HTTP/1.1-Antwort ohne `Connection: close` (oder eine HTTP/1.0-Antwort mit `keep-alive`), deren Body vollständig verbraucht wurde, lässt den Socket an einer sauberen Anfragegrenze zurück. Solche Sockets kommen in einen festen Pool von 256 untätigen Verbindungen zum Ursprungsserver, und die nächste weitergeleitete Anfrage nimmt einen davon, statt neu zu verbinden. Ohne den Pool öffnet jede weitergeleitete Anfrage eine neue Verbindung zum Ursprungsserver, und jede geschlossene Verbindung hält ihren lokalen Port in `TIME_WAIT`; unter Last gehen einem Proxy auf Loopback bald die ephemeren Ports aus, und sein Durchsatz bricht ein.

Ein Pool unter einer einzigen Sperre würde jede weitergeleitete Anfrage um diese Sperre konkurrieren lassen. Deshalb ist der Pool (`crates/net/src/pool.rs`) in Streifen aufgeteilt: 16 Streifen mit je 16 Sockets, jeder Streifen ein `Mutex<Vec<Arc<TcpStream>>>` auf einer eigenen Cache-Zeile. Einem Thread wird bei der ersten Nutzung reihum ein Heimatstreifen zugewiesen; er gibt zuerst an diesen ab und nimmt zuerst von ihm und schaut, ohne auf eine gehaltene Sperre zu warten, in die anderen Streifen, bevor er neu verbindet.

```
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()
    })
}
```

Ein Socket im Pool kann vom Ursprungsserver geschlossen worden sein, während er untätig war. Der Proxy bemerkt das auf eine von zwei Arten, durch einen fehlgeschlagenen Schreibvorgang oder durch ein Stream-Ende vor dem ersten Antwortbyte. In beiden Fällen hat den Client noch nichts erreicht, also schließt er den Socket und versucht es einmal erneut auf einer

frischen Verbindung, nie auf einer anderen aus dem Pool: Nach einer Ruhephase kann jeder Socket im Pool veraltet sein, und die erste Version dieser Wiederholung, die einen zweiten Socket aus dem Pool nahm, beantwortete nach jeder ruhigen Phase die erste Anfrage mit `502`. Die Wiederholung ist nur für eine sichere Methode (`GET`, `HEAD`, `OPTIONS`) erlaubt, deren Body vollständig gepuffert wurde. Eine fehlende Antwort belegt nicht, dass der Ursprungsserver keine Änderung festgeschrieben hat, und ein Body, der in Stücken weitergereicht wurde, kann nicht erneut gesendet werden; in beiden Fällen erhält der Client `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 {
    // Eine Wiederholung verbindet sich immer neu: Nach einer Ruhephase kann
    // jeder Pool-Socket veraltet sein; ein zweiter ließe die Anfrage scheitern.
    let pooled = if attempt == 0 { pool.take() } else { None };
    // ... verbinden, falls der Pool leer war, an Aktivität koppeln, austauschen
    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
                );
            // ... alles andere wird sofort gemeldet
        }
    }
}
```

Unit-Tests reichen feste Bytefolgen durch dieselben Funktionen und stellen sicher, dass die Ausgabe bis auf den Connection-Header byteidentisch ist (`crates/net/src/relay/tests.rs`). Ein End-to-End-Test des Daemons sendet zwei weitergeleitete Anfragen auf einem Client-Socket und prüft über einen Sequenz-Header, den der Stub-Ursprungsserver hinzufügt, dass die zweite die Verbindung zum Ursprungsserver aus dem Pool wiederverwendet hat. Ein Vektortest in `crates/net/tests/` spielt aufgezeichnete Proxy-Szenarien ab (eine rohe Clientanfrage und eine vorgefertigte Antwort des Ursprungsservers) und vergleicht jedes Byte, das Ursprungsserver und Client empfangen.

Warum der Socket zum Ursprungsserver ein `Arc` ist Ein Socket-Deskriptor ist eine kleine Ganzzahl, die der Kernel wiederverwendet, sobald sie geschlossen ist. Das Herunterfahren muss einen laufenden Austausch unterbrechen, darf aber nie einen Deskriptor berühren, der inzwischen für eine andere Verbindung wiederverwendet wurde. Eine Tabelle aktiver Deskriptoren unter einer Sperre könnte das verhindern, um den Preis einer Sperre

bei jedem Austausch. Sibuna braucht keine solche Tabelle: Der Socket zum Ursprungsserver ist ein `Arc<TcpStream>`. Der Pool hält einen Handle, der Austausch hält einen, und die `Activity` der Verbindung hält einen Klon, solange der Austausch läuft. Der Deskriptor wird geschlossen, wenn der letzte Handle freigegeben wird; er kann also nicht geschlossen oder wiederverwendet werden, solange ihn noch jemand erreichen kann.

Leerlauffristen

Jede Clientverbindung registriert ihren Socket in einer `IdleTable` (`crates/net/src/idle.rs`) und erhält eine `Lease`. Die `Activity` der Lease hält den Zeitpunkt, zu dem sich zuletzt Bytes auf der Verbindung bewegt haben. Der Daemon aktualisiert ihn nach jeder bedienten Anfrage; der Proxy aktualisiert ihn bei jedem Lesevorgang vom Ursprungsserver und je 16 KiB Upload und hängt den Socket zum Ursprungsserver des laufenden Austauschs an. Der Reaper-Thread wacht nach jeweils einem Viertel der kürzesten Frist auf (zwischen 200 ms und einer Sekunde) und schließt jede Verbindung, die länger als ihre Frist still war, zusammen mit dem angehängten Socket zum Ursprungsserver. So sieht der auf einer der beiden Seiten blockierte Thread ein Stream-Ende und gibt seinen Slot frei.

Fristen messen daher Stille, nie die Länge einer Antwort: Ein großer Download, der in Bewegung bleibt, wird nie abgeschnitten. Upload-Bytes kommen von der nicht vertrauenswürdigen Seite und erneuern den Zeitstempel nur je 16 KiB. Damit wird von einem Client mindestens 16 KiB pro Leerlaufperiode verlangt, die Abwehr langsamer Bodys durch Anfragefristen mit Mindestrate.

Die Tabelle hat eine kleine Sperre pro Slot und keine tabellenweite Sperre. Sie hält ein `try_clone` jedes Client-Sockets und einen `Arc` jedes angehängten Sockets zum Ursprungsserver, aus dem oben genannten Grund. Eine `Lease` meldet ihre Verbindung in `Drop` ab, und ein abgeräumter Slot bleibt bis dahin im Besitz seiner Lease, sodass eine alte Verbindung nie einen Slot freigeben kann, den eine neue übernommen hat.

Verbindungen nach einem Upgrade

Nach einer validierten `101` wird ein WebSocket-Upgrade in beide Richtungen weitergereicht. Jede Richtung läuft auf einem eigenen Thread mit einem festen Puffer: Der eigene Thread der Verbindung trägt die Bytes vom Ursprungsserver zum Client, und ein Scoped Thread mit einem 128-KiB-Stack trägt die Bytes vom Client zum Ursprungsserver. Ein Thread könnte mit `poll(2)` beide Richtungen bedienen, aber die Rust-Standardbibliothek stellt das nicht bereit; zwei Threads mit Socket-Timeouts halten die Weiterleitung frei von `unsafe`. Eine Verbindung nach einem Upgrade hält daher zwei Threads, sodass die Zahl der Verbindungsthreads das Doppelte von `--max-connections` erreichen kann, wenn jede Verbindung ein Upgrade erhalten hat.

Beide Richtungen teilen eine Leerlauffrist (`--websocket-idle-timeout`, standardmäßig 300 Sekunden): Bytes, die sich in einer der beiden Richtungen bewegen, erneuern einen gemeinsamen Zeitstempel. Jedes Warten auf einen Socket trägt ein Timeout gleich der verbleibenden Zeit; ein Warten, das in das Timeout läuft, liest den gemeinsamen Zeitstempel neu und wartet entweder den Rest ab oder beendet die Weiterleitung. Erreicht eine Richtung das Stream-Ende, wird die Sendeseite des anderen Sockets geschlossen (ein Half-Close); schlägt eine Richtung

fehl, werden beide Sockets geschlossen, sodass auch die andere Richtung endet. Bevor ein Upgrade weitergeleitet wird, gibt die Verbindung ihre zwischengespeicherte Richtlinien-Engine frei, sodass eine Verbindung, die Stunden leben kann, keine alte Engine am Leben hält.

Herunterfahren

`Running::stop`, und ebenso `Drop` für `Running`, hält den Daemon in einer festen Reihenfolge an. Es setzt das Anhalte-Flag und weckt jeden Accept-Worker, indem es sich mit dem Listener verbindet, joint die Worker, schließt jeden registrierten Client-Socket und seinen angehängten Socket zum Ursprungsserver (`IdleTable::shutdown`), hält den Pool zum Ursprungsserver an, joint jeden Verbindungsthread, joint den Reaper und schließt die Sockets im Pool.
SIGINT und SIGTERM erreichen diese Abfolge über gewöhnliches Rust. Bevor er irgendeinen Thread startet, blockiert der Daemon beide Signale (jeder spätere Thread erbt die Maske) und widmet einen Thread `sigwait`, der die Anforderung festhält und den Hauptthread sofort weckt. Kein Code läuft in einem Signalhandler, und kein Thread fragt ein Flag ab. Das Signalmodul (`cmd/sibuna/src/signals.rs`) enthält den einzigen `unsafe`-Code des Daemons: die Deklarationen der wenigen Funktionen der C-Bibliothek, die `std` nicht kapselt.

Freigabe durch `Drop` Explizite Aufräumaufrufe müssen in umgekehrter Reihenfolge der Einrichtung geschrieben werden, und jede vorzeitige Rückkehr muss gegen sie gelesen werden. In Sibuna gibt sich jede Ressource in `Drop` selbst frei: `Daemon` und `Running` halten den Server an, eine `Lease` meldet ihre Verbindung ab, `Counted` gibt einen Platz innerhalb der Verbindungsgrenze zurück, und `Finished` gibt einen Task-Slot frei. Der Compiler fügt jede Freigabe auf jedem Pfad aus dem Gültigkeitsbereich ein, auch wenn eine Panic ihn abwickelt, sodass kein Pfad eine vergessen kann.

Was wir gemessen haben

Die Tabelle zeigt Shield im Reverse-Proxy-Modus auf einem Ryzen 9 5950X: der Daemon auf vier CPUs festgelegt, `wrk` mit zwei Threads und 64 Verbindungen auf acht anderen CPUs, Läufe von fünf Sekunden, der Median aus drei. Die Zeilen sind aus `benchmarks/results/product-latest.md` übernommen; Teil VIII enthält das vollständige Protokoll.

Tabelle 7 Shield im Reverse-Proxy-Modus, aus `benchmarks/results/product-latest.md`

Arbeitslast	Status	Anfr./s	p50 / p99 (µs)	CPU pro Anfrage (µs)
mit Sitzung zugelassen, weitergeleitet	200	138.965	438 / 857	28,57
mit Challenge belegt (Zwischenseite)	200	195.188	162 / 328	13,73

Arbeitslast	Status	Anfr./s	p50 / p99 (µs)	CPU pro Anfrage (µs)
statisch erlaubt, weitergeleitet	200	145.673	415 / 746	27,27
von der WAF abgelehnt (mit Sitzung)	403	209.507	151 / 306	12,17

`wrk` hält 64 Anfragen gleichzeitig in Bearbeitung, sodass Anfragen auf den CPU-gebundenen weitergeleiteten Pfaden auf die vier CPUs warten; das p99 ist dort das Verteilungsende des Schedulers und nicht die Kosten einer Anfrage. Ein Lastgenerator mit offener Schleife, der mit fester Rate sendet und jede Anfrage ab ihrem vorgesehenen Sendezeitpunkt misst, misst stattdessen die Latenz bei gleicher angebotener Last. Für zugelassene Anfragen im Reverse-Proxy-Modus beträgt das p99 65 µs bei 10.000 Anfragen pro Sekunde und 100 µs bei 100.000; bei 150.000 angebotenen Anfragen sind die vier CPUs gesättigt, 134.344 Anfragen pro Sekunde werden erreicht, und die Warteschlange wächst auf Sekunden (`benchmarks/results/latency-latest.md`). Eine warme Keep-alive-Verbindung bedient jede Entscheidung ohne Heap-Allokation: `cmd/sibuna/tests/steady_state.rs` zählt die Allokationen des Daemons, und `crates/net/tests/steady_state.rs` findet über 500 weitergeleitete Keep-alive-Anfragen hinweg keine.

Übung 5.1. Ein Client sendet einen Upload von 200 KB. Verfolgen, welche Bytes im 64-KiB-Puffer liegen, welche `relay_body` durch den Upload-Puffer weiterreicht und warum der Proxy den Kopf *vor* dem Weiterreichen an den Ursprungsserver schreiben muss.

Nebenläufiger Zustand ohne Allokation

Lernziele Die Shard-Sperren, die Robin-Hood-Verbrauchsmenge, den GCRA-Begrenzer, die versionierte Sperrtabelle, den MPSC-Ring für Vorfälle und die Read-Copy-Update-Veröffentlichung der Richtlinien-Engine mit `Arc` und einem generationsgeprüften Cache analysieren.

Jeder Speicher in `sibuna-store` allokiert seine feste Kapazität einmal beim Aufbau und allokiert danach nie wieder und blockiert nie auf einer globalen Sperre. Keiner braucht einen Hintergrund-Aufräumer: Abgelaufener Zustand wird an Ort und Stelle zurückgewonnen.

Sperren und Sharding

Die kritischen Abschnitte in Sibunas Tabellen sind einige Dutzend Instruktionen lang, weit kürzer als ein Rundlauf über einen Kernel-Futex. Tabellen sind nach Schlüsselhash in 16 Shards

aufgeteilt, sodass zwei Operationen mit Wahrscheinlichkeit 1/16 konkurrieren, und jeder Shard liegt auf einer eigenen Cache-Zeile, sodass benachbarte Sperren kein False Sharing verursachen. Die Tabellen verwenden `std::sync::Mutex`: ein Compare-and-Swap ohne Konkurrenz, ein kurzes Spinnen, solange der Halter läuft, dann ein Futex-Warten. Die Alternative ist ein Spinlock mit zwei atomaren Zuständen und einem Spin-Loop-Hinweis in der Warteschleife. SID 0010 verlangte, zwischen beiden anhand von Messungen zu wählen. Ein Spinlock spart ohne Konkurrenz etwa eine Nanosekunde pro Operation, verliert aber unter Konkurrenz:

Tabelle 8 Durchsatz der Shard-Sperren auf einem Ryzen 9 5950X (Oktober 2026)

Arbeitslast	`Mutex`	Spinlock
Verbrauchsmenge, 16 Threads	10,8 M/s	8,6 M/s
Verbrauchsmenge, 64 Threads (mehr als CPUs)	11,2 M/s	6,9 M/s
GCRA auf sechs Hotspot-Adressen, 16 Threads	44,9 M/s	19,3 M/s
GCRA auf sechs Hotspot-Adressen, 64 Threads	37,4 M/s	6,5 M/s

Ein Edge-Server führt mehr Verbindungsthreads aus, als er CPUs hat, und ein Spinlock, dessen Halter verdrängt wird, lässt jeden Wartenden seine Zeitscheibe verbrennen; deshalb behalten wir den Mutex. Eine vergiftete Sperre wird betreten, statt die Vergiftung weiterzugeben: Keine Operation gerät in Panic, während sie einen Shard hält, und der Shard bliebe strukturell gültig, selbst wenn eine es täte.

Die Robin-Hood-Verbrauchsmenge

Nur angenommene Nachweise gelangen in die Verbrauchsmenge (Teil III). Sie hat 16 Shards, jeder ein Array mit offener Adressierung aus 4.096 Slots vom Typ `Option<Entry>`, wobei ein Eintrag `{ tag: [u8; 16], expires_at: u64, dist: u8 }` ist. Robin-Hood-Einfügen (Celis, 1986) verdrängt jeden Eintrag, der seiner Heimatposition näher ist als der eingehende Schlüssel, und hält so die Sondierungslängen eng beieinander; Suchen halten an, sobald sie auf einen Eintrag treffen, der seiner Heimatposition näher ist als sie der ihren.

Implementierung: `Table::insert` Fügt ein verbrauchtes Tag mit seinem Ablauf ein und gewinnt unterwegs abgelaufene Slots zurück. In `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(());
        };
        // Einen abgelaufenen Eintrag zu überschreiben ist nur sicher, wenn
        // die neue Distanz nicht kleiner ist; so bestehen Schlüssel, die
        // hier vorbeisondieren, weiterhin den Test für den frühen Abbruch.
        if e.is_expired(now) && carry.dist >= e.dist {
            *e = carry;
            return Ok(());
        }
        if carry.dist > e.dist {
            std::mem::swap(&mut carry, e);
            // Ein verdrängter abgelaufener Eintrag wird einfach verworfen.
            if carry.is_expired(now) {
                return Ok(());
            }
        }
        idx = (idx + 1) % SHARD_CAPACITY;
        carry.dist += 1;
    }
    Err(SpendError::Full)
}
```

Die Regel für abgelaufene Slots ist der heikle Teil: Ein abgelaufener Eintrag darf nur von einem Schlüssel an Ort und Stelle überschrieben werden, der mindestens ebenso weit von seiner Heimatposition entfernt ist; sonst würde ein späterer Schlüssel, der über den Slot hinaus sondiert, zu früh anhalten und verloren gehen. Abgelaufene Einträge, die verdrängt werden, werden einfach verworfen. Ein Aufräum-Thread ist nicht nötig. Das Einfügen simuliert zuerst den Verdrängungspfad, ohne etwas zu ändern (`can_insert`), und prüft, dass begrenzte Verdrängung gelingen kann, ohne ein lebendes Tag zu verlieren.

> **Invariante** Ein abgewiesenes Einfügen verdrängt nie ein lebendes Tag. Ein Tag, das verbraucht und nicht abgelaufen ist, wird daher immer gefunden, und die Abweisung (`SpendError::Full`) kann den Dienst verweigern, aber nie eine Wiederholung ermöglichen.

Ein Slot ist `Option<Entry>`. Ein leerer Slot hat kein Tag zum Lesen, also gibt es kein `occupied`-Flag, dessen Prüfung ein Leser vergessen könnte, und `match` kompiliert erst, wenn beide Fälle behandelt sind. Der Fehlertyp ist ebenso genau: Die Verbrauchsmenge kann auf zwei Arten scheitern (`DoubleSpend` und `Full`), und `SpendError` benennt genau diese.

GCRA: Ratenbegrenzung in einer Ganzzahl

Der Generic Cell Rate Algorithm (ATM Forum, 1996) ist die Virtual-Scheduling-Form des Leaky Bucket. Pro Client speichert er eine *theoretische Ankunftszeit* (TAT). Mit dem Emissionsintervall $T = \max(1, \lceil W/N \rceil)$ für ein positives Limit N pro Fenster W und der Bursttoleranz $\tau = (N-1)T$ gilt:

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

Satz (Token-Bucket-Schranke) Nach der ersten konformen Ankunft bei t_1 erfüllt die k-te konforme Ankunft $t_k - t_1 \ge (k-1)T - \tau$. Daher enthält jedes Intervall der Länge L höchstens $N + \lfloor L/T \rfloor$ konforme Ankünfte: einen Burst von N, dann eine pro T. Über den definierten Burst hinaus gibt es kein Artefakt fester Fenstergrenzen und keine Historie pro Anfrage.

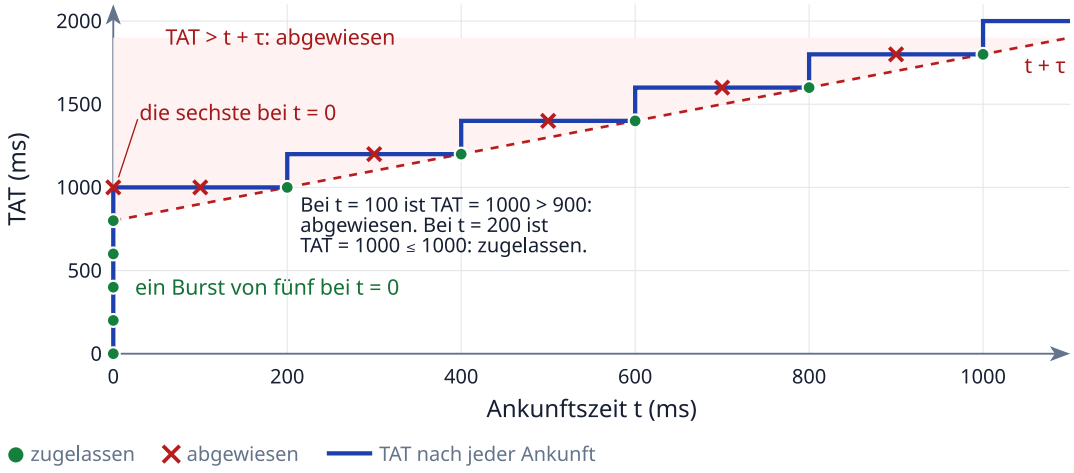


Abb. 7 GCRA-Zulassung für ein Limit von 5 pro Sekunde: ein Burst von fünf, dann eine alle 200 ms. Eine Ankunft wird zugelassen, wenn die TAT, die sie vorfindet, auf oder unter der Linie $t + \tau$ liegt.

Durchgerechnetes Beispiel: fünf pro Sekunde von Hand Wir nehmen $N = 5$ und $W = 1000$ ms, also $T = 200$ ms und $\tau = 800$ ms, und ein bisher inaktiver Client sendet sechs Anfragen bei $t = 0$. Die erste findet eine leere Zelle vor: $\mathrm{TAT} = \max(0, 0) = 0 \le 800$, also ist sie konform, und $\mathrm{TAT} \leftarrow 200$. Die nächsten vier sind auf dieselbe Weise konform und hinterlassen $\mathrm{TAT} = 1000$. Die sechste findet $1000 > 0 + 800$ vor und wird mit `Retry-After` $\mathrm{TAT} - \tau - t = 200$ ms abgewiesen. Bei $t = 199$ findet eine Anfrage noch $1000 > 999$ vor und wird abgewiesen; bei $t = 200$ findet sie $1000 \le 1000$ vor, ist konform und hinterlässt $\mathrm{TAT} = 1200$. Der Unit-Test `gcra_admits_exactly_the_burst_then_drains_at_the_emission_rate` in `crates/store/src/limiter.rs` prüft genau diese Zahlen.

Implementierung: `arrive` Wendet eine Ankunft auf die Zellen eines gesperrten Shards an und gibt die Entscheidung, die Wiederholungsverzögerung und das verbleibende Budget zurück. In `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)
}
```

`check` des Shards nimmt die Sperre und ruft `arrive` mit den gesperrten Zellen auf; der Guard gibt die Sperre frei, wenn `check` zurückkehrt. Zellen sind 16 Bytes groß (ein Schlüssel und eine TAT) und liegen in 16 Shards zu 512 Slots mit einem Sondierungsfenster von 16 Slots, insgesamt etwa 128 KiB. Eine Zelle, deren TAT älter als $t - \tau$ ist, ist leergelaufen und wird sofort zurückgewonnen. Der `Retry-After`-Header wird mit derselben Arithmetik berechnet. Gesättigte Sondierungsfenster weisen neue Clients ab, statt aktiven Kontingentzustand zu verdrängen, und die Entscheidung sagt das (`capacity_exhausted`), sodass ein Aufrufer eine Abweisung wegen Sättigung nicht zu einer Sperre eskaliert. Eine konfigurierte Rate von null weist Anfragen ab. Ein Begrenzer bedient mehrere Regeln: Der Zellschlüssel ist ein Wyhash der Clientadresse, mit dem Geltungsbereich der Regel als Seed, sodass derselbe Client in verschiedenen Geltungsbereichen unabhängige Budgets hat.

Die versionierte Sperrtabelle

Sperren verwenden 4.096 Slots mit atomaren Feldern für Version, Schlüssel und Ablauf, offen adressiert über einen schlüsselbasierten Hash der gesperrten Adresse mit einem Sondierungs-

fenster von 8 Slots. Ein Schreiber, durch einen Mutex mit den anderen Schreibern serialisiert, umklammert jede Änderung mit Versionserhöhungen; Leser nehmen keine Sperre, akzeptieren nur eine stabile gerade Version und wiederholen bei gleichzeitigen Änderungen.

Implementierung: `BanTable::ban` und `BanTable::read_slot` Die Klammer des Schreibers und der konsistente Schnappschuss des Lesers. In `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)];
    // Schlüssel und Ablauf sind ein logischer Wert: Eine ungerade Version
    // klammert die Ersetzung ein, damit kein Leser einen alten Schlüssel mit
    // dem Ablauf des nächsten Schlüssels kombiniert.
    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);
        }
    }
}
```

Sequenzielle Konsistenz verhindert, dass eine alte Identität mit einem neuen Ablauf vermischt wird: Jeder Zugriff gehört zu einer einzigen totalen Ordnung, was das Argument einfach hält. Leser nehmen keinen Mutex, können aber auf einen Schreiber warten und sind nicht wartefrei. Diese atomaren Operationen haben echte Kosten, die indirekt im HTTP-Messprogramm gemessen und nicht als kostenlos angenommen werden. Ein volles Sondierungsfenster überschreibt einen abgelaufenen Slot, andernfalls den Slot, der zuerst abläuft. `clear` hebt jede Sperre auf, lässt aber die Schlüssel stehen, weil ein geleerter Slot einen kollidierenden Schlüssel weiter hinten in einer Sondierungskette verbergen könnte.

Der MPSC-Ring für Vorfälle

Wenn die Betriebsform Shield eine Anfrage ablehnt oder der Honeypot auslöst, wird der Vorfall in einen festen Datensatz von etwa 1,3 KB kopiert (Clientadresse, User-Agent, Methode, Pfad, Kategorie und Nutzlast, jeweils auf ihre Kapazität gekürzt) und in einen begrenzten

Ring mit mehreren Produzenten und einem Konsumenten geschoben (Vyukovs Entwurf mit Sequenzstempeln, 512 Slots, `BoundedQueue<IncidentRecord, 512>`). Produzenten sind Verbindungsthreads; der Konsument ist der Speicherthread. Ein voller Ring verwirft den neuesten Datensatz, statt eine Antwort zu blockieren: eine ausdrückliche Verlustrichtlinie, keine Haltbarkeitsgarantie. Der Verwerfungszähler macht diesen Verlust beobachtbar.

```
/// Kopiert `incident` in den Ring; ist er voll, wird verworfen und gezählt.
pub(crate) fn record(&self, incident: &Incident<'_>) {
    if self.queue.push(IncidentRecord::from_incident(incident)).is_err() {
        self.metrics.incidents_dropped.bump();
    }
}
```

Nachdem der Speicherthread einen Datensatz in einen ausstehenden Batch verschoben hat, behält er ihn zusammen mit den vorbereiteten Zeilen, bis der Commit bestätigt ist; bei einem Datenbankfehler verwirft er ihn nicht stillschweigend, und eine Wiederholung schreibt den identischen Batch.
Jeder Slot trägt eine Sequenznummer, die Produzenten sagt, ob der Slot frei ist, und dem Konsumenten, ob er voll ist, sodass keine Seite eine Sperre nimmt. Ein solcher Ring lässt sich nicht in sicherem Rust schreiben: Produzenten und Konsument berühren gleichzeitig verschiedene Slots eines Arrays, und nur das Sequenzprotokoll, nicht das Typsystem, beweist, welcher Thread welchen Slot berühren darf. Das Modul `crates/store/src/ring.rs` hält seine Werte daher in Slots vom Typ `UnsafeCell<MaybeUninit<T>>`, und jeder `unsafe`-Block nennt den Schritt des Protokolls, der ihn rechtfertigt:

```
if diff == 0 {
    // Relaxed: Der CAS entscheidet nur, welcher Produzent `pos` erhält.
    match self.tail.0.compare_exchange_weak(
        pos, pos.wrapping_add(1), Ordering::Relaxed, Ordering::Relaxed,
    ) {
        Ok(_) => {
            // SAFETY: Der CAS hat diesem Thread die Position `pos` gegeben, und
            // `sequence == pos`: Der Slot ist leer; kein anderer Produzent
            // kann `pos` beanspruchen, und der Konsument liest ihn nicht vor
            // dem Release-Store unten.
            unsafe { (*slot.value.get()).write(value) };
            // Release: macht den Wert für den Konsumenten sichtbar.
            slot.sequence.store(pos.wrapping_add(1), Ordering::Release);
            return Ok(());
        }
        Err(current) => pos = current,
    }
} else if diff < 0 {
    return Err(value);
}
```

Zwei Konventionen des Rings sind Typen, keine Kommentare. Bei einem vollen Ring gibt `push` `Err(value)` zurück und reicht den Datensatz zurück, sodass der Aufrufer entscheidet, ob er ihn zählt oder verwirft; ein bloßes `false` würde ihn verlieren. Der Ring hat einen einzigen Konsumenten: `pop` nimmt `&mut self`, und gemeinsam genutzter Code muss zuerst den

eindeutigen `Consumer` beanspruchen, sodass ein zweiter Konsument nicht kompiliert oder zur Laufzeit abgewiesen wird.

Veröffentlichung der Engine per Read-Copy-Update

Die Richtlinien-Engine wird abseits des Hot Path neu gebaut, wenn sich Richtlinien ändern. Anfragen dürfen nie eine halb gebaute Engine sehen, und eine Engine darf nie freigegeben werden, solange eine Anfrage sie noch liest. Sibuna veröffentlicht jede Engine als `Arc<Engine>` in einem `EngineSlot`, und jede Verbindung liest sie über einen `EngineCache`:

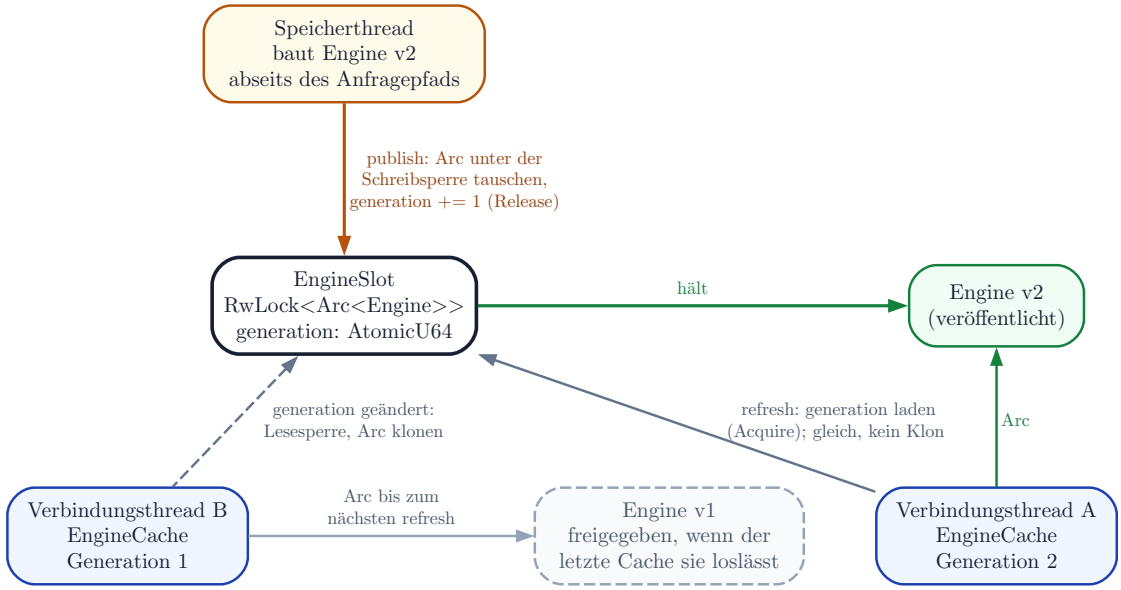


Abb. 8 Veröffentlichung der Engine. Der Speicherthread veröffentlicht eine vollständige Engine und erhöht die Generation; jede Verbindung prüft die Generation mit einem Acquire-Ladevorgang und klont den veröffentlichten Arc nur nach einer Veröffentlichung. Eine alte Engine wird freigegeben, wenn der letzte Cache, der sie hält, sie loslässt.

Implementierung: `EngineSlot::publish` und `EngineCache::refresh` Die Veröffentlichung einer vollständigen Engine und die Sicht einer Verbindung darauf. In `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);
    // Die alte Engine wird außerhalb der Sperre freigegeben (als letzter Halter).
    drop(previous);
}

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

pub fn refresh(&mut self, slot: &EngineSlot) -> &Engine {
    let generation = slot.generation();
    if self.generation != generation {
        // Eine Veröffentlichung zwischen Laden und Klonen versieht eine neuere
        // Engine mit einer älteren Generation; der nächste refresh lädt erneut.
        self.generation = generation;
        self.engine = None;
    }
    self.engine.get_or_insert_with(|| slot.load())
}
```

Die Generation wird mit Release-Ordnung gespeichert, nachdem die Engine an ihrem Platz ist, und mit Acquire-Ordnung geladen: Ein Leser, der eine neue Generation sieht, sieht auch die zuvor gespeicherte Engine. Der im Kommentar genannte Wettlauf ist harmlos. Fällt eine Veröffentlichung zwischen das Laden der Generation durch einen Leser und sein Klonen, so speichert der Leser die neuere Engine unter der älteren Generationsnummer, und seine nächste Aktualisierung bemerkt die Abweichung und lädt erneut. Ein Leser kann für eine Anfrage eine Engine im Rückstand sein; eine halb gebaute Engine kann er nie sehen, weil nur vollständige Engines im Slot gespeichert werden.

Invariante Jede Engine, die eine Anfrage liest, ist vollständig, und keine Engine wird freigegeben, solange eine Anfrage oder ein Verbindungscache sie hält. Das Erste gilt, weil eine Engine vollständig gebaut ist, bevor `publish` sie speichert; das Zweite, weil jeder Leser einen `Arc` hält und die Engine erst mit ihrem letzten `Arc` freigegeben wird.

Warum ein Cache und nicht einfach eine Sperre? SID 0010 verlangte, die Lesekosten mit 16 Threads zu messen und zwischen `RwLock<Arc<Engine>>` und einer sperrfreien Alternative in sicherem Rust zu wählen. Das Lesen über die Sperre berührt das Sperrwort und den Referenzzähler, zwei Cache-Zeilen, die jeder Kern beschreibt. Die Generation dagegen wird einmal pro Veröffentlichung geschrieben und sonst nur gelesen, sodass jeder Kern seine Kopie behält:

Tabelle 9 Kosten eines Engine-Lesevorgangs, Ryzen 9 5950X, 16 Threads lesen in einer Schleife (Oktober 2026)

Leser	1 Thread	16 Threads
nur `RwLock`-Lese-Guard	4,2 ns	752 ns

Leser	1 Thread	16 Threads
`RwLock`-Lesen und `Arc`-Klon	7,6 ns	833 ns
Generationsprüfung mit zwischengespeichertem `Arc`	0,8 ns	1,8 ns

Unter Konkurrenz ist das zwischengespeicherte Lesen mehr als 400-mal billiger als die Sperre, daher nutzt der Anfragepfad den Cache und den `RwLock` nur auf dem langsamen Pfad einer Aktualisierung. Die Messung ist der ignorierte Test `read_cost_under_contention` in `cmd/sibuna/src/slot.rs`. Jede Verwendung auf dem Anfragepfad aktualisiert, sodass eine Anfrage die zu diesem Schritt aktuelle Engine sieht. Die Engine selbst ist klein: 209.052 Bytes, etwa 204 KiB, kompakte Heap-Tabellen (`engine_bytes` in `benchmarks/results/primitives-latest.json`), mit Kapazität für 128 Regeln, Regelnamen von 128 Bytes und 8.192 Trie-Knoten.

Warum `Arc` und keine Leserzähler Die Alternative mit festem Speicher hält zwei Engine-Slots, jeder mit einem Leserzähler und einer eigenen Allokationsarena. Eine Anfrage fixiert einen Slot, indem sie seinen Zähler erhöht und den Slot-Zeiger erneut prüft, und beide atomaren Variablen müssen sequenziell konsistent sein, um den Wettlauf zu schließen, in dem ein Schreiber zwischen Laden und Erhöhen durch den Leser tauscht. Der Builder tauscht den Zeiger und spinnt dann, bis die Leser des alten Slots abgeflossen sind, bevor er die Arena dieses Slots zurücksetzt und neu in sie baut. Ein Leser, der einen Slot über einen Neubau hinweg hält, blockiert daher den Builder, und ein langsamer Ursprungsserver würde die nächste Veröffentlichung aufhalten, sofern nicht jede Anfrage ihren Slot freigibt, bevor die Proxy-I/O beginnt.
Referenzzählung mit `Arc` braucht nichts davon. Der Speicherthread baut aus jedem Schnappschuss eine frische `Engine` und veröffentlicht sie; nichts wird wiederverwendet, also gibt es keinen Reserveslot, kein Zurücksetzen einer Arena und kein Spinnen: `publish` wartet nie auf Leser. Eine alte Engine lebt einfach, bis ihr letzter Halter sie loslässt, und ein langsamer Ursprungsserver kann nur diesen Zeitpunkt verzögern, nie die nächste Veröffentlichung. Das Urteil kopiert den Namen der getroffenen Regel in einen 128-Byte-Puffer, um eine Ausleihe zu beenden: Die Entscheidung leiht die Engine über den Cache der Verbindung, und der Antwortpfad braucht den ganzen Verbindungskontext veränderlich. Eine Verbindung nach einem Upgrade, die Stunden leben kann, gibt ihre zwischengespeicherte Engine frei, bevor sie weitergeleitet wird.

Übung 5.2. Mit dem GCRA-Satz die Höchstzahl der Anfragen berechnen, die ein einzelner Client in den ersten 3 Sekunden nach einer Ruhephase durchbringen kann, für `--rate-limit 100 --rate-window 10`. Mit einem Zähler mit festem 10-Sekunden-Fenster und mit einem echten Sliding-Window-Log vergleichen.

Die Invariante erklären. An einem Beispiel mit drei Slots erklären, warum ein abgelaufener Robin-Hood-Eintrag nur von einem Schlüssel mit mindestens ebenso großer Distanz überschrieben werden kann.

TEIL VI

Automaten, Tries und die semantische Firewall

Wir untersuchen die Algorithmen hinter der Klassifikation in weniger als einer Mikrosekunde: den markierten Aho-Corasick-Automaten, den Radix-Trie für IPv4/IPv6, die Byteklassen-Tokenizer der semantischen WAF und die deklarative Richtlinien-Engine mit WEIGH-Bewertung.

Suche nach vielen Mustern in einem Durchlauf

Lernziele Am Ende dieses Kapitels sollten wir die Linearzeitgarantie des Aho-Corasick-Automaten herleiten, das dichte Tabellenlayout und die Groß-/Kleinschreibungsfaltung über Byteklassen erklären und begründen können, warum keine SIMD-Literal-Engine übernommen wurde.

Aho-Corasick als dichter DFA

Sibuna hat zwei Mengen literaler Muster: etwa vierzig Bot-Signaturen und etwa hundert Angriffssignaturen. Ein Feld nacheinander gegen N Muster zu durchsuchen kostet $O(NM)$; der Automat von Aho und Corasick (1975) durchsucht es nach dem Aufbau eines Tries mit Fehlerkanten einmal in $O(M)$. Sibuna faltet die Fehlerkanten beim Aufbau in die Übergangstabelle, sodass die Suche ein Tabellenzugriff pro Eingabebyte ist, ohne Verzweigungen über die Mustermenge. Der Automat liegt im Crate `sibuna-patterns`:

```
/// Rückt von `state` (einem Zeilenoffset) über `byte` vor.
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
}
```

Ein Zustand wird als sein Zeilenoffset in einer flachen Tabelle gespeichert, sodass ein Schritt aus einer Addition und einem Ladevorgang besteht. Die Zustände sind so nummeriert, dass jeder Trefferzustand nach allen anderen Zuständen kommt; einen Treffer zu erkennen ist dann ein einziger Vergleich mit `first_match`, ohne eine zweite Tabelle in der inneren Schleife zu laden.

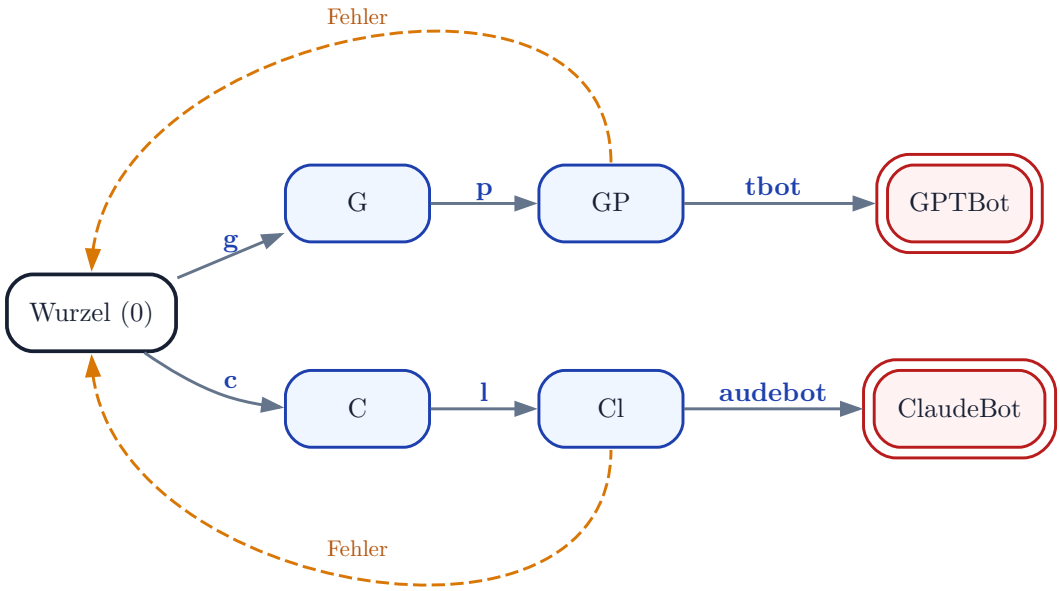


Abb. 9 Trie-Zustände und Fehlerkanten für zwei Signaturen

Das Zustandsbudget des Automaten ist eine Laufzeitoption. Die Richtlinien-Engine reserviert höchstens eine Wurzel plus die Summe aller Signaturlängen, eine sichere obere Schranke für die Trie-Zustände. Das vermeidet die früher übergroßen festen Tabellen und behält einen dichten Übergang für jede Byteklasse. Jedes Muster trägt eine kleine Kategoriemarke (0 bis 63, sodass eine Menge von Marken ein 64-Bit-Wort ist). Die Faltung der Groß-/Kleinschreibung nutzt eine Byteklassentabelle mit 256 Einträgen, die mit dem Automaten aufgebaut wird: Beide Schreibweisen eines Buchstabens teilen eine Klasse, und Bytes, die in keinem Muster vorkommen, teilen sich eine Klasse. Eine Tabellenzeile ist daher so breit wie die Zahl der verschiedenen Musterbytes plus eins statt 256. Die genauen Engine-Größen und der Vergleich mit der sequenziellen Teilzeichenkettensuche stehen in Teil VIII. Letztere nutzt dieselben Muster und dieselbe Semantik „irgendein Treffer“.

Reguläre Ausdrücke, SIMD-Engines und die Wahl

Für die Signaturstufe wurden drei Familien abgewogen:

- *Reguläre Ausdrücke mit Backtracking* (im Stil eines Backtracking-Regelwerks) führen Hunderte Muster pro Anfrage aus und allokieren pro Treffer; sie sind die langsamste Option und am anfälligsten für Fehlalarme.
- *SIMD-Literal-Engines* (Hyperscan, Vectorscan) filtern Blöcke von 16 bis 32 Bytes mit Nibble-Masken auf Shuffle-Basis vor (der „Teddy“-Algorithmus) und verifizieren Kandidaten mit Automaten. Bei langen Eingaben mit großen Literalmengen glänzen sie. Der

Preis ist eine C++-Abhängigkeit von mehreren Megabyte, eine zur Übersetzungszeit gebaute Musterdatenbank, kein WebAssembly-Ziel und, bei Eingaben von hundert Bytes, kein Vorteil: Das ganze Feld ist schneller durchsucht, als ein einziger Cache-Miss dauert.
- *Dichter Aho-Corasick plus kleine strukturelle Tokenizer*, die Linie von libinjection, liefert exaktes Linearzeitverhalten in einer festen Tabelle und braucht nichts außerhalb der Binärdatei.

Sibunas Felder sind kurz, daher gewinnt die dritte Familie sowohl beim Speicher als auch bei der Geschwindigkeit; Teil VIII verzeichnet etwa 1,56 ns pro Byte für einen 8-KiB-Body (der semantische Scan dauert 12.784,3 ns, `8kb_body_semantic_scan` in `benchmarks/results/primitives-latest.md`). Das ist die eine Arbeitslast, bei der sich ein SIMD-Vorfilter lohnen würde, und SID 0006 führt ihn als offenen Punkt.

Der Radix-Trie für IPv4 und IPv6

> **Lernziele** Eine Suche nach dem längsten Präfix durch den 128-Bit-Trie verfolgen, die Abkürzung über die IPv4-Wurzel erklären und sehen, wie der Trie zugleich als vom Cluster gespeister Reputationsspeicher dient.

Ein binärer Trie über 128-Bit-Schlüssel bedient beide Familien: IPv6-Präfixe werden unverändert eingefügt, IPv4-Präfixe werden in `::ffff:0:0/96` abgebildet, sodass ein `/24` zu einem Pfad der Tiefe 120 wird. Die Knoten liegen in einem flachen `Vec`, indiziert mit `u32`, wobei Index null zugleich Wurzel und „kein Kind“ bedeutet. Eine Suche besteht so aus höchstens 128 abhängigen Ladevorgängen ohne Zeigerverfolgung. Der Trie ist der `PrefixTrie` des Crates `sibuna-cidr`.

Die 96 Ebenen des abgebildeten Präfixes bei jeder IPv4-Suche zu durchlaufen, würde 96 abhängige Ladevorgänge kosten, bevor das erste Bit der IPv4-Adresse erreicht ist. Deshalb merkt sich der Trie bei der Initialisierung den Knoten auf Tiefe 96 und beginnt IPv4-Suchen dort:

```
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())
};
```

Eine IPv4-Adresse zu klassifizieren kostet 32,0 ns, eine IPv6-Adresse 79,3 ns (`ipv4_cidr_classification` und `ipv6_cidr_classification` in `benchmarks/results/primitives-latest.md`). Die Engine befragt den Trie zweimal: terminale `allow`/`deny`-Urteile vor der Regeltabelle, sodass eine clusterweite Sperre jede Regel schlägt, und `challenge`-Urteile danach.

Die semantische Firewall

Lernziele Die Byteklassen-Vorabsuche, den markierten Signaturautomaten, den SQL-Tokenizer mit Tautologieerkennung, die Strukturprüfungen für HTML und Shell, die Kanonisierung und die Disziplin gegen Fehlalarme verstehen, die eine echte Browseranfrage unverändert passieren lassen muss.

Eine Lektion über Fehlalarme

Die erste Version der WAF verglich eine flache Liste von Teilzeichenketten mit jedem Header. Die Liste enthielt `/*`, und jeder Browser sendet `Accept: text/html,...,*/*;q=0.8`. Jeder echte Browser wurde als SQL-Injection eingestuft und erhielt `403`. Die Lektion prägte den Entwurf: Ein Detektor darf nur bei *Struktur* anschlagen, und die Testsuite schickt jetzt eine vollständige Chrome-Anfrage, einschließlich eines Kommentar-Bodys mit `select`, `--` und einem Apostroph, durch die Engine und stellt sicher, dass sie sauber herauskommt.

Byteklassen in einem Durchlauf

Jedes geprüfte Feld wird einmal durchsucht, um festzustellen, welche Byteklassen vorkommen: Anführungszeichen, `=`, `<`, Shell-Trennzeichen, `%`, NUL, Auslöser der Kanonisierung und doppelte Leerzeichen. Jeder Detektor befragt die Klassen, statt erneut zu suchen, und der kanonisierte zweite Durchlauf läuft nur, wenn ein Auslöser gesehen wurde. Die Suche ist ein Zugriff auf eine Tabelle mit 256 Einträgen pro Byte, aus dem WAF-Modul von `sibuna-policy`. Vier Akkumulatoren halten die OR-Ketten kurz, und nach den beiden Bytepaaren (`/*` und zwei benachbarte Leerzeichen) wird nur gesucht, wenn noch kein einzelnes Byte die Kanonisierung verlangt hat:

```
pub(crate) fn scan(text: &[u8]) -> Classes {
    let (chunks, remainder) = text.as_chunks::<8>();
    // Vier Akkumulatoren halten die OR-Ketten kurz, damit die Loads dicht folgen.
    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)];
    }
    // Die Paare zählen nur, solange kein Byte allein die Kanonisierung verlangt
hat.
    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 }
}
```

Detektoren

- *Signaturen.* Alle starken Signaturen jeder Kategorie liegen in einem markierten Automaten; ein Treffer ist ein terminales `deny` mit dem Namen `waf:sqli`, `waf:xss`, `waf:path-traversal` oder `waf:rce`.
- *SQL-Injection.* Ein Wortscanner mit einem Durchlauf schlägt jedes alphanumerische Wort (höchstens sieben Bytes) in der Schlüsselworttabelle nach und prüft bei `or` / `and` auf eine Tautologie `literal = literal`; `id=1 or 1=1` ist terminal. Andernfalls ist ein Anführungszeichen-Byte Pflicht, und die Indizien müssen mindestens 4 Punkte erreichen aus: Anführungszeichen (1), bis zu zwei Schlüsselwörter (2), ein Kommentarzeichen (1), ein `=` (1). `admin'--` ist für sich allein terminal. „it's a group order from the shop“ erreicht 3 Punkte und passiert.
- *Cross-Site-Scripting.* Erfordert `<`; schlägt an, wenn ein Tag, das Skript aufnehmen kann, zusammen mit einem Attribut `on<event>=` an Attributposition vorkommt. `onboarding=1` zählt nicht.
- *Command Injection.* Erfordert ein Trennzeichen; schlägt an, wenn auf `;`, `|`, `&&`, `$(`, einen Backtick oder einen Zeilenumbruch ein bekannter Befehlsname folgt. `status ok; done | next` passiert.
- *Path Traversal.* Signaturen (`../`, `..%2f`, `/etc/passwd`, `id_rsa`, ...) sowie NUL und `%00`.

Strukturelle Header, deren Grammatik legitim Anführungszeichen und Sterne enthält (`Accept*`, `Content-Type`, `Sec-*`, `If-*`, ...), werden übersprungen; Pfad, Query, User-Agent, die übrigen Header und die ersten 8 KB des Bodys werden geprüft. Felder mit Prozent-Escapes, Pluszeichen, Kommentaröffnern oder zusammenfassbarem Leerraum werden in einen 8-KB-Stack-Puffer kanonisiert und erneut geprüft, sodass `1%27/**/UnIoN/**/SeLeCt` und `%252e%252e/` auf die rohen Signaturen zusammenfallen.

Kosten

Die Kosten der Prüfung hängen von den Feldern und Bytes ab, die der Engine vorgelegt werden, nicht nur von der Zahl der Anfragen. Das Benchmark-Kapitel berichtet getrennte Fälle für Gate, Shield und einen 8-KB-Body. Die Grenze von 8 KB ist ein Prüflimit, keine Aussage, dass der Rest eines Uploads sicher ist. Ein Eingangsproxy im Forward-Auth-Modus muss die relevanten Anfrageinformationen weitergeben; ausgelassene Bytes kann dieser Prozess nicht prüfen.

Die deklarative Richtlinien-Engine

Lernziele Das Regelmodell, die Auswertungsreihenfolge, die WEIGH-Bewertung mit Schwellen und die JSON-Grammatik der Richtlinien lesen.

Eine Regel ist eine Konjunktion optionaler Kriterien (Pfadmuster, User-Agent-Muster, bis zu vier Header, bis zu acht IPv4/IPv6-CIDR-Blöcke, ein JA4-Fingerabdruck oder -Präfix) mit einer Aktion `ALLOW`, `DENY`, `CHALLENGE` oder `WEIGH`, optionaler Schwierigkeit pro Regel in Arbeitsbits und optionalem Algorithmus sowie einem vorzeichenbehafteten `weight`. Auswertungsreihenfolge:

1. Semantische WAF (nur Shield): Ein Verstoß ist ein terminales `deny`.
2. Reputations-Trie: `deny`- oder `allow`-Präfixe sind terminal.
3. Regeln der Reihe nach: Der erste terminale Treffer liefert das Ergebnis; `WEIGH`-Treffer summieren sich.
4. Bewertung: Negative Summen ergeben `allow`; Summen ≥ 40 ergeben `deny`; Summen ≥ 10 ergeben `challenge` mit einem zusätzlichen Arbeitsbit je 5 Punkte über der Schwelle, höchstens sechs.
5. Statische Bypass-Pfade, dann `challenge`-Urteile des Tries, dann der Bot-Automat.
6. Standardaktion, `challenge`, sofern die Richtliniendatei nichts anderes sagt.

Eingebaute Regeln erlauben `/.well-known/*`, `/favicon.ico`, `/robots.txt` und `/__sibuna/*`, lehnen `CF-Worker`-Clients und `Amazonbot` ab und belegen jeden `Mozilla`-User-Agent mit einer Challenge: Eine Browserzeichenkette lässt sich kostenlos fälschen, sie zuzulassen würde das Gate zur Dekoration machen.

```
{
  "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 }
  ]
}
```

Durchgerechnete Ablaufverfolgung: ein kodierter Angriff mit gültiger Sitzung

Betrachten wir `GET /search?q=%2527%2520OR%25201%253D1` von einem Client mit gültiger Sitzung. Zuerst trennt der HTTP-Parser Pfad und Query, ohne ihre Bytes zu kopieren. Die lokalen Limits laufen, dann prüft Shield die Anfrage. Das Prozentzeichen markiert das Feld für die Kanonisierung. Ein Dekodierdurchlauf legt `%27%20OR%201%3D1` frei; ein zweiter legt `' OR 1=1` frei.

Der Tokenizer sieht nun ein Anführungszeichen, das Schlüsselwort `OR` und gleiche literale Operanden. Eine Ablehnung ist ein terminales Richtlinienergebnis. Die gültige Sitzung macht daraus keine Erlaubnis: Die Sitzungsprüfung löst nur eine Challenge-Entscheidung auf. Der Speicher-Hook kopiert einen begrenzten Vorfallsdatensatz; SQL läuft später. Der Antwortpfad

muss den Engine-Schnappschuss nicht halten, solange eine Verbindung zum Ursprungsserver untätig ist.

Übung 6.1. Die Nutzlast durch `O'Reilly` und dann durch `order=1` ersetzen. Warum sollte keines davon allein SQL-Injection belegen? Ein Beispiel für einen Fehlalarm angeben, den ein Detektor erzeugen könnte, der nur Teilzeichenketten vergleicht.

Übung 6.2. Die Einträge der Byteklassentabelle aufschreiben, die nötig sind, um einen Detektor für LDAP-Injection hinzuzufügen (Muster wie `)(|(`), und erklären, welche vorhandenen Klassenbits er wiederverwenden kann.

Die Invariante erklären. Einem Reviewer erklären, warum `Accept: */*` ein Fehlalarm war, welche strukturelle Eigenschaft der neue SQL-Detektor verlangt und wie Automat und Tokenizer die Arbeit aufteilen.

TEIL VII

Die Proof-of-Work-Engine im Browser

Wir betrachten das WebAssembly-Modul, das das Verifizierer-Crate des Servers selbst kompiliert, das Web-Worker-Protokoll, die JavaScript-Fallback-Beweiser und die Zwischenseite.

Eine Quelle, zwei Ziele

Lernziele Am Ende dieses Kapitels sollten wir erklären können, wie das Browsermodul aus dem eigenen Crate `sibuna-pow` des Servers gebaut wird, die exportierten Funktionen benennen und den Speicher beschreiben können, den der Beweiser für sequenzielle Arbeit in einem Tab braucht.

Verdrahtung des Builds

Der Browser-Löser kompiliert genau die Quellen, mit denen der Server verifiziert. `cmd/sibuna-pow-wasm` ist ein `no_std`-Bibliotheks-Crate vom Typ `cdylib`, gebaut für `wasm32-unknown-unknown`, dessen einzige Abhängigkeiten die Workspace-Crates `sibuna-pow` und `sibuna-crypto` sind:

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

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

# Cargo.toml (Wurzel des Workspace): Das Modul wird auf Größe optimiert gebaut
[profile.release.package.sibuna-pow-wasm]
opt-level = "z"
debug = false
strip = true
```

Der Daemon bettet das Modul ein. Sein Build-Skript (`cmd/sibuna/build.rs`) führt ein verschachteltes `cargo rustc -p sibuna-pow-wasm --target wasm32-unknown-unknown --profile wasm` in das Zielverzeichnis des Workspace aus (`target/wasm32-unknown-unknown/wasm/`). Der verschachtelte Build sperrt nur die Verzeichnisse des Profils `wasm` und wartet daher nie auf den äußeren Build. Der abschließende Link erhält `--compress-relocations`. Das Skript prüft, dass das Ergebnis mit den magischen Bytes von WebAssembly beginnt, und kopiert es in das Ausgabeverzeichnis des Builds. `SIBUNA_POW_WASM=<path>` bettet stattdessen ein vorgefertigtes Modul ein, für Builds ohne das WebAssembly-Ziel. Der Daemon nimmt die Bytes dann in seine Binärdatei auf:

```
/// Der Löser für den Browser, von `build.rs` aus `cmd/sibuna-pow-wasm` gebaut.
pub const WASM_SOLVER: &[u8] =
    include_bytes!(concat!(env!("OUT_DIR"), "/sibuna-pow.wasm"));
/// Der Web Worker des Lösers.
pub const WORKER_JS: &str = include_str!("../../../web/worker.js");
```

Es gibt keine zweite Implementierung, die auseinanderlaufen könnte. Der Daemon liefert das Modul unter `/__sibuna/wasm/sibuna-pow.wasm` mit einem Cache-Header für eine Stunde aus (`Cache-Control: public, max-age=3600`). Mit dem obigen Profil gebaut, misst das Modul mit beiden Stufen 14.744 Bytes (`wasm_bytes` in `benchmarks/results/primitives-latest.json`).

Exporte

Das Modul exportiert sechs Funktionen, und das Worker-Skript ruft sie beim Namen auf:

Export	Zweck
`sibuna_get_buffer_ptr()` / `sibuna_get_buffer_len()`	256-Byte-Eingabepuffer, in den der Worker die Challenge-Zeichenkette schreibt
`sibuna_solve_step(ptr, len, bits, start, max_steps)`	Hashcash: durchsucht `max_steps` Nonces ab `start`; gibt die Nonce oder den Sentinel aus lauter Einsen zurück
`sibuna_solve_sha256(ptr, len, bits)`	Hashcash in einem Batch ab Nonce 0 (`sibuna_solve_step` mit `u32::MAX` Schritten)
`sibuna_posw_solve(len, depth, challenges)`	Sequenzielle Arbeit: führt den Beweiser über den Puffer aus und gibt die Länge des Nachweises zurück, oder 0 bei ungültigen Argumenten
`sibuna_posw_proof_ptr()`	Beginn der Nachweisbytes im Arbeitsbereich des Beweisers

Der Arbeitsbereich des Beweisers, also die behaltenen oberen Ebenen, der Geschwisterstack und der Nachweis, ist ein statischer Bereich von etwa 92 KiB; es ist kein Allokator eingebunden, und das Modul hat keine Importe. `Workspace::new` ist eine `const fn`, daher kostet der statische Bereich nur genullten Speicher, keine initialisierten Daten im Modul.

Implementierung: `sibuna_solve_step` Der Hashcash-Export: leiht die Challenge aus dem linearen Speicher und führt einen Batch der gemeinsamen Nonce-Suche aus. In `cmd/sibuna-pow-wasm/src/lib.rs` (Crate `sibuna-pow-wasm`).

```
#[unsafe(no_mangle)]
pub unsafe extern "C" fn sibuna_solve_step(
    challenge_ptr: *const u8,
    challenge_len: usize,
    difficulty_bits: u32,
    start_nonce: u64,
```

```
    max_steps: u32,
) -> u64 {
    let challenge: &[u8] = if challenge_len == 0 {
        &[]
    } else {
        // SAFETY: Der Aufrufer garantiert `challenge_len` lesbare Bytes ab
        // `challenge_ptr`, die sich während des Aufrufs nicht ändern.
        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)
}
```

Die Exporte bilden die Fremdfunktionsgrenze des Crates und sind der einzige Ort seines `unsafe`-Codes. Der Eingabepuffer und der Arbeitsbereich des Beweisers sind statische Variablen in einer `ExportCell`, einer Hülle um `UnsafeCell`, die mit angegebener Begründung als `Sync` deklariert ist: Eine WebAssembly-Instanz ohne gemeinsamen Speicher läuft in einem Thread, und jeder Export erzeugt seine Referenzen, nutzt sie und gibt sie frei, bevor er zurückkehrt, sodass höchstens eine Referenz in eine Zelle zugleich lebt. Auf dem Host, wo die Tests die Exporte direkt aufrufen, serialisiert ein nur für Tests vorhandener Mutex sie. Eine Panic löst einen Trap aus (die Instruktion `unreachable`), statt eine Meldung zu formatieren; das hält das Modul frei von Formatierungscode und Importen. Panics sind konstruktionsbedingt unerreichbar, weil die Funktionen von `sibuna-pow` total sind und nie eine Panic auslösen.

Vorab-Hashing des Präfixes

Die innere Hashcash-Schleife absorbiert die Challenge und den Doppelpunkt einmal und klont den SHA-256-Zustand pro Nonce, sodass jeder Kandidat eine Kompression statt zwei kostet. Die Suche ist `sibuna_pow::hashcash::Search`, die das Browsermodul und der native Referenzlöser gemeinsam nutzen:

```
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(); // Kopie des aufgenommenen Präfixes
    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
}
```

Der Hasher-Zustand ist ein einfacher Wert von etwa 110 Bytes, daher ist das Klonen billig. Der Beweiser für sequenzielle Arbeit nutzt denselben Trick, wobei die Aussage χ einmal in `base_hasher` absorbiert und für jeden Knoten geklont wird, und der Verifizierer tut das inzwischen auch: Er leitet jedes geforderte Blatt aus einem Zustand ab, der die Aussage einmal absorbiert hat. Das macht die Verifikation um t Kompressionen billiger, bei denselben Digests. Auf dem Server nutzt die Kompressionsfunktion die SHA-Erweiterungen von x86-64 oder die SHA-2-Instruktionen von ARMv8, zur Laufzeit erkannt. Im Browser gibt es solche Instruktionen nicht, und der Löser verbringt fast seine gesamte Zeit in der portablen Kompressionsfunktion (`crates/crypto/src/sha256/portable.rs`). Diese Funktion ist für WebAssembly geschrieben: Die 64 Runden sind ausgerollt, der Nachrichtenplan liegt in einem Fenster von 16 Wörtern, in dem jede Runde ab Runde 16 das Wort überschreibt, das sie nicht mehr braucht, und die acht Arbeitsvariablen werden von Runde zu Runde umbenannt, statt verschoben zu werden:

```
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);
    // ... Runden 16 bis 55
    eight_rounds!(w, 56, a, b, c, d, e, f, g, h);
    for (word, value) in state.iter_mut().zip([a, b, c, d, e, f, g, h]) {
        *word = word.wrapping_add(value);
    }
}

/// Gibt das Wort W(i) des Nachrichtenplans zurück; ab Runde 16 vor Ort berechnet.
#[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]
}
```

Jedes `eight_rounds!` ist eine volle Umdrehung der Variablennamen: Das Makro übergibt sie pro Runde um eine Position rotiert, sodass eine Runde nur zwei von ihnen schreibt und die anderen sechs nie bewegt. Mit einem literalen Rundenindex ist jede Position im Plan eine Konstante, und der ganze Block wird zu geradlinigem Code, dessen Werte in lokalen WebAssembly-Variablen bleiben. Dieses Layout läuft in V8 etwa 20 % schneller als ein Plan mit 64 Wörtern und einer nicht ausgerollten Schleife, und nativ etwa 14 % schneller (gemessen an der Rust-Implementierung, Oktober 2026).

Das Worker-Protokoll

> **Lernziele** Das Nachrichtenprotokoll lesen, die Fallback-Hierarchie verstehen und den Sentinel-Fehler sehen, den die Grenze vorzeichenbehafteter Ganzzahlen zwischen WebAssembly und JavaScript verursacht hat.

Die Seite sendet eine Nachricht an `web/worker.js` und empfängt Fortschritts-, Fallback- und Ergebnisnachrichten:

```
// Seite -> Worker
{ challenge: spec.id, algorithm: spec.algorithm, difficulty: spec.difficulty,
    challenges: spec.challenges }

// Worker -> Seite
{ 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: '...' }
```

Der Worker versucht zuerst WebAssembly und weicht auf reines JavaScript aus, wenn `WebAssembly` fehlt, durch eine Richtlinie blockiert ist oder sich nicht instanziieren lässt. Die JavaScript-Beweiser bilden den Rust-Beweiser von `sibuna-pow` Label für Label nach und erzeugen *byteidentische* Ausgabe. Die Tests legen feste Nonces und Nachweise für beide Stufen fest, und SID 0010 verlangt, dass der Löser in einem Headless-Browser sowohl mit WebAssembly als auch mit dem JavaScript-Fallback besteht. Die JavaScript-Beweiser sind viel langsamer, garantieren aber, dass jeder Browser bestehen kann.

> **Der vorzeichenbehaftete Sentinel** WebAssembly gibt eine 64-Bit-Ganzzahl als *vorzeichenbehafteten* `BigInt` an JavaScript zurück. Der Hashcash-Export signalisiert „Batch erschöpft“ mit lauter Einsen (`EXHAUSTED`, `u64::MAX`), was JavaScript als `-1n` sieht, nicht als `18446744073709551615n`. Der ursprüngliche Worker verglich mit dem vorzeichenlosen Wert, sodass die Schleife nach dem ersten Batch mit einer falschen Nonce endete. Die Korrektur ist ein einziger Aufruf, `BigInt.asUintN(64, result)`, und der Worker behält ihn:
>
> ```
> // WebAssembly liefert i64-Ergebnisse als vorzeichenbehaftete BigInts, daher
> // kommt der u64-Sentinel "exhausted" (alle Bits gesetzt) als -1n zurück.
> ```

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

Die Lektion lässt sich verallgemeinern: Jeder u64, der die Grenze überquert, muss normalisiert werden. Rust kann dies auf der JavaScript-Seite nicht erzwingen; der Typ des Exports sagt `u64`, und die Umwandlung geschieht in der Engine.

Kalibrierung

Lernziele Schwierigkeitseinstellungen zur Echtzeit in einer Browser-Engine und auf nativer Hardware in Beziehung setzen.

Keine Schwierigkeit aus der durchschnittlichen Lösungszeit eines einzigen Laptops wählen. Hashcash hat ein geometrisches Verteilungsende; Browser-Uhren schließen Scheduling, thermischen Zustand, Start und Kompilierung ein. Eine Kalibrierung für einen Einsatz sollte Browserversion, Gerät, Algorithmus, Parameter und eine Verteilung wiederholter Lösungen festhalten. Abruf und Kompilierung des Moduls getrennt vom Hashen im eingeschwungenen Zustand halten.

Durchgerechneter Kalibrierungsplan b festhalten, das Modul aufwärmen und auf jeder Zielgeräteklasse mindestens 100 unabhängig ausgestellte Rätsel lösen. Median- und hohe Perzentildauern, Fehlschläge und Nachweisgrößen festhalten. Bei $b + 1$ wiederholen. Die erwartete Versuchszahl verdoppelt sich; eine einzelne Stichprobe muss das nicht. Eine Einstellung anhand des Budgets für langsame Geräte wählen und dann den vollständigen Ablauf aus Abruf, Lösung und Verifikation testen.

Die nativen Primitive-Messungen des Repositorys sind kein Beleg für eine versprochene Lösungszeit auf einem Telefon oder in einem Browser. Ein WASM-Modul kann dieselbe Arithmetik ausführen und dabei andere Start-, Ausführungs- und Speicherkosten haben: Nativ hasht der Server mit SHA-NI, während der Browser die obige portable Kompressionsfunktion ausführt. Die Nachrichten des Workers bilden die Grenze, an der sich diese Kosten messen lassen, ohne dem Anfrage-Handler des Servers einen Timer hinzuzufügen.

Die Zwischenseite

Die Seite (`web/challenge.html`, im Daemon eingebettet) ist eine einzelne HTML-Datei mit einer Karte, einem unbestimmten Fortschrittsbalken und einer Statuszeile. Sie fordert die Challenge für ihre eigene Adresse an (`?path=`), startet den Worker, sendet die Lösung mit `fetch` und lädt bei Erfolg neu; `fetch`-Aufrufe werden mit exponentiellem Backoff mit Jitter wiederholt, und eine Verifikationsantwort `4xx` wird mit ihrem Diagnosetitel angezeigt. Die

`Content-Security-Policy` der Seite lässt genau ein Skript zu, den Löserblock, über seinen SHA-256-Hash, den der Daemon einmal aus der eingebetteten Seite berechnet. Ein Honeypot-Link außerhalb des sichtbaren Bereichs (`/__sibuna/honeypot`) ist für Menschen und für Browser, die Skripte ausführen, unsichtbar; blinde Crawler, die jedem `href` folgen, sperren sich selbst.

Übung 7.1. Angenommen, der WebAssembly-Beweiser löst eine Challenge bei der Standardschwierigkeit auf einem Laptop in 20 ms, und der sequenzielle JavaScript-Beweiser ist 50-mal langsamer. Die Lösungszeit mit Fallback auf diesem Laptop und auf einem viermal langsameren Telefon abschätzen und eine Richtlinie für Clients vorschlagen, die den Fallback melden (Hinweis: Der Worker sendet eine `fallback`-Nachricht, die die Seite weiterleiten kann).

Die Invariante erklären. Erklären, warum die Kompilierung des Browsermoduls aus den Verifizierer-Quellen des Servers eine stärkere Garantie ist, als zwei Implementierungen gegeneinander zu testen.

TEIL VIII

Empirische Bewertung

Wir stellen Sibunas Messungen vor, aufgezeichnet von eingecheckten Messprogrammen: Latenzen der Primitiven, Gesamtprodukte unter einem externen Lastgenerator mit Erfassung von CPU und Speicher, Latenz bei gleicher angebotener Last aus einem Generator mit offener Schleife (open-loop) und ein Vergleich mit Anubis, einem weit verbreiteten Proof-of-Work-Gate. Ein Konsolenlauf mit acht Live-Dashboards vervollständigt die Aufzeichnung. Jede Zahl in diesem Teil stammt aus einer eingecheckten Ergebnisdatei, die jede Tabelle mit ihrer Erzeugungszeit nennt, oder aus einem Entwurfsdokument. Ein Funktionsdurchlauf und eine Leistungsmessung beantworten verschiedene Fragen, und eine Messung auf einem Host ist keine Rangfolge auf einem anderen.

Methodik

Lernziele Am Ende dieses Kapitels sollten wir jede Zahl in diesem Teil reproduzieren, angeben, was jedes Messprogramm einschließt und ausschließt, erklären, was die Streuungsspalten bedeuten, und einen gemessenen Wert von einem aufgezeichneten oder veröffentlichten unterscheiden können.

Die Messprogramme sind Befehle eines Rust-Werkzeugs, `tools/bench` (die Binärdatei `sibuna-bench`), ausgeführt über `cargo xtask bench`. Ein weiteres Werkzeug zeichnet den Konsolenlauf auf.

Tabelle 11 Messprogramme und ihre Ergebnisdateien, alle unter `benchmarks/results/`

Messprogramm	Was es misst	Ergebnisdatei
`cargo xtask bench primitives`	Sechzehn Arbeitslasten von Primitiven plus zwei Zeilen für den gehashten Pfad des Ratenbegrenzers, auf einer CPU: sieben Batches, Median pro Operation mit der Min-Max-Streuung	`primitives-latest.json`, `primitives-latest.md`
`cargo xtask bench product`	Vollständige Daemons unter `wrk`: Durchsatz, Latenzperzentile, CPU-Zeit pro Anfrage und maximaler residenter Speicher für Gate und Shield, im Forward-Auth- und im Reverse-Proxy-Modus	`product-latest.json`, `product-latest.md`
`cargo xtask bench latency`	Latenzperzentile zugelassener Anfragen bei festen angebotenen Raten, aus einem Generator mit offener Schleife	`latency-latest.json`, `latency-latest.md`

Messprogramm	Was es misst	Ergebnisdatei
`cargo xtask bench product --anubis`	Die Arbeitslasten des Gesamtprodukts mit Anubis als weiterem Produkt, hinter `caddy respond`	`comparison-latest.json`, `comparison-latest.md`
`cargo xtask bench admission --anubis`	Zulassungsoperationen pro Sekunde von Sibuna und Anubis im Forward-Auth-Modus, von zwei Keep-alive-Clients	`admission-latest.json`, `admission-latest.md`
`sibuna-browser dashboards` (`tools/browser`)	Ein Funktionslauf der Konsole: acht Live-Dashboards in Headless Chromium, während ein Generator Datenverkehr sendet	`console-8-dashboards-2026-10-15.md`

Die `.md`-Dateien sind Zusammenfassungen. Die `.json`-Dateien enthalten jede Wiederholung, die Maschine, die Revision sowie Pfad, Größe und SHA-256 jeder gemessenen Binärdatei. Git bewahrt die Historie, daher enthält das Verzeichnis nur die neuesten Dateien. Einige Dateien wurden von einer früheren Version des Messprogramms aufgezeichnet und enthalten auch Zeilen eines zweiten Daemon-Builds, der im selben Lauf gemessen wurde; dieser Teil berichtet Sibunas Zeilen.

Primitive

Jede Arbeitslast läuft in sieben zeitgemessenen Batches nach einem nicht gemessenen Aufwärmen von höchstens 1.000 Operationen. Veränderlicher Zustand (die Verbrauchsmenge, der Ratenbegrenzer) wird vor jedem Batch außerhalb der Zeitmessung neu aufgebaut. Die Zeitmessung umschließt ganze Batches, nie einzelne Operationen: Eine Uhrabfrage pro Operation würde so viel kosten wie die Arbeit unter 100 ns, die sie misst. Perzentile pro Operation werden daher nicht berichtet, und die Streuung bezieht sich auf die Batches. Wir berichten den Median-Batch pro Operation, mit dem schnellsten und dem langsamsten Batch daneben. Die Suite läuft in einem Kindprozess (`sibuna-bench primitives-run`), der an eine CPU gebunden ist. Auf einem gemeinsam genutzten Host verlangsamt ein ausgelasteter SMT-Geschwisterkern oder ein niedrigerer Takt einen ganzen Suite-Lauf. Daher führt das Werkzeug die Suite dreimal aus (`--rounds`) und behält pro Zeile die Runde mit dem niedrigsten Median. Innerhalb einer Runde gilt die Statistik wie oben.

Das Gesamtprodukt

Für jeden Modus (Forward-Auth, Reverse Proxy) und jede Betriebsform (Gate, Shield) startet das Messprogramm den Daemon im Speicher mit `--algorithm hashcash --difficulty 8 --rate-limit 100000000 --idle-timeout 5` und vier Workern, einem pro CPU seiner Menge. Es erhält eine Sitzung, indem es die Hashcash-Challenge des Daemons mit `sibuna-pow` löst, wie ein Browser es täte. Dann treibt es vier Arbeitslasten mit `wrk` über Keep-alive-Verbindungen: 2 Threads, 64 Verbindungen, 5 Sekunden, 3 Wiederholungen.

- `admitted`: `GET /private` mit einem gültigen Sitzungscookie. Im Reverse-Proxy-Modus erreicht die Anfrage den Ursprungsserver und dessen Antwort wird weitergeleitet; im Forward-Auth-Modus antwortet der Daemon selbst mit `200`.

- `challenged`: dieselbe Anfrage ohne Cookie, mit einem Browser-User-Agent. Der Daemon liefert seine Challenge aus: `200` mit HTML im Reverse-Proxy-Modus, `401` im Forward-Auth-Modus.
- `allowed_static`: `GET /robots.txt` von `curl/8.0`, das beide Betriebsformen ohne Challenge zulassen.
- `attack`: eine gültige Sitzung plus `q=' OR 1=1--`. Nur Shield weist sie ab (`403`). Gate lässt sie zu und leitet sie im Reverse-Proxy-Modus an den Ursprungsserver weiter.

