castle

Published April 22, 2026 · Last revised April 22, 2026

Fingerprint harvesting in the bot ecosystem

bot management · fingerprinting

2026

Table of Contents

1. Executive Summary

This report analyzes the emergence of structured fingerprint harvesting within bot and automation ecosystems.

Research question

How are browser fingerprints collected, structured, and operationalized by bot-adjacent services, and what does this imply for impersonation and anti-detection risk?

Castle's Research Team conducted community monitoring, manual reverse engineering, and ecosystem-level measurement across 811 bot- and fraud-adjacent websites, of which 678 were reachable and analyzed.

We identified multiple services deploying client-side scripts that collect high-entropy fingerprinting signals including commercial anti-bot vendors signals. In several cases, these implementations:

- Segment fingerprint payloads by commercial anti-bot and anti-fraud products, including PerimeterX, Incapsula, Akamai, Adyen, and hCaptcha.
- Embed signal structures, challenge logic, and string-level artifacts derived from commercial anti-bot scripts. These characteristics are consistent with structured fingerprint harvesting rather than generic analytics or site-local fraud detection.

Within bot ecosystems, realistic fingerprints are treated as operational assets. Community discussions—and observed willingness to pay significant recurring fees for real-world fingerprint collection—indicate clear economic demand.

Harvested fingerprints reduce the cost of realism and increase the scalability of automation. Instead of synthetically spoofing isolated attributes (often inconsistently), operators can replay coherent device environments and vendor-aligned telemetry to:

- Reduce detection by bot and fraud systems,
- Run higher volumes of automated traffic while maintaining diversity across apparent client profiles. At the ecosystem level, 12.5 percent of analyzed bot- and fraud-adjacent websites deployed fingerprinting-related scripts under a conservative heuristic. While many scripts corresponded to analytics or defensive tooling, a subset demonstrated vendor-specific telemetry replication, cross-context signal collection, and patterns consistent with harvesting-oriented reuse.

For fraud and bot detection teams, the risk extends beyond attribute spoofing. It includes the replay of:

- Real device fingerprints,
- Vendor-aligned challenge outputs,
- Structurally valid payloads modeled after commercial anti-bot systems.

Controls that rely on static JavaScript fingerprinting scripts, stable payload schemas, or predictable signal definitions are more exposed to reverse engineering and replay. Fingerprint data must be treated as untrusted client-side input.

An effective defensive posture requires:

- Reducing the replay value of collected fingerprints,
- Introducing controlled variability in client-side logic and payload structure,
- Validating internal consistency across execution contexts,
- Combining fingerprint-derived signals with behavioral, network, and server-side telemetry. Importantly, fingerprint harvesting depends on access to clean, real-world device data. Bot-adjacent services can obtain such data from ordinary user traffic. Organizations should therefore assume that client-side telemetry executed in untrusted environments may be collected by adversaries. Limiting unnecessary exposure of production devices to automation-adjacent ecosystems — and reducing execution of third-party fingerprinting scripts through the use of content-blocking extensions (e.g., uBlock Origin) or privacy-focused browsers with anti-fingerprinting protections — can decrease the quality and reliability of data available for harvesting.

As fingerprint harvesting becomes operationalized within automation ecosystems, detection systems must remain effective under replay conditions rather than relying on the secrecy or stability of individual client-side signals.

Disclaimer

This report documents how certain bot-adjacent services target commercial anti-bot systems by harvesting fingerprinting signals. We do not assess whether these techniques are effective in practice, as modern detection systems rely on multiple layers beyond fingerprints, including network, behavioral, and server-side signals. More broadly, being targeted is a normal aspect of operating in the bot detection space: all major anti-bot vendors, including Castle, are subject to ongoing evasion attempts. The key differentiator is not whether a system is targeted, but how effectively it adapts to and mitigates these evolving techniques.

2. Background

2.1 Browser fingerprinting in fraud and bot mitigation

Browser fingerprinting is widely used in fraud prevention and bot mitigation systems. Modern web applications rely on client-side telemetry to distinguish legitimate users from automated traffic, detect account compromise, and enforce risk-based controls.

A fingerprint is typically derived from a combination of browser, device, rendering, and environment attributes exposed through JavaScript APIs. These attributes are aggregated into a profile of the client environment that can be evaluated for consistency, anomaly detection, or device recognition.

Depending on the defender's objectives, fingerprints are commonly used for:

- **Device identification**: recognizing a returning browser across sessions, including after cookie deletion or IP rotation.
- **Human fraud detection**: identifying anomalous account activity such as account takeover, payment fraud, or multi-account abuse by comparing observed device characteristics to historical baselines.
- **Bot detection**: identifying automation frameworks, anti-detect browsers, or instrumented environments by detecting inconsistencies, abnormal feature combinations, or timing anomalies. In all three use cases, fingerprint data is treated as one signal within a broader risk evaluation framework. It is not inherently authoritative, but it can materially influence access control decisions, step-up challenges, and transaction approvals.

This defensive reliance on client-side fingerprint telemetry creates an incentive for attackers to analyze, replicate, and potentially harvest those signals.

2.1.1 Client-side fingerprint collection workflow

A typical client-side fingerprinting workflow follows these steps:

1. A website loads a JavaScript snippet.
2. The script queries browser and device properties through exposed APIs (for example, `window` and `navigator`).

3. The resulting payload is transmitted to a collection endpoint for evaluation. The remainder of this section examines prior research showing that attackers have already moved beyond simple spoofing. In multiple ecosystems, browser fingerprints have been collected, packaged, and reused to bypass device-based security controls.

2.2 Prior research and related ecosystems

This report examines whether browser fingerprints are being collected, structured, and operationalized by bot-adjacent services. To frame this analysis, we review previously documented instances of fingerprint harvesting and reuse in criminal and automation contexts.

The cases below establish that:

- Browser fingerprints can be harvested at scale.
- They can be packaged and sold as standalone assets.
- They can be injected into automated environments to bypass device-based controls. These precedents inform the analysis conducted in later sections.

2.2.1 Genesis Marketplace: fingerprints as criminal assets

Research published by Shape Security (later acquired by F5) and subsequently reported on by F5 Labs documented the operations of Genesis Marketplace, a criminal marketplace specializing in the sale of compromised digital identities. A coordinated law enforcement operation later disrupted the marketplace's infrastructure.

Within the Genesis ecosystem, listings bundled stolen authentication material (such as credentials or active session artifacts) together with detailed browser and device fingerprint data.

The objective of these bundles was to enable account access that appeared consistent with a victim's historical activity. By replaying both authentication material and device characteristics, attackers could bypass controls relying on device recognition, fingerprint consistency, and anomaly detection.

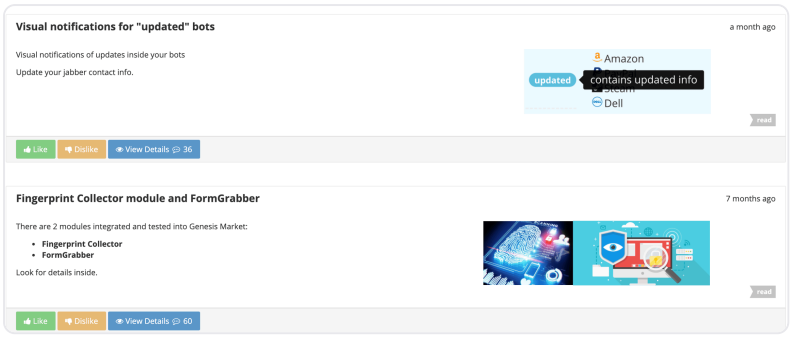


Figure 1: Genesis Marketplace interface showing the "Fingerprint collector" module and inventory of compromised browser profiles ("bots"), as documented by F5 Labs.

