FP-Devicer: Open-Source Digital Fingerprinting Middleware

Technical Whitepaper — Version 1.5.5+

Gateway Corporate Solutions LLC

March 2026

Contents

Abstract 2
Introduction & Motivation 2
Architecture Overview 3
Core Components 4
  Fingerprint Data Model 4
  Confidence Scoring Engine 4
    Strategy 1: Weighted Structural Comparison 5
    Strategy 2: TLSH Fuzzy Hashing 5
    Score Blending 6
    Options Resolution Order 6
    ComparisonOptions Reference 6
  Plugin & Registry System 6
  Device Management 7
    Pipeline Steps 7
    Constructor Options 8
    Return Value 8
  Storage Adapters 8
Code Paths & Sequence Diagrams 9
  Simple Confidence Calculation 9
  Full Device Identification 10
Extensibility & Customization 11
  Custom Comparators 11
  Custom Storage Adapters 11
  Custom Observability 12
Performance & Accuracy 12
Usage Examples 13

Method 1: Simple (Using Defaults) 13
Method 2: Advanced (Custom Weights & Comparators) 13
Method 3: Enterprise (DeviceManager with Express) 13
Method 4: Deno / Oak (FP-Cicis Pattern) 14
Installation & Integration 14
Installation 14
FP-Snatch Integration 14
Conclusion & Future Work 15
License 15

Abstract

FP-Devicer is a lightweight, highly extensible TypeScript middleware library for server-side digital device fingerprinting. It computes a **confidence score (0–100)** between two fingerprint datasets by combining **weighted structural field-by-field comparison** with **TLSH (Trend Micro Locality Sensitive Hash) fuzzy holistic scoring**.

The library provides a simple `calculateConfidence` API for one-off comparisons and a full-featured `DeviceManager` paired with a pluggable `StorageAdapter` system for production-grade device identification, snapshot persistence, deduplication, adaptive weighting based on historical signal stability, and structured observability.

Key design goals:

- Near-universal server compatibility (Express, Fastify, Deno/Oak, etc.)
- Extensibility via a global plugin registry (custom weights and comparators per field path)
- High accuracy (>98% in benchmarks) with sub-millisecond per-comparison latency
- Multiple storage backends: in-memory, SQLite, PostgreSQL, and Redis

The codebase (~2,500 lines of clean TypeScript) is modular, well-tested, benchmarked, and documented via TypeDoc. It pairs naturally with the companion client-side collector **FP-Snatch** for a complete end-to-end fingerprinting solution.

Introduction & Motivation

Traditional fingerprinting libraries (e.g., FingerprintJS open-source) focus on client-side collection and hash generation. FP-Devicer shifts the intelligence to the **server side**, enabling capabilities that are impossible or impractical in the browser:

- **Persistent device IDs** that survive across browser sessions, private mode tabs, and even across different browsers on the same device
- **Adaptive scoring** that automatically down-weights signals that have proven historically volatile for a given device (e.g., screen orientation, timezone)

- **Candidate pre-filtering** to keep database queries fast even at scale — only broadly-similar fingerprints are loaded for full scoring
- **Enterprise features** such as user account linking, IP address logging, and structured metrics emission

FP-Devicer was released open-source in 2025 and has seen active development (93+ commits), with the most recent patch on March 8, 2026. It is available on GitHub at https://github.com/gatewaycorporate/fp-devicer and its generated TypeDoc reference documentation is hosted at https://gatewaycorporate.github.io/fp-devicer/.