Der Ursprungsserver ist ein Stub (`sibuna-bench origin`): nur Standardbibliothek, ein Thread pro Verbindung, Keep-alive und eine 41-Byte-Seite. Die CPU-Zeit pro Anfrage ist die Benutzer- plus Systemzeit des Daemons aus `/proc/<pid>/stat` über den Lauf (Ticks von 10 ms), geteilt durch die Anfragen, die `wrk` gezählt hat; die CPU des Ursprungsservers ist nicht enthalten. Der Spitzenspeicher ist `VmHWM` aus `/proc/<pid>/status`, vor jedem Lauf über `/proc/<pid>/clear_refs` zurückgesetzt, sodass jeder Lauf seine eigene Spitze meldet. Jeder Wert ist der Median der Wiederholungen; Minimum und Maximum stehen daneben in der JSON-Datei.
`wrk` ist ein Generator mit geschlossener Schleife: Er hält 64 Anfragen gleichzeitig in Bearbeitung. Ein schnellerer Daemon erhält daher mehr Anfragen, und die Perzentile von `wrk` lassen die Anfragen aus, die er nicht gesendet hat, während er auf eine langsame Antwort wartete (Coordinated Omission). Sein p99 zeigt, wie sich der Daemon unter seiner eigenen Sättigung verhält, nicht seine Latenz bei einer gegebenen Last. Diese Frage beantwortet das Latenz-Messprogramm.

Latenz bei gleicher Last

Der Generator mit offener Schleife (`sibuna-bench generate`) betreibt einen blockierenden Thread pro Verbindung. Jede von 64 Verbindungen sendet `GET /private` mit einer Sitzung nach einem festen Zeitplan: Anfrage k einer Verbindung ist zum Zeitpunkt $t_0 + o + kT$ fällig, wobei o der Versatz der Verbindung und T ihr Intervall ist, unabhängig von früheren Antworten. Kommt eine Antwort an, nachdem die nächste Anfrage fällig war, geht diese Anfrage sofort hinaus, und ihre Latenz zählt ab dem Fälligkeitszeitpunkt, wie bei `wrk2`. War die Verbindung zum Fälligkeitszeitpunkt im Leerlauf, zählt die Latenz ab dem tatsächlichen Senden, sodass das Überschießen des Schlafs im Generator selbst nicht dem Daemon angelastet wird.

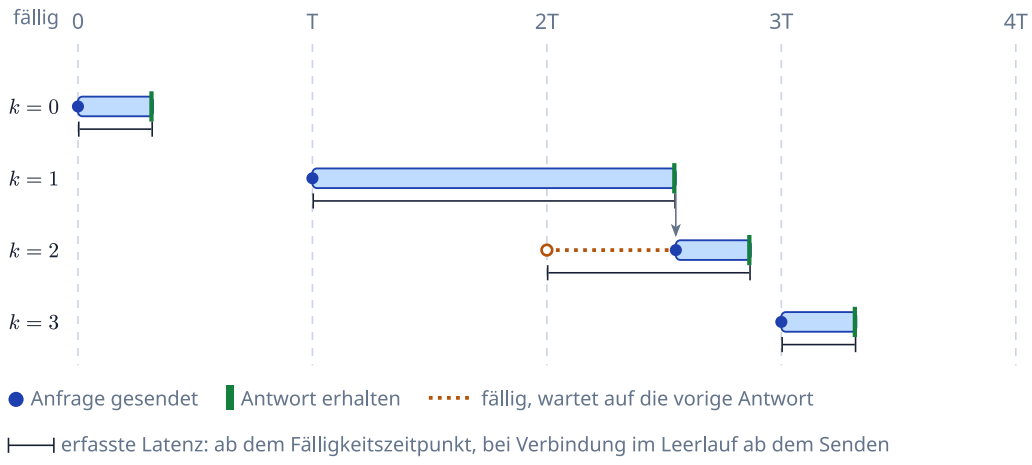


Abb. 10 Eine Verbindung des Generators mit offener Schleife. Anfrage 1 wird beantwortet, nachdem Anfrage 2 fällig war, also wird Anfrage 2 sofort gesendet, und ihre Latenz zählt ab 2T; Anfrage 3 findet die Verbindung im Leerlauf vor und zählt ab ihrem Senden.

Das Messprogramm betreibt die Betriebsform Shield und die Arbeitslast mit zugelassenen Anfragen bei fünf angebotenen Raten: 10.000, 25.000, 50.000, 100.000 und 150.000 Anfragen pro Sekunde, jeweils mit 2 Sekunden Aufwärmen, 10 gemessenen Sekunden und 3 Läufen. Das Messprogramm kann zwei Daemon-Builds in einem Lauf messen: Es startet dann beide auf derselben CPU-Menge und wechselt bei jeder Rate und Wiederholung zwischen ihnen, wobei der ruhende keine CPU nutzt, sodass eine langsame Drift der Maschine beide gleich trifft. Der aufgezeichnete Lauf tat dies. Perzentile sind Nearest-Rank-Perzentile über jede gemessene Anfrage eines Laufs; jeder Wert ist der Median der drei Läufe, mit dem Min-Max von p99 daneben.

CPU-Mengen

Die CPU-Mengen sind wichtiger als jede andere Einstellung. Die Daemons laufen auf den CPUs 0 bis 3 (`--cpus`; `--workers` ist die Größe dieser Menge). `wrk`, der Latenzgenerator und der Ursprungsserver laufen auf den CPUs 8 bis 15 (`--load-cpus`). Die Primitive-Suite läuft auf CPU 4 (`--cpu`). Jeder Prozess wird vor seinem Start mit `taskset -c` gebunden, sodass jeder Thread, den er erzeugt, die Menge erbt. Das Zulassungs-Messprogramm nutzt zwei CPUs, 0 und 1, für das Produkt und die CPUs 8 bis 15 für seine Clients.

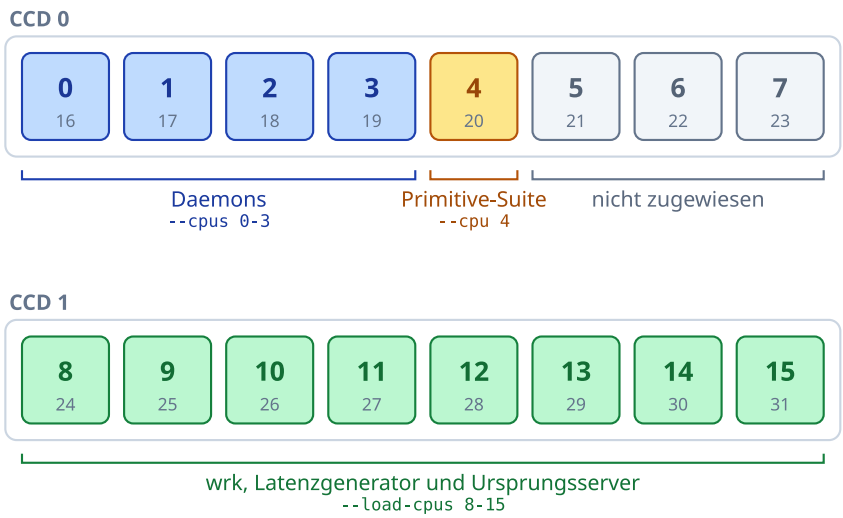


Abb. 11 CPU-Mengen auf dem AMD Ryzen 9 5950X. Die CPUs 0 bis 7 sind die acht Kerne des ersten CCD, 8 bis 15 die des zweiten, und 16 bis 31 ihre SMT-Geschwister. Daemon und Lastgenerator teilen nie einen Kern.

Auf anderen Maschinen wählen wir Mengen, die keine Kerne teilen (`lscpu -e` zeigt die Topologie).

Maschine und Bedingungen

Die Dateien wurden auf `paxos-zig` aufgezeichnet: einem AMD Ryzen 9 5950X 16-Core Processor (32 logische CPUs), Linux 7.0.0-28-generic x86_64, Governor `powersave` mit aktiviertem Boost, rustc 1.99.0. Die Dateien zu Primitiven, Produkten und Latenz wurden am 15. Oktober 2026 aufgezeichnet, der Vergleich mit Anubis am 16. Oktober 2026. Der Daemon ist `cargo build --release -p sibuna`, der Standard-Build mit SQLite-Speicher und einkompilierter Konsole (SID 0010); die Produkt- und Latenzdateien nennen eine Binärdatei von 49.113.856 Bytes. Jede Datei verzeichnet ihre Revision (`efce14d` für die Primitiven, `249d913` für die Produkt- und Latenzläufe) mit `dirty: true`; der Arbeitsbaum der letzteren enthielt bereits die wiederverwendeten Verbindungsthreads, die SID 0002 festhält, und SID 0002 zitiert `latency-latest.md` für ihre Wirkung. Die Vergleichs- und Zulassungsdateien nennen die Revision `54e4ec5` mit `dirty: true`, weil der Arbeitsbaum die Ergänzungen des Messprogramms in `tools/bench` enthielt; ihr Daemon ist ein Release-Build dieser Revision (37.450.232 Bytes). Ihre Lastmittel über eine Minute betrugen 3,53 vor dem Vergleichslauf und 0,56 am Ende des Zulassungslaufs.
Die Dateien verzeichnen Lastmittel über eine Minute von 2,46 beim Primitive-Lauf, 1,30 vor und 52,31 nach dem Latenzlauf sowie 52,31 vor und 33,39 nach dem Produktlauf. Die hohen Werte stammen von den 64 Threads des Latenzgenerators, auf die der Produktlauf direkt folgte.

Messumfang Es werden nur lokale Messungen ausgegeben. Jeder Lauf nutzt einen Host über Loopback: kein TLS, kein WAN, keine echten Browser. Die Daemons laufen im Speicher, ohne Speicherung. Die Zeitmessung umschließt Batches; Zurücksetzen des Zustands und Aufwärmen werden nicht gemessen. Das Messprogramm zählt keine Allokationen (die Dateien sagen „not instrumented; API/source audit only“); der allokationsfreie Anfragepfad wird stattdessen durch die Tests für den eingeschwungenen Zustand in Teil V geprüft. Kein Messprogramm wird in den Daemon gelinkt oder fügt dem Anfragepfad einen Hook hinzu: Die Produkt- und Latenz-Messprogramme führen die unveränderte Release-Binärdatei aus, identifiziert durch ihren SHA-256. Eine Rangfolge auf diesem Host ist keine Rangfolge auf einem anderen.

Reproduktion

Wir brauchen Linux, `taskset` (util-linux) und `wrk` im `PATH`.

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

`cargo xtask bench` baut zuerst den Daemon mit `cargo build --release -p sibuna` und führt dann `sibuna-bench` im Release-Modus aus. Das Werkzeug findet den Daemon neben seiner eigenen Binärdatei; `--rust <path>` überschreibt das. `cargo xtask bench help` listet die Optionen auf, darunter `--cpus`, `--load-cpus`, `--cpu`, `--rounds`, `--repetitions`, `--seconds`, `--rates` und `--warmup`. Der Abschnitt über Anubis weiter unten nennt die Befehle, mit denen dieser Vergleich aufgezeichnet wurde.

Latenzen der Primitiven

Die Tabelle enthält jede Zeile des Primitive-Laufs. Die JSON-Datei benennt jede Zeile nach Subsystem und Arbeitslast (zum Beispiel `pow_verify` und `hashcash_16_bits`).

Tabelle 12 Zeitmessungen der Primitiven in Nanosekunden pro Operation auf einem Kern des Ryzen 9 5950X (`benchmarks/results/primitives-latest.md`, erzeugt 2026-10-15T22:34:59Z)

Arbeitslast	ns/op	min-max
Hashcash-Verifikation (16 Bits)	70,2	69,9-71,0
PoSW-Verifikation (Tiefe 13, t = 16)	22272,4	22257,0-22316,4
Bot-Signaturen (40, Aho-Corasick)	55,7	55,3-57,2
Bot-Signaturen (40, sequenzielle Suche)	763,0	762,2-763,5
IPv4-CIDR-Suche	32,0	31,8-32,2
IPv6-CIDR-Suche	79,3	78,5-79,6
Sitzungstoken (BLAKE3 mit Schlüssel)	155,8	155,3-156,4
Sitzungstoken (Ed25519)	40587,5	40544,6-40631,1
Verbrauchsmenge (Robin Hood), verbrauchen und nachschlagen	21,0	20,8-21,3
Ratenbegrenzer (GCRA), Client-Bytes	12,8	12,7-12,8
Ratenbegrenzer, vier Regelbereiche, Client-Bytes	13,1	13,0-13,2
Ratenbegrenzer (GCRA), gehashter Client	6,2	6,1-6,3
Ratenbegrenzer, vier Regelbereiche, gehashter Client	6,3	6,2-6,4
HTTP-Parsen und Cookie	132,5	132,3-133,2
Klassifikation, Shield-Profil	790,8	787,2-793,1
Klassifikation, Gate-Profil	75,9	75,6-76,1
WAF-Body-Scan (8 KiB)	12784,3	12771,5-12809,2
WAF-Body-Scan (8 KiB), SQL-Injection im Audit-Modus	15485,4	15460,9-15530,1

Die Streuungen sind gering: Kein Batch einer Zeile liegt mehr als 2,8 % vom Median seiner Zeile entfernt. Die Abbildung zeigt dieselben Zeilen auf einer logarithmischen Skala, auf der Zeilen von 6 ns und von 40 µs nebeneinander lesbar bleiben.

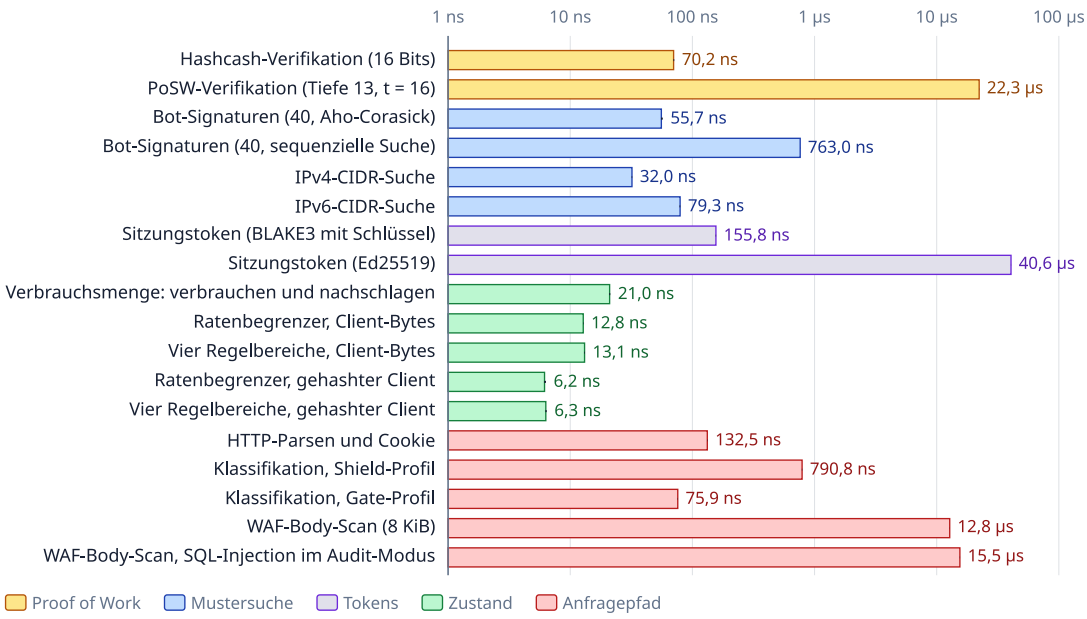


Abb. 12 Zeitmessungen der Primitiven auf einer logarithmischen Skala, gezeichnet aus `benchmarks/results/primitives-latest.json`.

Proof-of-Work-Verifikation

Die Hashcash-Verifikation hasht eine Challenge, ein Trennzeichen und eine dezimale Nonce: 70,2 ns. Der Verifizierer für sequenzielle Arbeit prüft Graphöffnungen: 22.272,4 ns. Seine Hash-Aufrufe und die Zahl der SHA-256-Kompressionen unterscheiden sich von denen bei Hashcash, weil die Eingaben unterschiedlich lang sind. Beide Zeilen sind durch SHA-256 begrenzt, und `sibuna-crypto` nutzt die SHA-NI-Instruktionen, wenn die CPU sie hat, zur Laufzeit erkannt (SID 0010).

Mustersuche

Der dichte Aho-Corasick-Automat und die sequenzielle Teilzeichenkettensuche nutzen identische Muster und dieselbe Semantik „irgendein Treffer“. Der Automat ist auf denselben 40 Signaturen 13,7-mal schneller als die sequenzielle Suche (55,7 gegenüber 763,0 ns). Seine Tabellen nutzen Byteklassen statt 256 Spalten pro Zustand (SID 0006, Teil F), was die Tabelle der Bot-Signaturen bei 42.244 Bytes hält (siehe die Tabelle zum Speicherbedarf unten). Die CIDR-Klassifikatoren brauchen 32,0 ns (IPv4) und 79,3 ns (IPv6).

Tokens

Das BLAKE3-MAC-Token wird in 155,8 ns verifiziert. `sibuna-crypto` implementiert BLAKE3 portabel, ohne SIMD-Pfade; diese gewinnen bei großen Eingaben, aber Sibuna hasht mit BLAKE3 Eingaben unter 100 Bytes, bei denen der portable Code schnell ist. Kompakte

Ed25519-Tokens werden in 40.587,5 ns verifiziert: Der Verifizierer nutzt NAF-Ziffern der Breiten 5 und 8 und behält den dekodierten öffentlichen Schlüssel. Eine Ed25519-Verifikation kostet etwa so viel wie 260 MAC-Verifikationen; der MAC ist Sibunas Standard-Tokenverfahren.

Zustand

Die Zeile der Verbrauchsmenge fügt ein frisches Tag ein und schlägt es erneut nach: 21,0 ns. Die Tabelle in dieser Zeile hat 16 Shards zu 4.096 Slots (65.536 Tags; der Standardwert des Daemons ist 1.048.576), und sie wird vor jedem Batch an Ort und Stelle geleert, außerhalb der Zeitmessung. Jeder Batch greift daher auf residenten Speicher zu. Eine frühere Version der Suite baute für jeden Batch eine frische, genullte Tabelle in der Standardgröße des Daemons; sie maß, wie der Kernel die Seiten der Tabelle beim ersten Schreiben einblendet, Kosten, die ein laufender Daemon einmal bezahlt, und ist nicht die Zeile, die wir berichten.
Die Zeilen des Ratenbegrenzers erfordern Sorgfalt. Die beiden Zeilen für Client-Bytes (`gcra_check` und `gcra_four_rule_scopes`) rufen die Byte-API auf: Jede Prüfung nimmt die Adresse des Clients als Bytes und berechnet zuerst ein SipHash-1-3 mit Schlüssel darüber, unter einem Geheimnis aus der Zufallsquelle des Betriebssystems. Der geheime Schlüssel hindert jeden daran, Adressen zu berechnen, die das Sondierungsfenster eines Opfers teilen (SID 0010 und die Dokumentation von `sibuna-store`). Diese Zeilen brauchen 12,8 und 13,1 ns.
Der Daemon nutzt die Byte-API nicht auf dem Anfragepfad. Er hasht den Client einmal pro Anfrage (`ClientHasher`) und übergibt den Hash an jede Tabelle, die er befragt. Die beiden Zeilen für einen gehashten Client (`gcra_check_hashed` und `gcra_four_rule_scopes_hashed`) messen diesen Pfad, wobei die Hashes vor der Schleife berechnet werden: 6,2 und 6,3 ns pro Prüfung. Der Abstand zwischen den Medianen der Byte- und der gehashten Zeilen, 6,6 ns, entspricht etwa dem Preis des Hashs mit Schlüssel für diese Eingaben, den der Daemon einmal pro Anfrage statt einmal pro Tabelle bezahlt.

HTTP-Parser

Die Parser-Zeile parst eine 400-Byte-Browseranfrage und schlägt ein Cookie nach: 132,5 ns. Der eigene Zeitmessungstest des Crates stimmt damit überein (`crates/http`: etwa 135 bis 180 ns auf derselben Maschine). Der Parser arbeitet ohne Kopien; jedes Feld ist ein Slice des Verbindungspuffers.

Richtlinien-Engine

Eine vollständige Klassifikation einer Browseranfrage dauert 790,8 ns, das Gate-Profil 75,9 ns. Die Zeilen lassen die geparste Clientadresse der Anfrage leer, sodass die Engine den Adresstext des Clients während der Auswertung parst. Der Daemon parst die Adresse einmal pro Anfrage vor der Auswertung, daher enthalten diese Zeilen kleine Kosten, die der Daemon außerhalb der Engine bezahlt. Die Heap-Tabellen der Engine umfassen insgesamt 209.052 Bytes (SID 0002).

WAF-Prüfung

Der semantische Scan eines 8-KiB-Bodys dauert 12.784,3 ns, also $12784.3/8192 \approx 1.56$ ns pro Byte. Die zweite Zeile scannt denselben Body, wobei die Kategorie SQL-Injection im Audit-Modus und die anderen im Durchsetzungsmodus laufen: 15.485,4 ns. Wenn eine Kategorie nur

auditiert statt durchgesetzt wird, wird jedes Feld pro Kategorie geprüft (SID 0004, „inspection modes“). Die Engine scannt das Feld einmal mit dem Signaturautomaten, um die Menge der Kategorien zu bestimmen, deren Signaturen vorkommen, und sucht dann nur nach den gewünschten Kategorien darunter. Eine Kategorie zu auditieren kostet daher etwa ein Fünftel mehr, als alle durchzusetzen, nicht einen Scan pro Kategorie.

Vergleich der Gesamtprodukte

Lernziele Durchsatz, Latenz, CPU-Zeit und Speicher des Daemons als vollständiger Prozess lesen, seine Betriebsformen und Modi vergleichen und angeben, in welchen Zeilen der Daemon und in welchen der Lastgenerator die Grenze setzt.

Die Tabellen zeigen die Mediane von drei fünfsekündigen `wrk`-Läufen für jede Betriebsform und Arbeitslast. Der Status neben jeder Arbeitslast ist die Antwort, die das Messprogramm erwartet und erhalten hat.

Forward-Auth-Modus

Der Eingangsproxy fragt den Daemon, ob eine Anfrage zugelassen werden soll; es gibt keinen Ursprungsserver und keine Body-Weiterleitung. Dies ist daher die reinste Messung der Zulassungskosten.

Tabelle 13 Forward-Auth-Modus, Gate und Shield; Latenz in Mikrosekunden (`benchmarks/results/product-latest.md`, erzeugt 2026-10-15T23:15:24Z)

Betriebsform	Arbeitslast (Status)	Anfr./s	p50	p90	p99	CPU µs/Anfr.	Spitzen-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

Im Forward-Auth-Modus ist der Daemon nicht der Engpass. Seine belegten Kerne (CPU-Zeit über Echtzeit, `cores_busy_median` in der JSON-Datei) bleiben zwischen 2,47 und 2,72 der vier Daemon-CPUs, und jede Zeile liegt zwischen 197.532 und 211.636 Anfragen pro Sekunde. Das deutet auf den Lastgenerator als Obergrenze hin. Nahe dieser Obergrenze schwankt ein

Unterschied von wenigen Prozent zwischen den Zeilen von Lauf zu Lauf: Allein die Gate-Zeile `admitted` reicht über ihre drei Läufe von 205.151 bis 211.513 Anfragen pro Sekunde. Der nützlichere Wert in diesem Modus ist die CPU-Zeit pro Anfrage: 11,73 bis 13,77 µs, etwas mehr für Shield als für Gate und am meisten für die `challenged`-Zeilen, die die Challenge-Antwort rendern.

Reverse-Proxy-Modus

Der Daemon sitzt vor dem Stub-Ursprungsserver und leitet zugelassene Anfragen und Antworten weiter. Weitergeleitete Zeilen enthalten daher die eigenen Kosten des Ursprungsservers und einen zusätzlichen Loopback-Sprung.

Tabelle 14 Reverse-Proxy-Modus, Gate und Shield; Latenz in Mikrosekunden (`benchmarks/results/product-latest.md`, erzeugt 2026-10-15T23:15:24Z)

Betriebsform	Arbeitslast (Status)	Anfr./s	p50	p90	p99	CPU µs/Anfr.	Spitzen-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

Hier sind die weitergeleiteten Zeilen (`admitted`, `allowed_static` und Gates `attack`, die Gate weiterleitet) CPU-gebunden: Die belegten Kerne liegen bei 3,96 bis 3,98 von vier. Sie bedienen 138.965 bis 147.303 Anfragen pro Sekunde bei 26,88 bis 28,57 µs CPU pro Anfrage, etwa doppelt so viel wie eine Antwort, die der Daemon selbst gibt. Die `challenged`-Zeilen und Shields `attack`-Zeile beantwortet der Daemon selbst, und sie sehen aus wie Forward-Auth-Zeilen. Unter Sättigung ist das p99 von `wrk` das Verteilungsende des Schedulers bei der eigenen Rate des Daemons; daher lesen wir stattdessen die Verteilungsenden im nächsten Kapitel.

In drei Reverse-Proxy-Zeilen zählte `wrk` Antworten außerhalb der Klassen 2xx und 3xx, wo keine erwartet wurden: 3 von 2.162.579 Anfragen (Gate `attack`), 1 von 2.082.640 (Shield `admitted`) und 1 von 2.204.358 (Shield `allowed_static`), summiert über die drei Läufe. Die Dateien verzeichnen nicht, welcher Status das war. Eine Wiederholung des Laufs mit den Metriken des Daemons zeigte `503`-Antworten, gezählt in `sibuna_overloaded_total`: `wrk` öffnet seine 64 Verbindungen von einer Adresse, genau dem Standardwert von `--max-connections-per-ip`, und wenn eine Verbindung nach ihrer 4.096. Anfrage geschlossen wurde, verband sich `wrk` neu, bevor die alte Verbindung ihren Platz zurückgegeben hatte. Der Daemon gibt den

Platz jetzt zurück, bevor er die abschließende Antwort schreibt (SID 0002, „bug fix“); die Tabellen sind älter als diese Korrektur.

Speicherbedarf

Tabelle 15 Größen und residenter Speicher des Daemons (`benchmarks/results/product-latest.json`, erzeugt 2026-10-15T23:15:24Z, und `primitives-latest.json`, erzeugt 2026-10-15T22:34:59Z)

Messgröße	Wert
Binärdatei des Daemons	49.113.856 Bytes
Residenter Speicher im Leerlauf, Produktläufe	5.592 bis 5.788 KiB
Maximaler residenter Speicher während einer Arbeitslast	15.896 bis 16.348 KiB
Residenter Speicher nach den Arbeitslasten	16.176 bis 16.348 KiB
Residenter Speicher im Leerlauf, Primitive-Lauf	6.052 KiB
Tabellen der Richtlinien-Engine	209.052 Bytes (Heap)
Tabelle der Bot-Signaturen	42.244 Bytes (Heap)
WAF-Tabelle	164.908 Bytes (Heap)
Browser-Löser-Modul (WebAssembly)	14.744 Bytes

Die Bereiche erstrecken sich über die vier Produktläufe (zwei Modi, zwei Betriebsformen), und für die Spitze über deren sechzehn Arbeitslasten. Die Tabellengrößen sind von der Engine gemeldete Heap-Größen. Die Aufzeichnung schlüsselt die Größe der Binärdatei nicht auf.

Latenz bei gleicher angebotener Last

Lernziele Eine Latenzverteilung bei fester angebotener Last lesen, ein Perzentil des Verteilungsendes zusammen mit seiner Streuung lesen und einen Defekt im Verteilungsende von seiner Messung bis zu seiner Korrektur verfolgen.

Beide Tabellen stammen aus einem Lauf von `cargo xtask bench latency`. Die Fehlerspalte zählt die Verbindungs-, E/A- und Rahmungsfehler des Generators.

Tabelle 16 Forward-Auth-Modus, Shield, zugelassene Anfragen bei festen angebotenen Raten; Latenz in Mikrosekunden (`benchmarks/results/latency-latest.md`, erzeugt 2026-10-15T23:07:14Z)

Angebotene Anfr./s	Erreichte Anfr./s	p50	p90	p99 (min-max)	p99.9	max	Fehler	CPU µs/Anfr.
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

Angebotene Anfr./s	Erreichte Anfr./s	p50	p90	p99 (min-max)	p99.9	max	Fehler	CPU µs/Anfr.
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

Tabelle 17 Reverse-Proxy-Modus, Shield, zugelassene Anfragen bei festen angebotenen Raten; Latenz in Mikrosekunden (`benchmarks/results/latency-latest.md`, erzeugt 2026-10-15T23:07:14Z)

Angebotene Anfr./s	Erreichte Anfr./s	p50	p90	p99 (min-max)	p99.9	max	Fehler	CPU µs/Anfr.
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

Im Forward-Auth-Modus bleibt das p50 bei jeder Rate zwischen 24 und 30 µs. Das p99 bleibt bis 150.000 Anfragen pro Sekunde bei höchstens 51 µs und das p99.9 bei höchstens 74 µs. Der Daemon wendet 13,48 bis 16,92 µs CPU pro Anfrage auf.

Im Reverse-Proxy-Modus beträgt das p50 bis 100.000 Anfragen pro Sekunde 47 bis 56 µs, das p99 bei 100.000 liegt bei 100 µs und das p99.9 bei 150 µs. Bei 150.000 Anfragen pro Sekunde ist der Daemon gesättigt: Er schafft 134.344 Anfragen pro Sekunde, und die Latenz wächst auf Sekunden, wie es ein Test mit offener Schleife oberhalb der Kapazität zeigen sollte. Diese Rate liegt nahe am weitergeleiteten Durchsatz mit geschlossener Schleife aus dem vorigen Kapitel (138.965 Anfragen pro Sekunde für Shield `admitted`). Die vier Fehler dieser Zeile traten in der Überlast auf.

Unterhalb der Sättigung sind die Streuungen des p99 gering; die breiteste ist das Reverse-Proxy-p99 bei 50.000 Anfragen pro Sekunde, 58 bis 66 µs. Die Spalte max ist eine einzige Anfrage von bis zu 1,5 Millionen in einem Lauf. Auf einem gemeinsam genutzten Host lesen wir sie als schlechtesten Moment des Hosts, nicht als Eigenschaft des Daemons.

Was die Messung verändert hat

Externe Last hat Sibunas Entwurf verändert. Der Daemon bedient jede Verbindung auf einem eigenen Thread (SID 0002, SID 0010), und seine ersten Latenzläufe fanden ein Verteilungsende, das `wrk` nicht gezeigt hatte. Mit 64 Verbindungen, die der Daemon jeweils nach 4.096 Anfragen schloss, lag das 99,9. Perzentil bei 100.000 zugelassenen Anfragen pro Sekunde bei 0,3 bis 2 ms. Die Diagnose, festgehalten in SID 0002 und in der Dokumentation von `cmd/sibuna/src/tasks.rs`, ist der Wechsel von Threads. Einen Thread zu erzeugen und zu beenden blendet

seinen Signal-Stack ein und aus und gibt seine Stack-Seiten zurück, und diese Änderungen am Adressraum brachten die anderen Threads des Daemons ins Stocken. Ein Experiment brachte Klarheit: Ein Build, der nie Verbindungen schloss, sodass während des Laufs kein Thread erzeugt wurde, ergab bei derselben Rate 54 bis 60 µs. Die Korrektur verwendet Verbindungsthreads wieder: Ein Thread, der seine Verbindung beendet, parkt und übernimmt die nächste, sodass ein stetiger Strom neuer Verbindungen keine Threads erzeugt. Ein Thread, der 10 Sekunden geparkt war, endet und wird vom nächsten Start oder vom nächsten Thread, der eine Aufgabe beendet, eingesammelt (join), sodass ein Burst seine Stacks nicht resident zurücklässt. Ein Thread, dessen Verbindung eine Panic auslöst, endet, statt zu parken. Höchstens `--max-connections` Verbindungen werden gleichzeitig bedient, wie zuvor.
Die Tabellen oben stammen aus dem Lauf nach der Korrektur: ein p99.9 von 62 µs im Forward-Auth-Modus und 150 µs im Reverse-Proxy-Modus bei 100.000 Anfragen pro Sekunde. Die Läufe vor der Korrektur sind nicht als Ergebnisdateien erhalten; deshalb erscheinen ihre Werte nur in SID 0002 und in diesem Abschnitt.

Durchgerechnetes Beispiel: warum sich das Stocken bei p99.9 zeigte Bei 100.000 Anfragen pro Sekunde über 64 Verbindungen trägt jede Verbindung $100000/64 = 1562.5$ Anfragen pro Sekunde. Der Daemon schließt eine Keep-alive-Verbindung nach 4.096 Anfragen (`MAX_REQUESTS_PER_CONNECTION` in `cmd/sibuna/src/connection.rs`), also hält eine Verbindung $4096/1562.5 \approx 2.62$ s, und der Generator öffnet etwa $64/2.62 \approx 24.4$ neue Verbindungen pro Sekunde. Mit einem Thread pro Verbindung sind das etwa 24 Thread-Starts und 24 Thread-Enden pro Sekunde.
Ein 10-Sekunden-Lauf bei dieser Rate misst 1.000.000 Anfragen. Sein p99.9 wird von 1.000 davon überschritten, 100 pro Sekunde; sein p99 von 1.000 pro Sekunde. Verzögert jeder Thread-Wechsel etwa $100/24.4 \approx 4$ laufende Anfragen auf anderen Threads um einen Bruchteil einer Millisekunde, so füllen allein die verzögerten Anfragen die obersten 0,1 %, während das oberste 1 % noch aus gewöhnlichem Warten in Warteschlangen besteht. Ein seltenes, periodisches Stocken verschiebt das äußerste Verteilungsende lange, bevor es das p99 verschiebt. Deshalb braucht ein solcher Defekt p99.9 und eine Zeitmessung an jeder Anfrage, um sichtbar zu werden.

Sibuna und Anubis

Lernziele Sibunas Werte neben denen eines etablierten Produkts unter derselben Last lesen und angeben, welche Unterschiede von den Produkten und welche vom Aufbau herrühren.

Anubis von Techaro ist ein weit verbreitetes Proof-of-Work-Gate, das vor Websites sitzt, ähnlich wie Sibunas Gate. Wir haben die offizielle Release-Binärdatei von Anubis v1.27.0 (`anubis-1.27.0-linux-amd64`, 40.001.698 Bytes) neben Sibuna auf demselben Host, auf denselben CPUs und mit denselben Arbeitslasten gemessen. Die beiden Ergebnisdateien wurden aufgezeichnet mit:

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

Anubis wurde so gestartet, wie die früheren Python-Treiber dieses Projekts es starteten: `--bind`, `--metrics-bind`, `--target=` (leer im Forward-Auth-Modus, der Ursprungsserver im Reverse-Proxy-Modus), `--difficulty 2`, `--slog-level ERROR`, `--cookie-secure=false` und eine Richtlinie, die jeden `Mozilla`-User-Agent mit einer Challenge belegt, mit deaktiviertem DNSBL und Honeypot. `GOMAXPROCS` entspricht der Worker-Zahl von Sibuna, und jeder Thread beider Produkte ist an dieselbe CPU-Menge gebunden. Die Hashcash-Arbeit ist angeglichen: 8 Nullbits für Sibuna, 2 hexadezimale Nullziffern für Anubis, dieselben erwarteten 256 Hashes. Das Messprogramm erhält die Sitzung jedes Produkts wie ein Browser. Für Anubis ruft es die Challenge-Seite ab, liest die Challenge aus `<script id="anubis_challenge">`, findet die Nonce, deren `SHA-256(randomData || nonce)` mit `00` beginnt, und übergibt sie mit dem Verifikationscookie der Seite an `pass-challenge`. Im Forward-Auth-Modus fragen die Arbeitslasten `admitted` und `challenged` den Prüfendpunkt von Anubis (`/.within.website/x/cmd/anubis/api/check`).

Arbeitslasten des Gesamtprodukts

Die vier Arbeitslasten, die Last und die Statistik sind die des Messprogramms für das Gesamtprodukt, mit einer Änderung: Der Ursprungsserver ist `caddy respond`, gebunden an die Last-CPUs, wie es die früheren Treiber nutzten. Vor jedem Lauf wartet das Messprogramm, bis weniger als 3.000 Sockets in `TIME_WAIT` verbleiben, weil die Reverse-Proxy-Läufe von Anubis Zehntausende zurücklassen.

Tabelle 18 Forward-Auth-Modus: Mediane von drei fünfsekündigen `wrk`-Läufen (`benchmarks/results/comparison-latest.md`, erzeugt 2026-10-16T00:55:19Z).

Arbeitslast	Kennzahl	Sibuna Gate	Sibuna Shield	Anubis 1.27.0
`admitted`	Anfr./s (Status)	210.682 (200)	208.635 (200)	36.509 (200)
	p99 µs	305	308	7.477
	CPU µs/Anfr.	12,09	13,21	106,30
	Spitzen-RSS KiB	16.228	16.316	32.624
`challenged`	Anfr./s (Status)	198.251 (401)	193.087 (401)	113.170 (401)
	p99 µs	326	342	3.781
	CPU µs/Anfr.	13,49	15,02	32,96
	Spitzen-RSS KiB	16.240	16.328	32.696
`allowed_static`	Anfr./s (Status)	212.569 (200)	212.444 (200)	89.904 (200)
	p99 µs	304	304	4.547
	CPU µs/Anfr.	11,73	11,95	41,63

Arbeitslast	Kennzahl	Sibuna Gate	Sibuna Shield	Anubis 1.27.0
	Spitzen-RSS KiB	16.240	16.328	33.396
`attack`	Anfr./s (Status)	212.086 (200)	211.347 (403)	36.853 (200)
	p99 µs	305	306	7.283
	CPU µs/Anfr.	12,07	12,56	104,88
	Spitzen-RSS KiB	16.240	16.328	33.764

Tabelle 19 Reverse-Proxy-Modus, hinter `caddy respond`: Mediane von drei fünfsekündigen `wrk`-Läufen (`benchmarks/results/comparison-latest.md`, erzeugt 2026-10-16T00:55:19Z).