According to F5's analysis, Genesis Marketplace advertised approximately 323,000 compromised browser environments ("bots") at the time of reporting. Each listing represented a harvested user environment, including fingerprint data.

To operationalize these profiles, Genesis provided a custom Chromium-based browser extension (.crx) that injected victim-specific browser and device attributes at runtime. When combined with residential or geo-matched proxy infrastructure, this allowed attackers to approximate both the device fingerprint and network location of the victim.

Europol's post-takedown reporting confirmed that buyers were provided not only with stolen data, but also with tooling to mimic the victim's browser environment to avoid triggering:

- Detection of new or unrecognized devices
- Fingerprint mismatches
- Anomalous login locations Genesis Marketplace demonstrated that browser fingerprints can be monetized as standalone assets and replayed to bypass device-based security controls. This case establishes that fingerprint harvesting and replay are viable at scale.

2.2.2 Bablosoft ecosystem: integrated fingerprint collection and replay

Commercial browser automation platforms have incorporated fingerprint manipulation and replay as built-in capabilities. Bablosoft, through its BrowserAutomationStudio (BAS) ecosystem, provides tooling that combines automation with fingerprint switching and injection.

This ecosystem is relevant because it shows how fingerprint collection, storage, and replay can be integrated directly into automation workflows.

2.2.3 Prior reporting: large-scale fingerprint collection

Research published by Group-IB documented a campaign in which Bablosoft-related fingerprinting code (`clientsafe.js`) was injected into more than 115 compromised e-commerce websites. The script collected high-entropy browser and device attributes and transmitted them to Bablosoft-controlled infrastructure, including `https://customfingerprints[.]bablosoft[.]com/save`

This reporting established that Bablosoft tooling had been used to harvest real-world fingerprints from third-party traffic at scale.

2.2.4 Independent verification and continued deployment

Castle's Research Team conducted independent analysis to determine whether Bablosoft fingerprint collection infrastructure remains active.

As of February 2026, Bablosoft services remain publicly accessible. A query on urlscan.io for `fingerprints[.]bablosoft[.]com` returned more than 1,500 indexed results.

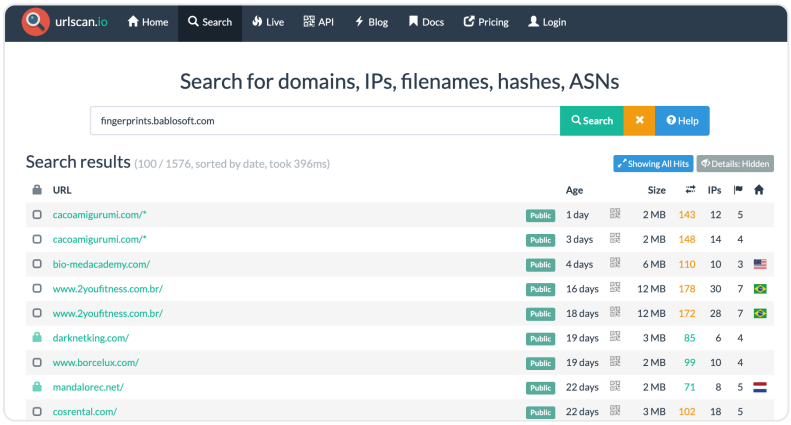


Figure 2: urlscan.io search results for fingerprints[.]bablosoft[.]com, showing more than 1,500 indexed scans referencing Bablosoft fingerprint collection infrastructure.

We identified live domains loading Bablosoft's `clientsafe.js`, including `coamigurumi[.]com`.

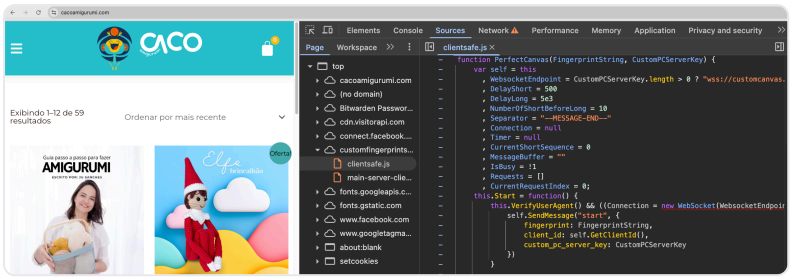


Figure 3: Live website (`cacoamigurumi[.]com`) loading Bablosoft's `clientsafe.js` fingerprinting script, confirming active deployment on third-party domains.

These findings confirm that Bablosoft-related fingerprint collection infrastructure remains operational.

2.2.5 Script analysis and PerfectCanvas mechanism

Beyond confirming deployment, we analyzed currently served Bablosoft fingerprinting scripts. The scripts continue to reference logic consistent with the **PerfectCanvas** mechanism previously described in public reporting.

Bablosoft documents the PerfectCanvas feature on their wiki. It is designed to bypass canvas-based antifraud controls by separating fingerprint generation from the automation client. According to their documentation:

> We are offering approach, which can bypass even the most sophisticated canvas-based antifraud systems. The idea is following:
>
> - Render canvas on remote machine.
> - Send canvas data to your PC.
> - Replace canvas data inside browser.

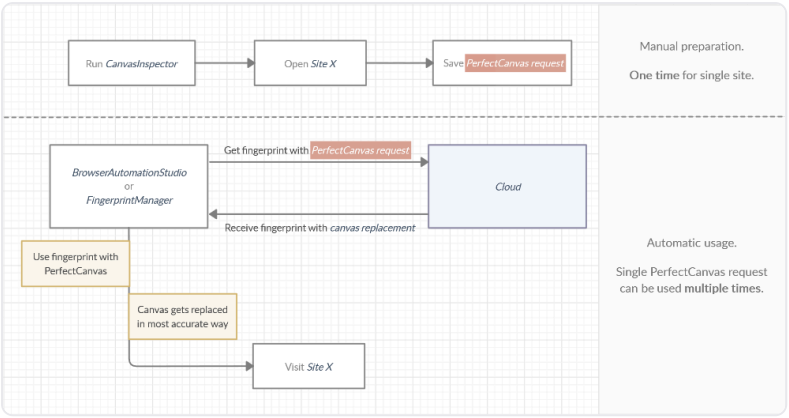


Figure 4: Bablosoft documentation diagram illustrating the PerfectCanvas workflow, where canvas rendering is performed on a remote device and injected into the local browser session.

In this model, canvas output is generated on a remote device and injected into the automation browser, replacing locally rendered output. Instead of synthetically spoofing canvas output, which often introduces detectable inconsistencies, PerfectCanvas enables replay of outputs generated on real hardware.

The continued presence of this mechanism indicates ongoing support for high-fidelity fingerprint replay.

2.2.6 Fingerprint application within automation workflows

Bablosoft's ecosystem does not only support fingerprint collection. It also provides tooling to apply and replay collected fingerprints within automated sessions.

One such component is FingerprintSwitcher, a module integrated into Bablosoft's BrowserAutomationStudio environment. FingerprintSwitcher enables operators to inject stored or predefined browser fingerprints into automated browser instances, modifying exposed attributes to align with selected device profiles.

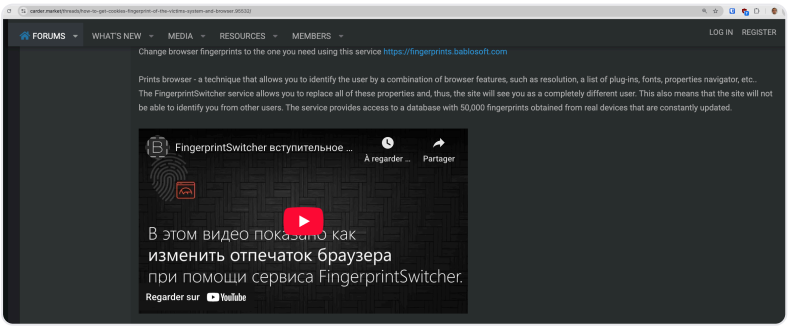


Figure 5: Carding forum post recommending Bablosoft's FingerprintSwitcher module for reducing detection in fraud and automation workflows.