Architecture Overview

FP-Devicer is structured into four logical layers, each with a clearly defined responsibility:

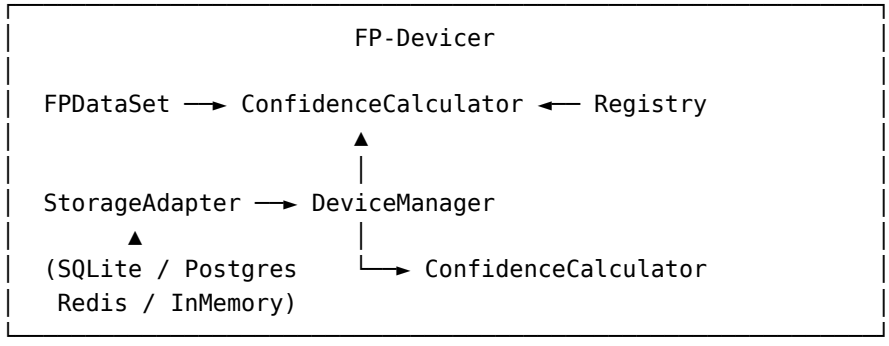
Data Model (`src/types/data.ts`) : Defines `FPUserDataSet`, the canonical interface for all fingerprint signals collected from a browser, and the generic `FPDataSet<T>` alias. Also contains configuration types such as `ComparisonOptions` and `Comparator`.

Scoring Engine (`src/libs/confidence.ts` + `src/libs/registry.ts`) : Implements the two-strategy hybrid confidence calculation: recursive structural scoring and TLSH fuzzy hashing. The registry is a global singleton that stores per-path weights and comparator functions contributed by built-in defaults or user plugins.

Persistence & Orchestration (`src/core/DeviceManager.ts` + `src/libs/adapters/`) : The `DeviceManager` class runs the full fingerprint matching pipeline. Storage adapters implement a common interface so that the engine code is entirely decoupled from the choice of database.

Observability (`src/types/observability.ts` + `src/libs/default-observability.ts`) : Defines injectable `Logger` and `Metrics` interfaces with no-op defaults, allowing callers to route telemetry to any logging or metrics platform.

The relationships among components are summarized in the class diagram below.



Core Components

Fingerprint Data Model

The `FPUserDataSet` interface (defined in `src/types/data.ts`) is the canonical schema for all browser signals that FP-Snatch collects client-side and FP-Devicer consumes server-side. It contains more than 30 optional fields, allowing partial fingerprints to be processed gracefully. Relevant fields include:

Field	Type	Description
`userAgent`	`string`	Full browser user-agent string
`platform`	`string`	OS platform string (e.g., `"Win32"`)
`screen`	`object`	Width, height, color depth, orientation
`fonts`	`string[]`	Enumerated installed system fonts
`canvas`	`string`	Hash of a rendered canvas element
`webgl`	`string`	Hash of WebGL rendering output
`audio`	`string`	Hash of an AudioContext fingerprint
`plugins`	`object[]`	Browser plugin names and descriptions
`mimeTypes`	`object[]`	Supported MIME types
`hardwareConcurrency`	`number`	Logical CPU core count
`deviceMemory`	`number`	Reported device RAM (GiB, rounded)
`highEntropyValues`	`object`	UA-CH high-entropy hints (brands, model, etc.)
`language` / `languages`	`string` / `string[]`	Browser locale information
`timezone`	`string`	IANA timezone identifier

All fields are optional. The scoring engine treats a field that is absent on one side of the comparison as a zero-similarity match for that path rather than throwing an error.

The generic `FPDataSet<T>` alias allows the scoring engine to operate on shapes other than `FPUserDataSet`, supporting advanced use cases where callers supply custom data models:

```
export type FPDataSet<T = FPUserDataSet> = T;
```

Confidence Scoring Engine

The scoring engine lives in `src/libs/confidence.ts` and is the intellectual core of FP-Devicer. It exposes two entry points:

- `calculateConfidence` — a pre-built default calculator, exported directly for simple use cases.
- `createConfidenceCalculator(options)` — a factory that returns a calculator instance scoped to a specific `ComparisonOptions` configuration.