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

Im Forward-Auth-Modus bediente Sibuna 193.087 bis 212.569 Anfragen pro Sekunde bei einem p99 von 304 bis 342 µs, und Anubis 36.509 (`admitted`) bis 113.170 (`challenged`) bei einem p99 von 3.781 bis 7.477 µs. Sibunas belegte Kerne blieben zwischen 2,49 und 2,90 von vier, also setzte der Lastgenerator seine Grenze; bei Anubis waren es 3,73 bis 3,88, also war Anubis CPU-gebunden. Der klarste Wert ist die CPU-Zeit pro Anfrage: Eine zugelassene Anfrage kostete Sibuna Gate 12,09 µs und Anubis 106,30 µs. Ein wahrscheinlicher Teil dieses Unterschieds ist das Sitzungstoken. Anubis signiert Sitzungen standardmäßig mit Ed25519, und Sibuna prüft einen BLAKE3-MAC mit Schlüssel (155,8 ns in der Primitive-Tabelle); die Zulassungsmessungen unten, bei denen HS512-Tokens die Sitzungsprüfungen von Anubis um mehr als die Hälfte steigern, stützen diese Lesart.
Im Reverse-Proxy-Modus bedienten Sibunas weitergeleitete Zeilen 103.298 bis 106.819 Anfragen pro Sekunde bei 2,77 bis 2,98 belegten Kernen. Sie liegen unter den 138.965 bis 147.303 des Stub-Ursprungsservers in `product-latest.json`, und der Daemon ist nicht CPU-gebunden, also liegt die Grenze außerhalb von ihm: Der Ursprungsserver, `caddy respond`, teilt die Last-CPUs mit `wrk`. Sein Verteilungsende zeigt sich im p90 von 2,1 bis 2,4 ms. Anubis bediente 19.398 bis 28.862 Anfragen pro Sekunde bei 3,68 bis 3,97 belegten Kernen und 135 bis 190 µs CPU pro Anfrage. Sein residenter Speicher stieg von 39.980 KiB in der Arbeitslast `admitted` auf 516.872 KiB in `challenged` und 616.548 KiB in `allowed_static`, und nach den Arbeitslasten hielt es 589.276 KiB gegenüber 21.552 KiB im Leerlauf. Der Anstieg begann in `challenged`, wo jede Anfrage eine frische Challenge mit eigener ID erhält; das deutet auf einen Datensatz pro ausgestellter Challenge hin, aber das Messprogramm hat die Ursache nicht bestätigt. Sibunas Spitze blieb in jeder Zeile beider Modi zwischen 16.228 und 16.544 KiB.
Auch mehrere Zeilen sind nicht direkt vergleichbar:

- Im Forward-Auth-Modus beantwortet Anubis `/robots.txt` und den `attack`-Pfad selbst mit `200`, ohne Ursprungsserver; Sibuna beantwortet sie, wie es in diesem Modus jede Anfrage beantwortet.
- Anubis hat keine Anfrageprüfung, daher lässt es die SQL-Injection der Arbeitslast `attack` zu. Shield weist sie nach mehr Arbeit mit `403` ab; Gate lässt sie zu, wie Anubis.
- Die Produkte signieren Sitzungen unterschiedlich (standardmäßig Ed25519 bei Anubis, ein BLAKE3-MAC mit Schlüssel bei Sibuna), und sie rendern unterschiedliche Challenge-Seiten.

Zulassungsoperationen

Das Zulassungs-Messprogramm misst die Zeit der vier Operationen, aus denen die Zulassung besteht, im Forward-Auth-Modus, mit jedem Produkt auf zwei CPUs (zwei Sibuna-Worker, `GOMAXPROCS=2`). Zwei Client-Threads auf den Last-CPUs, jeder mit einer Keep-alive-Verbindung, teilen sich jeden Batch: 200 Operationen oder, für die Verifikation, 64 Nachweise, die vor dem Batch vorbereitet werden. Wir berichten den Median von sieben Batches.

- Sitzungsprüfung: der Prüfendpunkt mit einer gültigen Sitzung (`200`).
- Nicht authentifizierte Prüfung: dasselbe ohne Cookie (`401`).
- Challenge-Start: was ein Browser abruft, um eine Challenge zu beginnen. Für Sibuna sind das zwei Anfragen, die Zwischenseite und der Challenge-Datensatz; für Anubis eine, die Seite mit der eingebetteten Challenge.

- Nachweisprüfung: ein frischer Nachweis pro Operation. Sibunas Challenges werden dafür im Abstand von 25 ms ausgestellt, unterhalb der 50 pro Sekunde, ab denen seine adaptive Schwierigkeit Bits hinzufügt, sodass die angeglichene Arbeit erhalten bleibt. Sibuna läuft mit `--challenge-rate-limit 100000000`, sodass sein Begrenzer geprüft, aber nie erschöpft wird.

Anubis lief zweimal, mit seinen standardmäßigen Ed25519-Tokens und mit `--hs512-secret`.

Tabelle 20 Zulassungsoperationen pro Sekunde, zwei Clients, Produkt auf zwei CPUs (`benchmarks/results/admission-latest.md`, erzeugt 2026-10-16T00:55:44Z)

Produkt (Token)	Sitzungsprüfung	Nicht authentifizierte Prüfung	Challenge-Start	Nachweisprüfung	RSS danach, KiB
Sibuna (BLAKE3 mit Schlüssel)	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

Die Bereiche der sieben Batches stehen in der Datei. Die von Sibuna liegen innerhalb von 11 % um seine Mediane; die von Anubis sind breiter (seine nicht authentifizierten Prüfungen mit Ed25519-Tokens reichen von 25.394 bis 32.741), und ein Batch von 200 Operationen dauert nur 2 bis 25 ms, daher lesen wir diese Werte auf etwa ein Zehntel genau. Die Zeit pro Operation des langsameren Clients ergibt die Umlaufzeit: 22,7 µs für eine Sitzungsprüfung bei Sibuna, 182,5 µs für eine Prüfung bei Anubis mit Ed25519-Tokens und 117,7 µs mit HS512.

In keiner der Zeilen oben bediente Anubis mehr Anfragen oder nutzte weniger CPU oder Speicher als Sibuna. Das ist eine Aussage über diese Arbeitslasten auf diesem Host und nicht mehr.

Wie die Anubis-Zeilen zu lesen sind Anubis ist ein leistungsfähiges, weit verbreitetes Produkt, und diese Zeilen behaupten nicht, dass es langsam sei. Es läuft auf einer Laufzeitumgebung mit Garbage Collection, signiert Sitzungen standardmäßig mit Ed25519, sodass die Verifikation kein gemeinsames Geheimnis braucht, bietet mehrere Challenge-Verfahren und eine reichhaltige Richtliniensprache und hält den Challenge-Zustand in einem Speicher; das sind Entwurfsentscheidungen mit Vorteilen, die dieses Messprogramm nicht misst. Die Zeilen zeigen, was eine Anfrage jedes Produkt auf einem Host unter einer Lastform im Oktober 2026 gekostet hat. Sibuna ist neu, und dies sind seine ersten Messungen neben einem etablierten Produkt.

Overhead der Konsole

SID 0007 legt den Isolationsvertrag der Konsole fest. Die aktivierte Konsole darf höchstens 1 % des Durchsatzes der Datenebene und höchstens 10 % der p99-Latenz kosten, gemessen über eine Matrix von Konfigurationen (nicht einkompiliert, einkompiliert aber deaktiviert, im Leerlauf und acht aktive Dashboards) und Arbeitslasten, mit wiederholten, verschränkten Läufen und angegebener Unsicherheit. Nicht schlüssige oder verrauschte Läufe belegen keine Einhaltung, und Basiswerte der Primitiven sowie Smoke-Läufe können dieses Abnahmekriterium nicht ersetzen.
SID 0010 verzeichnet eine Messung: Mit einkompilierter Konsole, deaktiviert, nur mit Speicherung oder aktiviert und im Leerlauf, bleiben die 16 Produkt-Arbeitslasten innerhalb von etwa 1 % des ohne sie gebauten Daemons, also im Rauschen. Das Dokument nennt dafür keine Ergebnisdatei. Die Konfiguration mit acht aktiven Dashboards unter Last ist nicht gemessen worden, daher bleibt das formale Abnahmekriterium von SID 0007 offen.
Der Lauf mit acht Dashboards prüft die Funktion, nicht den Overhead. `sibuna-browser dashboards` startete einen Release-Daemon auf einem SQLite-Knoten mit seiner Konsole, vier Workern und einer Richtlinie, die `/static/*`, `/about`, `/missing/*` und `/broken/*` erlaubt und alles andere mit einer Challenge belegt; eine Ursprungsserver-Attrappe beantwortet `/missing/*` mit 404 und `/broken/*` mit 500. Ein Generator sendete 30 Sekunden lang 200 Anfragen pro Sekunde und wechselte dabei durch zehn Pfade (einer davon eine SQL-Injection), drei User-Agents und 500 weitergeleitete Adressen. Jedes Dashboard lief in einem eigenen Chromium mit eigener Sitzung: die Dashboards 1 bis 3 als Administratorsitzungen, die Dashboards 4 bis 8 als Wandanzeigen, geöffnet mit Codes, die das erste ausgestellt hatte, weil die Konsole fünf Anmeldungen pro Minute pro Adresse und pro Konto erlaubt.

Tabelle 21 Acht Traffic-Dashboards auf einem Knoten (`benchmarks/results/console-8-dashboards-2026-10-15.md`, aufgezeichnet 2026-10-15, Chromium 155.0.8059.39). Ergebnis: PASS

Messgröße	Wert
Anfragen der Datenebene	6.000 in 30,0 s (200 pro Sekunde, 4 Threads, eine Verbindung pro Anfrage)
Beantwortet	6.000: 200 x 1.200, 401 x 3.000, 403 x 600, 404 x 600, 500 x 600
Fehler (keine Antwort)	0
Latenz p50 / p90 / p99 / max	246 / 311 / 373 / 552 µs (einschließlich Verbindungsaufbau)
VmRSS des Daemons beim Start	36.152 KiB
VmRSS des Daemons mit 8 offenen Dashboards	38.028 KiB
VmRSS des Daemons nach dem Datenverkehr	41.496 KiB (VmHWM 41.496 KiB)

Die Kacheln wurden während des Datenverkehrs alle zwei Sekunden und 3 Sekunden danach gelesen. Jedes Dashboard endete mit denselben Summen (Requests 6.000, Admitted 2.400,

Challenged 3.000, Policy denied 600, Origin 4xx 600, Origin 5xx 600), erhielt 134 bis 139 `stats`-Deltas, hielt seinen WebSocket offen und meldete keine Ausnahme, keinen Konsolenfehler und keinen Verstoß gegen die Content-Security-Policy. Acht Browser-Abonnenten liegen weit unter dem Limit der Konsole von 64. Der Daemon dieses Laufs hat SQLite-Speicher und die Konsole, anders als die Daemons im Speicher beim Produkt-Messprogramm; sein residenter Speicher ist daher nicht mit deren vergleichbar.

Nicht gemessen

Tabelle 22 Was dieser Teil nicht misst und was an die Stelle einer Messung tritt

Gegenstand	Warum es hier nicht gemessen wird	Was an seine Stelle tritt
SafeLine Community Edition 9.x	Wird nur als Docker-Compose-Stack ausgeliefert; der Host hatte keinen Docker-Daemon, und das Detektor-Image ist kein quelloffener Build, der sich dort kompilieren ließe	Veröffentlichte, vom Hersteller gemeldete Angaben (geprüft September 2026): Erkennungsrate 71,65 % (balance) und 76,17 % (strict), Fehlalarme 0,07 % (balance) und 0,22 % (strict); Mindest-Host Linux x86_64 oder arm64 mit SSSE3, 1 CPU-Kern, 1 GB RAM, 5 GB Festplatte, Docker 20.10.14+ und Compose 2.0+
Cloudflare WAF	Ein gehosteter Dienst; eine Loopback-Messung würde den Netzwerkpfad messen, nicht die WAF	Veröffentlichte Tarifkontingente (geprüft September 2026): eigene Regeln Free 5, Pro 20, Business 100, Enterprise 1000; Ratenlimit-Regeln Free 1, Pro 2, Business 5, Enterprise 100
Ein Cluster unter Last	Noch treibt kein Messprogramm mehrere Knoten auf einer PostgreSQL-Datenbank	Die Protokolleigenschaften, die SID 0011 spezifiziert (ein gemeinsamer Seed, an den Aussteller gebundene Challenges), und ihr Funktionstest mit drei Knoten (Kriterium 2); keine Messung
Speicherung auf dem Anfragepfad	Beide Daemons laufen in jedem Messprogramm im Speicher	SID 0011: Ein Speicherthread besitzt alles SQL, und der Anfragepfad wartet nie auf ihn
TLS, WAN, echte Browser	Die Messprogramme nutzen Loopback und HTTP; Sitzungen	Nichts; ein Einsatz misst seinen eigenen Pfad

Gegenstand	Warum es hier nicht gemessen wird	Was an seine Stelle tritt
	werden pro Fall einmal von `sibuna-pow` gelöst	
Die Lösungszeit des Browser-Lösers	Das Messprogramm löst bei 8 Bits mit dem nativen Löser	Die Modulgröße oben; Teil VII erklärt, warum native Zeitmessungen nicht für Lösungszeiten im Browser stehen
Allokationszahlen	Das Messprogramm instrumentiert den Allokator nicht	Die Tests für den eingeschwungenen Zustand in Teil V
Isolation der Konsole mit acht aktiven Dashboards unter Last	Das formale Abnahmekriterium von SID 0007 ist nicht ausgeführt worden	Der Wert aus SID 0010 (innerhalb von etwa 1 %, Konsole im Leerlauf) und der Funktionslauf oben

Eine Zeile in dieser Tabelle ist keine Aussage über die Leistung des Produkts. SafeLines eigene Zahlen betreffen die Erkennungsqualität, nicht den Durchsatz, und Cloudflare veröffentlicht Tarifkontingente statt Kosten pro Anfrage. Wer einen Docker-Host hat, kann SafeLine hinter denselben `wrk`-Arbeitslasten betreiben.

Übung 8.1. `cargo xtask bench primitives` auf einer anderen Maschine ausführen und die Zeilen für Hashcash-Verifikation und Token-Verifikation mit dem Referenzhost vergleichen. Welche der beiden hängt am stärksten von Hardware-Hash-Instruktionen ab, und warum?

Übung 8.2. In der Forward-Auth-Tabelle die belegten Kerne einer Zeile berechnen: ihre CPU-Mikrosekunden pro Anfrage mal ihre Anfragen pro Sekunde, geteilt durch eine Million. War der Daemon CPU-gebunden? Dann ihre CPU-Mikrosekunden pro Anfrage mit den Kosten der Klassifikation aus der ersten Tabelle vergleichen. Warum sind die CPU-Kosten pro Anfrage so viel höher, und welche Bestandteile erklären den Unterschied? *Hinweis:* Alles zwischen `accept` und der abgeschickten Antwort auflisten, was die Primitive-Suite nicht misst.

Die Invariante erklären. Jeder Wert in diesem Teil ist ein Median über Wiederholungen, mit seiner Streuung daneben. Den Unterschied zwischen einem Median über Batches und einem p99.9 pro Anfrage erklären, warum die Primitive-Tabelle ersteren und die Latenztabelle letzteren berichtet und warum nur letzterer das oben beschriebene Stocken durch Thread-Wechsel aufdecken konnte.

TEIL IX

Betrieb und Einsatz

Wir behandeln die beiden Betriebsformen, jede Kommandozeilenoption, Rezepte für Forward-Auth, Richtliniendateien, persistente Speicherung in SQLite oder PostgreSQL, Cluster über eine gemeinsame Datenbank und die Paketierung.

Betriebsformen und Konfiguration

Lernziele Am Ende dieses Kapitels sollten wir Sibuna als Reverse Proxy oder als Forward-Auth-Validator betreiben, eine Betriebsform wählen, die Proof-of-Work-Stufe und die Schwierigkeit festlegen und das Master-Geheimnis verwalten können.

Eine Betriebsform wählen

- *Gate* (`--gate`, Alias `--no-waf`): Proof-of-Work-Zulassung, Sitzungen, deklarative Regeln, Reputation, GCRA-Limits, Sperren.
- *Shield* (Standard, `--shield`, Alias `--waf`): Gate plus die semantische Firewall.
- *Verteilter Einsatz* (eine Option für beide Betriebsformen): `--data-dir` hält Richtlinien, Reputation und Forensik in einer lokalen SQLite-Datei; `--database-url` mit `--cluster-node` verbindet mehrere Knoten mit einer PostgreSQL-Datenbank (SID 0011).

Kommandozeilenreferenz

Die Tabelle listet jede Option von `sibuna --help`. Die Wertebereiche sind diejenigen, die der Parser durchsetzt.

Tabelle 23 Daemon-Optionen

Option	Standard	Bedeutung
`--port, -p`	`8080`	Port, auf dem gelauscht wird, 1 bis 65535
`--host, -h`	`0.0.0.0`	Adresse, auf der gelauscht wird, ein IP-Adressliteral
`--upstream-host`	`127.0.0.1`	Host des Ursprungsservers (Reverse-Proxy-Modus)
`--upstream-port, -u`	`3000`	Port des Ursprungsservers
`--mode, -m`	`reverse_proxy`	`reverse_proxy` oder `forward_auth` (auch `proxy` und `auth`), genau einmal angegeben
`--workers, -w`	eine pro CPU	Accept-Threads, die sich den lauschenden Socket teilen, 0 (eine pro CPU) bis 64; jede ange-

Option	Standard	Bedeutung
		nommene Verbindung erhält danach einen eigenen Thread
`--max-connections`	`1024`	Gleichzeitig bediente Verbindungen, 1 bis 8192; weitere werden mit `503` beantwortet
`--max-connections-per-ip`	`64`	Verbindungen, die eine Peer-Adresse eines Sockets halten darf, 0 (keine Grenze) bis 8192; ein IPv6-Peer zählt mit seinem `--ipv6-prefix`-Netz, und weitergeleitete Header spielen keine Rolle; weitere werden mit `503` beantwortet
`--idle-timeout`	`15`	Längste Stille in Sekunden, 0 bis 86400. Ein Anfragekopf muss innerhalb dieser Zeit eintreffen. Eine Antwort des Ursprungsservers erneuert die Frist bei jedem Lesen, sodass ein langsamer Stream lebt, solange Bytes fließen, und ein stummer Ursprungsserver auf beiden Sockets getrennt wird. Ein weitergeleiteter Upload muss pro Periode 16 KiB liefern (eine Mindestrate gegen Slow-Body-Angriffe). Der Client-Socket trägt ein Schreib-Timeout derselben Länge, sodass ein Client, der so lange kein Byte annimmt, die Verbindung verliert
`--websocket-idle-timeout`	`300`	Leerlauffrist einer per Upgrade umgestellten Verbindung in Sekunden, 0 bis 86400
`--trust-forwarded`	aus; an im Forward-Auth-Modus	`X-Forwarded-For` / `X-Real-IP` vom Peer berücksichtigen
`--no-trust-forwarded`	aus	Sie nie berücksichtigen, auch nicht im Forward-Auth-Modus
`--trusted-hops`	`1`	Bei vertrauenswürdiger weitergeleiteter Identität ist der Client der Eintrag von `X-Forwarded-For`, der so viele Positionen von rechts steht, 1 bis 8
`--ja4-header`	aus	Der Anfrage-Header, in dem ein vertrauenswürdiger Eingangsproxy den JA4-TLS-Fingerabdruck des Clients sendet, etwa `cf-ja4`; 1 bis 64 Token-Zeichen, nicht `X-Forwarded-*`, `X-Real-IP`, `X-Sibuna-*` oder `Forwarded`. Er wird nur gelesen, wenn der weitergeleiteten Identität ver-

Option	Standard	Bedeutung
		traut wird (siehe „JA4-Fingerabdrücke vom Eingangsproxy“)
`--ipv6-prefix`	`64`	IPv6-Präfixlänge, 32 bis 128, für die Ratenlimits, Sperren und die Verbindungsgrenze pro Client gelten
`--algorithm, -a`	`posw`	`posw` oder `hashcash`
`--difficulty, -d`	`16`	Arbeitsbits, 1 bis 64: Nullbits bei Hashcash oder PoSW-Tiefe plus drei
`--posw-challenges`	`16`	Öffnungen pro Nachweis sequenzieller Arbeit, 1 bis 32
`--token-scheme`	`mac`	`mac` (BLAKE3 mit Schlüssel) oder `ed25519`
`--token-ttl`	`86400`	Lebensdauer der Sitzung, Sekunden (1 bis 31.536.000)
`--challenge-ttl`	`300`	Lebensdauer der Challenge, Sekunden (1 bis 86.400)
`--secret-file, -s`	zufällig	64 Hex-Zeichen oder 32 Rohbytes; `SIBUNA_SECRET` (64 Hex-Zeichen) wird ebenfalls akzeptiert
`--cookie-name`	`__sibuna_token`	Name des Sitzungscookies, 1 bis 64 Token-Zeichen
`--secure-cookie`	aus	Das Attribut `Secure` hinzufügen
`--gate` / `--shield`	shield	Auswahl der Betriebsform
`--rate-limit`	`100`	Anfragen pro Fenster und Client (GCRA-Burst)
`--rate-window`	`10`	Fenster in Sekunden, 1 bis 86400
`--challenge-rate-limit`	`30`	Ausgaben und Prüfungen von Challenges pro Fenster und Client, getrennt vom Anfragebudget
`--ban-seconds`	`3600`	Sperrdauer durch den Honeypot, 0 bis 31.536.000
`--policy-file, -P`	keine	Deklarative JSON-Richtlinie
`--rate-table-cells`	`262144`	Zellen pro Tabelle des Ratenbegrenzers, je 16 Bytes, 4.096 bis 16.777.216, aufgerundet auf eine Zweierpotenz

Option	Standard	Bedeutung
`--ban-table-slots`	`65536`	Slots der Sperrtabelle, je 24 Bytes, 1.024 bis 16.777.216, aufgerundet auf eine Zweierpotenz
`--spent-tags`	`1048576`	Gelöste Challenges, die gegen Wiedereinspielung gespeichert werden, je 24 Bytes, 65.536 bis 16.777.216
`--data-dir, -D`	keine	Ein Knoten: dauerhaft in der SQLite-Datei `<path>/sibuna.db` speichern
`--database-url`	keine	Ein Cluster: die gemeinsame PostgreSQL-Datenbank (`postgres://` oder `postgresql://`)
`--database-url-file`	keine	Die Datenbank-URL aus einer Datei lesen, die nur der Eigentümer lesen darf
`--cluster-node`	keine	Die Kennung dieses Knotens; bindet Challenge-Schlüssel und die oberen Bits der Vorfallskennungen; mit einer Datenbank-URL erforderlich
`--storage-poll-ms`	`500`	Taktintervall des Speicherthreads, 1 bis 60.000 ms
`--verbose, -v`	aus	Ausführliche Protokollierung
`--help` / `--version`		Die Hilfe (zuerst die Konsolenoptionen, wenn die Konsole einkompiliert ist) oder die Version ausgeben und beenden

`--data-dir` und eine Datenbank-URL schließen einander aus, ebenso `--database-url` und `--database-url-file`. Eine Datenbank-URL ohne `--cluster-node` wird abgewiesen. Die Zaxonlite-Clusteroptionen `--cluster-listen`, `--cluster-peer`, `--cluster-secret-file` sowie `--cluster-tls-cert`, `--cluster-tls-key` und `--cluster-tls-ca` gibt es nicht mehr; wird eine davon angegeben, bricht der Start mit `RemovedOption` und einem Hinweis auf SID 0011 ab. Die Optionen der Konsole stehen im nächsten Kapitel.

Das Master-Geheimnis Ohne `--secret-file` oder `SIBUNA_SECRET` erzeugt der Daemon einen zufälligen Seed, und das Banner gibt „random (set –secret-file)“ aus. Tokens und Challenges werden dann mit dem Ende des Prozesses ungültig, und ein Cluster, dessen Knoten verschiedene Seeds halten, weist die Cookies der jeweils anderen ab. Einen Seed mit `head -c 32 /dev/urandom | xxd -p -c 64 > /etc/sibuna/secret` und Modus 0600 erzeugen.

Vorschau der Konsole

Die Konsole wird über das Standard-Feature `console` in den Daemon einkompiliert, startet aber nur mit `--console`. Sie braucht eine Speicherung: `--console` ohne `--data-dir` oder `--database-`

`url` wird mit `CONSOLE001` (`StorageRequired`) abgewiesen. Mit `cargo build --release -p sibuna --no-default-features --features sqlite` lässt sie sich herauskompilieren; ein Build mit `--no-default-features` allein hat weder Speicherung noch Konsole. Die lokale Ersteinrichtung erfolgt bei angehaltenem Daemon:

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

`sibuna console init-admin` ist derselbe Befehl. `http://127.0.0.1:19446/console/` öffnen und das erzeugte temporäre Passwort ersetzen, das bei der ersten Anmeldung geändert werden muss und nach einer Stunde abläuft. Die Navigation nach der Anmeldung bietet den animierten Länderglobus, Anfrage- und Challenge-Statistiken, die Untersuchung von Vorfällen, Editoren für Richtlinien und Prüfung, Benutzer, API-Tokens mit begrenztem Geltungsbereich, Audit sowie Steuerelemente für den bedienenden Knoten. Die Authentifizierungsoberfläche lädt weder Globusgeometrie noch Telemetrie. Die Oberfläche ist ein WebAssembly-Modul, das das Build-Skript des Daemons aus `cmd/console-ui` (Crate `sibuna-console-ui`) baut; es wird zusammen mit dem eingecheckten Stylesheet in gewöhnlichen Builds ausgeliefert.

Tabelle 24 Konsolenoptionen

Option	Bedeutung
`--console <ip:port>`	Die Konsole auf diesem Listener starten (IPv6 in eckigen Klammern, Port ungleich null); erfordert `--data-dir` oder `--database-url`
`--console-key-file <path>`	64 Hex-Zeichen, Rechte nur für den Eigentümer; schützt gespeicherte Geheimnisse für den zweiten Faktor und für Webhooks
`--console-origin <https-origin>`	Der kanonische HTTPS-Origin hinter einem Proxy
`--console-behind-proxy`	Ein TLS-terminierender Proxy liefert jede Anfrage
`--console-trusted-proxy <cidr>`	Ein Netz, dessen Verbindungen weitergeleitete Felder tragen dürfen; wiederholbar, höchstens 16
`--console-advertise <origin>`	Der Link, den andere Konsolen für diese anzeigen
`--console-location <latitude,longitude>`	Die angegebene Position dieses Knotens auf dem Globus
`--console-probe <node-id>=<http://ip:port>`	Ein Data-Plane-Listener eines Peers, dessen Zustand geprüft wird; wiederholbar, höchstens acht
`--console-peer <node-id>=<https://origin>`	Ein Management-Peer; wiederholbar, höchstens acht
`--console-peer-key-file <path>`	Der Peer-Schlüssel: nur für den Eigentümer, 64 Hex-Zeichen, unabhängig vom Konsolenschlüssel

Option	Bedeutung
`--console-peer-ca-file <path>`	Optionale PEM-Vertrauensanker für Peers, die die Systemwurzeln ersetzen
`--console-query-steps <n>`	SQLite-Schritte pro Aggregatabfrage, 100.000 bis 50.000.000, Standard 4.000.000
`--console-capture-heads`	Geschwärzte Köpfe von Anfrage und Ursprungsantwort pro Vorfall speichern; standardmäßig aus
`--console-capture-header <name>`	Den Wert dieses Headers in gespeicherten Köpfen behalten; wiederholbar, höchstens 16; erfordert `--console-capture-heads`
`--console-cookie-secure`	Konsolen-Cookies ohne Proxy das Attribut `Secure` geben (vom Parser akzeptiert, nicht in `--help` aufgeführt)

Eine Konsolenoption ohne `--console`, eine wiederholte Option oder ein fehlerhafter Wert bricht den Start mit `CONSOLE002` und dem Fehlernamen der Option ab. Eine Konfiguration, die eine Regel der Konsole verletzt (zum Beispiel ein Listener außerhalb des Loopbacks ohne vertrauenswürdigen Proxy), bricht ihn mit `CONSOLE001` ab.

Ein Rundgang durch die Konsole

Die folgenden Bildschirmfotos stammen von einem Release-Build auf einem Loopback-Knoten mit SQLite-Speicher und einer Richtlinie aus vier `ALLOW`-Regeln über einer Standard-Challenge, nach etwa vier Minuten synthetischen Datenverkehrs: Adressen aus Dokumentationsbereichen, Sondierungen auf SQL-Injection und Cross-Site-Scripting, drei Honeypot-Treffer und sieben in Headless Chromium gelöste Challenges. Jede Zahl ist daher nur ein Beispiel. Wir haben sie mit `cargo xtask browser console --viewport-only` aufgenommen; der Befehl durchläuft die Konsole in Chromium mit 1440 mal 960 Pixeln und bewertet jede Seite. Jede Seite beantwortet eine Frage, die in ihrem Titel steht, und zeigt an derselben Stelle das Produkt, den bedienenden Knoten, die Seite und den Weg zurück.

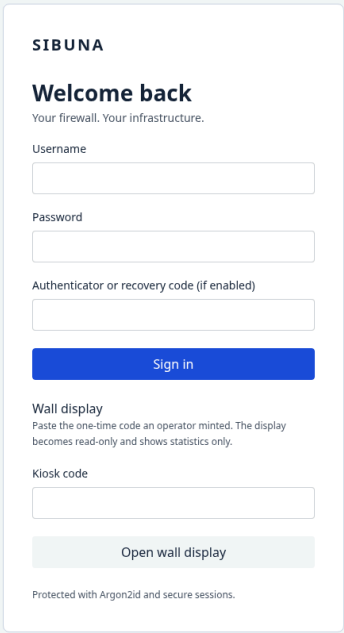


Abb. 13 Anmeldung. Die öffentliche Oberfläche lädt weder Globusgeometrie noch GeoIP-Daten noch Telemetrie; das zweite Formular tauscht einen Code für eine Wandanzeige gegen eine schreibgeschützte Statistiksitzung.

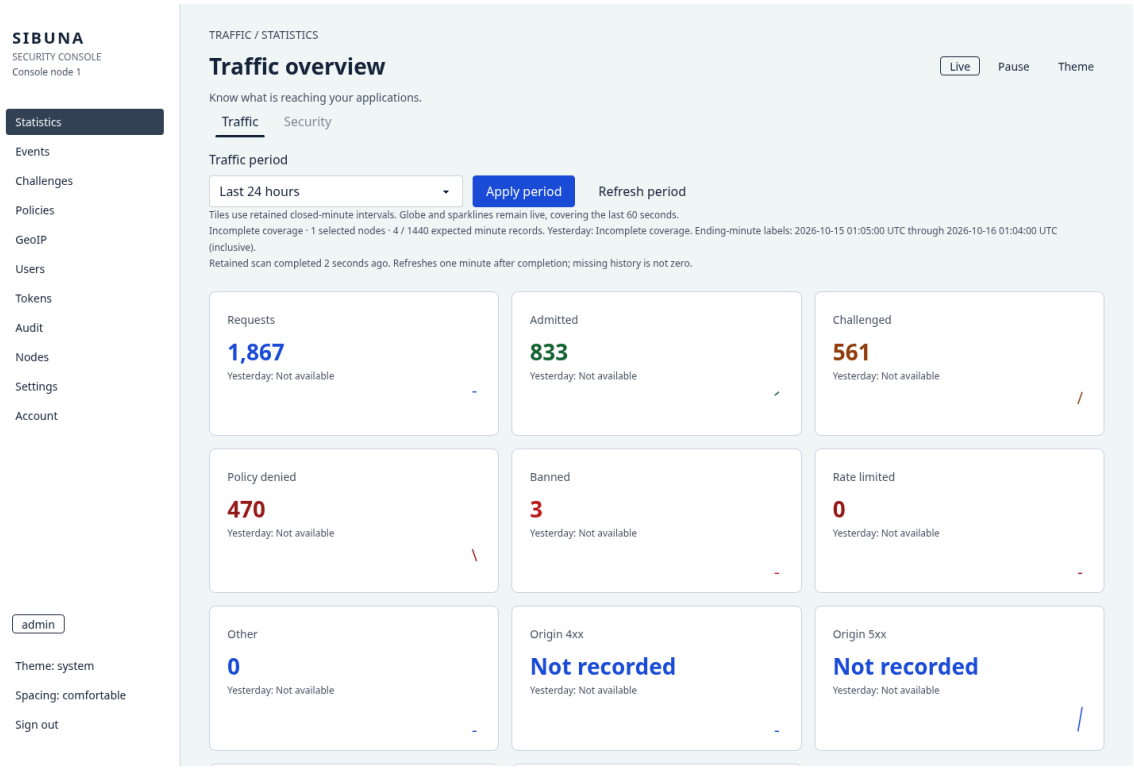


Abb. 14 Statistics, Traffic, über die letzten 24 Stunden. Der Hinweis unter der Zeitraumauswahl nennt die Abdeckung: Nur vier von 1.440 erwarteten Minutendatensätzen existieren, daher zeigt jedes „Yesterday“ den Wert „Not available“. Kacheln zeigen aufbewahrte Zählungen abgeschlossener Minuten mit einer Live-Sparkline über 60 Sekunden; die Entscheidungsfarben (zugelassen grün, mit Challenge belegt bernsteinfarben, abgelehnt rot) sind in jedem Feld gleich. Die Kacheln Origin 4xx und Origin 5xx zeigen für einen aufbewahrten Zeitraum „Not recorded“, weil die Minutendatensätze den Proxy-Modus nicht festhalten, der einer Ursprungszählung ihre Bedeutung gibt; die Live-Summen seit dem Start zeigen sie. Unter den sichtbaren Kacheln liegen zwei Zustandskacheln, aktive Sperreinträge und gesunde Knoten, die Zeitleiste sowie Stichprobenfelder, die Anfragepfade und verweisende Hosts ordnen und Betriebssysteme, Browser und Antwortstatus der Clients zählen.

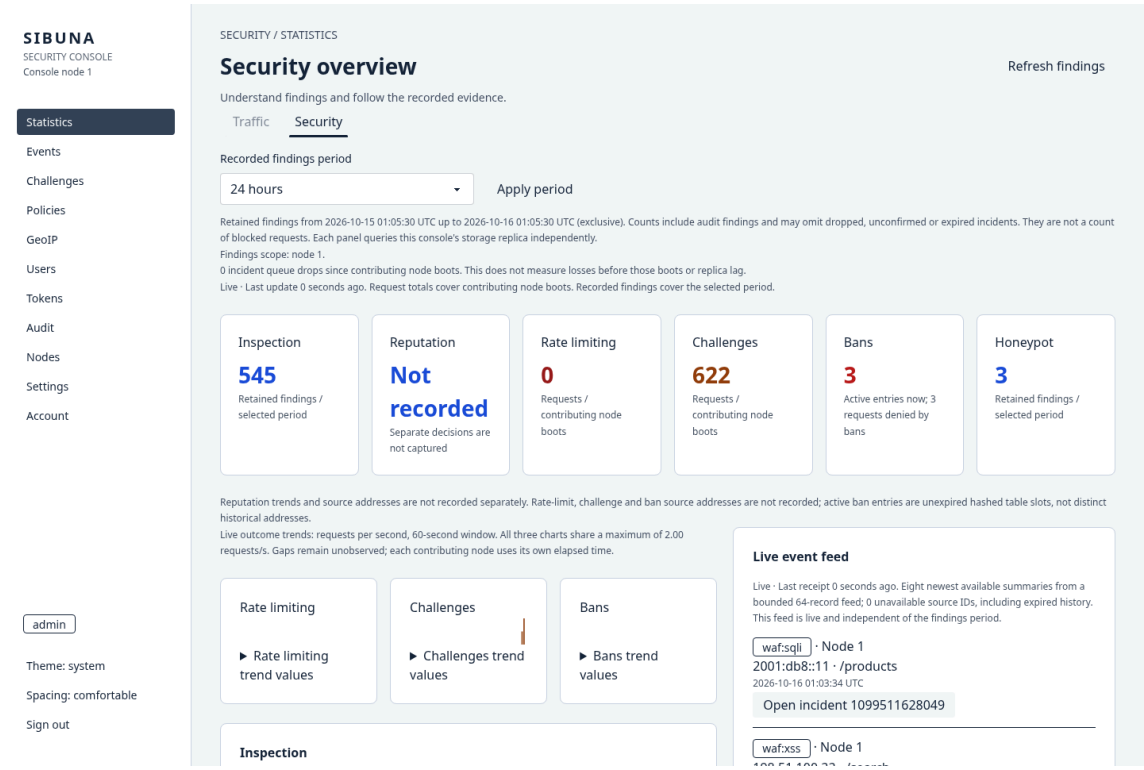


Abb. 15 Statistics, Security. Eine Kachel pro Modul über den gewählten Zeitraum, Live-Trends der Ergebnisse und der Live-Ereignisstrom mit seinen neuesten Befunden (hier `waf:sqli` und `waf:xss`). Nicht erfasste Werte sagen das, statt null zu zeigen: Die Kachel Reputation zeigt „Not recorded“. Weiter unten auf der Seite stehen Befunde über sechzig gleiche Intervalle, Angriffskategorien und angegriffene Pfade.