Independent monitoring of carding forums shows FingerprintSwitcher being recommended to reduce detection during fraud operations. This indicates active adoption of fingerprint replay capabilities within fraud-focused communities.

This prior research establishes that fingerprint harvesting and replay are not theoretical constructs. They are implemented within operational tooling. The following sections examine whether similar structured harvesting behavior is observable across a broader set of bot-adjacent services.

3. Analysis

3.1 Indicators of demand for harvested fingerprint data

This report examines how browser fingerprints are collected and operationalized within bot ecosystems. Before analyzing specific harvesting implementations, we evaluated whether demand for realistic fingerprint data is visible in practice.

Evidence of demand for realistic fingerprint data is visible across both commercial services and bot-focused communities.

One illustrative example is the service impersonate[.]pro, which markets itself with the slogan “Impersonate like a pro” and advertises support for device impersonation using a “comprehensive TLS, HTTP/2, HTTP/3, and JavaScript fingerprint collection.”

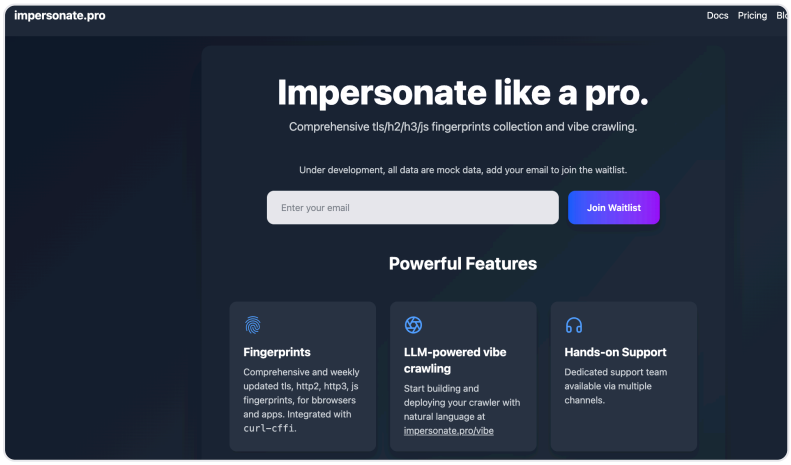


Figure 6: Marketing page of impersonate[.]pro advertising TLS and JavaScript fingerprint impersonation capabilities.

While services of this type do not disclose how fingerprints are sourced, their marketing framing treats device fingerprints as an input that can be collected, controlled, and optimized, rather than as a fixed property of a user's environment.

In parallel, we observed recurring discussions in Discord and Telegram communities where bot developers explicitly sought methods to collect fingerprints from real users.

In multiple Discord conversations, operators discussed embedding custom JavaScript on websites in order to gather fingerprint data. In some cases, this approach was explicitly framed as "monetizing traffic to collect fingerprints".

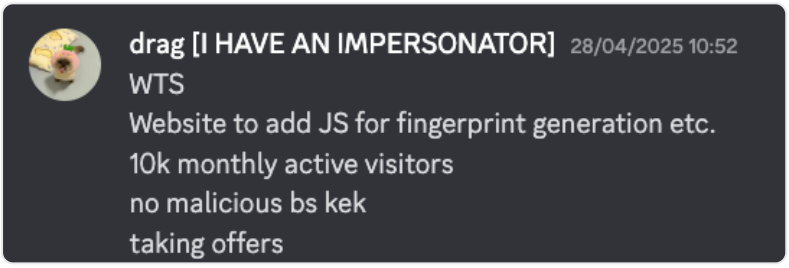


Figure 7: Discord discussion describing monetizing web traffic to collect browser fingerprints.

Other discussions focused on sourcing fingerprints directly from third parties. In one example, a user asked for a provider offering device fingerprints suitable for automation use cases.

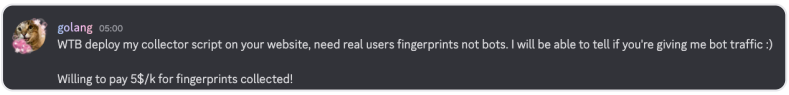


Figure 8: Forum discussions where operators express willingness to pay up to $5 per 1,000 real-world fingerprints.

In another exchange, a user seeking realistic fingerprints was advised to deploy a fingerprinting script and acquire traffic, including through paid advertising, in order to collect real-world browser data at scale.

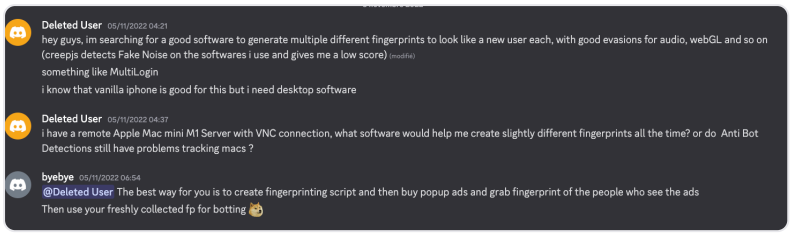


Figure 9: Discord exchange where a user seeking realistic fingerprints is advised to deploy fingerprinting scripts and acquire traffic, including via paid ads, to collect real-world browser data at scale.

Documented willingness to pay up to $5 for 1,000 real-world fingerprint indicates that browser fingerprints are treated as operational assets.

Castle's Research Team therefore shifted from ecosystem-level indicators of interest in realistic fingerprints (for example, forum discussions and marketing claims) to technical validation.

- Are bot- and fraud-adjacent services deploying client-side scripts that collect browser fingerprints in a manner consistent with large-scale harvesting rather than site-specific analytics or defensive security use cases? To address this question, we moved from anecdotal observations to direct technical analysis. This included:
- Identifying domains referenced by operators claiming to collect fingerprints.
- Manually inspecting client-side JavaScript behavior on those domains.
- Expanding the scope to an ecosystem-level measurement of bot-, proxy-, and automation-adjacent websites. The following section describes the initial investigative lead that grounded this analysis in observable artifacts.

3.2 Initial investigative lead: operator claims of fingerprint collection

The investigation was initiated following a public statement from the operator of cybertemp[.]xyz, a disposable email service active in bot-focused communities. The operator stated that cybertemp[.]xyz and several related websites under their control were used to collect real user browser fingerprints.