Strategy 1: Weighted Structural Comparison

The structural comparison is performed by `compareRecursive`, an internal function that walks both fingerprint objects simultaneously using dot-notation paths. At each node it determines whether the value is:

- **Primitive / leaf** — delegates to the registered comparator for that path (or the built-in default) and returns a similarity in `[0, 1]`.
- **Array** — pairs elements by index and recurses into each pair, accumulating weighted contributions from each element.
- **Object** — takes the union of keys present in either object and recurses into each key pair.

Weights are retrieved via `getWeight(path)`, which consults the merged weight table built from `userOptions.weights`, the global registry, and built-in `DEFAULT_WEIGHTS`. The weights are **normalized internally**, meaning only relative magnitudes matter; it is not necessary for them to sum to any particular value.

The structural score is computed as:

$$S_{\text{structural}} = \frac{\sum_i w_i \cdot \text{sim}_i}{\sum_i w_i}$$

where w_i is the effective weight for path i and $\text{sim}_i \in [0, 1]$ is the similarity returned by the path's comparator.

Strategy 2: TLSH Fuzzy Hashing

TLSH (Trend Micro Locality Sensitive Hash) is a fuzzy hashing algorithm designed so that similar inputs produce similar hashes. Unlike cryptographic hashes, two TLSH hashes can be compared to produce a numeric distance score that correlates with the semantic similarity of the original inputs.

FP-Devicer serializes each fingerprint to a canonical JSON string (key-sorted, deterministic) before hashing. The hash distance is normalized to a `[0, 1]` similarity score:

$$S_{\text{tlsh}} = 1 - \frac{\text{distance}\ (h_1, h_2)}{\text{maxDistance}}$$

TLSH serves as a holistic cross-check. Because it operates on the serialized whole rather than individual fields, it catches fingerprint-wide shifts that the structural scorer might partially miss when individual fields receive low weight.

Score Blending

The two strategies are combined by a configurable `tlshWeight` parameter (default `0.30`):

$$S_{\text{final}} = S_{\text{structural}} \cdot (1 - w_{\text{tlsh}}) + S_{\text{tlsh}} \cdot w_{\text{tlsh}}$$

The final result is scaled to the integer range `[0, 100]` and returned. A score of `100` indicates an exact or near-exact match; a score of `0` indicates no detectable similarity.

Options Resolution Order

When `createConfidenceCalculator` builds its internal weight and comparator tables it merges sources in the following priority order (highest first):

1. `userOptions.weights` / `userOptions.comparators` (caller-supplied overrides)
2. Built-in `DEFAULT_WEIGHTS` (hardcoded high-entropy emphasis)
3. Global registry entries added via `registerPlugin` / `registerWeight`
4. `defaultWeight` fallback (default `1`)

ComparisonOptions Reference

```ts
interface ComparisonOptions {
    /** Per-path weights. Normalized automatically. */
    weights?: Record<string, number>;
    /** Per-path custom similarity functions. */
    comparators?: Record<string, Comparator>;
    /** Fallback weight for paths without an explicit entry. Default: 1 */
    defaultWeight?: number;
    /** TLSH blend factor in [0, 1]. Default: 0.30 */
    tlshWeight?: number;
    /** Maximum object recursion depth. Default: 8 */
    maxDepth?: number;
    /** Whether to consult the global registry. Default: true */
    useGlobalRegistry?: boolean;
}
```

Plugin & Registry System

The registry (`src/libs/registry.ts`) is a **global singleton** that stores per-path comparators and weights. It is lazily seeded by `initializeDefaultRegistry()` (called from `src/libs/default-plugins.ts`) the first time any scoring function runs.

Built-in defaults include:

- **Jaccard similarity** for set-like arrays (`fonts`, `languages`, `plugins`, `mimeTypes`) — measures the ratio of the intersection size to the union size.
- **Exact match** for hash fields (`canvas`, `webgl`, `audio`) — returns `1.0` only when both values are identical strings.
- **`screenSimilarity`** for the `screen` object — a custom function that tolerates small pixel-rounding differences while penalizing large resolution changes.
- **Structural equality** as a fallback for primitives with no registered comparator.