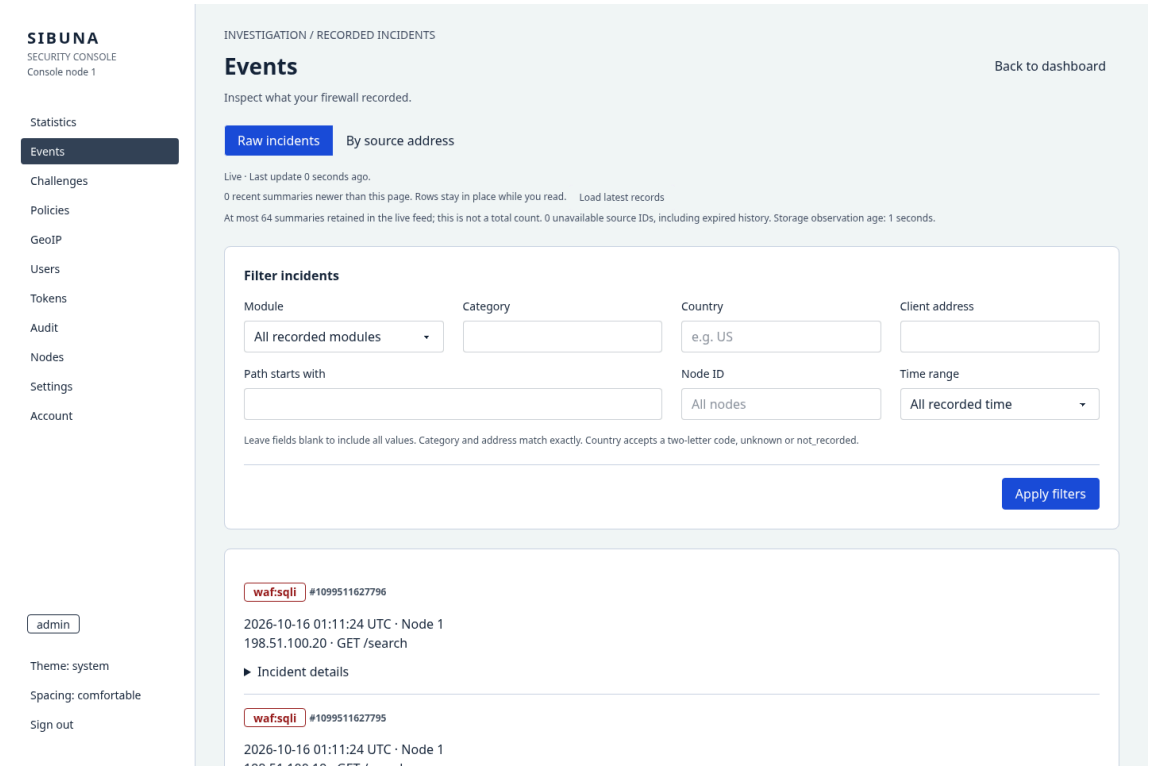


Abb. 16 Events. Das Filterformular über den aufgezeichneten Vorfällen, die neuesten zuerst. Jeder Vorfall öffnet sich direkt in der Liste mit seinen Belegen: der gewählten lokalen Antwort, Bytelängen, dem Kampagnenkandidaten und ausdrücklichen „not recorded“-Einträgen für die getroffene Regel, die Bewertungsterme und den JA4-Fingerabdruck. Mit `--console-capture-heads` öffnet sich auf Anforderung der geschwärzte Anfragekopf (und bei auditierten Zulassungen über den Reverse Proxy der Kopf der Ursprungsantwort), dargestellt als UTF-8 oder Latin-1, mit Angabe des Antwortzustands (erfasst, lokal, bei Forward-Auth nicht beobachtet, Ursprungsserver nicht verfügbar) und einem Befehl zum Kopieren als cURL; nur aufgelistete Header-Werte werden behalten, und `--console-capture-header <name>` erweitert diese Liste. Andernfalls gelten Köpfe als nicht erfasst. Die Aktionen Deny und Allow öffnen das Formular für IP-Gruppen mit der Adresse als Entwurf.

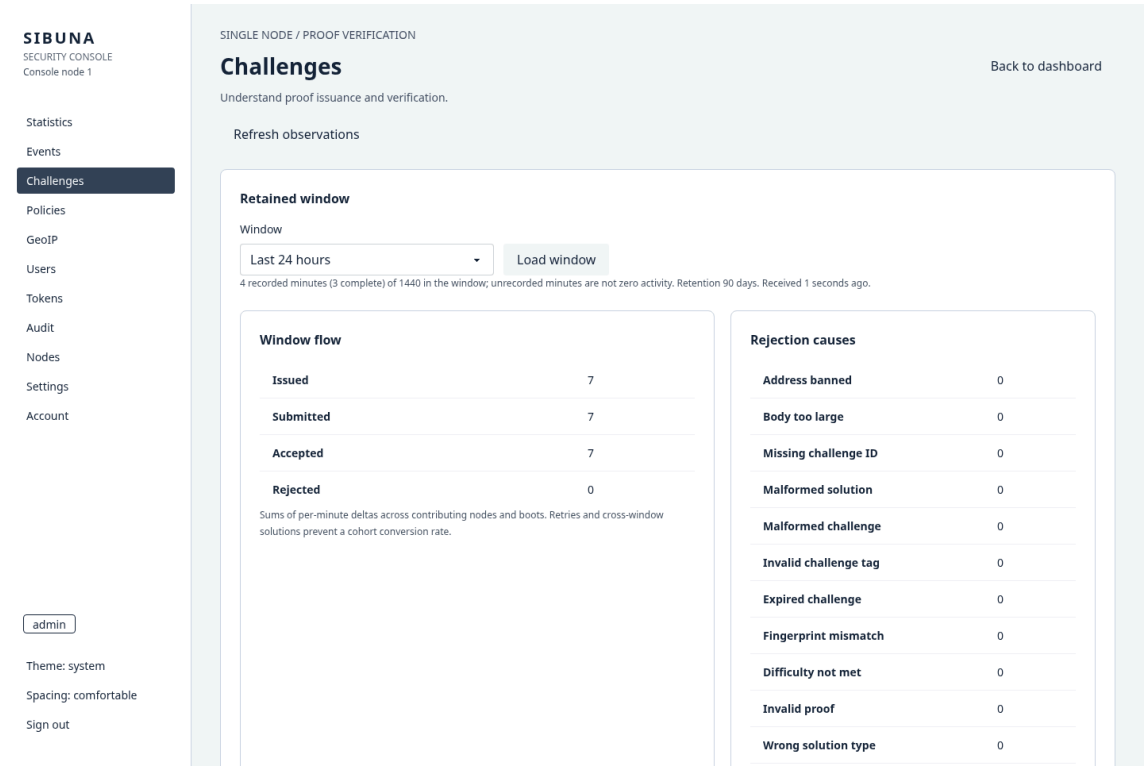


Abb. 17 Challenges. Ein aufbewahrtes Fenster (eine Stunde, ein Tag oder sieben Tage) summiert dauerhafte Datensätze pro Minute und gibt an, wie viele Minuten erfasst und vollständig waren; hier wurden sieben Challenges ausgegeben, eingereicht und akzeptiert, und alle Ablehnungsgründe stehen auf null. Weiter unten stehen die Live-Summen seit diesem Start, die konfigurierten und zuletzt ausgegebenen Parameter, das Histogramm der Lösungszeiten akzeptierter Challenges, aufgeteilt nach Algorithmus und Parameterklasse, Übergänge der adaptiven Schwierigkeit und Datensätze pro Adresse mit Abkürzungen zum Ablehnen und Erlauben, sieben Tage aufbewahrt.

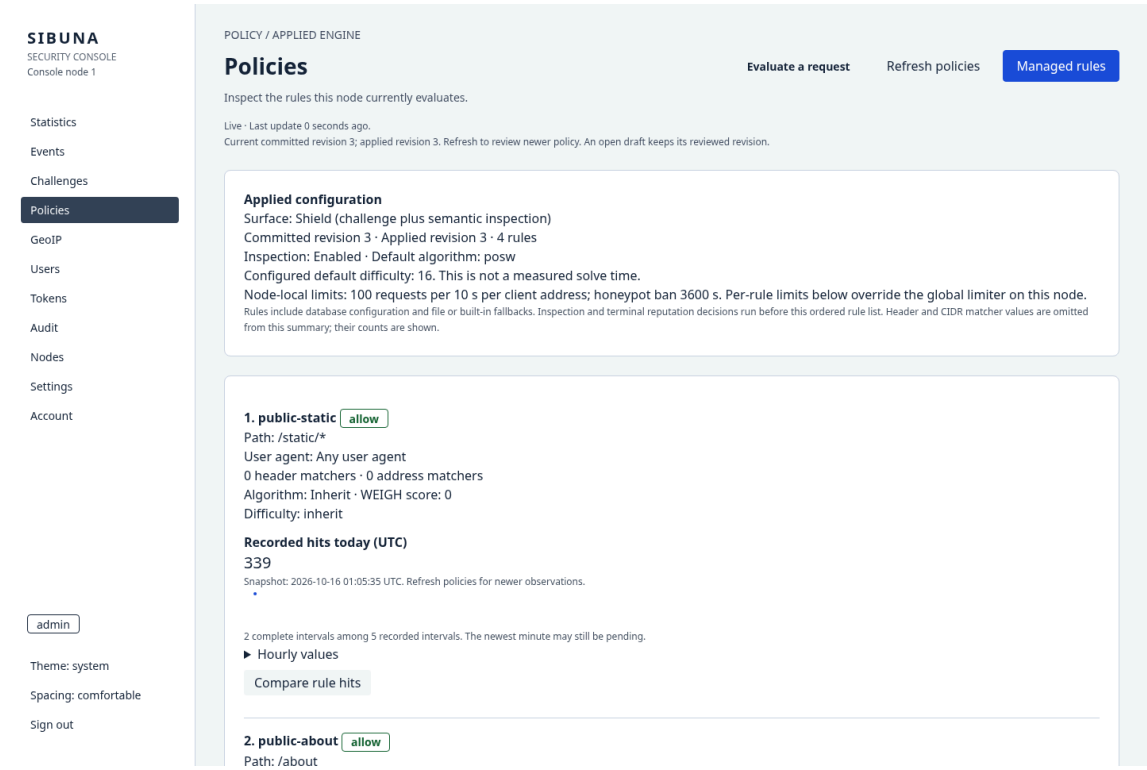


Abb. 18 Policies. Die angewandte Engine nennt ihre Betriebsform, Revisionen, Standardalgorithmus und Schwierigkeit sowie ihre knotenlokalen Limits und listet dann die Regeln in Auswertungsreihenfolge mit ihren heutigen Treffern und der Abdeckung dieser Zählungen. Weiter unten stehen die Matrix der Prüfmodi, der Anfragetester und die IP-Gruppen. *Managed rules* öffnet den Editor; das Speichern einer Änderung zeigt zuerst eine Durchsicht der geänderten Felder, und die Bestätigung validiert den ganzen Kandidaten und erzeugt eine neue Revision.

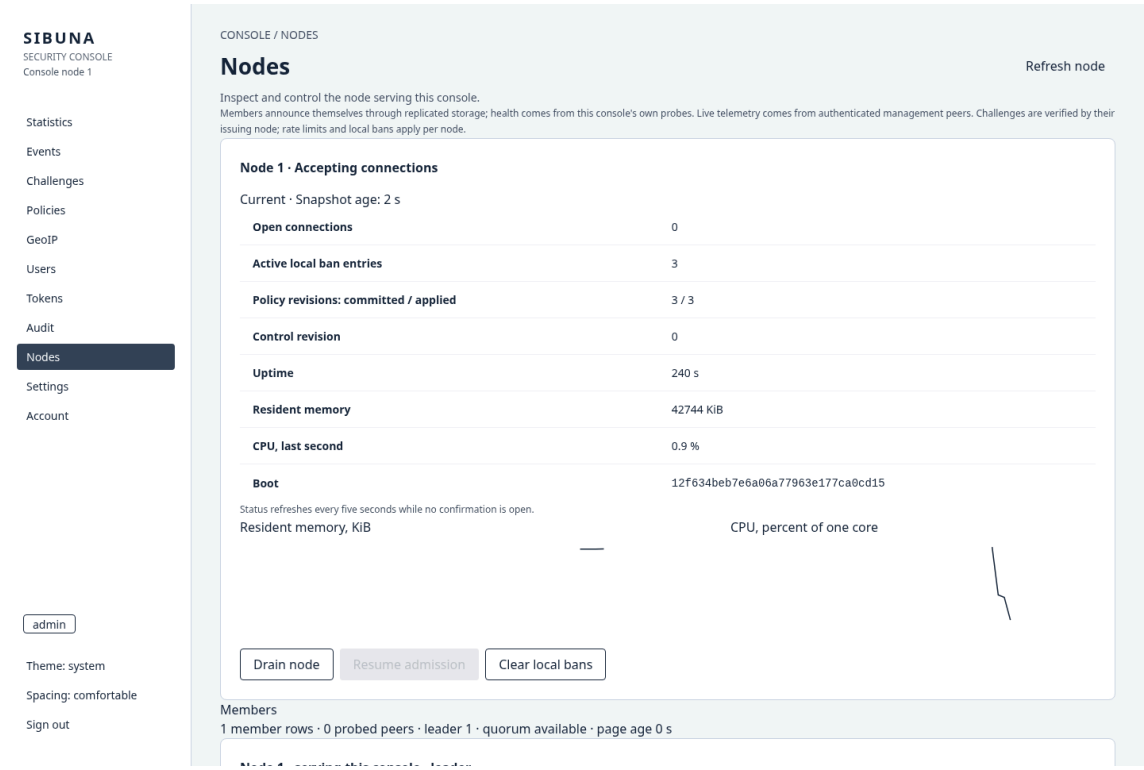


Abb. 19 Nodes. Status und Befehle des bedienenden Knotens, mit residentem Speicher und CPU der letzten Sekunde und ihren Sparklines über sechzig Sekunden; darunter jedes angekündigte Mitglied mit seiner angewandten Richtlinienrevision im Vergleich zur festgeschriebenen Revision, seiner Management-Adresse und seinem Prüfergebnis (SID 0007, SID 0013).

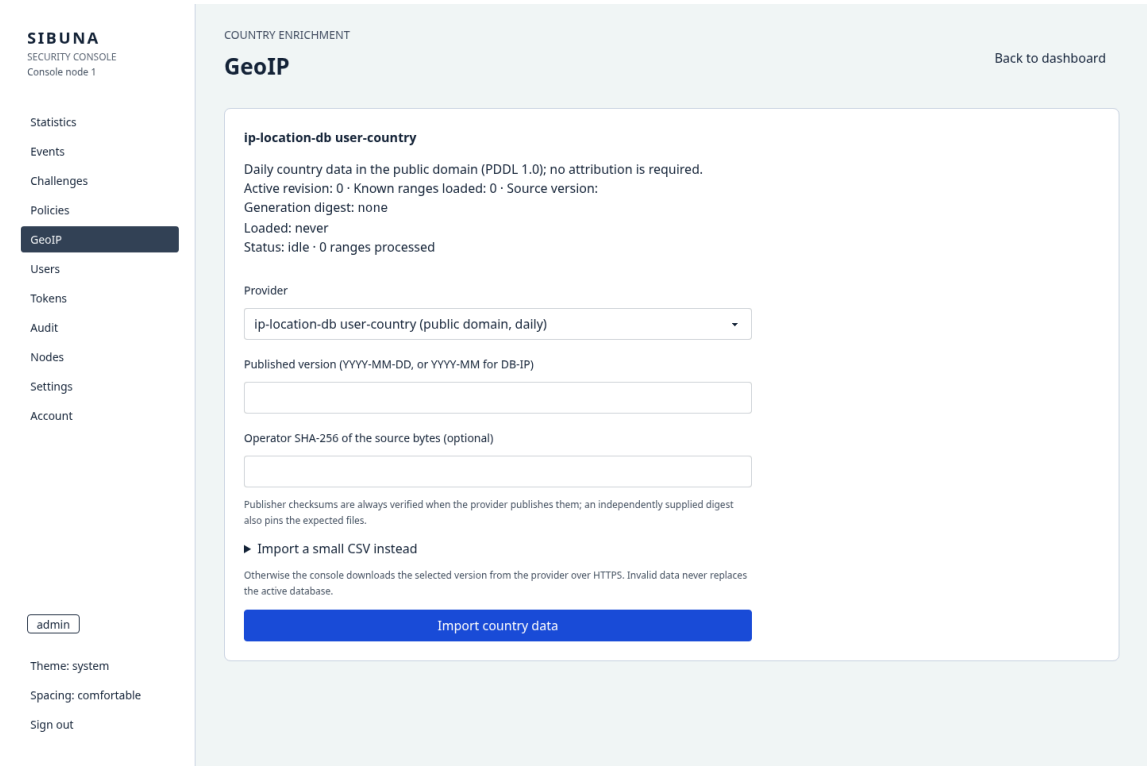


Abb. 20 GeoIP. Anbieter, Lizenz, Digest der aktiven Generation, Ladezeit und das Importformular; hier wurde noch keine Generation geladen. Ein fehlgeschlagener Import ersetzt nie die aktive Generation.

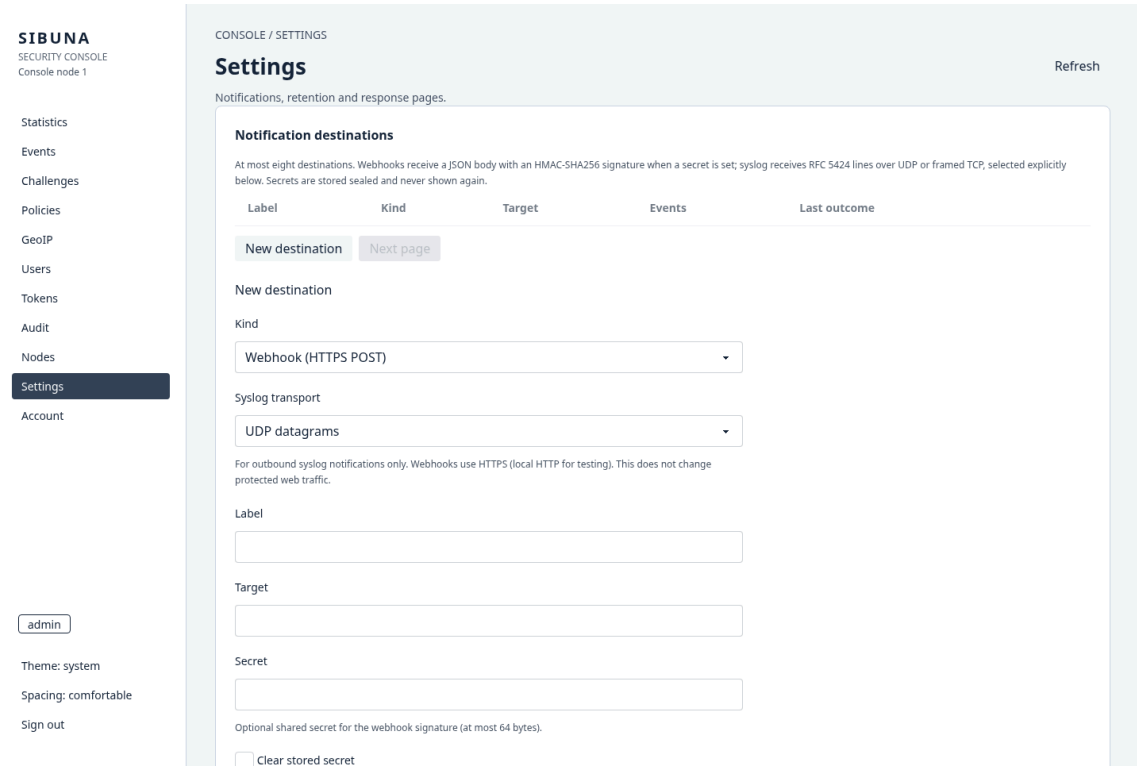


Abb. 21 Settings, noch ohne Benachrichtigungsziel und mit dem Formular für ein neues. Weiter unten auf der Seite stehen Schwellen für Ablehnungsspitzen, die Aufbewahrung mit einer ausdrücklichen Bestätigung pro Fenster, Vorlagen für Antwortseiten und About.

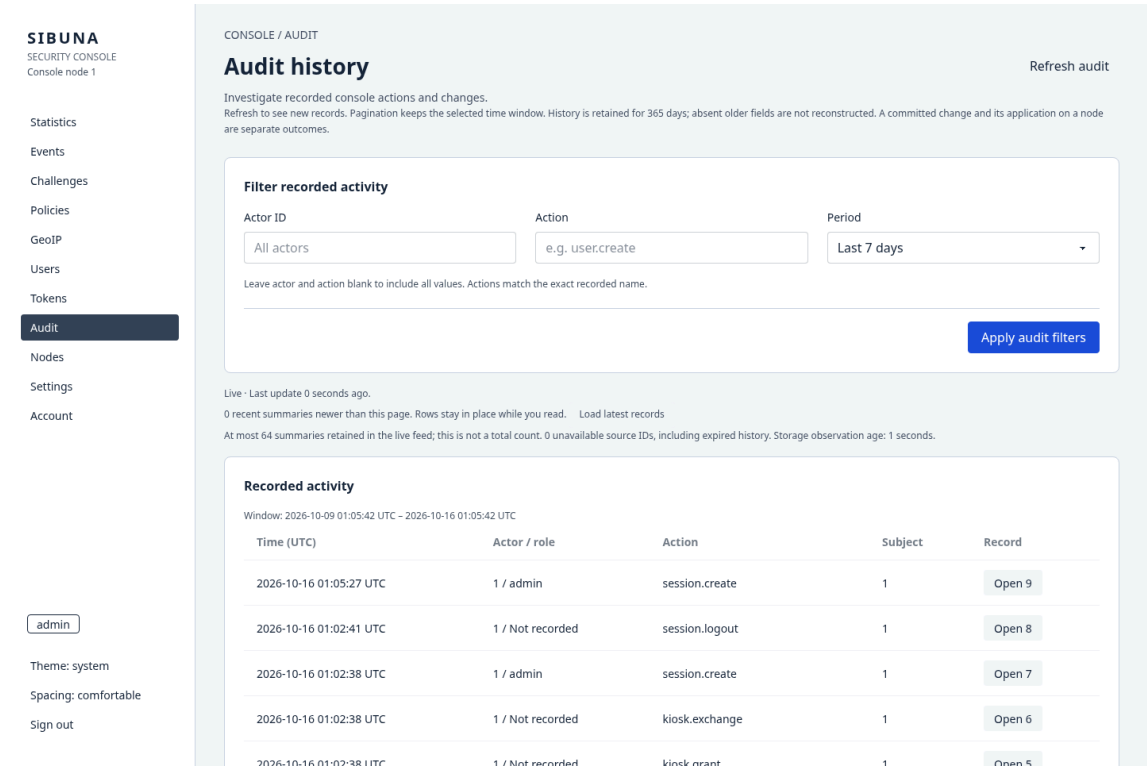


Abb. 22 Audit. Eine nur erweiterbare Historie mit Akteur und Rolle, Aktion, Gegenstand und einer Detailansicht mit geschwärzten Zusammenfassungen des Vorher und Nachher; abgewiesene Anmeldungen werden ebenfalls erfasst. Jede Management-Änderung und jede Authentifizierungszeile hält die Client-Adresse fest, die die Anmeldedaten vorgelegt hat; Zeilen, die das System selbst schreibt, zeigen sie als nicht erfasst. Ein Richtliniendatensatz bietet *Revert this change*; das stellt nach einer Bestätigung das vor dieser Revision erfasste Dokument als neue, auditierte Revision wieder her und wird abgewiesen, wenn sich der Regelsatz inzwischen weiterentwickelt hat.

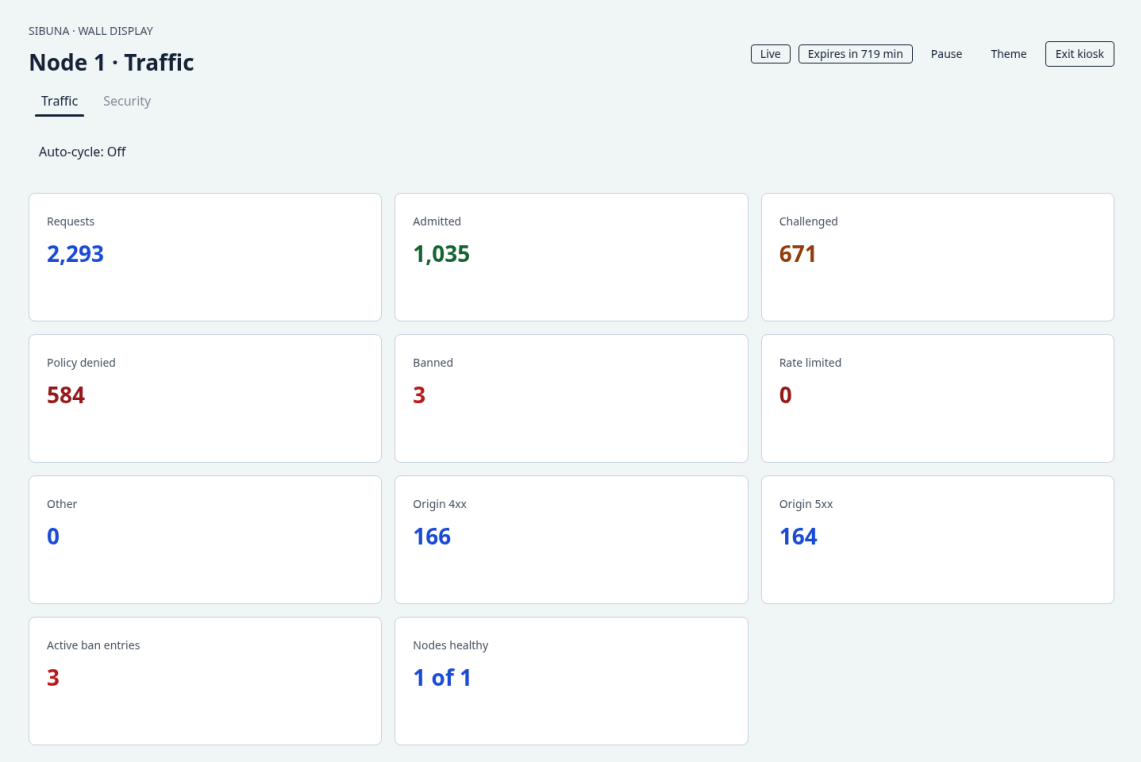


Abb. 23 Eine Wandanzeige. Eine Kiosk-Sitzung zeigt nur die Ansichten Traffic und Security, ohne Navigation, und gibt an, wann sie abläuft.

Die optionale Angabe `--console-location <latitude,longitude>` legt die Position dieses Knotens fest, zum Beispiel `1.3521,103.8198` für einen Einsatz in Singapur. Der Globus ist zunächst dort zentriert; Center Sibuna kehrt nach einer Drehung zu diesem Punkt zurück. Die Aktivität der Länder folgt animierten Großkreisbögen zum Knoten, mit Abschneiden an der Rückseite der Halbkugel und an der Datumsgrenze der flachen Karte. Länderpositionen sind repräsentative Schwerpunkte, und Pfeile stellen das beobachtete Fenster von sechzig Sekunden dar, nicht einzelne Verbindungen. Eine nicht gesetzte Serverposition bleibt ausdrücklich unbekannt. Die konfigurierte Markierung bleibt bei Telemetrieausfällen sichtbar; veralteter Datenverkehr wird nicht animiert.

Richtlinienvorschauen bauen einen privaten Kandidaten einschließlich der konfigurierten Dateiregeln. Beim Speichern wird die erwartete Revision verglichen, und ein Feldvergleich muss bestätigt werden. Historische Rücknahmen vergleichen mit der aktuellen Regel und erzeugen eine neue Revision. Festgeschriebene und lokal angewandte Revisionen sind verschieden. Audit-Details behalten begrenzte Entscheidungsänderungen und die tatsächlich handelnde Rolle. Werte von Matchern werden geschwärzt, und bei alten Datensätzen bleibt fehlender Kontext ausdrücklich als fehlend gekennzeichnet. Drain, Resume und das Löschen lokaler Sperren erfordern eine Befehlsvorschau. Ein Befehl, der eintrifft, während auf dem Knoten ein anderer läuft, wird mit `503 CONSOLENODE` abgewiesen, statt in eine Warteschlange zu kommen. Nach einer verlorenen Antwort vor einem neuen Versuch die dauerhafte Quittung prüfen; eine festgeschriebene Absicht allein belegt nicht, dass die Wirkung zur Laufzeit abgeschlossen ist.

Für den Fernbetrieb einen HTTPS-Proxy verwenden und `--console-origin`, `--console-behind-proxy`, ausdrückliche CIDRs für `--console-trusted-proxy` und `--console-key-file` konfigurieren. Die Schlüsseldatei enthält 64 hexadezimale Zeichen, hat Rechte nur für den Eigentümer und schützt gespeicherte Geheimnisse für den zweiten Faktor; sie ist über Neustarts hinweg getrennt vom Challenge-Seed aufzubewahren. Der konfigurierte HTTPS-Ursprung und die Allowlist der Proxys sind für Zugriffe außerhalb des Loopbacks Pflicht. Auch auf dem Konsolen-Listener terminiert Sibuna kein eingehendes TLS; die Konsole nutzt TLS nur für ihre ausgehenden Verbindungen, über rustls (SID 0012).
Die native CLI greift über dieselben Autorisierungs- und Speicherverträge auf die laufende Konsole zu. Sie liest Anmeldedaten aus privaten Dateien, nicht aus Werten der Kommandozeile:

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

Eine Passwortdatei nur für den Eigentümer verwenden, zuerst die Passworteinrichtung abschließen und bei Bedarf `--factor-file` für einen Authenticator- oder Wiederherstellungscode hinzufügen; `--token-file` ersetzt Benutzername und Passwort bei Konto- und GeoIP-Operationen. Der Client spricht HTTP/1.1, verlangt HTTPS außer bei einer literalen Loopback-Adresse, folgt nie Weiterleitungen und begrenzt jede Anfrage auf 20 Sekunden und jede Antwort auf 16 KiB. `geoip status` liest die aktive Generation. Aktualisierungen laden die veröffentlichten Dateien des gewählten Anbieters über HTTPS herunter (standardmäßig den gemeinfreien Datensatz `user-country` oder DB-IP Lite mit `--provider dbip --version YYYY-MM`), prüfen die Prüfsummen des Herausgebers (oder die mit `--checksum` angegebene), validieren begrenzte Bereiche über `sibuna-geoip`, speichern die neue Generation dauerhaft und aktivieren sie lokal. Fehlgeschlagene Aktualisierungen erhalten die vorherige Generation. Ein CLI-Timeout (`--timeout`, Standard 1200 Sekunden) beendet das Warten; es bricht bereits übergebene Speicherarbeit nicht ab. DB-IP Lite verlangt eine Namensnennung nach CC BY 4.0, die nur angezeigt wird, solange seine Daten aktiv sind. Ein Build mit `SIBUNA_GEOIP_DATA=<snapshot>` bettet einen validierten Snapshot ein (geschrieben vom Werkzeug `geoip-snapshot`), der bis zum ersten dauerhaften Import gilt; ein Build ohne Konsole weist ihn mit `GEOIP001` ab. Unbekannte Adressen, Stichprobenverluste und veraltete Daten bleiben sichtbar; der Import von Länderdaten erzeugt keinen Datenverkehr und reichert keine bereits abgelaufenen Stichproben an.
Cluster-Mitglieder kündigen sich über die Mitgliedertabelle der gemeinsamen Datenbank (`console_nodes`) an: beim Start, jede Minute und innerhalb von zwei Sekunden nach einer Änderung der angewandten Richtlinien- oder Steuerrevision; sie erscheinen auf der Seite Nodes jeder Konsole. Es gibt keine Leader-Wahl: Ein SQLite-Knoten meldet die Rolle `single` und sich selbst als Leader, und ein PostgreSQL-Knoten meldet die Rolle `unknown` und keinen Leader, weil das Failover von PostgreSQL für Sibuna nicht sichtbar ist (SID 0007). `--console-probe <node-id>=<http://ip:port>` für die Peers konfigurieren, deren Data-Plane-Listener diese Konsole auf ihren Zustand prüfen soll, und `--console-advertise <origin>` für den Link, den andere Konsolen anzeigen. Direkte Live-Telemetrie verwendet `--console-peer <node-id>=<https://origin>` (wiederholbar, höchstens acht) und eine eigene `--console-peer-key-file`. Peers erfordern `--console-behind-proxy`. Beide Endpunkte müssen einander

aufführen und einen vertrauenswürdigen HTTPS-Eingangsproxy nutzen. Der Peer-Schlüssel ist eine Datei nur für den Eigentümer mit 64 Hex-Zeichen, die unabhängig vom Konsolenschlüssel und vom Challenge-Seed bereitgestellt wird. Peers kommunizieren über einen authentifizierten TLS-WebSocket, über rustls mit dem Provider `ring` (SID 0012). `--console-peer-ca-file` liefert optional PEM-Vertrauensanker für eine private Management-PKI; andernfalls nutzt der Client die Systemwurzeln. Die Prüfung des Hostnamens gilt immer und lässt sich nicht abschalten: Der Host jedes Management-Ursprungs, ein DNS-Name oder ein IP-Adressliteral, muss im Zertifikat des Peers genannt sein. Ein abgewiesenes Zertifikat markiert den Peer als `rejected` und schreibt eine Audit-Zeile `peer.rejected`. Schlüsselrotation und ein Wechsel der CA-Datei erfordern einen koordinierten Neustart; das Zertifikat eines Peers lässt sich ohne Neustart erneuern. Die Knoten-API und ihr Abonnement melden das Alter der Quittung, die Uhrenabweichung, Neustarts und Stichprobenverluste. Fehlende Beobachtungen bleiben nicht verfügbar, und getrennte Werte bleiben veraltet; empfangene Statistiken werden nie zum eigenen Beitrag eines anderen Knotens. Der *Live traffic scope* des Dashboards wählt einen Knoten oder kombiniert die konfigurierten Knoten. *Node coverage and locations* zeigt fehlende, veraltete und zeitlich abweichende Quellen, Beobachtungszeiten und konfigurierte Ziele. Länderranglisten zeigen die Unsicherheit durch ausgelassene Quellzeilen. Pfeile behalten ihren empfangenden Knoten, und fehlende Knotenpositionen werden nicht erfunden. Die kombinierte Rate bleibt unbeobachtet, wenn einer beitragenden Quelle ein lückenloses Intervall fehlt. Einen Knoten wählen, um seine aufbewahrten Sekunden und die Pfadranglisten der laufenden Minute zu untersuchen. Entfernte Abfragen nutzen die authentifizierte Peer-Verbindung; eine Antwort muss in eine WebSocket-Nachricht von 4 KiB passen, daher halbiert die bedienende Konsole die Zeilen einer Seite, die nicht passt. Fehlende Peers bleiben nicht verfügbar, und Verlaufscursor können keinen Neustart eines Knotens überqueren. Der Minutenverlauf behält seine eigene Knotenauswahl und Zeilen pro Knoten. Navigation und mobiler Kopfbereich nennen den bedienenden Konsolenknoten unabhängig von der gewählten Quelle des Datenverkehrs. Lokale Befehle bleiben pro Knoten.

Die Traffic-Kacheln öffnen mit den letzten 24 Stunden aufbewahrter Datensätze abgeschlossener Minuten für die gewählten Knoten. Die Zeitraumauswahl bietet außerdem eine Stunde, sieben oder neunzig Tage und die Live-Summen seit dem Start. Abweichungen gegenüber gestern erfordern vollständige passende Abdeckung von jedem gewählten Knoten. Globus und Sparklines beschreiben weiterhin ihr eigenes Live-Fenster von 60 Sekunden. Eine aufbewahrte Auswertung wird eine Minute nach ihrem Abschluss aufgefrischt und hält ihre vorigen Werte und deren Alter bis zur Ersetzung sichtbar; ein Wechsel der Quelle oder des Zeitraums verwirft den alten Umfang. Fehlender Verlauf wird nie zu null.

Compare retained traffic zeigt zwei abgeschlossene Minutenfenster nebeneinander: dasselbe Fenster gestern, den vorhergehenden Zeitraum oder einen zweiten Knoten. Steuerelemente für Dauer und Endversatz frieren beide UTC-Bereiche ein. Jede Aktion liest höchstens sechzehn kompakte Seiten pro Seite des Vergleichs, mit 96 gespeicherten Datensätzen pro Seite. Das deckt ein Fenster von 24 Stunden ohne überlappende Neustarts in einem Durchgang ab; Continue behält für längere Auswertungen dieselben Grenzen bei. Vollständige Abdeckung, unvollständige Abdeckung und fehlender Verlauf bleiben ausdrücklich sichtbar. Prozentuale Raten erfordern vollständige, nicht überlappende Intervalle in gleichen UTC-Fenstern, normiert auf beobachtete Millisekunden; eine Referenz von null mit neuem Datenverkehr zeigt

„New“. Der Hauptzeitraum der Seite Traffic bleibt unabhängig. Das Minutenformat erfasst den historischen Proxy-Modus nicht, daher bleiben Vergleiche von Ursprungsantworten nicht verfügbar, statt Nullen aus dem Forward-Auth-Modus als beobachtete Antworten zu deuten. *Compare retained path rankings* wählt zwei abgeschlossene Fenster oder Knoten. Knoten 0 umfasst alle aufbewahrten Knoten, auch ausgeschiedene Mitglieder. Eine Aktion liest höchstens sechzehn unveränderliche Archive, prüft ihre Identitäten und Prüfsummen und führt jeden Zähler zusammen, bevor sie Anzeigezeilen auswählt. Archive werden nach dem Fenster von 60 Sekunden für verspätete Stichproben versiegelt, sodass die neueste abgeschlossene Minute noch ausstehen kann. Continue bewahrt die ursprünglichen Fenster und den Cursor. Die Grenzen beschreiben stichprobenartig erfasste Pfadpräfixe, keine exakten Anfragesummen. Die Seite zeigt unvollständige Auswertungen, Abschneidungen, Aufbewahrungsgrenzen und gemeldete Verluste in der Warteschlange; fehlende Archive können weder ausbleibenden Datenverkehr noch vollständige Abdeckung belegen.

Die Seite Security öffnet mit den letzten 24 Stunden und verwendet sechzig gleiche Zeitintervalle über den gewählten Zeitraum. Ihre Aggregate laufen auf dem Speicherthread unter einer Arbeitsgrenze. Unter SQLite ist die Grenze `--console-query-steps` Schritte der virtuellen Maschine (100.000 bis 50.000.000, Standard vier Millionen). Sie wird zusammen mit `--console` gesetzt; ungültige oder wiederholte Werte werden abgewiesen. Unter PostgreSQL ist die Grenze ein Statement-Timeout von 2.000 ms, das in der Transaktion der Leseoperation gesetzt wird (SID 0013); eine Änderung von `--console-query-steps` ändert es nicht. Erschöpfte Abfragen bitten um einen engeren Zeitraum. Schrittzahlen begrenzen die Arbeit einer Abfrage, nicht die verstrichene Zeit. Diagramme teilen sich eine Skala; die Trend-Werte aufklappen, um die UTC-Intervallanfänge und die exakten gruppierten Zählungen zu sehen. Dies sind aufbewahrte Befunde, daher darf fehlende Vorfallsabdeckung nicht als Abwesenheit von Angriffen gedeutet werden.

Wandanzeigen verwenden Kiosk-Sitzungen. Unter Account benennt ein Betreiber die Anzeige und wählt Create display code. Die Anzeige fügt den Einmalcode innerhalb von zehn Minuten in das Anmeldeformular ein; ihre Sitzung ist schreibgeschützt, auf Statistiken beschränkt und läuft innerhalb von zwölf Stunden ab. Die Kontoseite löscht den angezeigten Code beim Navigieren oder wenn sie verborgen wird. Das Verbergen widerruft eine unbenutzte Berechtigung nicht; die Frist für den Austausch gilt weiterhin. Codes gehören nie in URLs. Traffic und Security teilen sich die Anzeige: Security zeigt aggregierte Modultrends und Anfrageergebnisse ohne Vorfallsadressen oder Nutzlastbelege. Automatisches Durchschalten ist optional, anfangs aus und wird bei reduzierter Bewegung, veralteten Daten oder Pause ausgesetzt.