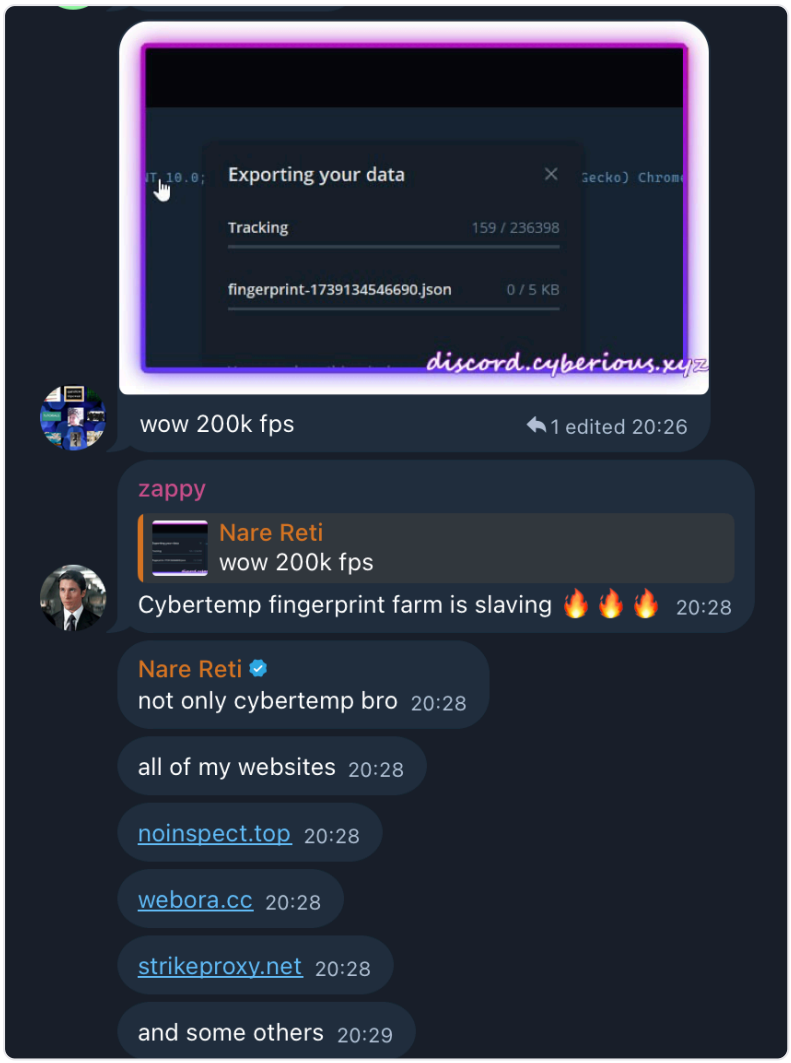


Figure 10: Telegram message from operator of cybertemp[.]xyz claiming use of multiple websites to collect real browser fingerprints.

This claim is operationally significant because the operator explicitly described collecting real device fingerprints from live user traffic, with the stated purpose of reuse. The traffic received by these services was framed as a source of clean, real-world fingerprint data rather than as input for analytics or defensive controls.

While the initial claim alone does not establish that harvesting was occurring, it links previously observed ecosystem demand to identifiable domains and infrastructure.

Notably, after the initial phase of this research was conducted, the same operator posted an additional message in a Discord community offering access to fingerprint-related infrastructure and datasets. Here, “fp” refers to browser fingerprints:

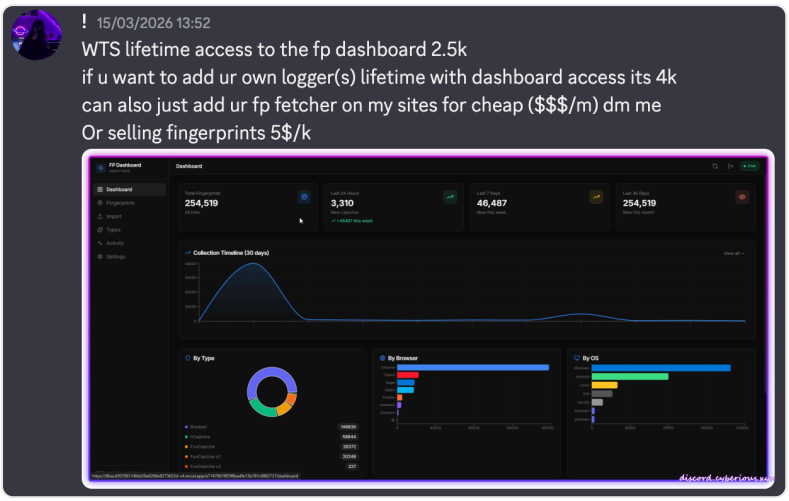


Figure 11: Discord message from operator of cybertemp[.]xyz selling access to his fingerprinting dashboard so people can add their own device fingerprinting collector.

This message is significant for two reasons:

- It demonstrates direct monetization of collected fingerprints, including per-volume pricing.
- It suggests the existence of dedicated infrastructure (“fp dashboard”) for managing and distributing fingerprint data, as well as a model for embedding third-party collectors (“loggers”) into existing traffic sources. Castle’s Research Team therefore conducted a technical review of cybertemp[.]xyz and related properties to determine whether client-side behavior was consistent with structured fingerprint harvesting.

The following section documents the results of that analysis.

3.3 Targeted technical validation of referenced domains

The Telegram claim described in the previous section (Figure 10) provided a concrete investigative lead. In that exchange, the operator of cybertemp[.]xyz asserted that cybertemp[.]xyz and several other websites under their control were used to collect real user browser fingerprints at scale.

3.3.1 Scope of technical review

Castle’s Research Team conducted a manual inspection of the websites referenced by the operator. The objective was to determine whether these domains deployed client-side JavaScript consistent with browser fingerprinting behavior.

For each website, we reviewed:

- JavaScript resources loaded during page initialization.
- Network requests triggered during page load.
- Access to browser attributes commonly used in fingerprinting, including properties of `navigator`, `screen`, Canvas APIs, WebGL APIs, and related feature-detection mechanisms. This phase was exploratory and qualitative. The goal was not to measure prevalence, but to validate whether the claim of fingerprint collection corresponded to observable technical behavior.

3.3.2 Observed fingerprinting implementations

Across multiple referenced domains, we observed JavaScript files exhibiting fingerprinting-related behavior. Several sites loaded identical or closely related scripts, indicating potential infrastructure reuse.

noinspect[.]top

- Observed script: fp-flame[.]vercel[.]app/_next/static/chunks/app/page-ce580fedf644df13.js?dpl=dpl_GH1GsZTKtcXyT6a5g3LfNX2Hyi4z **webora[.]cc**
- Observed inclusion of the same script: fp-flame[.]vercel[.]app/_next/static/chunks/app/page-ce580fedf644df13.js?dpl=dpl_GH1GsZTKtcXyT6a5g3LfNX2Hyi4z **cyberious[.]xyz**
- Observed script: preferencenail[.]com/sfp.js
- This script was blocked by EasyList at the time of analysis. We did not conclusively attribute it to the same operator as the other sites. **cybertemp[.]xyz**
- Observed script: cybertemp[.]xyz/_next/static/chunks/8022.6ccd48f583f01430.js?dpl=dpl_8TYCjpXRpumdRAQFKTW4D392LxoE These scripts accessed multiple browser and device properties commonly associated with fingerprinting, including:
- `navigator` attributes (for example, user agent, hardware concurrency).
- Screen characteristics.
- Rendering-related APIs such as Canvas and WebGL.

3.4 Ecosystem-level measurement methodology

3.4.1 Limitations of manual investigation

Manual inspection was sufficient to validate specific implementations and confirm collector reuse. It was not sufficient to assess ecosystem-wide patterns.

Specifically, manual review could not determine:

- The prevalence of fingerprinting scripts across bot-adjacent services
- The extent of collector reuse across unrelated domains
- Whether fingerprinting behavior was isolated or systematic To address these gaps, we conducted an ecosystem-level measurement of client-side fingerprinting behavior across a larger corpus of bot- and fraud-adjacent websites.

3.4.2 Dataset construction

Castle's Research Team constructed a corpus of **811 websites** associated with botting, automation, and fraud-adjacent use cases. Sources included:

- Domains identified during prior research, including disposable email providers and temporary phone number services.
- Services shared within Telegram and Discord communities focused on bot development and automation.
- Commercial offerings marketed toward automation workflows, including proxy providers, CAPTCHA solvers, and bot tooling platforms.
- Engagement manipulation services, such as providers of fake followers, likes, or views on social networks and streaming platforms. At the time of measurement, 678 of the 811 websites were reachable and successfully analyzed. The remaining domains were offline, defunct, or blocked automated access.

3.4.3 Data collection methodology

Each reachable website was visited using a Puppeteer-based browser automation framework.