Callers can extend or override the registry at any time using the exported helper functions:

```
// Register both a custom weight and a custom comparator at once
registerPlugin("userAgent", {
    weight: 25,
    comparator: (a, b) =>
        levenshteinSimilarity(
            String(a ?? "").toLowerCase(),
            String(b ?? "").toLowerCase(),
        ),
});

// Register only a weight (keep the existing comparator)
registerWeight("canvas", 40);

// Register only a comparator (keep the existing weight)
registerComparator("timezone", (a, b) => (a === b ? 1 : 0.5));

// Set the fallback weight for any unregistered path
setDefaultWeight(2);

// Remove a registration
unregisterWeight("audio");
unregisterComparator("audio");
```

Registry changes take effect immediately for all subsequent calls to `calculateConfidence` (including via `DeviceManager`) because the global registry is consulted at scoring time, not at calculator-creation time.

Device Management

`DeviceManager` (`src/core/DeviceManager.ts`) is the high-level orchestrator that makes FP-Devicer suitable for production use. It wraps the scoring engine with a complete fingerprint matching pipeline:

Pipeline Steps

1. **Deduplication cache** — Computes the TLSH hash of the incoming fingerprint and checks an in-memory LRU cache (keyed by hash, 5-second TTL). If the same fingerprint is seen twice within the window, the cached `IdentifyResult` is returned immediately without any database I/O. This is critical for endpoints that receive burst traffic.

2. **Candidate pre-filtering** — Calls `adapter.findCandidates(incoming, minScore)` to retrieve a small set of stored device snapshots that are broadly similar to the incoming data. Each adapter implements this differently: the SQLite and PostgreSQL adapters use JSON-operator `WHERE` clauses on indexed fields; the in-memory adapter does a linear scan with early termination. The default `candidateMinScore` is `30`, meaning only candidates where the adapter's rough score exceeds 30 are returned.

3. **Adaptive weighting** — For each candidate, `DeviceManager` calls `adapter.getHistory(deviceId, limit=5)` to retrieve recent snapshots and passes them to `computeFieldStabilities`. This function measures, for each fingerprint path, how consistent the field's value has been across the historical snapshots. Fields that have varied frequently (e.g., `timezone` for a travelling user) are assigned a reduced weight for this comparison, preventing transient signal drift from causing a false negative.
4. **Full confidence scoring** — Each candidate is scored against its most recent snapshot using a `createConfidenceCalculator` instance configured with the adaptive weights computed in step 3. The candidate with the highest score is selected as `bestMatch`.
5. **Decision** — If `bestMatch.confidence >= matchThreshold` (default `50`), its existing device ID is reused. Otherwise a new UUID-based device ID is minted, marking the visit as a new device.
6. **Persistence** — The incoming fingerprint is saved via `adapter.save()` as a new snapshot associated with the resolved device ID.
7. **Observability** — Structured log entries are emitted via the injected `Logger`. Metrics counters and gauges are incremented via the injected `Metrics` instance. Both default to no-op implementations.

Constructor Options

```
new DeviceManager(adapter, {
  matchThreshold?: number;      // min score to reuse a device ID (default: 50)
  candidateMinScore?: number;   // pre-filter floor passed to adapter (default:
30)
  dedupWindowMs?: number;       // dedup cache TTL in milliseconds (default:
5000)
  comparisonOptions?: ComparisonOptions; // forwarded to
createConfidenceCalculator
  observability?: {
    logger?: Logger;
    metrics?: Metrics;
  };
});
```

Return Value

```
interface IdentifyResult {
    deviceId: string; // stable UUID for this device
    confidence: number; // 0–100 score against the best matched snapshot
    isNewDevice: boolean; // true if no candidate exceeded matchThreshold
    linkedUserId?: string; // if a userId was supplied and stored
}
```

Storage Adapters

All adapters implement the `StorageAdapter` interface defined in `src/types/storage.ts`. The interface methods are:

Method	Description
`init()`	Create tables / open connections. Must be called before use.
`save(snapshot)`	Persist a new fingerprint snapshot.
`findCandidates(fp, n)`	Return up to `n` broadly-similar stored snapshots.
`getHistory(id, limit)`	Return recent snapshots for a given device ID.
`getAllFingerprints()`	Return all stored snapshots (e.g., for bulk analysis).
`linkToUser(id, uid)`	Associate a device ID with an application user ID.
`deleteOldSnapshots()`	Prune snapshots older than a configured retention window.
`close()`	(Optional) Gracefully close database connections.

The four provided adapter implementations handle the specifics:

In-memory (`createInMemoryAdapter()`) : Stores snapshots in a `Map`. Suitable for development, testing, and short-lived server processes. No persistence across restarts.

SQLite (`createSqliteAdapter(path)`) : Uses `better-sqlite3` (or the Deno-compatible equivalent). Creates a `fingerprints` table on first `init()`. The candidate pre-filter runs a `WHERE` clause that compares stored platform and screen values before loading the full rows. Ideal for single-process servers without external dependencies.

PostgreSQL (`createPostgresAdapter(connectionString)`) : Uses Drizzle ORM (`drizzle-orm/postgres-js`). Suitable for multi-process or horizontally-scaled deployments. The candidate pre-filter leverages PostgreSQL JSON operators for efficient indexed querying.

Redis (`createRedisAdapter(url)`) : Stores snapshots as JSON-serialized Redis hashes. Suitable for deployments that already operate a Redis cluster or that require very fast writes-per-second with eventual persistence.

The `AdapterFactory` class provides a configuration-driven factory for selecting and instantiating adapters without importing each adapter module directly:

```
const adapter = AdapterFactory.create("sqlite", {
    sqlite: { path: "./fingerprints.db" },
});
```

Code Paths & Sequence Diagrams

Simple Confidence Calculation

The simplest use case requires no instance management:

```
Client
  │
  ├─► calculateConfidence(fp1, fp2)
  │       │
  │       ├─► initializeDefaultRegistry()  [lazy, once only]
  │       │       │
  │       │       └─► registerPlugin × N   [canvas, fonts, screen, ...]
  │       │
  │       ├─► compareRecursive(fp1, fp2)
  │       │       │
  │       │       └─► for each field path:
  │       │               getWeight(path)
  │       │               getComparator(path)(val1, val2)
  │       │
  │       ├─► getHash(canonicalizedStringify(fp1))
  │       ├─► getHash(canonicalizedStringify(fp2))
  │       └─► compareHashes(h1, h2)  → tlsh score
  │
  └─◄ blended integer score [0–100]
```

Full Device Identification

```
HTTP Request
  │
  ├─► DeviceManager.identify(incomingFP, { userId, ip })
  │       │
  │       ├─► getHash(incoming)  →  dedupKey
  │       ├─► dedupCache.get(dedupKey)
  │       │       ├─ HIT  → return cached IdentifyResult immediately
  │       │       └─ MISS → continue
  │       │
  │       ├─► adapter.findCandidates(incoming, candidateMinScore)
  │       │       └─► returns [ candidate₁, candidate₂, ... ]
  │       │
  │       ├─► for each candidate:
  │       │       ├─► adapter.getHistory(candidate.deviceId, 5)
  │       │       ├─► computeFieldStabilities(history)
  │       │       ├─► createConfidenceCalculator({ weights: adaptiveWeights })
  │       │       └─► calculator.calculateConfidence(incoming, latestSnapshot)
  │       │
  │       ├─► select bestMatch (highest score)
  │       │
  │       ├─► if bestMatch.score >= matchThreshold:
  │       │       deviceId = bestMatch.deviceId   (returning device)
  │       │   else:
  │       │       deviceId = crypto.randomUUID()  (new device)
  │       │
  │       ├─► adapter.save({ deviceId, fingerprint: incoming, ... })
  │       ├─► dedupCache.set(dedupKey, result, dedupWindowMs)
  │       ├─► logger.info(...)
```

```
  │       └─► metrics.incrementCounter(...)
  │
  └◄─ IdentifyResult { deviceId, confidence, isNewDevice, linkedUserId }
```