Benachrichtigungsziele (signierte Webhooks und Syslog für Ablehnungsspitzen, Sperren, unerreichbare Mitglieder und Leader-Wechsel) liegen unter Settings; Webhook-Geheimnisse brauchen `--console-key-file`. Die Art Leader-Wechsel bleibt definiert, damit gespeicherte Ziele ihre Bedeutung behalten, aber kein Knoten löst sie aus, weil es keine Leader-Wahl gibt. Jedes Ziel hat seine eigene Abklingzeit und ein Budget von drei Zustellversuchen. Audit-Datensätze der Zustellung zeigen die Ergebnisse; der Verlauf abgeschlossener Warteschlangeneinträge ist auf sieben Tage und 4.096 Ereignisse begrenzt. Webhooks enthalten einen stabilen `Idempotency-Key`, damit Empfänger wiederholte Wirkungen nach unsicherem Netzwerkabschluss unterdrücken können. Webhooks werden über HTTPS gesendet, oder über reines HTTP nur an eine literale Loopback-Adresse, und der Antwortbody wird bis zu 64 KiB

gelesen. Die Wahl zwischen UDP und TCP für Syslog gilt nur für ausgehende Benachrichtigungen, unabhängig vom geschützten Web-Datenverkehr. Manuelle Tests erfassen Absicht und Abschluss im Audit und frischen das Ergebnis des Ziels auf; ein unbestätigter Audit-Abschluss wird ausdrücklich angezeigt, bevor ein Betreiber es erneut versucht.

Settings steuert auch die Aufbewahrung: ein bis 90 Tage für den Minutenverlauf, ein bis sieben für Ranglisten, ein bis 30 für Vorfälle und ein bis 365 für das Audit. Die oberen Werte sind die Standardwerte; Ranglisten behalten ihr Kontingent von 512 MiB. Das Speichern eines Aufbewahrungswerts erfordert eine Bestätigung, weil die Bereinigung ältere Datensätze in begrenzten Durchgängen endgültig entfernt. Ein späteres Erhöhen des Werts stellt gelöschten Verlauf nicht wieder her. Veraltete Änderungen werden abgewiesen, und erfolgreiche Änderungen erscheinen im Audit.

Antwortseiten (Challenge, abgelehnt, Ratenlimit erreicht, gesperrt, überlastet) lassen sich unter Settings als begrenztes HTML mit festen Platzhaltern bearbeiten; Entwürfe werden in einem Tab mit Sandbox als Vorschau gezeigt, und gespeicherte Seiten werden ab dem nächsten Richtlinienschnappschuss ausgeliefert.

Richtlinien-Workflows auf der Seite Policies: verwaltete Regeln neu ordnen, einen Entwurf gegen aufbewahrte Prüfbefunde abspielen, IP-Gruppen und Ländersperren verwalten, die an die aktive GeoIP-Generation gebunden sind, und den verwalteten Satz exportieren oder atomar importieren (auch `sibuna console policies export` und `sibuna console policies import --file <set.json>`). Ein Import ist eine Anfrage mit 1 bis 128 Dokumenten von je höchstens 4 KiB und insgesamt höchstens 256 KiB; der Hinweis der Seite nennt den Import über die Kommandozeile für Sätze über etwa 96 KiB. Angewandte Regeln zeigen die heute erfassten Treffer, stündliche Sparklines und eine barrierefreie Tabelle. Ein Treffer ist eine erfolgreiche Auswertung eines deklarativen Matchers: Passende WEIGH-Regeln und der erste terminale Treffer zählen; eine frühere Prüf- oder Reputationsentscheidung kann die Auswertung verhindern. Private Tests erhöhen diese Zähler nicht. *Compare rule hits* friert zwei abgeschlossene Zeiträume für einen erfassten Knoten ein. Der Revisionsverlauf bietet *Compare hits around this edit*; dabei wird die Minute der Änderung ausgeschlossen, und es werden gleiche verfügbare Zeiträume bis zur gewählten Dauer verwendet. Optionale Filter nach angewandter Revision halten fremde Generationen aus dem Vergleich heraus. Start, Umstellung, fehlende Schreibvorgänge und Aufbewahrung hinterlassen sichtbare Abdeckungslücken; Prozentangaben erfordern vollständige Abdeckung auf beiden Seiten. Regelbeobachtungen und stündliche/tägliche Zusammenfassungen folgen der Einstellung für die Minutenaufbewahrung. Eine beobachtete Änderung rund um eine Bearbeitung ist ein Vergleich, kein Beleg dafür, dass die Bearbeitung die Änderung des Datenverkehrs verursacht hat.

Der Import einer späteren GeoIP-Generation frischt vorhandene, aus Ländern abgeleitete Reputationszeilen nicht automatisch auf. Die Vorschau der Länderaktion ansehen, um hinzugefügte, behaltene und entfernte Präfixe zu vergleichen, und vor dem Anwenden durch den geprüften Diff blättern. Die Ersetzung entfernt veraltete Zeilen, die diesem Land gehören, und bewahrt unabhängig verwaltete Präfixe. Eine geänderte Generation oder Richtlinienrevision erfordert eine neue Vorschau, und überlappende unabhängige Änderungen werden abgewiesen.

Die Seite Events enthält aufbewahrte WAF-Befunde und Honeypot-Vorfälle; sie ist kein vollständiges Zugriffsprotokoll. Mit geladenem GeoIP erfasst der Speicherthread Land und Generation jedes Vorfalls in der Transaktion des Vorfallsstapels. Das ist eine Zuordnung

zum Zeitpunkt der Speicherung, keine rekonstruierte Position zum Zeitpunkt der Anfrage. Spätere Importe lassen erfasste Zuordnungen unverändert. Der Länderfilter akzeptiert einen großgeschriebenen Code aus zwei Buchstaben, `unknown` für eine Adresse, die in der geladenen Generation fehlt, oder `not_recorded` für einen Vorfall ohne Zuordnungsdaten. Quellgruppen melden gemischte Länder oder Abdeckung ausdrücklich. Die Aktion *View events* des Globus behält den gewählten Knoten bei und öffnet die aufbewahrten Vorfälle des Landes für die letzte Stunde. Die Ansicht *Security* der Seite Statistics kombiniert Live-Raten von Ratenlimits, Challenges und Sperren mit aufbewahrten Prüf- und Honeypot-Befunden über eine Stunde, einen Tag, sieben Tage oder dreißig Tage. Links zu Kategorie, Quelle und Pfad öffnen den normalen Vorfalls-Workflow mit demselben Knoten und einer festen Zeitgrenze. Das Anwenden von Vorfallsfiltern beginnt einen neuen Zeitraum. Befunde schließen Audit-Datensätze ein und sind von den Summen blockierter Anfragen verschieden; fehlende Zuordnungen zu Reputation und Regeltreffern werden als nicht erfasst angezeigt. Ereignis- und Audit-Filter halten Beschriftungen bei ihren Steuerelementen, richten die Aktion Apply getrennt aus und passen ihre Spalten an die verfügbare Inhaltsbreite an. Ein authentifizierter WebSocket überdauert die Navigation und überträgt Statistiken, Vorfallszusammenfassungen, Knotenstatus, Richtlinienrevisionen, Challenges und Audit-Zusammenfassungen. Neue Vorfalls- und Audit-Datensätze warten hinter *Load latest records*, damit die Tabelle beim Lesen an ihrem Platz bleibt. Richtlinienaktualisierungen zeigen die aktuellen festgeschriebenen und angewandten Revisionen, ohne einen offenen Entwurf zu ersetzen. Flusszähler werden live aktualisiert; eine gewählte, vom Standard abweichende Partition der Challenge-Zeiten ausdrücklich auffrischen. Historische Abfragen, Detailabrufe und Änderungen bleiben HTTP-Anfragen.

Seitenfragmente lassen sich als Lesezeichen speichern; Zurück und Vorwärts öffnen authentifizierte Seiten erneut. Die Seitenleiste merkt sich in diesem Browser außerdem die gewählten Einstellungen für Theme und Abstände; Standard-Theme ist System. SID 0007 hat den Status Committed. Die Konsole besteht ihre Golden Fixtures und ihre Browser-Workflows in Headless Chromium (`cargo xtask browser console`, SID 0010, Kriterium 5). Ihre End-to-End-Testsuite ist `cmd/sibuna/tests/console_*.rs`; deren PostgreSQL-Clusterszenarien laufen, wenn `SIBUNA_TEST_DATABASE_URL` gesetzt ist. SID 0010 hält fest, dass die 16 Produkt-Arbeitslasten mit einkompilierter Konsole, ob deaktiviert, nur mit Speicherung oder aktiviert und im Leerlauf, innerhalb von etwa 1 % des ohne Konsole gebauten Daemons bleiben, also im Rauschen. Das formale Isolationskriterium von SID 0007, mit acht aktiven Dashboards unter Last, ist noch nicht gemessen (Teil VIII). Die vollständige WebAssembly-Anwendung hat eine unkomprimierte Obergrenze von 768 KiB, die das Build-Skript des Daemons durchsetzt, indem es den Build scheitern lässt, und eine Prüfschwelle von 640 KiB. In 1.0.0-beta.3 misst das Modul 724.330 Bytes (272.568 mit gzip) und liegt damit über der Prüfschwelle, was SID 0007 zur Überprüfung festhält. Diese Projektgrenzen lassen Raum für die Workflows der Konsole; sie ersetzen keine Messungen von Ladezeit und Reaktionsfähigkeit im Browser. Der lineare Speicher des Moduls wächst bei Bedarf bis zu einem deklarierten Maximum von 16 MiB.

Einsatztopologien

Lernziele Den autonomen Reverse Proxy und den Forward-Auth-Validator hinter Nginx oder Caddy einsetzen.

Reverse Proxy

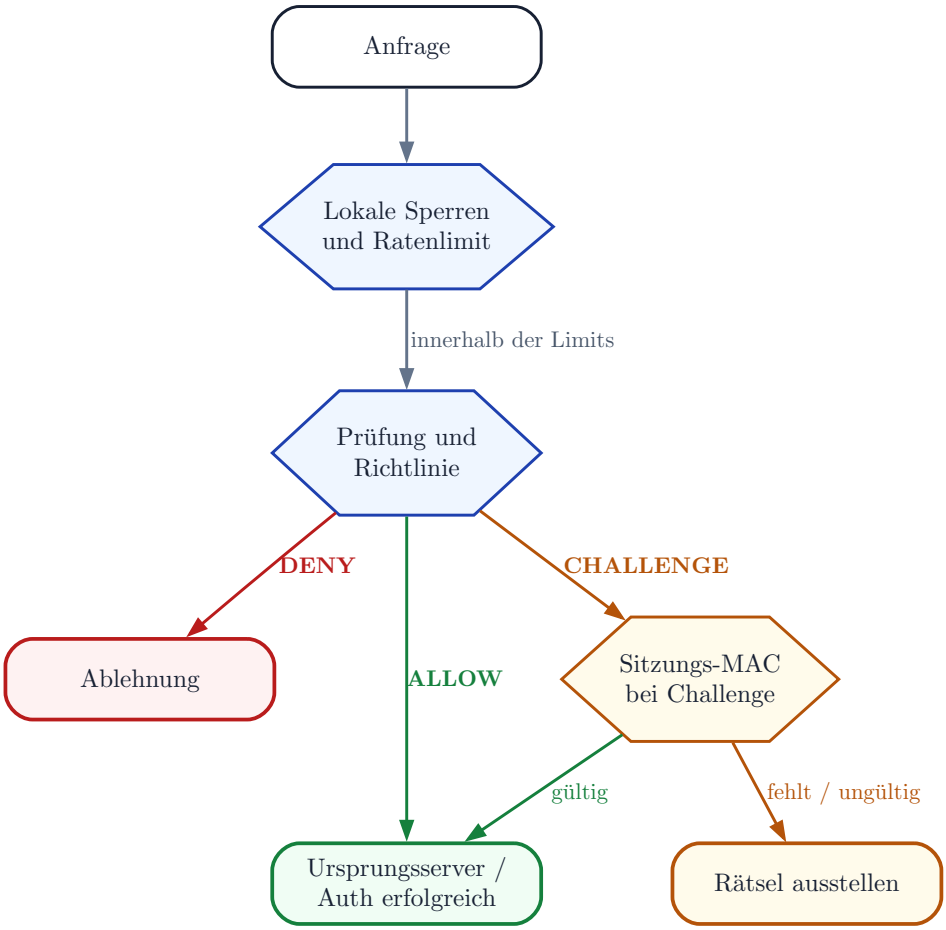


Abb. 24 Anfrage-Routing in der Betriebsform Shield

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

Zugelassene Anfragen erreichen den Ursprungsserver mit den Headern `X-Forwarded-For`, `X-Forwarded-Proto`, `X-Real-IP`, `X-Sibuna-Status` und `X-Sibuna-Rule`. Hop-by-Hop-Header, die Felder, die Sibuna selbst schreibt, und die alternativen Weiterleitungsfelder (`Forwarded`, `X-Forwarded-Host`, `X-Forwarded-Port`) werden entfernt, auch wenn ein Client ihre Namen mit Unterstrichen schreibt (`X_Forwarded_For`). `X-Real-IP` ist der Client, nach dem die Anfrage

beurteilt wurde. `X-Forwarded-For` ist allein der Socket-Peer, nie ein vom Client geschriebener Eintrag; wenn der weitergeleiteten Identität vertraut wird, ist es die empfangene Kette, gefolgt vom Socket-Peer. Ursprungsserver müssen den Client daher aus `X-Real-IP` oder von rechts aus `X-Forwarded-For` lesen, nie aus dessen linkestem Eintrag. Der Pfad, den der Ursprungsserver erhält, ist die Normalform, die die Richtlinie beurteilt hat (SID 0009); ein Pfad ohne Normalform wird mit `400` beantwortet.

WebSocket-Upgrades unter HTTP/1.1 durchlaufen Zulassung und Richtlinienprüfung, bevor der Handshake des Ursprungsservers angenommen wird. Die Aushandlung von Subprotokollen und Erweiterungen bleibt Ende-zu-Ende. Zwei Threads leiten Bytes weiter, einer pro Richtung, jeder mit einem festen Puffer, einschließlich vorab gelesener Bytes und halbseitiger Schließungen; eine per Upgrade umgestellte Verbindung hält daher zwei Threads, und die Zahl der Verbindungsthreads kann das Doppelte von `--max-connections` erreichen, wenn jede Verbindung umgestellt ist. Umgestellte Sockets bleiben im normalen Verbindungskontingent und im Shutdown-Register, verwenden aber `--websocket-idle-timeout` (standardmäßig 300 Sekunden) unabhängig von der HTTP-Leerlauffrist. Datenverkehr in einer der beiden Richtungen erneuert diese Grenze, aber ein Peer, der während der ganzen Periode kein Byte annimmt, beendet die Weiterleitung, wie beschäftigt die andere Richtung auch ist. Frames nach dem Handshake sind keine Eingaben der WAF-Prüfung. HTTPS/WSS nutzt einen TLS-terminierenden Eingangsproxy vor dem privaten HTTP/1.1-Listener von Sibuna; Sibuna terminiert Browser-TLS nicht selbst.

Der Eingangsproxy kann mit Browsern HTTP/2 aushandeln und HTTP/1.1 an Sibuna weiterleiten. Native HTTP/2-Unterstützung in Sibuna ist auf ein späteres Update verschoben; diese Anordnung bietet keine Ende-zu-Ende-Semantik von HTTP/2 für Anwendungen wie natives gRPC.

Anfragebodys mit Content-Length fließen durch feste Puffer, wobei Bytes und MIME-Header erhalten bleiben. Multipart-Uploads behalten Grenzen, wiederholte Feldnamen, Dateinamen und Teiltypen. Die WAF untersucht höchstens die ersten 8 KiB des Bodys: Multipart-Metadaten und Felder, die keine Dateien sind, bleiben Eingaben der Textprüfung, während Dateinutzlasten undurchsichtig sind. Ein Teil, dessen `filename` leer ist, ist ein gewöhnliches Feld. Ein Body eines erkannten binären MIME-Haupttyps (Bilder außer SVG, Audio, Video, PDF, ZIP, gzip, 7z, protobuf und octet-stream) ist nur dann undurchsichtig, wenn auch sein Präfix binär aussieht, das heißt, wenn mindestens eines von sechzehn Bytes ein Steuerbyte außer Tabulator, Zeilenvorschub, vertikalem Tabulator, Seitenvorschub und Wagenrücklauf ist; andernfalls wird er als Text geprüft (SID 0004). JSON, XML, SVG, URL-codierte Formulare und unbekannte Typen behalten die Textprüfung. Mehrdeutige oder fehlerhafte MIME-Metadaten fallen auf die Textprüfung zurück. Das Multipart-Parsing ist innerhalb desselben Präfixes auf 32 Teile und 2 KiB Header pro Teil begrenzt und akzeptiert das Transport-Padding nach RFC 2046 nach einem Trennzeichen.

Das ist Upload-Kompatibilität, keine Dateivalidierung und keine Malware-Prüfung. Das Backend muss akzeptierte Medientypen selbst durchsetzen, statt der Content-Type-Angabe eines Clients zu vertrauen. Felder nach dem Prüfpräfix, auch solche nach einer großen hochgeladenen Datei, werden nicht geprüft. Komprimierte Nutzlasten werden für die Prüfung nicht dekomprimiert. Uploads durchlaufen dennoch Zulassung sowie Pfad-, Query- und Header-Prüfungen und unterliegen weiterhin den Verbindungsfristen. Chunked-Anfragebodys werden

vor der Prüfung im Verbindungspuffer dekodiert (SID 0009). Ein Body, der innerhalb des Puffers endet, erreicht das Backend mit Content-Length. Ein längerer wird von Sibuna neu in Chunks zerlegt, ein Chunk pro Leseoperation, sodass das Backend nie die Chunk-Größen, Erweiterungen oder Trailer des Clients sieht. Eine Chunk-Zeile mit einem einzelnen CR oder LF, Leerraum um die Größe, einer fehlerhaften Erweiterung oder mehr als 4 KiB wird mit 400 abgewiesen. Ebenso ein Trailer-Abschnitt über 16 KiB. `Transfer-Encoding` mit einer anderen Kodierung erhält 501. Backends, die keine Chunked-Anfragen parsen können, erhalten für Bodys unter etwa 44 KiB weiterhin Content-Length. Eine Verbindung zum Ursprungsserver, die einen Anfragebody übertragen hat, wird nach ihrem Austausch geschlossen und kehrt nie in den Pool zurück. Clients, die auf `100-continue` warten, erhalten es lokal, bevor sie den Body senden; nicht unterstützte Erwartungen erhalten 417. Der Expect-Header wird vor der Weiterleitung an das Backend verbraucht.

Forward-Auth hinter einem Eingangsproxy

Mit `--mode forward_auth` beantwortet der Daemon die Unteranfrage des Eingangsproxys mit `200` (plus den Audit-Headern `X-Sibuna-Status`, `X-Sibuna-Rule` und `X-Sibuna-Rule-Hash`), `401` für eine Challenge, `403` für eine Ablehnung oder `429` bei erreichtem Ratenlimit. Der Eingangsproxy muss die Client-Adresse und die ursprüngliche URL weiterleiten; der Forward-Auth-Modus vertraut ihnen standardmäßig. Dieser Listener wird privat gebunden, damit sich nur der Eingangsproxy verbinden kann. `X-Forwarded-Uri` (Caddy) oder `X-Original-URI` (Nginx) und `X-Forwarded-Method` stellen die Anwendungsanfrage für die Richtlinienauswertung wieder her; der ursprüngliche Pfad wird in Normalform gebracht, so wie der Pfad einer Anfragezeile. Doppelte oder widersprüchliche Felder für die ursprüngliche URL und eine ursprüngliche URI, die nicht in Origin-Form vorliegt, werden mit `400` abgewiesen. Interne Routen des Daemons verwenden immer die tatsächliche Anfrage-URI. Forward-Auth prüft keinen Body, den der Eingangsproxy weglässt.

```
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
    }
}
```

Der Namensraum `/__sibuna/*` (Zwischenseite, Challenge, Verifikation, Assets des Lösers) muss in beiden Rezepten den Daemon direkt erreichen. Die Fehlerseite von nginx übergibt

die ursprüngliche URI und Methode, damit die Zwischenseite die Anforderung trägt, die die Autorisierung entschieden hat; ohne sie funktioniert die Seite trotzdem, und die Ausgabe fällt darauf zurück, die vom Browser gemeldete URL auszuwerten. Caddy liefert die Metadaten zu ursprünglicher URI und Methode selbst. Die exklusiven `handle`-Blöcke stellen sicher, dass Challenge- und Verifikationsrouten nicht in die Forward-Auth-Vorprüfung gelangen. Nginx benötigt für WebSockets der Anwendung die oben gezeigten ausdrücklichen Upgrade/Connection-Header. Der Fehlerhandler von Nginx stellt die interne Challenge-Route dar, ohne eine zweite Autorisierungsentscheidung zu treffen oder den Upload-Body zu verbrauchen. Sein Auth-Modul akzeptiert direkt nur 2xx, 401 und 403, daher übersetzt das Beispiel einen Auth-Fehler wegen Ratenlimit zurück in 429 mit Retry-After; andere Autorisierungsfehler bleiben mit 503 geschlossen. Nginx liefert seinen eigenen 403-Body, während Caddy die Ablehnungsseite von Sibuna weiterleitet. Beide Rezepte ersetzen eingehende Audit-Header von Sibuna durch die tatsächliche Autorisierungsantwort. Normale TLS-Zertifikate und die Upload- und Fristgrenzen der Anwendung werden am Eingangsproxy konfiguriert; die Beispiele zeigen die Routing-Logik. Das Repository testet diese Rezepte noch nicht gegen laufende Eingangsproxys; die Regeln für weitergeleitete Metadaten, auf die sie sich stützen, sind durch Unit-Tests in `sibuna-http` abgedeckt.

JA4-Fingerabdrücke vom Eingangsproxy

Ein JA4-Fingerabdruck fasst zusammen, wie ein Client eine TLS-Verbindung öffnet. Er wird aus der ersten Handshake-Nachricht des Clients berechnet, dem ClientHello. Die TLS-Bibliothek des Clients legt diese Nachricht fest, bevor ein einziges HTTP-Byte gesendet wird, daher kann ein Client seinen Fingerabdruck nicht aus seinen HTTP-Anfragen heraus ändern. Browser, Bibliotheken und Bots unterscheiden sich darin. Das macht JA4 zu einem stärkeren Signal als ein `User-Agent`-Wert, den jeder Client nach Belieben schreiben kann (SID 0014). JA4 wurde von FoxIO definiert. Ein Fingerabdruck hat 36 Zeichen in drei Teilen, die durch `_` verbunden sind:

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

a  Protokoll (t TCP, q QUIC, d DTLS), TLS-Version (13 ist TLS 1.3),
   d mit SNI oder i ohne, Anzahl der Cipher Suites (zweistellig),
   Anzahl der Erweiterungen (zweistellig), ALPN-Code (h2 ist HTTP/2)
b  12-stelliger Hexadezimal-Hash (Kleinbuchstaben) der sortierten Cipher Suites
c  12-stelliger Hexadezimal-Hash (Kleinbuchstaben) der sortierten Erweiterungen
   und Signaturalgorithmen
```

Nur die Komponente, die TLS terminiert, sieht das ClientHello, daher kann nur sie JA4 berechnen. Sibuna terminiert kein TLS: Es bedient einfaches HTTP/1.1 hinter einem Eingangsproxy, und der Eingangsproxy spricht TLS, HTTP/2 und HTTP/3 mit den Browsern. Mehrere Eingangsproxys berechnen JA4 bereits. Wir lassen daher den Eingangsproxy den Fingerabdruck berechnen und in einem Anfrage-Header senden, und Sibuna liest ihn von dort. `--ja4-header <name>` benennt diesen Header. Die Option ist standardmäßig aus und hat keinen Standardnamen; wir nennen stets den Header, den der Eingangsproxy verwendet. Sibuna speichert den Namen in Kleinbuchstaben und vergleicht ihn ohne Beachtung der Groß-/

Kleinschreibung. Es weist die Namen von Feldern ab, die es selbst schreibt oder entfernt (`X-Forwarded-*`, `X-Real-IP`, `X-Sibuna-*` und `Forwarded`, auch mit `_` statt `-`). Ist die Option gesetzt, gibt das Start-Banner eine Zeile wie `JA4: from cf-ja4 (trusted)` aus.
Der Header wird nur gelesen, wenn weitergeleiteten Feldern vertraut wird. Das ist dieselbe Entscheidung, die für `X-Forwarded-For` gilt: `--trust-forwarded` oder der Forward-Auth-Modus ohne `--no-trust-forwarded`. Von einem nicht vertrauenswürdigen Peer wird der Header ignoriert. Mit Vertrauen trägt eine Anfrage einen Fingerabdruck, wenn genau ein Feld dieses Namens vorhanden ist und sein Wert die JA4-Form hat. Ein fehlerhafter Wert oder ein wiederholtes Feld lässt die Anfrage ohne Fingerabdruck. Zwei Zähler beobachten dies: `sibuna_ja4_present_total` zählt Anfragen mit einem Fingerabdruck, und `sibuna_ja4_invalid_total` zählt Anfragen, deren Header fehlerhaft oder wiederholt war. Eine Anfrage von einem nicht vertrauenswürdigen Peer erhöht keinen der beiden Zähler.
Im Reverse-Proxy-Modus sieht der Ursprungsserver den Header nur, wenn Sibuna ihm vertraut. Ohne Vertrauen entfernt Sibuna jedes Feld dieses Namens aus der Anfrage, die es an den Ursprungsserver sendet, auch wenn ein Client den Namen mit `_` statt `-` schreibt. Ein Client kann dem Ursprungsserver daher unter diesem Namen keinen gefälschten Fingerabdruck unterschieben. Mit Vertrauen wird das Feld unverändert in seiner genauen Schreibweise weitergereicht, und ein Alias mit `_` wird dennoch entfernt. Ohne `--ja4-header` ist ein Feld dieses Namens ein gewöhnliches Feld.

> **Der Eingangsproxy muss den Header kontrollieren** Ein Fingerabdruck ist nur so vertrauenswürdig wie der Eingangsproxy, der ihn geschrieben hat. Sibuna kann einen Wert, den der Eingangsproxy berechnet hat, nicht von einem Wert unterscheiden, den der Client gesendet hat. Der Eingangsproxy muss daher jede Kopie des Headers, die der Client mitliefert, überschreiben oder entfernen, und zwar auf jeder Route, die Sibuna erreicht. Sibunas Listener darf Verbindungen nur vom Eingangsproxy annehmen, wie bei `X-Forwarded-For`.

Cloudflare. Die verwaltete Transformation „Add bot protection headers“ fügt der an den Ursprungsserver gesendeten Anfrage `cf-ja4` hinzu, neben `cf-bot-score`, `cf-verified-bot` und `cf-ja3-hash`. Sie erfordert einen Enterprise-Tarif mit Bot Management. Ein Eingangsproxy zwischen Cloudflare und Sibuna muss das Feld unverändert weiterreichen. Wir starten Sibuna mit:

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

Der Eingangsproxy vor Sibuna darf Verbindungen nur von den Adressen von Cloudflare annehmen; andernfalls kann ein Client Cloudflare umgehen und sein eigenes `cf-ja4` senden.
nginx. FoxIOs `ja4-nginx-module` berechnet JA4 innerhalb von nginx und stellt es als `$http_ssl_ja4` bereit. Es braucht nginx 1.26 oder neuer mit dem Patch des Moduls zum Erfassen des ClientHello, gebaut mit `--with-http_ssl_module`. `proxy_set_header` ersetzt ein Feld des Clients mit demselben Namen, daher setzt jede Location, die Sibuna erreicht, den Header selbst:

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

# Forward-Auth: Sibuna läuft mit --mode forward_auth --ja4-header x-ja4
location = /__sibuna_auth {
    internal;
    proxy_pass http://127.0.0.1:8080/;
    proxy_set_header X-JA4 $http_ssl_ja4;
    # ... die übrigen Felder des Forward-Auth-Rezepts oben
}
```

Im Forward-Auth-Rezept gehört dieselbe Zeile `proxy_set_header X-JA4 $http_ssl_ja4;` auch in `@sibuna_challenge` und in `location /__sibuna/`. Eine Location, die sie weglässt, reicht das eigene `X-JA4` des Clients an Sibuna weiter, das ihm vertraut.

Apache Traffic Server. Das Plugin `ja4_fingerprint` wird durch die Zeile `ja4_fingerprint.so` in `plugin.config` aktiviert. Es hat keine Optionen. Es fügt den Anfragen jeder TLS-Verbindung ein Feld `ja4` mit dem Fingerabdruck und ein Feld `x-ja4-via` mit dem Namen des Proxys hinzu. Wir starten Sibuna mit `--ja4-header ja4 --trust-forwarded`. Die Dokumentation des Plugins sagt nicht, ob es ein vom Client gesendetes Feld `ja4` ersetzt. Wir müssen dies für unsere Version feststellen oder das Feld des Clients entfernen, bevor das Plugin sein eigenes hinzufügt, bevor wir dem Wert vertrauen.

Eine Regel gleicht einen Fingerabdruck mit dem Feld `ja4` ab (Teil X, Richtlinienschema). Der Wert ist entweder ein ganzer Fingerabdruck, der gleich sein muss, oder ein Präfix gefolgt von *, mit dem der Fingerabdruck beginnen muss. `"*"` allein passt auf jeden Fingerabdruck. Eine Anfrage ohne Fingerabdruck passt nie auf eine `ja4`-Regel. Der Lader weist ein Muster, auf das kein Fingerabdruck passen könnte, etwa `T13D*` in Großbuchstaben, mit `INVALID POLICY JA4` ab. Der Abgleich vergleicht Bytes und allokiert nichts.

```
{ "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" }
] }
```

Die erste Regel lehnt genau einen Fingerabdruck ab; der Wert ist ein Beispiel, kein bekannter Scanner. Die nächsten beiden fügen der Bewertung der Anfrage Gewicht hinzu, statt zu entscheiden: ein Client mit TLS 1.3, der keinen Servernamen gesendet hat, und ein Headless-Browser, erkennbar an seinem `User-Agent`. Wenn keine spätere Regel zuerst entscheidet, erreicht eine Anfrage, die beide Signale zeigt, die Bewertung 45 und damit die Standardschwelle für Ablehnung von 40, während eine Anfrage, die eines zeigt, eine Challenge erhält. Die letzte Regel stellt per Präfix jedem Client eine Challenge, der auf dem Checkout-Pfad TLS 1.2 über

TCP ausgehandelt hat. Sie stellt eine Challenge, statt abzulehnen, weil viele legitime ältere Clients noch TLS 1.2 verwenden. Regeln sollten JA4 auf diese Weise mit anderen Signalen kombinieren. Ein JA4-Wert benennt eine TLS-Bibliothek und ihre Konfiguration, keine Person und kein Gerät. Viele Clients teilen sich eine Bibliothek, und spezialisierte Bibliotheken können das ClientHello eines Browsers nachahmen. JA4 erhöht die Kosten der Automatisierung; es beweist keine Identität. Aus demselben Grund führt Sibuna Sperren und Reputation nach Adresse und nie nach Fingerabdruck.

Vorfälle (WAF-Ablehnungen, Honeypot-Treffer und Audit-Befunde) zeichnen den Fingerabdruck ihrer Anfrage auf, wenn sie einen trug, in der Spalte `security_incidents.ja4`. Die Konsole zeigt ihn in den Belegen des Vorfalls als „JA4 fingerprint“, mit dem Wert oder „Not recorded“. Die Filter der Seite Events akzeptieren einen Fingerabdruck oder ein Präfix. Der Editor für verwaltete Regeln hat ein JA4-Feld, gespeichert in `policies.ja4_pattern`, und das Testformular für Richtlinien nimmt einen Fingerabdruck an.

Deklarative Richtlinie

> **Lernziele** Eine Richtliniendatei mit Regeln, Gewichten, Schwellen, IP-Regeln und dem WAF-Schalter schreiben.

```
{
  "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" }
  ]
}
```

Der Lader schlägt geschlossen fehl: Ein unbekannter Schlüssel, eine falsch geschriebene Aktion, eine fehlerhafte Adresse oder eine Zahl außerhalb des Bereichs weist die ganze Datei ab, die Diagnose nennt Regel und Feld, und der Daemon startet nicht. Eine Kommandozeile mit einer unbekannten Option oder einem Wert außerhalb des Bereichs wird auf dieselbe Weise abgewiesen. `rules` ersetzt, wenn vorhanden, die eingebaute Tabelle; `ip_rules` speist den Reputations-Trie, der auf Tausende von Präfixen skaliert; `waf: false` wählt die Betriebsform Gate aus der Datei. Zwei weitere Schlüssel gehören zum Schema (Teil X): `inspection` setzt

jede WAF-Kategorie auf `disabled`, `audit` oder `enforce`, und das Objekt `limits` einer Regel gibt ihr ein eigenes Kontingent. Das Feld `ja4` einer Regel gleicht den JA4-Fingerabdruck ab, den ein vertrauenswürdiger Eingangsproxy weitergeleitet hat, exakt oder per Präfix gefolgt von *; es braucht `--ja4-header` (siehe „JA4-Fingerabdrücke vom Eingangsproxy“). Pfadmuster passen auf den Anfragepfad in Normalform (SID 0003), daher muss ein Muster in dieser Form geschrieben sein.

Persistente Speicherung und Cluster

Lernziele SQLite- oder PostgreSQL-Speicher aktivieren, eine dynamische Richtlinie und eine Sperre per SQL hinzufügen, die Vorfallsforensik abfragen, die Kampagnenbildung verstehen und mehrere Knoten auf einer PostgreSQL-Datenbank betreiben.

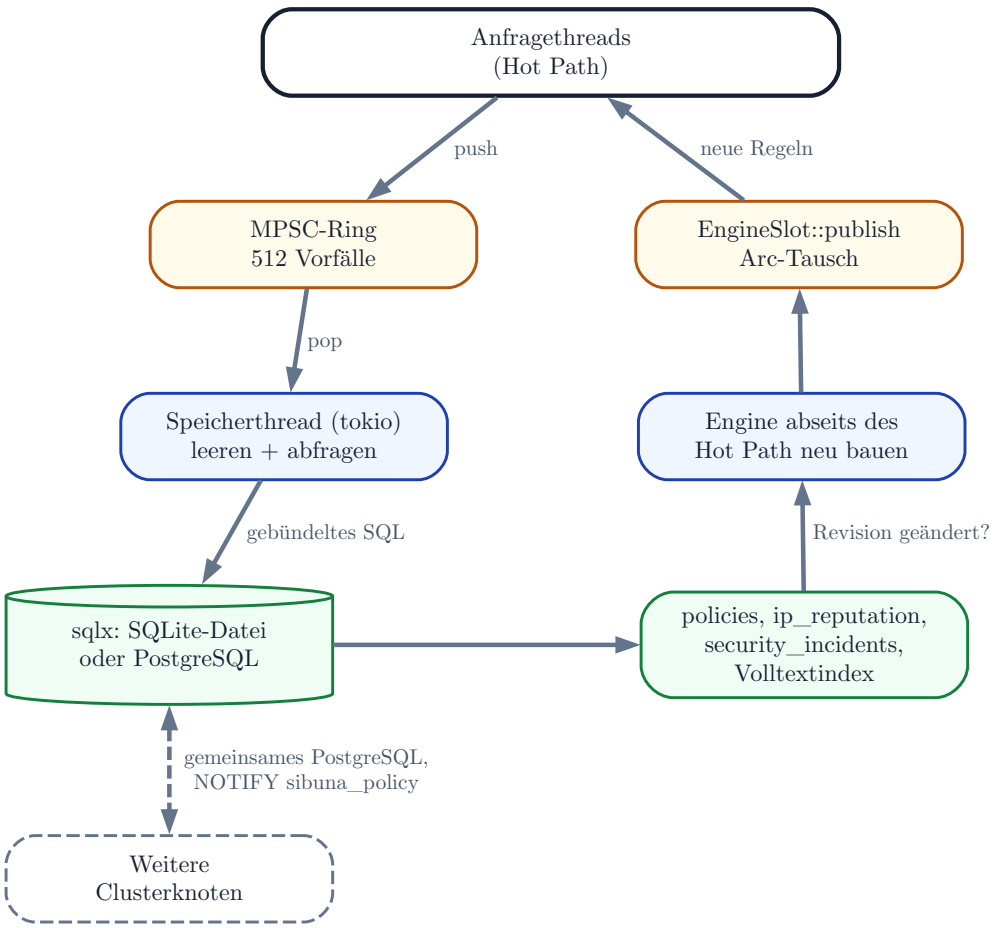


Abb. 25 Die Speicherarchitektur: Worker blockieren nie auf der Datenbank

`--data-dir /var/lib/sibuna` öffnet die SQLite-Datei `/var/lib/sibuna/sibuna.db` (WAL-Modus, `synchronous=NORMAL`, ein Busy-Timeout) und startet den Speicherthread. `--database-url`

(oder `--database-url-file`) verbindet stattdessen mit einer PostgreSQL-Datenbank. Der Speicherthread besitzt eine einfädige tokio-Laufzeit und einen kleinen Verbindungspool; er ist die einzige Stelle, die SQL ausführt. Beim Start, bei jeder Änderung der Richtlinienrevision (ein Schreibvorgang auf `policies`, `ip_reputation` oder `policy_inspection`) und wenn ein Reputationseintrag abläuft, baut der Thread die Engine aus der Richtliniendatei und der Datenbank neu auf und veröffentlicht sie; Anfragen warten nie auf SQL. Ein Neuaufbau liest die Revision vor und nach dem Laden und verwirft seinen Kandidaten, wenn ein Schreiber dazwischen festgeschrieben hat.