To reduce trivial blocking and allow client-side JavaScript to execute under conditions approximating a standard browser session, the crawler applied limited evasion steps, including:

- Modifying the reported user agent.

- Removing common automation indicators such as `navigator.webdriver`. The objective was not to bypass advanced bot mitigation systems, but to observe fingerprinting behavior that would execute during a typical page load.

3.4.4 Instrumentation of fingerprinting-related APIs

During each visit, we instrumented JavaScript execution to detect access to browser attributes commonly used in fingerprinting.

This was implemented by overriding selected objects, properties, and functions, including:

- Properties of `window.navigator`.
- Properties of `window.screen`.
- Canvas and WebGL-related APIs.
- Selected feature-detection mechanisms.

When an overridden attribute or function was accessed, the instrumentation recorded:

- The name of the attribute or API accessed.
- The value returned at the time of access.
- The script responsible for the access. To attribute access to specific scripts without interrupting page execution, overridden functions deliberately threw controlled exceptions that were immediately caught. The resulting stack traces enabled association of fingerprinting activity with the originating script.

3.4.5 Execution tracing and script attribution

For each website, the crawler generated a structured trace of fingerprinting-related activity. This included the list of scripts accessing fingerprint-relevant signals and a partial record of the attributes queried.

Example (truncated for readability):

```
{
        "website":"https://rayobyte.com/",
        "fingerprintingScripts":[
        {
        "url":"https://api.sb.rayobyte.com/fingernaut/pixel.deps.js",
                "trace":[
                        {"name":"navigator.hardwareConcurrency","value":14},
                        {"name":"navigator.vendor","value":"Google Inc."},
                        {"name":"screen.width","value":2560},
                        {"name":"screen.height","value":1440}
                ]
        }]
}
```


These traces do not capture full script logic. However, they are sufficient to:

- Identify scripts that actively query fingerprinting-relevant signals.
- Detect reuse of identical scripts across multiple domains.
- Cluster websites based on shared fingerprinting infrastructure.

3.4.6 Methodological limitations

This methodology captures client-side JavaScript behavior observable during page load and early execution. It does not capture:

- Server-side fingerprinting logic.
- Scripts triggered only after specific user interaction.
- Fingerprinting performed exclusively within execution contexts not instrumented by the crawler (for example iframes or workers). Despite these limitations, the approach is sufficient to identify widespread fingerprinting behavior, script reuse, and infrastructure overlap across bot- and fraud-adjacent services.

3.5 Measurement results: prevalence and script classification

This section summarizes findings from the ecosystem-level measurement described above. It combines quantitative prevalence observations with a qualitative classification of fingerprinting scripts identified across the dataset.

3.5.1 Heuristic identification of fingerprinting scripts

To focus the analysis on scripts relevant to browser fingerprinting, Castle's Research Team applied a filtering heuristic based on execution traces collected during page visits.

A script was flagged as fingerprinting-related if it accessed more than:

- 4 attributes from `window.navigator`, and
- 2 attributes from `window.screen` during execution.

This threshold was intentionally conservative. It excludes scripts with minimal or incidental access to browser attributes while accepting that some analytics and measurement scripts may still be included. At this stage, the objective is identification of signal collection behavior, not attribution of intent.

3.5.2 Prevalence within the analyzed dataset

Applying this heuristic to the 678 successfully analyzed bot- and fraud-adjacent websites yielded the following results:

- **85 websites (12.5%)** loaded at least one fingerprinting-related script.
- **593 websites (87.5%)** did not load scripts matching our fingerprinting heuristic.
- Among the 85 identified sites, several loaded multiple fingerprinting-related scripts. This indicates that client-side collection of browser attributes is not universal across the ecosystem, but is present in a measurable subset of automation-adjacent services.

We also observed reuse of identical third-party fingerprinting or tracking scripts across multiple domains. Examples include:

- `https://mc.yandex.ru/metrika/tag.js` loaded on **13** websites
- `https://bat.bing.com/bat.js` loaded on **9** websites
- `https://euob.blueridgeloop.com/sxp/i/636f8b858f681acb7bfa6f583a96630a.js` loaded on **3** websites
- `https://ssl.google-analytics.com/ga.js` loaded on **3** websites
- `https://mc.webvisor.org/metrika/tag_ww.js` loaded on **2** websites Most reused scripts are standard analytics, tracking, or ad-fraud detection libraries. Their presence alone is not inherently suspicious. This distinction between presence and purpose is critical. Detecting fingerprinting-related behavior does not demonstrate harvesting or reuse across properties.

That said, fingerprint collection appears on 12.5% of bot- and fraud-adjacent websites in our dataset. This establishes a structural baseline: collecting client-side environment data is common in parts of the automation ecosystem.

This baseline matters because it creates cover. In an environment where fingerprinting is routine, harvesting-oriented implementations can operate without standing out. The goal of the following deep dives is therefore to move beyond presence and understand intent. We analyze whether these scripts support legitimate use cases such as analytics and risk scoring, or whether they enable fingerprint aggregation and reuse across coordinated workflows.

3.5.3 Classification of observed fingerprinting scripts

Flagged scripts were grouped into categories based on runtime behavior, code structure, and publicly identifiable associations.

A large portion of flagged scripts were associated with analytics or advertising platforms. These scripts typically:

- Access basic browser and device attributes.
- Perform limited automation checks (for example, `navigator.webdriver`).
- Avoid exhaustive rendering or hardware-level probing. Representative examples observed include:
- https://mc.yandex.ru/metrika/tag.js
- https://mc.yandex.ru/watch/
- https://bat.bing.com/bat.js In these cases, attribute access appears consistent with measurement and attribution use cases.

Many websites in the dataset rely on advertising revenue and deploy scripts intended to detect invalid or automated traffic. These scripts generally exhibit broader signal collection, including:

- Canvas and WebGL probing.
- Feature-detection routines.
- Explicit checks for browser tampering. Representative examples observed include:
- https://cdn.cheqzone.com/cheq.min.js

- https://cdn.trafficguard.ai/trafficguard.min.js
- https://cdn.adscore.com/adscore.js
- https://static.escalated.io/ Here, fingerprinting is used in the context of traffic quality assessment and ad fraud mitigation.

A non-trivial subset of scripts were derived from open-source fingerprinting libraries, most commonly FingerprintJS.

These scripts typically:

- Collect a standardized set of attributes.
- Follow recognizable structural patterns, even when bundled or minified.
- Appear in multiple versions across different sites. Representative examples include:
- https://cdn.jsdelivr.net/npm/@fingerprintjs/fingerprintjs
- Self-hosted or bundled variants embedded in site-specific JavaScript bundles. In most cases, these libraries appear to support tracking, device recognition, or site-level security, rather than fingerprint harvesting for resale.

For example, the open-source FingerprintJS implementation relies on classical entropy sources such as canvas, WebGL, user agent, and fonts. It does not include the signal structures typically found in commercial anti-bot systems, such as challenge-response mechanisms, integrity checks, or server-side correlation hooks.

As a result, while it can generate stable identifiers at the page level, it provides limited value for operators seeking to build reusable, vendor-compatible fingerprint datasets.

Several websites loaded fingerprinting logic associated with commercial bot mitigation or security providers. In these cases, fingerprinting appears to serve a defensive purpose: protecting the site itself from abuse, fraud, or automated traffic.

Representative examples observed include:

- `https://newassets.hcaptcha.com/captcha/v1/`
- `https://js.maxmind.com/` The presence of such scripts does not indicate harvesting or reuse. Rather, it reflects the fact that some bot- and proxy-adjacent services are themselves targets of fraud and automation, and therefore deploy third-party bot mitigation or fraud prevention tooling.