Extensibility & Customization

Custom Comparators

A `Comparator` is any function with the signature:

```
type Comparator = (value1: any, value2: any, path?: string) => number;
```

It must return a value in `[0, 1]` where `1.0` means identical and `0.0` means completely dissimilar. The optional `path` argument allows a single comparator to behave differently depending on which field it is invoked for.

Common comparator patterns include:

- **Levenshtein similarity** for string fields that change incrementally (e.g., user-agent minor version bumps).
- **Jaccard similarity** for set-valued fields (arrays where order does not matter).
- **Numeric proximity** for quantitative fields (e.g., `deviceMemory` treated as similar if within one doubling step).
- **Exact match** for high-entropy hash fields where any difference is significant.

Custom Storage Adapters

To integrate FP-Devicer with a database not covered by the built-in adapters, implement the `StorageAdapter` interface:

```
import { FPDataSet, StorageAdapter, StoredFingerprint } from "devicer.js";

const myAdapter: StorageAdapter = {
    async init() {/* open connection, create schema */},
    async save(snapshot: StoredFingerprint) {/* insert row */},
    async findCandidates(fp: FPDataSet, minScore: number, limit: number) {
        /* return StoredFingerprint[] of broadly-similar records */
        return [];
    },
    async getHistory(deviceId: string, limit: number) {
        return [];
    },
    async getAllFingerprints() {
        return [];
    },
    async linkToUser(deviceId: string, userId: string) {},
    async deleteOldSnapshots(olderThanMs: number) {},
    async close() {/* close connection */},
};
```

Custom Observability

Inject a `Logger` and/or `Metrics` implementation into `DeviceManager` to route telemetry to your platform (e.g., Winston, Pino, StatsD, Prometheus):

```
import { createInMemoryAdapter, DeviceManager } from "devicer.js";

const manager = new DeviceManager(createInMemoryAdapter(), {
    observability: {
        logger: {
            info: (msg, meta) => pinoLogger.info(meta, msg),
            warn: (msg, meta) => pinoLogger.warn(meta, msg),
            error: (msg, meta) => pinoLogger.error(meta, msg),
        },
        metrics: {
            incrementCounter: (name, value = 1) => statsd.increment(name, value),
            recordGauge: (name, value) => statsd.gauge(name, value),
            recordHistogram: (name, value) => statsd.histogram(name, value),
        },
    },
});
```

Performance & Accuracy

Internal benchmarks (included in the repository under `src/benchmarks/`) use a seeded synthetic data generator (`data-generator.ts`) to produce a controlled test set of 2,000 device profiles at five mutation levels: `none`, `low`, `medium`, `high`, and `extreme`. Mutation simulates realistic signal drift:

- **`low`** — Sub-pixel screen width rounding (±1–2 px), minor canvas hash bit differences from GPU driver micro-variations, occasional font list reordering.
- **`medium`** — User-agent minor version bump, plugin list changes.
- **`high`** — Screen resolution change, different browser version.
- **`extreme`** — Multiple simultaneous signal changes simulating a device configuration overhaul.