Schema

Tabelle 25 Kerntabellen

Tabelle	Zweck und Schlüssel
`policy_revision`	Die Richtlinienrevision, eine Zeile; Trigger auf den drei Richtlinientabellen erhöhen sie.
`policies`	Geordnete dynamische Regeln, mit Schlüssel `id`. Das nullbare `ja4_pattern` enthält das JA4-Kriterium einer Regel, höchstens 37 Bytes.
`policy_inspection`	Der gespeicherte WAF-Modus pro Kategorie, eine Zeile.
`ip_reputation`	Bewertungen, Trefferzahlen und Ablauf, mit Adresse oder Präfix als Schlüssel.
`incident_receipts`	Die dauerhafte Vorfallsquittung jedes Knotens (die nächste Sequenznummer nach seinem letzten festgeschriebenen Stapel), mit dem Knoten als Schlüssel.
`security_incidents`	Ein begrenzter Ereignisdatensatz mit seinem Embedding aus 64 Komponenten (256 Bytes), mit Schlüssel `node_id << 40 | sequence`. Das nullbare `ja4` enthält den JA4-Fingerabdruck der Anfrage, genau 36 Bytes.
`incidents_fts`	Unter SQLite: eine FTS5-Tabelle über die kanonischen Tokens von Pfaden und Nutzlasten. Unter PostgreSQL sind dieselben Tokens die `tsvector`-Spalte `search` von `security_incidents` mit einem GIN-Index.
`sibuna_schema`	Jede angewandte Migration, mit Komponente und Version als Schlüssel.

Das Kernschema ist `crates/storage/src/schema.rs`, einmal pro Dialekt geschrieben. Mit der Konsole fügt die Komponente `console` 33 Tabellen hinzu, deren Namen mit `console_` beginnen (SID 0013; `crates/console-storage/src/schema/`). Migrationen sind pro Komponente nummeriert und in die Binärdatei einkompiliert. Die Kernmigration 2 fügt die beiden JA4-Spalten hinzu; eine vor ihr angelegte Datenbank wird beim Start migriert. Alle ausstehenden Migrationen laufen in einer Transaktion, und ein Knoten weist eine Datenbank ab, deren Version einer Komponente neuer ist, als er kennt. Ein gekürzt abgedrucktes Schema ist kein Upgrade-

Skript. SQLite-Tabellen sind `STRICT`, und `CHECK`-Constraints spiegeln die Rust-Typen, sodass die Datenbank eine Zeile abweist, die der Daemon nicht darstellen könnte. Aufzählungswerte werden in Kleinbuchstaben gespeichert: Ein direkter Schreibvorgang von `'DENY'` wird abgewiesen.

Der Betrieb

Die `sqlite3`-Shell auf `sibuna.db` verwenden (der WAL-Modus erlaubt ihr das Lesen, während der Daemon läuft) oder `psql` auf der Cluster-Datenbank:

```
-- Eine dynamische Regel hinzufügen; jeder Knoten baut beim nächsten Takt neu.
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());

-- Eine Adresse clusterweit für einen Tag sperren.
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());

-- Forensik: Volltextsuche über aufgezeichnete Nutzlasten.
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;
```

Die Anweisungen sind für SQLite geschrieben. Unter PostgreSQL ist `enabled` gleich `TRUE`, die Zeit ist `CAST(extract(epoch FROM now()) AS BIGINT)` statt `unixepoch()`, und die Suche lautet `WHERE search @@ CAST('union' AS tsquery)` auf `security_incidents`. `path` und `offending_payload` halten die rohen Anfragebytes (`BLOB` oder `BYTEA`). Ein direkter Schreibvorgang, der einen Constraint verletzt, wird von der Datenbank abgewiesen und erreicht nie einen Knoten. Unter PostgreSQL sendet der Revisionstrigger außerdem `NOTIFY sibuna_policy`, sodass die anderen Knoten sofort einen Takt ausführen; das Polling bleibt die Garantie.
Bewertungen bei oder unter −50 werden zu `deny`-Präfixen im Trie, Bewertungen bei oder über 50 zu `allow`, und `banned_until` begrenzt die Sperre. Honeypot-Treffer fügen automatisch einen Datensatz mit −100 ein, sodass mit einer gemeinsamen PostgreSQL-Datenbank eine auf einem Knoten ausgelöste Falle die Adresse auf allen Knoten sperrt.

Eine Transaktionsquittung macht Wiederholungen sicher

Der Speicherthread sammelt höchstens 32 ausstehende Datensätze und baut daraus eine SQL-Transaktion. Die Transaktion fügt Vorfälle ein, aktualisiert den Volltextindex und wendet Reputationsänderungen durch den Honeypot an. Jeder Aussteller (Knoten) hat eine monoton wachsende Quittung, seine Zeile in `incident_receipts`. Jede datenverändernde Anweisung ist durch diese Quittung geschützt; die Transaktion erhöht sie zuletzt, und nie rückwärts.

Durchgerechnetes Beispiel: Die Bestätigung geht verloren Angenommen, Aussteller 2 bereitet die Sequenzen 101 bis 132 vor, die bei Quittung 133 enden. Die Datenbank schreibt alle Datensätze und Quittung 133 fest, dann bricht die Verbindung ab, bevor die Bestätigung eintrifft. Der Speicherthread behält den vorbereiteten Stapel (Kennungen, Embeddings und Kampagnen) und versucht ihn erneut. Die Quittung ist bereits 133, daher bewirken die geschützten Schreibvorgänge nichts: Weder Vorfälle noch Honeypot-Trefferzahlen werden verdoppelt. Wurde die ursprüngliche Transaktion zurückgerollt, bleibt die alte Quittung bestehen, und der Wiederholungsversuch wendet alle Schreibvorgänge genau einmal an.

Ein fehlgeschlagenes Festschreiben behält den ausstehenden Stapel im Speicher und erhöht `incident_write_failures`. Der Takt versucht dennoch das Polling der Richtlinie. Eine volle Warteschlange mit 512 Datensätzen zählt `incidents_dropped`; sie blockiert nie eine HTTP-Antwort, die auf die Festplatte wartet. Der ausstehende Stapel bietet Platz für weitere 32 Datensätze. Stirbt der Prozess vor dem Festschreiben, können eingereihte Daten verloren gehen: Dies ist ein begrenzter asynchroner Forensikpfad, keine Nachrichtenwarteschlange, die schon beim Einreihen dauerhaft ist.

Übung 9.1. Die Aktualisierung der Quittung aus der Datentransaktion herausnehmen. Einen Absturzablauf konstruieren, der eine Reputationswirkung verdoppelt, und einen weiteren, der einen Vorfall verliert.

Kampagnenbildung

Jede Vorfallsnutzlast wird mit dem Hashing-Trick über Byte-Trigramme (Ziffern zusammengefasst, Groß- und Kleinschreibung zusammengefasst) als 64-dimensionaler Einheitsvektor eingebettet und als 256 Bytes in `security_incidents.embedding` gespeichert. Der Speicherthread hält die 16.384 neuesten Embeddings aller Knoten im Speicher (4 MiB) und frischt sie bei jedem Takt aus der Datenbank auf. Vor dem Einfügen sucht er die nächstgelegene davon, einschließlich früherer Datensätze desselben Stapels; ist der Cosinus-Abstand höchstens 0.35, schließt sich der neue Vorfall der `campaign_id` dieses Vorfalls an, andernfalls beginnt er eine Kampagne, die nach seiner eigenen Kennung benannt ist. Vorfälle, die älter als das Fenster sind, werden nicht verglichen. Ähnlichkeit ist eine Heuristik: Die Regressionsbeispiele gruppieren verwandte SQL-Nutzlasten, aber das garantiert nicht, dass jedes Paar von Angriffen eine Kampagne teilt. Es läuft kein Modell, und nichts wird trainiert.

Clusterbetrieb

Ein Cluster ist eine Menge von Knoten, die sich eine PostgreSQL-Datenbank teilen. Mit dem Feature `postgres-tls` bauen (`cargo build --release -p sibuna --features postgres-tls`; es schließt `postgres` ein) und jedes Mitglied mit der Datenbank und einer eigenen Knotenkennung starten:

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

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

Jedes Mitglied muss dasselbe Master-Geheimnis und ein eindeutiges `--cluster-node` von 1 bis 8.388.607 verwenden. Die URL-Datei enthält eine `postgres://`-URL und darf nur für ihren Eigentümer lesbar sein; das Banner gibt die URL nie aus. Schreibvorgänge gehen an den PostgreSQL-Primary. Der Speicherthread jedes Knotens sieht die Änderung der festgeschriebenen Revision (per `NOTIFY` oder per Polling) und baut seine Engine neu auf.
Knoten erreichen die gemeinsame Datenbank über das Netz, daher verschlüsseln und verifizieren wir diese Verbindung. Wir geben dem Server ein Zertifikat, das von unserer eigenen Zertifizierungsstelle signiert ist, und nennen diese Stelle in der URL: `postgres://sibuna:...@db.internal:5432/sibuna?sslmode=verify-full&sslrootcert=/etc/sibuna/db-ca.pem`. Auf dem Server akzeptieren wir TCP nur über `hostssl`-Zeilen in `pg_hba.conf`.
Beim Start gibt der Daemon eine Zeile über die Datenbankverbindung aus. Mit `verify-full` meldet er, dass Zertifikat und Hostname verifiziert sind. Mit jedem schwächeren Modus gibt er eine Warnung und einen Hinweis aus. Ohne `sslmode` ist der Modus `prefer`: Die Verbindung ist nur verschlüsselt, wenn der Server TLS anbietet, und der Server wird nicht verifiziert. Ein ohne `postgres-tls` gebauter Daemon weist `require`, `verify-ca` und `verify-full` ab und nennt das fehlende Feature; er fällt nie auf Klartext zurück. Ein abgewiesenes Zertifikat bricht den Start sofort ab; der Hinweis bittet uns, `sslrootcert` zu prüfen und ob der Host der URL ein Name im Zertifikat des Servers ist. `sslrootcert` fügt unsere Zertifizierungsstelle zum Vertrauensspeicher des Systems hinzu; es entfernt nicht die Zertifizierungsstellen des Systems (SID 0011, SID 0012).
Sibuna repliziert keine eigene Datenbank und wählt keinen Leader. Replikation, Failover und Backups sind Sache von PostgreSQL (Streaming-Replikation, Patroni oder ein verwalteter Dienst). Die Knotenansicht der Konsole hat keinen Leader: Sie zeigt die Rolle `single` für einen SQLite-Knoten und `unknown` für einen PostgreSQL-Knoten, und `quorum` bedeutet nur, dass der letzte Datenbank-Roundtrip des Knotens gelungen ist. Ist die Datenbank nicht erreichbar, bedient die zuletzt veröffentlichte Engine weiter, Vorfälle werden im begrenzten Ring eingereiht und mit einem Zähler verworfen, sobald er voll ist, und der Speicherthread versucht es mit Backoff erneut; ein Knoten behält seine letzte Richtlinie, bis er sich wieder verbindet. Ein Zaxonlite-Datenverzeichnis (SID 0005) wird nicht gelesen; Richtlinien werden mit `sibuna console policies export` und `sibuna console policies import` übertragen, und der Forensikverlauf wird nicht migriert.

Challenge-Routing im Cluster

Challenge-Schlüssel sind an `--cluster-node` gebunden; Ausgabe und Verifikation einer Challenge bleiben daher auf demselben Mitglied. Tokens verwenden den gemeinsamen Seed und funktionieren auf anderen Mitgliedern. Lokale Ratenkontingente werden nicht zu globalen Kontingenten, und Prozessneustarts leeren die Verbrauchsmengen. Mit drei Knoten auf einer lokalen PostgreSQL-Datenbank erreicht eine direkt geschriebene Richtlinienzeile alle drei innerhalb eines Takts, eine Honeypot-Sperre auf einem Knoten erreicht die anderen, und Vorfallskennungen kollidieren nie (`crates/storage/tests/multi_node.rs`, SID 0011, Kriterium 2; der Test läuft, wenn `SIBUNA_TEST_DATABASE_URL` gesetzt ist).

Paketierung

Lernziele Ein Release-Paket installieren und prüfen, Sibuna als Dienst, in einem Container oder unter Kubernetes betreiben und die Binärdatei selbst bauen.

Jedes Release auf GitHub enthält für jede Plattform dieselbe ausführbare Datei, gebaut aus dem Release-Tag mit den Standard-Features: SQLite-Speicher und Konsole. PostgreSQL-Cluster brauchen einen eigenen Build (siehe unten).

Plattform	Release-Dateien
Linux x86-64, ARM64	`sibuna-linux-amd64.tar.gz` und `sibuna-linux-arm64.tar.gz` enthalten eine statische ausführbare musl-Datei, die auf jeder Distribution läuft. `sibuna_<version>-1_amd64.deb`, `sibuna_<version>-1_arm64.deb`, `sibuna-<version>-1.x86_64.rpm` und `sibuna-<version>-1.aarch64.rpm` installieren sie mit einem systemd-Dienst; `sibuna-bin-<version>-1-x86_64.pkg.tar.zst` tut dasselbe unter Arch Linux.
macOS arm64, x86-64	`sibuna-macos-arm64.tar.gz`, `sibuna-macos-amd64.tar.gz`, für macOS 12 oder neuer.
Windows x86-64	`sibuna-windows-amd64.zip` enthält `sibuna.exe`, für Windows 10 oder Server 2016 oder neuer.
FreeBSD 15.1, OpenBSD 7.9 (amd64)	`sibuna-<version>-freebsd-15.1-amd64.pkg` und `sibuna-<version>.tgz`.
Homebrew, Kubernetes, Container	`sibuna.rb`, `helm-sibuna-<version>.tgz` und das Image `ghcr.io/insanai/sibuna:<version>` für amd64 und arm64, dessen Digest in `IMAGE-DIGEST.txt` steht.
Quellcode	`sibuna-<version>-source.tar.gz`.

Jedes Archiv und jedes Paket enthält `LICENSE`, `NOTICE`, `LICENSES/` (mit `THIRD-PARTY-RUST.txt`, den Crates in der ausführbaren Datei), `README.md`, `SECURITY.md` und `sibuna.build.json`. Das Manifest verzeichnet die Version, den Commit, das Rust-Target, die Compilerversion, die Features und den SHA-256 der ausführbaren Datei und ihrer beiden WebAssembly-Module. Der Release-Workflow startet jede ausführbare Datei auf ihrer eigenen Plattform, bevor er etwas veröffentlicht, und er installiert, reinstalliert und entfernt die Pakete unter Debian, Fedora, Arch Linux, FreeBSD und OpenBSD.
Die Dateien sind nicht signiert. Wir prüfen einen Download gegen `SHA256SUMS` aus demselben Release, bevor wir ihn installieren:

```
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
```

Unter macOS trägt ein Download aus dem Browser ein Quarantäne-Attribut, das eine unsignierte ausführbare Datei blockiert; `xattr -d com.apple.quarantine sibuna` entfernt es nach der Prüfung.

Der systemd-Dienst

Die Debian-, RPM- und Arch-Pakete installieren `/usr/bin/sibuna`, die Unit `sibuna.service`, das Konto `sibuna` (über `systemd-sysusers`), das Hilfsskript `/usr/lib/sibuna/seed.sh` und die Einstellungsdatei `/etc/sibuna/service.env`. Die Installation startet nichts. Wir tragen den Ursprungsserver, ein IP-Literal, in die Einstellungsdatei ein und starten den Dienst:

```
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
```

Der Dienst lauscht auf `127.0.0.1:8080`, solange `SIBUNA_HOST` und `SIBUNA_PORT` nichts anderes festlegen. `SIBUNA_OPTIONS` übergibt weitere Optionen, an Leerzeichen getrennt, zum Beispiel `--mode forward_auth` oder `--console 127.0.0.1:19446`. Vor jedem Start erzeugt `seed.sh` den Admission-Seed `/var/lib/sibuna/admission.seed` (64 Hexadezimalziffern, Modus 0600, Eigentümer `sibuna`), falls er nicht existiert; einen vorhandenen ersetzt es nie, denn ein neuer Seed würde jede Sitzung beenden. Der Daemon hält seine SQLite-Datenbank in `/var/lib/sibuna`, einem Verzeichnis mit Modus 0700. Die Unit läuft ohne Capabilities, mit schreibgeschütztem System (`ProtectSystem=strict`), privatem `/tmp` und privaten Geräten und ohne neue Privilegien.

Ein Upgrade behält die Einstellungsdatei, den Seed und den Zustand und startet den Dienst nicht neu; wir starten ihn nach einer Sicherung neu. Die Deinstallation stoppt und deaktiviert den Dienst, behält aber das Konto, den Seed, die Einstellungen und den Zustand, die wir selbst löschen, wenn wir den Host außer Betrieb nehmen.

Die FreeBSD- und OpenBSD-Pakete installieren nur `/usr/local/bin/sibuna` und seine Dokumente (`pkg add`, oder `pkg_add -D unsigned` für ein Paket, das nicht von den OpenBSD-Mirrors stammt). Sie legen weder Dienst noch Konto noch Zustand an: Wir betreiben den Daemon als unprivilegierten Benutzer mit eigenem Seed und eigenem Datenverzeichnis.

Homebrew

`sibuna.rb` installiert das Release-Archiv für die Maschine: macOS oder Linux, arm64 oder x86-64. Es baut nicht aus dem Quellcode, weil das Rust von Homebrew keine Standardbibliothek für `wasm32-unknown-unknown` hat, die der Build der beiden eingebetteten WebAssembly-Module braucht; die Archive enthalten die ausführbaren Dateien, die der Release-Workflow bereits geprüft hat. Homebrew installiert Formeln nur aus Taps, daher legen wir die Formel in einen lokalen Tap:

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

Die Formel startet keinen Dienst. Ihre Caveats zeigen, wie man einen Seed erzeugt; ihr Test startet den Daemon zweimal auf demselben Datenverzeichnis.

Container und Kubernetes

Das Image `ghcr.io/insanai/sibuna:<version>` enthält die statische ausführbare Datei auf einer Distroless-Basis: CA-Zertifikate, Benutzer 65532, keine Shell und kein Paketmanager. Der Einstiegspunkt ist `sibuna`, die Argumente sind also Daemon-Optionen. Der Release-Workflow betreibt es schreibgeschützt, ohne Capabilities, mit einem schreibgeschützt eingehängten Seed und einem Volume für den Zustand, und startet es einmal neu:

```
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
```

Das Helm-Chart `helm-sibuna-<version>.tgz` betreibt einen Pod dieses Images. Es verlangt ein Secret, das wir selbst anlegen (`secret.existingSecret`, Schlüssel `admission.seed`), sodass nie ein Seed in Values oder Templates erscheint. Der Pod läuft als UID 65532 mit schreibgeschütztem Root-Dateisystem unter dem Pod-Security-Standard Restricted. Ein einziges Replikat mit `Recreate`-Upgrades hält genau einen Schreiber auf dem ReadWriteOnce-Volume, auf dem Sibuna seinen SQLite-Zustand in `data/` hält; der Claim bleibt bei der Deinstallation erhalten. Die Probes rufen `/__sibuna/health` auf. Der Service ist vom Typ ClusterIP, das Chart legt weder Ingress noch Konsole an, und seine NetworkPolicy verweigert jeden Datenverkehr, bis `networkPolicy.ingressFrom` und `networkPolicy.egressTo` die Clients und den Ursprungsserver erlauben. `upstream.host` muss ein IP-Literal sein, und `extraArgs` übergibt weitere Daemon-Optionen. Der Workflow installiert, aktualisiert und deinstalliert das Chart in einem kind-Cluster, der den Standard Restricted durchsetzt.

```
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
```

Bauen aus dem Quellcode

Wir bauen und testen mit Rust 1.100; Rust 1.94 bleibt die niedrigste unterstützte Version. Wir installieren die Release-Werkzeugkette mit `rustup toolchain install 1.100.0 --profile minimal --target wasm32-unknown-unknown` und wählen sie in den folgenden Befehlen mit `cargo +1.100.0` aus.

- `cargo build --release -p sibuna` erzeugt `target/release/sibuna` mit den Standard-Features: SQLite-Speicher und Konsole. sqlx bündelt SQLite, und `ring` (rustls) braucht zur Build-Zeit einen C-Compiler; die Binärdatei bindet die C-Laufzeit des Systems dynamisch. Das Build-Skript baut außerdem den WebAssembly-Löser und die Konsolenoberfläche, wofür `rustup target add wasm32-unknown-unknown` nötig ist; `SIBUNA_POW_WASM=<path>` und `SIBUNA_CONSOLE_WASM=<path>` betten stattdessen vorgebaute Module ein.

- `cargo build --release -p sibuna --no-default-features` erzeugt die Firewall im Arbeitsspeicher ganz ohne externe Crates; `--data-dir` und `--database-url` werden dann beim Start mit `StorageDisabled` abgewiesen und `sibuna console` mit `CONSOLEBUILD`.
- `cargo build --release -p sibuna --features postgres-tls` fügt PostgreSQL-Cluster mit verifiziertem TLS zur Datenbank hinzu. Es braucht kein OpenSSL.
- Die statischen ausführbaren Linux-Dateien der Releases entstehen mit `cargo build --release --locked -p sibuna --target x86_64-unknown-linux-musl` (oder `aarch64-unknown-linux-musl`) und `musl-gcc` als C-Compiler (`CC_x86_64_unknown_linux_musl=musl-gcc`).

Das Release-Profil verwendet fettes LTO und eine Codegen-Einheit. Der Workflow baut die beiden WebAssembly-Module einmal und reicht sie an jede Plattform weiter, sodass alle ausführbaren Dateien eines Releases byteidentische Module einbetten. `cargo xtask licenses` erzeugt `LICENSES/THIRD-PARTY-RUST.txt` neu, nachdem sich eine Abhängigkeit geändert hat. Unter Windows stoppt der Daemon bei Ctrl+C, Ctrl+Break und beim Schließen der Konsole; er läuft nicht als Windows-Dienst. Windows-Dateien haben keine Modusbits, daher kann Sibuna eine Schlüssel-, Zugangsdaten- oder Datenbank-URL-Datei, die andere lesen dürfen, nicht abweisen: Wir beschränken diese Dateien mit ACLs. Die Speicher- und CPU-Anzeigen der Konsole werden aus `/proc` gelesen und sind auf anderen Systemen nicht verfügbar.

Übung 9.2. Die Richtliniendatei und die `ip_reputation`-Zeilen schreiben, die nötig sind, damit ein Staging-Netz (`10.20.0.0/16`) Challenges umgeht, `/admin/*` 20 Arbeitsbits sequenzieller Arbeit verlangt und ein durch `X-Partner-Key` identifizierter Partner-Scraper mit 30 Anfragen pro 10 Sekunden zugelassen wird.
Hinweis: Die Daemon-Optionen legen das Limit pro Client fest, auf das jede Anfrage zuerst trifft; eine Regel kann in `limits` ein eigenes Kontingent tragen.

Die Invariante erklären. Einem Betreiber erklären, warum das Hinzufügen einer Zeile zu `policies` ohne Neustart wirkt und ohne dass je eine Anfrage wartet, und zwar anhand des Speicherthreads und des Engine-Slots.

TEIL X

Nachschlagewerk und Diagnose

Endpunkte, Metriken, Fehlerkatalog, Richtlinienschema und Speichertabellen an einem Ort.

Endpunkte

Lernziele Jede Route unter `/__sibuna/` kennen und wissen, was jede zurückgibt.

Tabelle 27 Reservierte Routen

Endpunkt	Methode	Funktion
`/__sibuna/challenge`	GET	Die Zwischenseite
`/__sibuna/challenge.json?need=&path=`	GET	Stellt eine zustandslose Challenge aus: Das Anforderungsticket aus der Challenge-Antwort bestimmt sie, andernfalls wird die gemeldete URL in Normalform ausgewertet (`414` über 8 KiB, `400` für einen Pfad ohne Normalform). Gibt `{id, algorithm, difficulty, challenges, expires_at}` zurück
`/__sibuna/verify`	POST	Nimmt `{"challenge_id", "nonce"}` oder `{"challenge_id", "proof"}` an; `200` mit `Set-Cookie`, `400` mit einer Diagnose oder `413` für einen Body, der größer als der Verbindungspuffer ist
`/__sibuna/wasm/sibuna-pow.wasm`	GET	Das Löser-Modul, 14.744 Bytes (`wasm_bytes` in `benchmarks/results/primitives-latest.json`), eine Stunde lang cachebar
`/__sibuna/worker.js`	GET	Der Web Worker mit WASM- und JavaScript-Beweisern, eine Stunde lang cachebar
`/__sibuna/honeypot`	GET	Sperrt den Aufrufer für `--ban-seconds` und zeichnet einen Vorfall auf; `403`. Nur eine Anfrage auf oberster Ebene sperrt (kein `Sec-Fetch-Dest` oder `document` und kein websiteübergreifendes `Sec-Fetch-Site`); jede andere wird mit `404` beantwortet und gezählt

Endpunkt	Methode	Funktion
`/__sibuna/health`	GET	JSON-Lebenszeichen mit Engine-Name, Version, Proxy-Modus und Nachweisalgorithmus
`/__sibuna/metrics`	GET	Prometheus-Textformat (Version 0.0.4)

Jeder andere Pfad unter `/__sibuna/` wird mit `404` beantwortet. Ausstellung und Verifikation verbrauchen das getrennte Budget `--challenge-rate-limit`, und eine gesperrte Adresse wird auf jeder Route mit `403` abgewiesen. Challenge-Antworten tragen `X-Sibuna-Status: CHALLENGE`; zugelassene Anfragen tragen zum Ursprungsserver `X-Sibuna-Status: PASS` und `X-Sibuna-Rule` (und in Forward-Auth-Antworten `X-Sibuna-Rule-Hash` in hexadezimaler Form). Ein Client, der kein HTML akzeptiert, erhält `401` mit `{"error":"challenge_required","challenge":"/__sibuna/challenge.json?need=<ticket>"}`.
Der getrennte, optional aktivierte Verwaltungs-Listener bedient `/console/`. Sein Administrator wird lokal mit `sibuna init-admin` eingerichtet; danach wird das temporäre Passwort bei der ersten Anmeldung ersetzt. `/console/api/stats` und `/console/ws` erfordern eine authentifizierte Sitzung (eine Kiosk-Sitzung eingeschlossen). Der WebSocket bündelt Statistiken, Ereignisse, Knoten, Richtlinien, Challenges und Audit mit begrenzten Schnappschüssen, Deltas und Lückenwiederherstellung. Verwaltungs-Peers verbinden sich mit `/console/peer`, und jede API-Route liegt unter `/console/api/`. Fehlerzähler des Ursprungsservers werden im Forward-Auth-Modus nicht beobachtet. Teil IX dokumentiert die Workflows der Konsole, den Einsatz mit HTTPS und die aktuellen Grenzen von SID 0007.

Metriken

Zähler, die als `sibuna_<name>_total` im Prometheus-Textformat bereitgestellt werden, in dieser Reihenfolge:

Tabelle 28 Zähler

Zähler	Erhöht, wenn
`incidents_persisted`, `incident_batches`	Datensätze und Transaktionen, deren Commit der Speicherthread bestätigt hat
`incidents_dropped`, `incident_write_failures`	Abgewiesene Einreihungen, weil der Ring voll war, und fehlgeschlagene Commit-Versuche
`requests`	Eine geparste Anfrage hat die Verteilung (Dispatch) erreicht (reservierte Routen eingeschlossen)
`allowed`, `denied`, `challenged`	Eine Richtlinienentscheidung wurde getroffen (zugelassen, abgewiesen, Challenge ausgestellt oder erneut ausgestellt)
`challenges_issued`	`/__sibuna/challenge.json` hat einen Challenge-Datensatz ausgestellt

Zähler	Erhöht, wenn
`solutions_accepted`, `solutions_rejected`	`/__sibuna/verify` hat einen Nachweis angenommen oder abgewiesen
`rate_limited`	GCRA hat eine Anfrage abgewiesen (`429`): das Anfragelimit, das Challenge-Budget oder das Kontingent einer Regel
`banned`	Eine gesperrte Adresse wurde abgewiesen, oder der Honeypot hat eine gesperrt
`bans_issued`	Eine Sperre wurde verzeichnet (Honeypot oder die Sperre aus dem Kontingent einer Regel)
`proxied`, `upstream_errors`	Eine Anfrage wurde an den Ursprungsserver weitergereicht, oder der Ursprungsserver ist gescheitert (`502`)
`parse_errors`	Ein fehlerhafter Kopf, ein wiederholter `Host`, ein Ziel ohne Normalform (`400`) oder eine nicht unterstützte Transfer-Kodierung (`501`) wurde abgewiesen
`overloaded`	Eine Verbindung über `--max-connections` oder `--max-connections-per-ip` hinaus wurde mit `503` beantwortet
`bans_refused`	Eine Sperre wurde nicht verzeichnet, weil jeder Slot, den ihr Schlüssel belegen darf, eine wirksame Sperre hielt
`forwarded_invalid`	Eine vertrauenswürdige weitergeleitete Clientadresse ließ sich nicht parsen; die Anfrage wurde nach dem Socket-Peer beurteilt
`honeypot_refused`	Ein Honeypot-Treffer hat niemanden gesperrt, weil eine Seite und keine Person ihn angefordert hat
`rate_evictions`	Eine Ankunft im Begrenzer hat die Zelle eines anderen aktiven Clients verdrängt
`ja4_present`	Mit `--ja4-header` und vertrauenswürdiger Weiterleitung trug die Anfrage genau einen Header dieses Namens mit einem JA4-Fingerabdruck
`ja4_invalid`	Mit `--ja4-header` und vertrauenswürdiger Weiterleitung war der Header fehlerhaft oder wiederholt; die Anfrage wurde ohne Fingerabdruck beurteilt

Eine Wiederholung kann `incident_write_failures` erhöhen, ohne Datensätze zu verlieren, weil der ausstehende Batch behalten wird. Die Zuwächse über ein Intervall überwachen; Summen allein sind keine Warteschlangentiefe.

Statuscodes

Tabelle 29 Statuscodes der Datenebene

Code	Wann
200	Zugelassen (Forward-Auth), Zwischenseite (HTML-Navigation, die eine Challenge braucht), interne Routen
302	Im aktuellen Protokoll nicht verwendet; die Verifikation antwortet mit `200`, und die Seite lädt neu
400	Fehlerhafte Anfrage, ein wiederholtes `Host`-Feld, ein Ziel ohne Normalform, ungültige Weiterleitungsmetadaten, ein fehlerhafter Body oder eine abgewiesene Lösung mit einer Diagnose im Elm-Stil
401	Challenge erforderlich für einen Client, der kein HTML akzeptiert, oder im Forward-Auth-Modus
403	Ablehnung durch Richtlinie oder WAF, gesperrte Adresse, Honeypot
404	Ein unbekannter Pfad unter `/__sibuna/` oder ein Honeypot-Treffer, der keine Anfrage auf oberster Ebene ist
413	Lösungs-Body größer als der 64-KB-Verbindungspuffer
414	Eine gemeldete Challenge-URL über 8 KiB
417	Nicht unterstützte Anfrageerwartung; `100-continue` wird lokal behandelt
429	GCRA-Limit überschritten; `Retry-After` in Sekunden
431	Anfragekopf über 16 KB
501	`Transfer-Encoding` mit einer anderen Kodierung als chunked
502	Ursprungsserver nicht erreichbar, oder sein Antwortkopf war fehlerhaft oder größer als 16 KB
503	Verbindungslimit (`--max-connections` oder `--max-connections-per-ip`) erreicht, oder der Knoten ist im Drain-Zustand; der Socket wird nach der Antwort geschlossen

Fehlerkatalog

`INVALID COMMAND LINE` bricht den Start ab bei jeder Option, die der Daemon nicht kennt, bei jedem Wert außerhalb seines dokumentierten Bereichs und bei einer ungültigen, fehlenden oder doppelten Modusauswahl. Der Block nennt die Option, den angegebenen Wert, den erwarteten Bereich und den Fehler (`UnknownOption`, `MissingValue`, `InvalidValue`, `InvalidMode`, `MissingMode`, `DuplicateMode`, `RemovedOption` für eine Zaxonlite-Cluster-Option, `ConflictingOptions` für zwei Speicheroptionen, die einander ausschließen, `MissingOption` für eine Datenbank-URL ohne `--cluster-node`) und endet mit einem Hinweis. Eine Datenbank-URL wird als `[redacted]` angezeigt. `--mode reverse_proxy` oder `--mode forward_auth` genau

einmal angeben; `-m` ist die gleichwertige Kurzoption. Eine abgewiesene Richtliniendatei gibt einen Block aus, dessen Titel ihren Fehler nennt (zum Beispiel `INVALID POLICY ACTION`), mit Datei, Fehlername, Regel, Feld und Wert. Ein `ja4`-Muster, auf das kein Fingerabdruck passen könnte, gibt `INVALID POLICY JA4` (`InvalidJa4`) aus; ein Verwaltungsdokument der Konsole mit einem solchen Muster wird mit demselben Fehlernamen abgewiesen. Konsolenoptionen werden getrennt geprüft: `CONSOLE002` bezeichnet eine fehlerhafte oder wiederholte Konsolenoption und `CONSOLE001` eine Konsolenkonfiguration, die eine Regel verletzt (`StorageRequired` ohne Speicher).

Implementierung: `ErrorKind::explain` Ordnet jedem Domänenfehler eine Grenzzeile, eine Erklärung und einen `Hint:` zu. In `crates/core/src/errors.rs` (Crate `sibuna-core`).

Tabelle 30 Challenge- und Sitzungsfehler

Fehler	Ursache	Hinweis
`MalformedChallenge`	Die Kennung ist kein wohlgeformter Challenge-Datensatz	Eine frische Challenge abrufen und sie unverändert einreichen
`InvalidChallengeTag`	Das Tag authentifiziert nicht; nicht von diesem Cluster ausgestellt oder bearbeitet	Challenges lassen sich nicht fälschen; eine neue anfordern
`ChallengeExpired`	Älter als die Challenge-TTL oder in der Zukunft ausgestellt	Eine neue Challenge anfordern
`FingerprintMismatch`	Von einer anderen Adresse oder einem anderen User-Agent eingereicht	Vom Client einreichen, der sie abgerufen hat
`DifficultyNotMet`	Der Hashcash-Nonce fehlen die geforderten Nullbits	Weiter nach Nonces suchen
`InvalidProof`	Der Nachweis sequenzieller Arbeit öffnet die festgelegten Labels nicht	Den Beweiser für die ausgestellte Tiefe und die Öffnungen vollständig laufen lassen
`WrongSolutionType`	Eine Nonce für eine PoSW-Challenge oder ein Nachweis für Hashcash	Das Lösungsfeld an den Algorithmus anpassen
`DoubleSpendAttempt`	Die Challenge wurde bereits verbraucht	Challenges sind nur einmal verwendbar
`StoreFull`	Die Verbrauchsmenge ist voll	Die Challenge-TTL senken oder `--spent-tags` erhöhen
`InvalidTokenSignature`	Tag oder Signatur des Cookies ungültig	Über die Zwischenseite erneut authentifizieren

Fehler	Ursache	Hinweis
`TokenExpired`	Cookie nach seinem Ablauf	Erneut authentifizieren
`TokenBoundAddressMismatch`	Cookie von einer anderen Clientidentität vorgelegt	Cookies lassen sich nicht weitergeben

Die API der Konsole beantwortet einen Fehlschlag mit `{"error":"<CODE>","hint":"<sentence>"}`; der Hinweis ist der Text, den die Oberfläche anzeigt (`crates/console/src/http/reply.rs`).

Tabelle 31 Fehlercodes der Konsole, die Betreiber betreffen

Code	Status	Bedeutung
`CONSOLE002`	400	Die Anfrage ist fehlerhaft, zu groß oder in dieser Form nicht erlaubt
`CONSOLE003`	503 / 429	Die Speicher-Job-Warteschlange ist voll (32 Jobs), oder der Passwortverifizierer ist ausgelastet
`CONSOLEQUORUM`	503	Die Datenbank hat nicht innerhalb der Wartezeit der Anfrage von 10 Sekunden geantwortet; das Ergebnis ist unbekannt, und die Datenebene arbeitet mit der zuletzt angewendeten Richtlinie weiter
`CONSOLE004`	503	Der Dienst kann nicht antworten: in diesem Build nicht verfügbar, im Herunterfahren oder ein interner Fehler
`CONSOLENODE`	400 / 409 / 503	Ein Knotenbefehl wurde abgewiesen: Er nennt einen anderen Knoten (400), eine veraltete Boot- oder Steuerrevision (409), oder er traf ein, während ein anderer Befehl lief oder ein Abschluss nicht verzeichnet werden konnte (503); vor einer Wiederholung den Vorgangsbeleg prüfen
`CONSOLEMUTATION`	429	Mehr als sechzig Verwaltungsänderungen in einer Minute aus einer Sitzung

Richtlinienschema

```
{
  "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": [                                                    // bis zu 128
    {
```

```
      "name": "<kebab-case>",                                    // 1 bis 128
druckbare Bytes
      "path" | "path_regex": "<pattern>",                        // bis zu 512
Bytes
      "user_agent" | "user_agent_regex": "<pattern>",
      "headers" | "headers_regex": { "<Header>": "<pattern>" }, // bis zu 4
      "remote_addresses" | "cidrs": ["<cidr>", ...],             // bis zu 8, IPv4
oder IPv6
      "action": "ALLOW" | "DENY" | "CHALLENGE" | "WEIGH",
      "weight": int,                                             // nur WEIGH,
-1000..1000
      "challenge": { "difficulty": <work bits>, "algorithm": "hashcash" |
"posw" },
      "limits": { "rate": int, "window_seconds": int, "ban_seconds": int },
      "ja4": "<fingerprint>" | "<prefix>*"                       // 36 Bytes oder
Präfix und *
    }
  ]
}
```

`challenge_at` und `deny_at` liegen zwischen -1000 und 1000, mit `deny_at` mindestens gleich `challenge_at`, und `bits_step` zwischen 1 und 1000. Ein `challenge`-Block gehört nur zu `CHALLENGE`-Regeln, mit 1 bis 64 Arbeitsbits. `limits` gibt einer Regel ein eigenes Kontingent: `rate` 1 bis 1.000.000 und `window_seconds` 1 bis 86.400 sind Pflicht, `ban_seconds` (höchstens 86.400) ist optional, und `WEIGH`-Regeln können keine Limits tragen. `ja4` ist ein ganzer JA4-Fingerabdruck, der gleich sein muss, oder ein Präfix gefolgt von *, mit dem der Fingerabdruck beginnen muss; * allein passt auf jeden Fingerabdruck, und eine Anfrage ohne Fingerabdruck passt auf keine `ja4`-Regel. Ein Muster, auf das kein Fingerabdruck passen könnte, wird mit `InvalidJa4` abgewiesen (Teil IX, „JA4-Fingerabdrücke vom Eingangsproxy“). Der Reputations-Trie von `ip_rules` fasst höchstens 8.192 Knoten. Jede Regel darf von jedem Kriterium eine Schreibweise verwenden.