3.5.4 Custom fingerprinting implementations with ambiguous intent

One representative example is Rayobyte, a proxy and scraping infrastructure provider. During ecosystem measurement, we observed the script: `https://api.s-b.rayobyte.com/fingernaut/pixel.deps.js`

Execution traces show that this script assembles a structured browser fingerprint object containing device, platform, storage, rendering, and environment attributes. The collected fields include user agent data, hardware concurrency, device memory, screen and color properties, WebGL renderer information, storage availability, plugin enumeration, language and timezone configuration, and permission states.

A representative excerpt from the script is shown below:

```
a = {
    useragent: navigator.userAgent,
    platform: G(n.platform) ?? navigator.platform,
    doNotTrack: navigator.doNotTrack ?? null,
    hardwareConcurrency: G(n.hardwareConcurrency) ??
navigator.hardwareConcurrency ?? 0,
    deviceMemory: G(n.deviceMemory),
    screenResolution: N(G(n.screenResolution)),
    screenFrame: G(n.screenFrame) ?? [null, null, null, null],
    touchSupport: G(n.touchSupport) ?? {
        maxTouchPoints: navigator.maxTouchPoints ?? 0,
        touchEvent: !1,
        touchStart: !1
    },
    languages: G(n.languages) ?? [[navigator.language]],
    dateTimeLocale: G(n.dateTimeLocale) ?? navigator.language,
    timezone: G(n.timezone) ?? Intl.DateTimeFormat().resolvedOptions().timeZone,
    webGlBasics: {
        vendorUnmasked: o.vendorUnmasked ?? "",
        rendererUnmasked: o.rendererUnmasked ?? ""
    },
    colorDepth: G(n.colorDepth),
    colorGamut: G(n.colorGamut),
    vendor: G(n.vendor),
    vendorFlavors: G(n.vendorFlavors),
    plugins: G(n.plugins),
    sessionStorage: G(n.sessionStorage),
    localStorage: G(n.localStorage),
    indexedDB: G(n.indexedDB),
    cookiesEnabled: G(n.cookiesEnabled),
    pdfViewerEnabled: G(n.pdfViewerEnabled),
    osCpu: G(n.osCpu),
    invertedColors: z(G(n.invertedColors)),
    forcedColors: A(G(n.forcedColors)),
    monochrome: G(n.monochrome),
    contrast: B(G(n.contrast)),
    reducedMotion: G(n.reducedMotion),
    reducedTransparency: G(n.reducedTransparency),
    hdr: G(n.hdr),
    ...r,
    permissions: t
};
```

The overall signal set is consistent with standard browser fingerprinting practices. It captures a broad but conventional entropy surface suitable for device recognition, anomaly detection, or account-level abuse controls. The script also contains explicit

fingerprinting references such as `submitFingerprint` and `FingernautDeps`, reinforcing that this is an intentional fingerprinting telemetry component rather than incidental logging.

However, unlike the implementations analyzed in later deep dives (e.g., those referencing Akamai, PerimeterX, Adyen, or hCaptcha), the Rayobyte script does not exhibit vendor-prefixed functions, distinctive challenge seeds, or string-level artifacts that can be traced to commercial anti-bot client code. We did not observe segmented payload streams aligned with specific vendor ecosystems, nor structural indicators of extracted production challenge logic.

As a result, the operational intent of this fingerprint collection remains ambiguous.

On one hand, the collected signals are sufficient to support internal abuse prevention use cases, such as protecting authentication endpoints and detecting multi-account activity.

On the other hand, Rayobyte markets a custom Chromium-based anti-detect browser positioned around “real-world” fingerprint emulation (Figure 12).

After more than a decade helping teams collect public web data at scale, we ran into a hard truth: **proxies alone aren't enough anymore.**

Modern anti-bot systems don't just look at IPs — they fingerprint browsers at a deep level. And when you're scraping millions of pages a day, even small inconsistencies get you blocked fast.

So we did what we've always done at Rayobyte when the market came up short: **we built our own solution.**

Introducing Rayobrowse

Rayobrowse is a self-hosted, Linux-native stealth Chromium browser built for scraping at scale. Engineered at the Chromium C++ level, it emulates real-world fingerprints, runs without GPUs, and stays undetected on modern anti-bot systems.

It's Playwright-compatible, fully self-hostable, and free to get started for our proxy users — pip install, and you're live [Full installation guide available on GitHub]

Figure 12: Rayobyte announcement marketing a custom Chromium-based "anti-detect" browser positioned as emulating real-world fingerprints for automation workflows.

In that context, access to organic browser fingerprint distributions from real visitors could theoretically inform calibration of anti-detect profiles. Real-world telemetry improves the statistical realism of emulated environments.

That said, we did not identify technical evidence within the observed script indicating cross-property aggregation, resale, or integration with anti-detection tooling. The runtime behavior alone does not establish structured harvesting.

For this reason, Rayobyte is categorized under custom fingerprinting scripts with unclear intent. This case illustrates a broader gray zone within the ecosystem: fingerprint collection that is technically sophisticated and entropy-rich, yet not demonstrably aligned with vendor-specific harvesting or replay workflows.

3.5.5 Custom implementations with indicators of structured harvesting

A smaller subset of observed scripts exhibited characteristics that diverge from standard analytics, advertising telemetry, or routine site-level abuse prevention.

These implementations are distinguished by a combination of the following traits:

- Broad, high-entropy signal collection spanning browser, rendering, hardware, feature-detection, and automation-relevant APIs.
- Inclusion of signals that mirror the structure, naming conventions, or challenge logic of commercial anti-bot systems.
- Redundant collection of identical signals across multiple sub-objects, each aligned with different commercial anti-bot vendors, suggesting attempts to maximize compatibility with downstream fingerprinting or scoring systems.
- Obfuscation, hashing, or encryption layers that conceal structured payloads prior to transmission. Unlike basic device fingerprinting, which focuses on generating a stable identifier for session continuity or anomaly detection, these scripts replicate how commercial anti-bot vendors derive and structure client-side telemetry. The goal is not just device identification, but alignment with vendor-specific signal generation.

Representative examples include:

- `https[:]//fp-flame.vercel[.]app/_next/static/chunks/app/page-ce580fedf644df13.js`
- `https[:]//collect.bfgcollect[.]com/static/a/a.js?tag=ez-protal`
- Obfuscated collectors embedded within sneaker and automation-oriented services that execute vendor-style challenge logic client-side. In these cases, fingerprinting moves beyond site-level identification. The depth of signal extraction, the presence of vendor-specific artifacts, and the structured packaging of telemetry point toward implementations designed to mirror commercial anti-bot systems.

The following case studies examine these implementations in detail, focusing on payload structure, signal provenance, and vendor-specific patterns.

3.6 Case studies of fingerprint harvesting implementations

Castle's Research Team identified multiple fingerprinting implementations across bot- and fraud-adjacent services that extend beyond typical analytics or defensive bot detection. While ecosystem-level measurements highlight prevalence and script reuse, they do not explain how fingerprint data is actually collected, structured, and operationalized.

The following section examines a set of services where fingerprint collection exhibits characteristics consistent with large-scale harvesting. Each analysis moves from observable deployment to technical breakdown, and finally to an evidence-based assessment of intent.

Each case follows a consistent framework:

- **Operational context**: how scripts are deployed, including execution context (main page vs iframe), origins, and data collection endpoints.
- **Script and payload analysis**: signals collected, payload structure, and any compression, encoding, obfuscation, or encryption.
- **Indicators of harvesting**: technical or operational elements suggesting collection for reuse beyond the originating site. The first case examines cybertemp[.]xyz and shows how first-party scripts and embedded third-party collectors operate in parallel within a single service.