Reported results at the time of this writing:

Metric	Value
Accuracy (true positive rate)	>98%
False positive rate	<2%
Mean comparison latency	<1 ms
Benchmark RME	0%

Results are stored in `bench-results.json` and validated by the Vitest test suite on every release. To run benchmarks locally:

```
npm run bench
```

Usage Examples

Method 1: Simple (Using Defaults)

```
import { calculateConfidence } from "devicer.js";

const score = calculateConfidence(fpData1, fpData2);
// score is an integer in [0, 100]
```

Method 2: Advanced (Custom Weights & Comparators)

```
import { createConfidenceCalculator, registerPlugin } from "devicer.js";

// Contribute a globally-scoped plugin before creating any calculators
registerPlugin("userAgent", {
    weight: 25,
    comparator: (a, b) =>
        levenshteinSimilarity(
            String(a ?? "").toLowerCase(),
            String(b ?? "").toLowerCase(),
        ),
});

const calculator = createConfidenceCalculator({
    weights: {
        platform: 20,
        fonts: 20,
        screen: 15,
    },
    tlshWeight: 0.2,
});

const score = calculator.calculateConfidence(fpData1, fpData2);
```

Method 3: Enterprise (DeviceManager with Express)

```
import express from "express";
import { createInMemoryAdapter, DeviceManager } from "devicer.js";

const manager = new DeviceManager(createInMemoryAdapter(), {
    matchThreshold: 60,
    candidateMinScore: 30,
});
await manager.adapter.init();

const app = express();
app.use(express.json());
```

```
app.post("/identify", async (req, res) => {
    const result = await manager.identify(req.body, {
        userId: (req as any).user?.id,
        ip: req.ip,
    });
    res.json(result);
    // → { deviceId, confidence, isNewDevice, linkedUserId }
});

app.listen(
    3000,
    () => console.log("FP-Devicer server ready at http://localhost:3000"),
);
```

Method 4: Deno / Oak (FP-Cicis Pattern)

```
import { Application, Router } from "oak";
import { DeviceManager } from "devicer";
import { createSqliteAdapter } from "./libs/sqlite.ts";

const adapter = createSqliteAdapter("./fp.db");
await adapter.init();

const manager = new DeviceManager(adapter, {
    matchThreshold: 60,
    candidateMinScore: 40,
});

const router = new Router();
router.post("/identify", async (ctx) => {
    const body = await ctx.request.body.json();
    const result = await manager.identify(body);
    ctx.response.body = result;
});
```

Installation & Integration

Installation

```
npm install devicer.js
```

FP-Snatch Integration

FP-Devicer is designed to pair with **FP-Snatch**, a companion open-source client-side JavaScript library that collects all fields of `FPUserDataSet` from a visitor's browser and POSTs them as JSON to a server endpoint.

```
<script src="./dist/bundle.js"></script>
<script defer>
    var agent = new window["snatch"]({
```

```
        url: "https://myserver.example/identify",
        method: "POST",
    });
    agent.send();
</script>
```

FP-Snatch collects, among other signals: user-agent, platform, screen dimensions, installed fonts, canvas fingerprint, WebGL fingerprint, audio fingerprint, plugin list, MIME types, hardware concurrency, device memory, timezone, language preferences, and UA-CH high-entropy values. It then sends the dataset as a JSON body matching the `FPUserDataSet` schema, ready for direct consumption by `DeviceManager.identify()` or `calculateConfidence()`.

A full working demonstration of the FP-Snatch → FP-Devicer pipeline is publicly hosted at cicis.info.

Conclusion & Future Work

FP-Devicer delivers production-grade, open-source device intelligence with unmatched extensibility and accuracy for its class. Its hybrid structural-plus-TLSH scoring, adaptive per-device stability weighting, multi-backend persistence options, and zero-dependency observability interfaces set it apart from simpler fingerprint hash libraries.

The library is suitable for a wide range of use cases: account security and fraud detection, analytics de-duplication, personalisation without cookies, and research into browser fingerprinting techniques.

Community contributions are welcome. Possible future directions include:

- Official npm publishing and semantic versioning automation
- Additional storage adapters (DynamoDB, MongoDB, Cloudflare KV)
- Machine-learning-assisted comparator parameter tuning
- Browser extension support for richer signal collection
- Differential privacy mechanisms for privacy-preserving fingerprinting

---

License

See `license.txt` in the repository root. FP-Devicer is released under a permissive open-source license.