Mustergrammatik: `.*` oder * passen auf alles; `^...$` verankert einen exakten Pfad; ein abschließendes *, `/*` oder `.*` ist ein Präfix; ein Muster, das mit `/` beginnt, ist ein exakter Pfad; alles andere ist eine Teilzeichenkette ohne Beachtung der Groß-/Kleinschreibung. Pfadmuster werden mit dem Pfad in Normalform verglichen (ein Schrägstrich zwischen Segmenten, keine Punktsegmente, nicht reservierte Zeichen unmaskiert, andere Escapes mit großgeschriebenen Hexadezimalziffern), sodass ein Muster mit `;`, `%2F`, `%5C` oder `\` auf nichts passt, und der Pfadvergleich beachtet Groß-/Kleinschreibung (SID 0003).

Speichertabellen

`policy_revision`, `policies` (mit dem nullbaren Kriterium `ja4_pattern` von höchstens 37 Bytes), `policy_inspection`, `ip_reputation`, `incident_receipts`, `security_incidents` (mit dem 256-Byte-Embedding und dem nullbaren 36-Byte-Fingerabdruck `ja4`), der Volltextindex (`incidents_fts`, FTS5 über kanonische Tokens, unter SQLite; die `tsvector`-Spalte `search` mit einem GIN-Index unter PostgreSQL) und `sibuna_schema`; die Konsole fügt 33 `console_*`-Tabellen hinzu. Gespeicherte Namen von Aktionen, Algorithmen und Modi sind kleingeschrieben.

Vorfalls-IDs sind `node_id << 40 | sequence`, im ganzen Cluster ohne Koordination eindeutig; Knoten-IDs reichen von 1 bis 8.388.607.

Build-Ziele

Tabelle 32 Build- und Repository-Befehle

Befehl	Ergebnis
`cargo build --release -p sibuna`	Daemon mit SQLite-Speicher und Konsole (die Standard-Features)
`cargo build --release -p sibuna --no-default-features`	Daemon im Arbeitsspeicher ohne Speicherung oder Konsole und ohne externes Crate
`cargo build --release -p sibuna --features postgres-tls`	Daemon, der sich zusätzlich mit einer gemeinsamen PostgreSQL-Datenbank verbindet, mit verifiziertem TLS (das Feature `postgres` allein verbindet sich ohne TLS)
`cargo test`	Unit-Tests, Integrationstests gegen einen laufenden Daemon, Speicher- und Konsolentests (die PostgreSQL-Tests, wenn `SIBUNA_TEST_DATABASE_URL` gesetzt ist)
`cargo clippy --all-targets`, `cargo fmt --check`	Lints und Formatierung (`max_width = 99`)
`cargo xtask check`	Die Grenzen von 70 Zeilen pro Funktion, 1.408 Zeilen pro Datei und 99/108 Spalten pro Zeile; keine Tabulatoren
`cargo xtask browser`	Prüfungen in Headless Chromium (`sibuna-browser`)
`cargo xtask bench <primitives|product|latency>`	Benchmarks im Release-Modus; Ergebnisse in `benchmarks/results/<command>-latest.json`
`cargo build -p sibuna-pow-wasm --target wasm32-unknown-unknown --profile wasm`	Nur das Browsermodul (`-p sibuna-console-ui` für die Konsolenoberfläche)
`cargo run -p sibuna-geoip --bin geoip-snapshot -- ...`	Länderdaten validieren und einen einbettbaren GeoIP-Schnappschuss schreiben
`guidedog build` / `guidedog dsc`	Dieses Buch / die SID-Dokumente

Das Build-Skript des Daemons (`cmd/sibuna/build.rs`) baut beide WebAssembly-Module mit dem Profil `wasm` in das Zielverzeichnis des Workspace und lässt den Build scheitern, wenn die Konsolenoberfläche 768 KiB überschreitet. `SIBUNA_POW_WASM` und `SIBUNA_CONSOLE_WASM` benennen stattdessen vorgebaute Module, für Builds ohne das Ziel `wasm32-unknown-unknown`.

`SIBUNA_GEOIP_DATA` benennt einen einzubettenden GeoIP-Schnappschuss; er wird zur Build-Zeit dekodiert, und ein Build ohne Konsole weist ihn ab (`GEOIP001`).

Übung 10.1. Ein Client erhält `400` mit dem Titel `CLIENT FINGERPRINT MISMATCH`, nachdem er mitten im Lösen von WLAN zu Mobilfunk gewechselt hat. Die Ursache aus den Token- und Challenge-Formaten erklären und die kleinste Änderung an der Zwischenseite vorschlagen, mit der sie sich sauber erholt.

Die Invariante erklären. Ohne nachzusehen die Endpunkte aufzählen, die ein Reverse Proxy vor Sibuna an den Daemon statt an den Ursprungsserver routen muss, und für jeden sagen, warum er nötig ist.

TEIL XI

Kurzreferenzkarte

Neben dem Terminal bereithalten: bauen, ausführen, konfigurieren, routen, beobachten und diagnostizieren. Teil X enthält die vollständigen Tabellen; diese Karte enthält, was ein Betreiber eintippt.

Bauen

```
cargo build --release -p sibuna                          # Daemon + SQLite + Konsole
cargo build --release -p sibuna --no-default-features    # im RAM, ohne externe Crates
cargo build --release -p sibuna --features postgres-tls  # Cluster mit PostgreSQL
rustup target add wasm32-unknown-unknown   # oder SIBUNA_POW_WASM / SIBUNA_CONSOLE_WASM
cargo test            # Unit-, Live-Daemon-, Speicher- und Konsolentests
cargo clippy --all-targets && cargo fmt --check
cargo xtask check     # Grenzen: 70 Zeilen / 1408 Zeilen / 99 Spalten
cargo xtask browser   # Prüfungen in Headless Chromium
cargo xtask bench product   # Gesamtprodukte unter wrk -> benchmarks/results/
guidedog build        # dieses Buch (guidedog dsc: die SID-Dokumente)
```

Ausführen

```
# Shield (Standard): PoW + WAF, Proxy zu :3000
sibuna -p 8080 -u 3000 -s /etc/sibuna/secret
# Gate: nur PoW
sibuna --gate -p 8080 -u 3000 -s /etc/sibuna/secret
# Forward-Auth hinter Nginx/Caddy (vertraut X-Forwarded-For)
sibuna -m forward_auth --host 127.0.0.1 -p 8080 -s /etc/sibuna/secret
# Edge: persistente Richtlinien, Reputation, Forensik in <dir>/sibuna.db
sibuna -D /var/lib/sibuna -s /etc/sibuna/secret
# JA4 vom vertrauenswürdigen Proxy (Header von Cloudflare; nginx-Modul: x-ja4; ATS: ja4)
sibuna --ja4-header cf-ja4 --trust-forwarded -p 8080 -u 3000 -s /etc/sibuna/secret
# Cluster-Mitglied: gemeinsames PostgreSQL, eindeutige Knotenkennung
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
```

Konsole (optionale Vorschau)

```
# Bei gestopptem Daemon initialisieren, dann die Konsole starten
sibuna init-admin admin --data-dir ./data
sibuna --data-dir ./data --console 127.0.0.1:19446
```

`http://127.0.0.1:19446/console/` öffnen und das temporäre Passwort ersetzen. Mit `--console-location 1.3521,103.8198` wird der Knoten auf dem Globus platziert. Fernzugriff erfordert einen HTTPS-Proxy, einen ausdrücklichen Origin und CIDRs vertrauenswürdiger Proxys; dazu Teil IX folgen. SID 0007 hat den Status Committed. Die Konsole besteht ihre Browser-Workflows in Headless Chromium (SID 0010, Kriterium 5); ihr formales Isolationskriterium für die Leistung ist noch offen (Teil VIII).

Die wichtigsten Optionen

Option	Standard	Bedeutung
`-d, --difficulty`	16	Arbeitsbits (PoSW-Tiefe = Bits - 3)
`-a, --algorithm`	posw	`posw` oder `hashcash`
`--posw-challenges`	16	Öffnungen pro Nachweis
`--token-ttl`	86400	Sitzungsdauer in Sekunden
`--challenge-ttl`	300	Challenge-Dauer in Sekunden
`--token-scheme`	mac	`mac` oder `ed25519`
`--rate-limit / --rate-window`	100 / 10	GCRA-Burst pro Fenster
`--ban-seconds`	3600	Honeypot-Sperre
`-w, --workers`	CPUs	Accept-Threads
`--max-connections`	1024	danach `503`
`--max-connections-per-ip`	64	pro Peer-Adresse (IPv6: pro `--ipv6-prefix`), danach `503`
`--trusted-hops`	1	Client = n-ter Eintrag von `X-Forwarded-For` von rechts
`--ja4-header`	aus	Header mit dem JA4-Fingerabdruck des Eingangsproxys; nur bei vertrauenswürdiger Weiterleitung gelesen
`--idle-timeout`	15	Reaper-Sekunden
`-P, --policy-file`	keine	JSON-Richtlinie
`--trust-forwarded`	aus	an im Forward-Auth-Modus

Option	Standard	Bedeutung
`--secure-cookie`	aus	fügt `Secure` hinzu

Endpunkte unter `/__sibuna/`

Route	Antwort
`challenge`	HTML der Zwischenseite
`challenge.json?need=&path=`	`{id, algorithm, difficulty, challenges, expires_at}`
`verify` (POST)	`{challenge_id, nonce | proof}` -> `200` + `Set-Cookie`, sonst `400`
`wasm/sibuna-pow.wasm`	Löser mit 15.181 Bytes
`worker.js`	Worker mit WASM- und JS-Beweisern
`honeypot`	sperrt den Aufrufer, `403`; `404` bei einem Abruf einer Unterressource oder websiteübergreifend
`health`	`{"status":"ok",...}`
`metrics`	Prometheus-Text

Alle an den Daemon routen, nie an den Ursprungsserver.

Statuscodes

Code	Wann
200	zugelassen (Forward-Auth), Zwischenseite, interne Routen
400	fehlerhafte Anfrage oder fehlerhaftes Ziel, abgewiesene Lösung (Body mit Diagnose)
401	Challenge erforderlich: Nicht-HTML-Client oder Forward-Auth
403	Ablehnung durch Richtlinie oder WAF, Sperre, Honeypot
404	unbekannte `/__sibuna/`-Route, abgewiesener Honeypot-Treffer
413	Lösungs-Body über dem 64-KB-Puffer
414	gemeldete Challenge-URL über 8 KiB
417	nicht unterstützte Anfrageerwartung
429	GCRA-Limit, `Retry-After` in Sekunden
431	Kopf über 16 KB

Code	Wann
501	Transfer-Kodierung außer chunked
502	Ursprungsserver nicht erreichbar oder fehlerhaft
503	Verbindungslimits erreicht oder Knoten im Drain-Zustand

Header

- *Upstream (Proxy)*: `X-Forwarded-For` (Socket-Peer, bei Vertrauen nach der empfangenen Kette), `X-Forwarded-Proto`, `X-Real-IP` (der beurteilte Client), `X-Sibuna-Status: PASS`, `X-Sibuna-Rule: <rule>`; Hop-by-Hop-Felder und alternative Weiterleitungsfelder werden verworfen, auch Schreibweisen mit Unterstrich.
- *Forward-Auth-Antwort*: `X-Sibuna-Status`, `X-Sibuna-Rule`, `X-Sibuna-Rule-Hash`.
- *Challenge-Antwort*: `X-Sibuna-Status: CHALLENGE`.
- *Cookie*: `__sibuna_token=<75 chars, 139 with ed25519>; Path=/; Max-Age=<ttl>; HttpOnly; SameSite=Lax[; Secure]`

Gerüst der Richtliniendatei

```
{ "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: ein ganzer Fingerabdruck oder ein Präfix und *; ohne Fingerabdruck kein Treffer. Muster: * oder .* beliebig; ^...$ exakt; abschließendes * Präfix; führendes / exakter Pfad; sonst Teilzeichenkette ohne Beachtung der Groß-/Kleinschreibung. Pfade werden in Normalform verglichen. Reihenfolge: WAF -> Trie deny/allow -> Regeln -> Bewertung -> Bypass -> Trie challenge -> Bots -> Standard.

Speicherung (Edge) in SQL

```
INSERT INTO policies (id, name, priority, path_pattern, action, difficulty,
algorithm,
  header_matchers, cidr_matchers, weight, enabled, created_at, updated_at)
```

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

SQLite (`sqlite3 <dir>/sibuna.db`); Aktionen werden kleingeschrieben. Bewertung ≤ -50 -> deny-Präfix; ≥ 50 -> allow. Cluster: dasselbe Geheimnis auf jedem Knoten, eine PostgreSQL-Datenbank, `--features postgres-tls`, `--database-url-file <file> --cluster-node N` mit einem eindeutigen N von 1 bis 8.388.607; keine Mitgliederliste, kein Leader und kein Cluster-TLS (SID 0011).

Diagnosen

Titel	Bedeutung
`MALFORMED CHALLENGE`	Kennung bearbeitet oder gekürzt; eine neue abrufen
`INVALID CHALLENGE TAG`	nicht von diesem Seed oder Knoten ausgestellt
`CHALLENGE EXPIRED`	älter als die TTL oder Uhrenabweichung > 60 s
`CLIENT FINGERPRINT MISMATCH`	Adresse oder User-Agent geändert
`DIFFICULTY NOT MET`	Nonce fehlen Nullbits
`INVALID PROOF OF SEQUENTIAL WORK`	Öffnungen erreichen die Wurzel nicht
`WRONG SOLUTION TYPE`	Nonce für PoSW oder Nachweis für Hashcash
`DOUBLE SPEND PREVENTED`	Challenge bereits verwendet
`STORE FULL`	Verbrauchsmenge gesättigt; TTL senken oder `--spent-tags` erhöhen
`NETWORK BINDING MISMATCH`	Cookie auf einen anderen Client kopiert
`INVALID COMMAND LINE`	unbekannte Option, ungültiger Wert oder eine entfernte Zaxonlite-Option

Metriken für Alarme: `denied`, `rate_limited`, `overloaded`, `upstream_errors`, `incidents_dropped`, `incident_write_failures`, `bans_refused`, `forwarded_invalid` (alle `sibuna_*_total`).

Zahlen zum Merken

Challenge-Kennung 70 Zeichen (36-Byte-Nutzlast + 16-Byte-Tag); Token 75 Zeichen (40 + 16), 139 mit Ed25519; Nachweisbytes $32(1 + t(n + 1))$; PoSW-Tiefe = Bits - 3, Bereich 4 bis

24; Anfragepuffer 64 KB; Kopflimit 16 KB; geprüfter Body 8 KB; Verbrauchsmenge 1.048.576 Tags (`--spent-tags`); Ratenzellen 262.144 pro Tabelle (`--rate-table-cells`); Sperren 65.536 Slots (`--ban-table-slots`); 64 Verbindungen pro Clientadresse; Vorfallsring 512; Batch 32; Kampagnenabstand 0,35 über die 16.384 jüngsten Embeddings; Keep-Alive 4.096 Anfragen pro Verbindung; Pool zum Ursprungsserver 256 Sockets; Speicher-Job-Warteschlange 32.

Routing am Eingangsproxy

Das Rezept für Nginx oder Caddy aus Teil IX, „Forward-Auth hinter einem Eingangsproxy“, verwenden. `/__sibuna/*` direkt routen; Anwendungsanfragen mit der ursprünglichen URI, der Methode und der vertrauenswürdigen Clientadresse autorisieren. Eingehende Sibuna-Audit-Header durch das Autorisierungsergebnis ersetzen. Nginx braucht eine ausdrückliche Behandlung von Challenge, 429 und nicht verfügbarer Autorisierung.
`reverse_proxy` trägt zugelassene HTTP/1.1-Bodies und WebSockets; `forward_auth` erteilt die Zulassung, und der Eingangsproxy trägt sie. Ausgelassene Auth-Bodies und Antworten des Ursprungsservers kann Sibuna weder prüfen noch zählen. TLS terminiert am Eingangsproxy; natives HTTP/2 ist zurückgestellt. Uploads verwenden Content-Length oder Chunked-Kodierung; die Prüfung sieht höchstens ein Präfix von 8 KiB, die Dateibytes bleiben undurchsichtig.

Ausgewählte Lösungen

Diese Lösungen sind Prüfsteine für die Argumentation, kein Ersatz für die Übungen. Weicht ein Ergebnis ab, vergleichen wir zuerst die Annahmen: Die Endpunkte von Zeitintervallen, was als Arbeit gezählt wird und ob ein Schreibvorgang bestätigt ist, sind häufige Ursachen für Abweichungen.

Teil I: Kosten und Wahrscheinlichkeit

1.1. Eine Million Versuche, verteilt auf 100 Anfragen, ergeben 10.000 Versuche pro Anfrage. Eine gemeinsam genutzte Sitzung ändert diese Rechnung nicht, wenn die Gesamtzahl der Anfragen bei 100 bleibt. Das tatsächliche System muss entscheiden, ob ein zweiter Prozess die Bindungen an Fingerabdruck und Routing erfüllen kann; diese Bedingungen liegen außerhalb der Amortisierungsgleichung.
1.2. Der Überschuss beträgt $1000 - 600 = 400$ Datensätze/s. Eine leere Warteschlange mit 512 Datensätzen verschafft im Fluidmodell $512/400 = 1.28$ Sekunden. Echte Erzeuger liefern in Bursts, und der Verbraucher bestätigt gebündelt, daher hängt der genaue Zeitpunkt des ersten Verwerfens von den Ankunfts- und Commit-Zeiten ab.

Teil II: Stand der Technik

2.1. Eine zweimal kompilierte Quelle kann sich nicht selbst widersprechen, was die Aussage, die Kodierung der Nonce, die Baumform oder das Layout der Öffnungen betrifft; eine Drift erforderte einen Compilerfehler. Sie sagt nichts darüber, ob die Konstruktion korrekt ist (kryptographische Begutachtung), ob die Browser-Laufzeit sie richtig ausführt (das deckt ein bytegenauer Vergleich zwischen dem WebAssembly- und dem JavaScript-Beweiser ab) oder ob die Parameter kalibriert sind (Messung).
2.2. (a) Sibuna Gate oder Anubis, eine Binärdatei; (b) Sibuna Shield, SafeLine oder Cloudflare (Pro-Tarif oder höher); (c) Sibuna Edge mit einer gemeinsamen PostgreSQL-Datenbank (`--database-url`), Anubis mit einem gemeinsamen Valkey-Speicher (für Zustand, nicht für Sperren) oder Cloudflare. Das einzige Produkt, das alle drei ohne Docker-Host oder Abonnement erfüllt, ist Sibuna Shield mit Knoten, die eine PostgreSQL-Datenbank teilen (`--database-url` und `--cluster-node`); die Zeilen „Läuft als“, „Prüfung“ und „Mehrere Knoten“ entscheiden es.

Teil III: Kryptographie

3.0. Die Schätzung des Verteilungsendes ist $e^{-4.6} \approx 0.010$. Den Nonce-Raum aufzuteilen ändert die Rate, mit der Versuche abgeschlossen werden, nicht die Erfolgswahrscheinlichkeit jedes unabhängigen Versuchs. Doppelte Versuche oder Koordinationsarbeit können eine echte parallele Implementierung weniger effizient machen.

3.1. Gelingt jedes Commitment mit Wahrscheinlichkeit q, so scheitern alle g mit Wahrscheinlichkeit $(1-q)^g$. Mindestens eines gelingt mit Wahrscheinlichkeit $1-(1-q)^g$. Die Formel zählt weder den Aufbau der Commitments noch Anfragebudgets, Abhängigkeiten zwischen Versuchen oder die Arbeit, die nötig ist, um ein Commitment mit einer bestimmten Annahmewahrscheinlichkeit zu finden.
3.2. Bei Tiefe 17 und 16 Öffnungen hat der Nachweis $32(1+16\times 18)=9248$ Bytes. Bei 32 Öffnungen hat er $32(1+32\times 18)=18464$ Bytes. Die Bytes der Öffnungen verdoppeln sich; die 32-Byte-Wurzel wird weiterhin einmal gesendet.
3.3. Ohne die Arbeitsstufe würde eine auf einer billigen Route erworbene Sitzung eine Anfrage an eine teure Route zulassen, die mehr Arbeit verlangt; ohne `timestamp` könnte ein Token „in der Zukunft“ ausgestellt werden und seine Richtlinie überleben; ohne `expiry` würde es nie ablaufen; ohne `rule_hash` könnte der Upstream nicht erfahren, welche Regel den Client zugelassen hat; ohne `fingerprint` würde ein auf eine andere Maschine kopiertes Cookie akzeptiert. Das Tag muss alle fünf abdecken, weil jedes Feld außerhalb davon frei bearbeitet werden könnte, ohne dass der Verifizierer es bemerkt.

Teil IV: Protokoll

4.1. Beide Browser teilen Adresse und User-Agent, also teilen sie den Fingerabdruck, die Ratenlimit-Zelle und den Sperr-Slot. Sie teilen nicht die Challenge (jeder ruft seinen eigenen Datensatz mit eigener Nonce ab), den Eintrag in der Verbrauchsmenge oder das Cookie. Um das Cookie des anderen Browsers wiederzuverwenden, müsste ein Angreifer hinter demselben NAT es von der anderen Maschine lesen: Der Fingerabdruck würde dann passen. Deshalb sind `HttpOnly` und `SameSite` bei einer gemeinsam genutzten Adresse wichtiger als die Bindung.

Teil V: Zustand und Reihenfolge

5.1. Der Kopf und der erste Teil des Bodys (so viel, wie nach dem Kopf in 64 KB passt) liegen im Verbindungspuffer und werden mit `write_all(request.body)` gesendet. Die übrigen Bytes gelangen nie als Ganzes in den Puffer: `relay_body` liest sie in Stücken von 16 KB vom Client-Reader und schreibt jedes zum Ursprungsserver. Der Kopf muss zuerst gehen, weil der Ursprungsserver `Content-Length` vor dem Body braucht und weil die Audit-Header Teil des Kopfs sind.
5.2. Das Emissionsintervall beträgt 100 ms und die Bursttoleranz 9.900 ms. Ein inaktiver Client darf sofort 100 Anfragen senden, danach eine pro 100 ms. Bis einschließlich des Endpunkts bei 3.000 ms beträgt die Schranke 130. Ein fester 10-Sekunden-Zähler würde 100 im ersten Fenster erlauben und 100 weitere in dem Moment, in dem das Fenster wechselt; ein gleitendes Protokoll würde in beliebigen 10 Sekunden genau 100 erlauben, muss aber 100 Zeitstempel pro Client speichern.

Teil VI: Algorithmen

6.1. Ein Apostroph in einem Namen und ein gewöhnlicher Parameter namens `order` begründen keinen SQL-Ausdruck. Ein Teilzeichenketten-Detektor könnte beides für Syntax halten. Der

strukturelle Detektor sucht nach weiteren Indizien, darunter Operatoren und literale Beziehungen. Er bleibt eine beschränkte Prüfung, kein Parser für jede Anwendungssprache.
6.2. LDAP-Filter bestehen aus Klammern, `|`, `&`, `!` und `=`. Der Detektor braucht eine Byteklasse für `(` und `)` (neu) und kann die vorhandenen Klassen für `=` und für Shell-Trennzeichen für `|` und `&` wiederverwenden. Er sollte bei einer *Folge* wie `)(` oder `(|(` anschlagen, nicht bei einer einzelnen Klammer, aus demselben Grund, aus dem der SQL-Detektor ein Anführungszeichen oder eine Tautologie verlangt.

Teil VII: Browser-Engine

7.1. Bei den standardmäßigen 16 Arbeitsbits (Tiefe 13 mit 16 Öffnungen) werden aus den angenommenen 20 ms in WebAssembly $50 \times 20 = 1000$ ms, eine Sekunde, in JavaScript auf dem Laptop und vier Sekunden auf dem langsameren Telefon. Die Zahlen sind Annahmen; wir messen die Geräte unserer eigenen Clients, bevor wir eine Richtlinie festlegen. Eine vernünftige Richtlinie: Die Seite die `fallback`-Nachricht weiterleiten lassen, sie serverseitig als Metrik erfassen und die Schwierigkeit für eine Regel senken, die auf Clients passt, von denen bekannt ist, dass sie WebAssembly blockieren, statt für alle.

Teil VIII: Auswertung

8.2. Für die Zeile `admitted` von Gate ergeben sich $11.84 \times 211,345 / 10^6 \approx 2.50$ belegte Kerne von vier; der Daemon war also nicht CPU-gebunden, und der Lastgenerator setzte die Grenze. Die 11,84 µs CPU pro Anfrage liegen weit über den Kosten der Klassifikation von weniger als einer Mikrosekunde, weil eine Produktzeile auch das Lesen und Schreiben des Sockets im Kernel, den Parser, die Prüfung des Sitzungstags, das Erhöhen der Metriken, das Formatieren der Antwort, das Leeren des Writers und die Übergabe durch den Scheduler zwischen Lastgenerator und Verbindungsthread enthält. Die Primitive-Suite misst nichts davon.

Teil IX: Betrieb

9.1. Wird zuerst der Datenschreibvorgang bestätigt und die Quittung getrennt aktualisiert, hinterlässt ein Absturz dazwischen die alte Quittung, und eine Wiederholung kann die Wirkung erneut anwenden. Wird die Quittung zuerst bestätigt, kann ein Absturz vor dem Datenschreibvorgang dazu führen, dass die Wiederholung Daten überspringt, die nie existierten. Eine einzige atomare Transaktion schließt beide Lücken. Geht die Bestätigung der Transaktion verloren, ist weiterhin die aufbewahrte Wiederholung nötig.
9.2. `ip_rules: {"10.20.0.0/16": "ALLOW"}` (oder eine `ip_reputation`-Zeile mit Wert 100) lässt Staging zu; eine Regel `{"name": "admin", "path": "/admin/*", "action": "CHALLENGE", "challenge": {"difficulty": 20, "algorithm": "posw"}}` verlangt die Arbeit; eine Regel, die auf `headers: {"X-Partner-Key": ".*"}` passt, mit `ALLOW` und `"limits": {"rate": 30, "window_seconds": 10}` lässt den Partner mit 30 Anfragen pro 10 Sekunden zu. Das Kontingent der Regel wird für jede Anfrage geprüft, auf die sie passt, nach dem Daemon-Limit pro Client (`--rate-limit`, `--rate-window`), das deshalb mindestens ebenso großzügig sein muss.

Teil X: Referenz

10.1. Der Challenge-Datensatz trägt den schlüsselbasierten Fingerabdruck der Adresse, die ihn abgerufen hat; die Lösung kommt von einer neuen Adresse, daher gibt `verify_and_mint FingerprintMismatch` zurück, bevor der Nachweis betrachtet wird. Die kleinste Abhilfe liegt in der Zwischenseite: Bei einer `400` mit dem Titel `CLIENT FINGERPRINT MISMATCH` eine frische Challenge abrufen und erneut lösen, statt den Fehler anzuzeigen, da die bereits geleistete Arbeit nicht übertragen werden kann.

Die Untersuchung fortsetzen

Jeweils eine Invariante in einem Wegwerftest ändern. Die erneute Prüfung des Schnappschusses entfernen, die Vorfallsquittung weglassen oder das Ratenintervall abrunden. Einen Ablauf des Fehlschlags vorhersagen, bevor wir den Test ausführen. Eine nützliche Regressionsprüfung tut mehr, als die Ausgabe des Normalfalls zu prüfen: Sie zeigt, warum die weggelassene Bedingung notwendig ist.

Glossar

Zulassung
: Die Entscheidung, eine Anfrage an den Ursprungsserver weiterzuleiten. In Sibuna wird sie durch eine gültige Sitzung, eine `ALLOW`-Regel, einen statischen Bypass-Pfad oder ein `allow`-Präfix der Reputation getroffen, und sie hebt nie eine WAF-Ablehnung auf.

Aho-Corasick-Automat
: Ein endlicher Automat, der jedes Muster aus einer festen Menge in einem Durchlauf über die Eingabe findet. Sibuna faltet die Fehlerkanten in eine dichte Übergangstabelle, sodass jedes Byte einen Tabellenzugriff kostet.

Sperrtabelle
: 4.096 sperrfreie Slots gesperrter Clientadressen mit Ablaufzeit, geschrieben vom Honeypot und von der im Cluster gemeinsam genutzten Reputation, gelesen bei jeder Anfrage.

Kampagne
: Eine Gruppe von Sicherheitsvorfällen, deren Nutzlast-Embeddings höchstens den Cosinus-Abstand 0,35 voneinander haben; eine heuristische Gruppierung, kein Beweis gemeinsamer Herkunft.

Challenge-Datensatz
: Die 70 Zeichen lange zustandslose Kennung, die ein Client lösen muss: Version, Algorithmus, Schwierigkeit, Öffnungen, Ausgabezeit, Fingerabdruck des Clients, PRF-Nonce, Regelhash und ein 16-Byte-Tag aus BLAKE3 mit Schlüssel. Ihr Ausstellen schreibt nichts auf dem Server.

Schwierigkeit, Arbeitsbits
: Die eine Ganzzahl, die beide Stufen skaliert: die Zahl führender Nullbits bei Hashcash oder drei mehr als die Baumtiefe bei sequenzieller Arbeit.

Edge
: Ein Sibuna-Einsatz mit `--data-dir` (eine SQLite-Datei auf einem Knoten) oder mit `--database-url` und `--cluster-node` (eine gemeinsame PostgreSQL-Datenbank für einen Cluster), der beiden Betriebsformen persistente Richtlinien, gemeinsame Reputation und Vorfallsforensik gibt.

Engine-Slot
: Der Ort, an dem die veröffentlichte Richtlinien-Engine als referenzgezählter Zeiger liegt. Worker halten beim Klassifizieren eine Referenz; der Speicherthread baut eine neue Engine abseits des Anfragepfads und veröffentlicht sie per Zeigertausch, und die alte Engine wird freigegeben, wenn ihr letzter Leser sie loslässt.

Fingerabdruck
: Ein schlüsselbasierter Hash von Clientadresse und User-Agent, in Challenges und Tokens gebunden, sodass keines von beiden von einem anderen Client wiederverwendet werden kann.

Forward-Auth
: Einsatzmodus, in dem ein Eingangsproxy Sibuna fragt, ob eine Anfrage zugelassen (`200`)

oder mit einer Challenge belegt, abgelehnt oder begrenzt werden soll (`401`, `403`, `429`), und die Anfrage selbst an den Ursprungsserver weiterleitet.

Gate
: Die Betriebsform mit Proof-of-Work-Zulassung, Sitzungen, Regeln, Reputation, GCRA-Limits und Sperren, aber ohne Anwendungsprüfung (`--gate`).

GCRA
: Der Generic Cell Rate Algorithm: eine theoretische Ankunftszeit pro Client, die einen Burst von N und danach eine Anfrage pro Emissionsintervall zulässt, ohne Fenstergrenzen.

Hashcash
: Stufe eins: eine Nonce finden, sodass SHA-256 über die Challenge, einen Doppelpunkt und die dezimale Nonce b führende Nullbits hat. Im Erwartungswert 2^b Versuche, verifiziert mit einem einzigen Hash.

Honeypot
: Ein unsichtbarer Link zu `/__sibuna/honeypot`; ein Client, der ihm folgt, wird gesperrt, und ein Vorfall mit der Reputation -100 wird aufgezeichnet.

Idle-Reaper
: Ein Thread, der Verbindungen schließt, die länger als `--idle-timeout` inaktiv waren, sodass ein langsamer Client keinen Verbindungsthread blockieren kann.

Zwischenseite
: Die eingebettete HTML-Seite, die anstelle einer geschützten Seite ausgeliefert wird; sie holt eine Challenge, löst sie in einem Web Worker, sendet die Lösung und lädt neu.

JA4
: Ein von FoxIO definierter Fingerabdruck aus 36 Zeichen dafür, wie ein Client eine TLS-Verbindung öffnet: Protokoll, TLS-Version, Servername, Anzahl der Cipher Suites und Erweiterungen, ALPN sowie Hashes der sortierten Cipher Suites und Erweiterungen. Sibuna berechnet ihn nicht; es liest den Wert, den ein vertrauenswürdiger Eingangsproxy in dem mit `--ja4-header` benannten Header weiterleitet, und Regeln gleichen ihn mit dem Feld `ja4` ab (SID 0014). Er hat nichts mit dem Fingerabdruck des Clients zu tun, der in Challenges und Tokens gebunden ist.

BLAKE3 mit Schlüssel
: Die Pseudozufallsfunktion hinter Sibunas Schlüsselplan, den Challenge-Tags, Sitzungstags, Fingerabdrücken und Challenge-Nonces.

MAC-Token
: Das Standard-Sitzungscookie: eine 40-Byte-Nutzlast (Version, Arbeitsstufe, Zeitstempel, Ablauf, Regelhash, Fingerabdruck) und ein 16-Byte-Tag aus BLAKE3 mit Schlüssel, verifiziert in konstanter Zeit.

Arbeitsstufe
: Das Verfahren und die Arbeitsbits, die der Inhaber einer Sitzung tatsächlich gelöst hat, getragen im Token; eine Route lässt eine Sitzung nur zu, wenn deren Stufe die Anforderung der Route erreicht.

Öffnung
: In einem Nachweis sequenzieller Arbeit ein Blattlabel plus die Geschwisterlabels entlang seines Pfads zur Wurzel; der Verifizierer berechnet den Pfad neu und vergleicht mit dem Commitment.

Proof of Sequential Work (PoSW)
: Stufe zwei: der Hash-Graph nach Cohen-Pietrzak, dessen Labels von allen früheren Labels abhängen. Er erzwingt $2^{n+1} - 1$ sequenzielle Hashes, unabhängig davon, wie viele Kerne der Beweiser hat. Verifiziert mit $t(n + 1)$ Hashes.

Reputations-Trie
: Der 128-Bit-Radix-Trie mit `allow-`, `deny-` und `challenge`-Präfixen aus der Richtliniendatei und aus der Tabelle `ip_reputation`.

Robin-Hood-Verbrauchsmenge
: Die Open-Addressing-Tabelle fester Kapazität für gelöste Challenge-Tags, die verhindert, dass eine gültige Lösung zweimal eingereicht wird.

Regelhash
: Ein 64-Bit-Hash des Namens der Regel, die eine Challenge verlangt hat, getragen in der Challenge, im Token und im Header `X-Sibuna-Rule-Hash`.

Shield
: Die Standard-Betriebsform: Gate plus die semantische Firewall (`--shield`).

SID
: Ein Sibuna-Discussion-Dokument: ein Entwurfsdokument im Stil eines Fachartikels unter `docs/sid/records` mit den Annahmen, Beweisen und Messungen hinter einem Teil des Systems.

Betriebsform
: Eine der beiden Produktformen, Gate oder Shield; Speicherung und Clusterbildung sind für beide optional.

WEIGH
: Eine Regelaktion, die einen vorzeichenbehafteten Wert zur Bewertung einer Anfrage addiert, statt zu entscheiden; Summen an oder über den Schwellen führen zu einer Challenge mit zusätzlichen Arbeitsbits oder zur Ablehnung.

Zaxonlite
: Die eingebettete SQLite-Replikations-Engine (Multi-Paxos) aus SID 0005, in SID 0011 durch Speicherung in SQLite und PostgreSQL ersetzt; ihre Cluster-Optionen brechen den Start jetzt mit `RemovedOption` ab.

Literaturverzeichnis

- Aho, A. V., und 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 (der Generic Cell Rate Algorithm).
- Back, A. „Hashcash: a denial of service counter-measure.“ Technischer Bericht, 2002.
- Blocki, J., Lee, S., und 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*. Dissertation, University of Waterloo, 1986.
- Cohen, B., und Pietrzak, K. „Simple proofs of sequential work.“ *EUROCRYPT*, 2018.
- Dwork, C., und Naor, M. „Pricing via processing or combatting junk mail.“ *CRYPTO*, 1992.
- Fiat, A., und Shamir, A. „How to prove yourself: practical solutions to identification and signature problems.“ *CRYPTO*, 1986.
- Fielding, R., Nottingham, M., und Reschke, J. *HTTP Semantics*, RFC 9110, und *HTTP/1.1*, RFC 9112. IETF, 2022.
- FoxIO. *JA4+ Network Fingerprinting*. https://github.com/FoxIO-LLC/ja4, 2023 (der JA4-Fingerabdruck von TLS-Clients).
- Hanson, N. „libinjection: SQL injection detection by tokenization.“ Black Hat USA, 2012.
- Juels, A., und 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; und „Paxos made simple.“ *ACM SIGACT News* 32(4), 2001.
- Mahmoody, M., Moran, T., und Vadhan, S. „Publicly verifiable proofs of sequential work.“ *Innovations in Theoretical Computer Science (ITCS)*, 2013.
- O'Connor, J., Aumasson, J.-P., Neves, S., und Wilcox-O'Hearn, Z. *BLAKE3: one function, fast everywhere*. Spezifikation, 2020.
- Vyukov, D. „Bounded MPMC queue.“ 1024cores.net, 2011 (der sequenzmarkierte Ring der Vorfallswarteschlange).
- Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J., und Zhu, H. „Hyperscan: a fast multi-pattern regex matcher for modern CPUs.“ *NSDI*, 2019.
- Sibuna Discussions 0001 bis 0014, `docs/sid/records`, 2026: Prozess, Grundarchitektur, deklarative Richtlinien-Engine, semantische Prüfung, Speicherarchitektur von Zaxonlite, mathematische Grundlagen, Verwaltungsoberfläche der Konsole, Überwachung des Datenverkehrs von KI-Bots, Chunked-Request-Bodies, die Portierung nach Rust, Speicherung über sqlx, ausgehendes TLS für die Konsole über rustls, Konsolenspeicher und vertrauenswürdige JA4-Fingerabdrücke vom Eingangsproxy.

Stichwortverzeichnis

A
Aho-Corasick-Automat, **154**
Arbeitsstufe, **155**

B
Betriebsform, **156**
BLAKE3 mit Schlüssel, **155**

C
Challenge-Datensatz, **154**

E
Edge, **154**
Engine-Slot, **154**

F
Fingerabdruck, **154**
Forward-Auth, **154**

G
Gate, **155**
GCRA, **155**

H
Hashcash, **155**
Honeypot, **155**

I
Idle-Reaper, **155**

J
JA4, **155**

K
Kampagne, **154**

M
MAC-Token, **155**

O
Öffnung, **155**

P
Proof of Sequential Work (PoSW), **156**

R
Regelhash, **156**
Reputations-Trie, **156**
Robin-Hood-Verbrauchsmenge, **156**

S
Schwierigkeit, Arbeitsbits, **154**
Shield, **156**
SID, **156**
Sperrtabelle, **154**

W
WEIGH, **156**

Z
Zaxonlite, **156**
Zulassung, **154**
Zwischenseite, **155**