3.7 Case study 1: cybertemp[.]xyz

3.7.1 Operational context

Sites analyzed:

- Main site: `http://cybertemp[.]xyz`
- Embedded iframe: `https://fp-flame.vercel.app/` Cybertemp[.]xyz is a disposable email service offering both free and paid temporary inboxes. Disposable email platforms are frequently used in automated workflows, including fake account creation, spam campaigns, testing flows, and abuse of signup systems.

The screenshot below shows the homepage of cybertemp[.]xyz at the time of analysis.

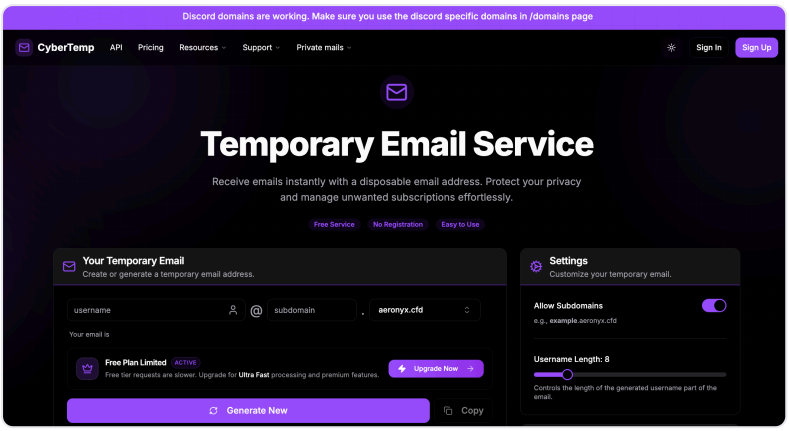


Figure 13: Homepage of cybertemp[.]xyz at time of analysis.

Fingerprint collection occurs through two parallel mechanisms:

1. First-party JavaScript loaded directly from `cybertemp[.]xyz`
2. An embedded iframe loading fingerprinting scripts from `fp-flame.vercel.app`

Observed data collection endpoints:

- `POST https://monitor[.]ac/collector` (fp-flame iframe)
- `X-Client-Meta` HTTP header appended to first-party API requests (cybertemp script)

Script URLs:

- First-party script: `https://www.cybertemp[.]xyz/_next/static/chunks/881.35be087306d117a4.js`
- Embedded iframe: `https://fp-flame.vercel.app/`

  Sub-scripts:

  - `https://fp-flame.vercel.app/a/f.js`
  - `https://fp-flame.vercel.app/a/g.js`
  - `https://fp-flame.vercel.app/a/w.js`
  - `https://fp-flame.vercel.app/a/c.js` The `fp-flame` iframe also loads a nested iframe from:

- `https://ss.10512[.]top/hf_v6.html?h=ss.10512.top` (This nested iframe was not analyzed in this report.)

3.7.2 Fingerprint script and payload analysis

Castle's Research Team analyzed the first-party JavaScript bundle loaded from:

`https://www.cybertemp[.]xyz/_next/static/chunks/881.35be087306d117a4.js`

This script generates a structured fingerprint object combining:

- Device characteristics
- Browser attributes
- Canvas and WebGL rendering outputs
- Automation indicators
- Session-level behavioral telemetry A representative fingerprint object is shown below.

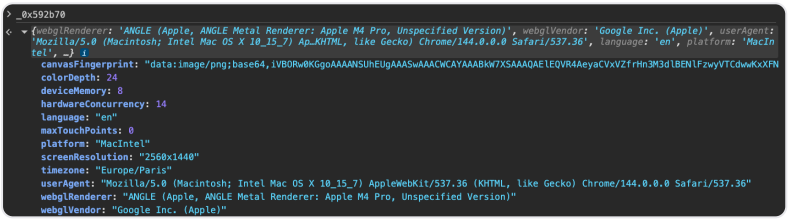


Figure 14: Structured fingerprint object generated by cybertemp's first-party JavaScript bundle prior to encryption.

In addition to static device attributes, the script collects runtime session information, including:

- Device classification (mobile vs. desktop)
- Computed risk score
- Automation detection flags
- Session duration
- Interaction summary and event types

```
return {
    'fingerprint': _0x46abb2,
    'webglRenderer': _0x2d5d43,
    'webglVendor': _0x59a473,
    'browserData': JSON['stringify'](_0x28284b),
    'expiry': Date[_0x460fe5(0x194)]() + 0x36ee80,
    'isAutomated': _0x3c62e3[_0x460fe5(0x142)],
    'automationReasons': _0x3c62e3[_0x460fe5(0x162)],
    'riskScore': _0x3c62e3[_0x460fe5(0x151)],
    'browserType': _0x437d87(),
    'deviceType': _0x232a05[_0x460fe5(0x110)] ? _0x460fe5(0xe7) : 'desktop',
    'metaVersion': _0x232a05[_0x460fe5(0x110)] ? _0x460fe5(0x18a) :
_0x460fe5(0xff),
    'mobileChecks': _0x232a05['checks'],
    'timestamp': Date[_0x460fe5(0x194)](),
    'sessionDuration': Date[_0x460fe5(0x194)]() - _0x446a59,
    'interactionSummary': {
        'total': _0x31103a[_0x460fe5(0x111)],
        'types': Array[_0x460fe5(0x15d)](new Set(_0x31103a[_0x460fe5(0x163)]
(_0x118b86 => _0x118b86[_0x460fe5(0x17a)]('_')[0x0])))
    },
    'sessionStart': _0x446a59,
    'interactionCount': _0x31103a['length']
};
```

The fingerprint object is serialized, compressed, encrypted, and stored in `window.__x_client_meta`.

Rather than issuing a dedicated fingerprint submission request, the script patches `window.fetch` so that outbound API requests automatically include the encrypted fingerprint inside a custom `X-Client-Meta` header. This embeds the fingerprint into otherwise legitimate application traffic.

The screenshot below shows the header attached to a request to `https://www.cy-bertemp[.]xyz/api/inbox`.

Referer	https://www.cybertemp.xyz/
Sec-Ch-Ua	"Not(A:Brand";v="8", "Chromium";v="144", "Google Chrome";v="144"
Sec-Ch-Ua-Mobile	?0
Sec-Ch-Ua-Platform	"macOS"
Sec-Fetch-Dest	empty
Sec-Fetch-Mode	cors
Sec-Fetch-Site	same-origin
User-Agent	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/144.0.0.0 Safari/537.36
X-Client-Meta	H4sIAAAAAAAAA1VU226jMBD9FctPrZQLBAxJ9inb267atFXSdlfaVJWBgVgBjGyTXqL8+44htyoP8RzPnDlzMRuaijIDVSlRGjqmkc8834NwOErjQegyJ0wZ4ynzPI8ljCGWDN0gHrlh6kAAzI8gjGEICY/daOhHtEPfIcryGZQJKFBIObm/ubsiZ5OqyqFDWmsKhudk7zQmzSWZ+uRRyQ55LnUFsUgFJOQFIBayPN8Tv2CMtLQ3UmYY87uMezty6xMp+a5BXXLD0WezoDVakwxKs6DjBZ3KL5HnvM96Djmb8hiLlnr5A1kM5AQB8jAnf4nrvLnsLTxvdf2B6FaYPvPCnheQs9tfT9O7DsnFCsgNxCt5Ti6WShbQd32/59gfmfOUK7ELWdDOgua8zGqeQSMDym+YRvDfAU1V93q2OzV/UHaf5wv6iscq5yaVqmiL4XGju3Gq8joTZcNkHXWswPKNsQXvIjFLPA5Y4ODVEkS2tO1AudaOZS7VJVStj2+5xAfkR2SLkBEFfMmylX9VK1lB/xFL1E3y/e1Dmmqw1N0mk9BTGYncBqU814BQ0QAXS+xbq3WLQ4OPSqhPFBSGTsAcI4280aCD4ZPayIlbSPYElLcIbsQMuJalRg5aCK1xi0ncjlGourSCSGVVKiNAY45cFAKJCE4cFI8tA0nE2m6X+cTeUqxlNY+lAiz5sEdPnxXaG5TSjpiOjapRRptp1iY6iPuGPh6z7++F/qn4Gk7sa6EgIR8nSLs5J8BVkp1GPCAp39nbDk1gLWJoZaKhV0ZWWG2B72v3co54d+3aq5MJNMvSTE8bXlS7CfgB201Ag7YUl7VqWo6dCQJ3xFDHsYvzuii4nd6GGomvGkmQEgW1o5H4AAuJRb9uD3xzw5U5JBuMPHcUfuO8kLX9Grnb/zJX26fBAAA

Figure 15: Encrypted fingerprint payload embedded in the `X-Client-Meta` header of a first-party API request.

In parallel, cybertemp[.]xyz embeds an iframe loading:

`https://fp-flame.vercel.app/`

This iframe loads multiple components:

- `a/f.js` – font enumeration, canvas fingerprinting, plugin inspection, screen analysis
- `a/g.js` – feature detection, media queries, sensor capability checks, mathematical functions behavior
- `a/w.js` – WebGL fingerprinting and graphics capability extraction
- `a/c.js` – aggregation, encryption, and transmission The aggregated iframe payload is transmitted via:

`POST https://monitor[.]ac/collector`

Notably, the same `monitor[.]ac` endpoint was observed on additional services associated with the same operator (including `noinspect[.]top` and `webora[.]cc`), suggesting shared collection infrastructure across related properties.

Representative excerpts are shown below.

```
const canvasFingerprint = function () {
    ;
    if (arguments["length"] > 0 && void 0 !== arguments[0] && arguments[0]) return !1;

    var Bt = document["createElement"]("canvas");
    if (!Bt["getContext"]) return !1;

    try {
        var Ft = [];
        Bt["width"] = 2e3, Bt["height"] = 200, Bt["style"]["display"] = "inline";
        var Ut = Bt["getContext"]("2d");
        return !(!Ut) && (Ut["rect"](0, 0, 10, 10), Ut["rect"](2, 2, 6, 6), Ft["push"]("canvas winding:"["concat"]
    } catch (t) {
        return !1;
    }
};
```

Figure 16: Code related to the canvas fingerprint component.

```
var E = function () {
    var t, e, n, r;
    try {
        var Jn = {};
        Jn["colorScheme"] = window["matchMedia"]("(prefers-color-scheme: dark)")["matches"] ? "dark" : "light",
        var zn = (function (t) {
            var c = Object["keys"](t);
            c["sort"](function (t, e) {
                return t["localeCompare"](e);
            });
            for (var s = [], f = 0; f < c["length"]; f++) {
                var l = c[f], p = t[l];
                s["push"](l + "=" + p);
            }
            return s["join"](";");
        })(Jn);
        let str = JSON["stringify"](zn);
        return md5(str)
    } catch (t) {
        return null;
    }
}
```

Figure 17: Code related to the media query fingerprint component.


```
payload
▾ {fe: {…}, general: {…}, wgl: {…}} i
  ▾ fe:
      f: "e4500e866f8ad9641b45de6e9e985d00"
    ▸ fe: (23) ['DNT:unknown', 'L:en', 'D:24', 'PR:1.5', 'S:2560,1440', 'AS:2560,1410', 'TO:-60', 'SS:true', 'LS:true', 'IDB:true', 'B:false
    ▸ fe_unformatted: {DNT: 'unknown', L: 'en', D: 24, PR: 1.5, S: Array(2), …}
      ife_hash: "3beb528e31297ce428804cc7524dd70d"
      jsbd: "{\"HL\":3,\"NCE\":true,\"DT\":\"\",\"NWD\":\"false\",\"DMTO\":1,\"DOTO\":1}"
    ▸ [[Prototype]]: Object
  ▸ general: {user_agent_data_brands: 'Not(A:Brand,Chromium,Google Chrome', user_agent_data_mobile: false, navigator_connection_downlink: 10
  ▾ wgl:
      webgl_aliased_line_width_range: "[1, 1]"
      webgl_aliased_point_size_range: "[1, 511]"
      webgl_antialiasing: "yes"
      webgl_bits: "8,8,24,8,8,0"
      webgl_extensions: "ANGLE_instanced_arrays;EXT_blend_minmax;EXT_clip_control;EXT_color_buffer_half_float;EXT_depth_clamp;EXT_disjoint_t
      webgl_extensions_hash: "9cbeeda2b4ce5415b07e1d1e43783a58"
      webgl_fsf_params: "23,127,127,23,127,127,23,127,127"
      webgl_fsi_params: "0,31,30,0,31,30,0,31,30"
      webgl_hash_webgl: "4f2acc6fcdbb7936a3163cb9b0ee218a"
      webgl_max_params: "16,32,16384,1024,16384,16,16384,30,16,16,1024"
      webgl_max_viewport_dims: "[16384, 16384]"
      webgl_renderer: "WebKit WebGL"
      webgl_shading_language_version: "WebGL GLSL ES 1.0 (OpenGL ES GLSL ES 1.0 Chromium)"
      webgl_unmasked_renderer: "ANGLE (Apple, ANGLE Metal Renderer: Apple M4 Pro, Unspecified Version)"
      webgl_unmasked_vendor: "Google Inc. (Apple)"
      webgl_vendor: "WebKit"
      webgl_version: "WebGL 1.0 (OpenGL ES 2.0 Chromium)"
      webgl_vsf_params: "23,127,127,23,127,127,23,127,127"
      webgl_vsi_params: "0,31,30,0,31,30,0,31,30"
```


Figure 18: Full iframe-generated fingerprint object prior to encryption.

Some collected attributes appear consistent with signals commonly used by commercial anti-bot vendors. These elements are examined in detail in the following section.

3.7.3 Indicators of fingerprint harvesting

Several elements observed on cybertemp[.]xyz are consistent with fingerprint harvesting rather than site-local abuse prevention.

The operator explicitly referenced fingerprint collection activity in Telegram discussions (see Figures 10 and 11). While public claims alone do not establish intent, they are materially relevant when evaluated alongside the technical findings.

Cybertemp deploys:

1. A first-party script embedding encrypted fingerprint data into application traffic (`X-Client-Meta`)

2. A separate iframe bundle transmitting structured telemetry to `monitor[.]ac` This layered architecture exceeds what is typically required for a disposable email platform's abuse prevention. The parallel collection channels and shared infrastructure across related properties increase the plausibility of centralized aggregation.

Within `https://fp-flame.vercel.app/a/g.js`, Castle's Research Team identified feature-detection logic structurally aligned with commercial bot-detection scripts.

Signals are stored under non-semantic identifiers such as:

- `3f76dd27`
- `5dd48ca0`
- `c2d2015`

Example:

```
"3f76dd27": screen && screen["orientation"] && screen["orientation"]["type"] ?
screen["orientation"]["type"] : null,
"5dd48ca0": (function () {
    for (var f = [window["RTCPeerConnection"], window["mozRTCPeerConnection"],
window["webkitRTCPeerConnection"]], l = 0, p = 0; p < f["length"]; p++) f[p] &&
(l |= 1 << p);

    return l;
})(),
"c2d2015": (function () {
    var V = [["accelerometer", typeof DeviceMotionEvent === "undefined" ?
"undefined" : r(DeviceMotionEvent)], ["gyroscope", typeof DeviceOrientationEvent
=== "undefined" ? "undefined" : r(DeviceOrientationEvent)], ["ambient light
sensor", typeof AmbientLightSensor === "undefined" ? "undefined" :
r(AmbientLightSensor)], ["ambient temperature sensor", typeof
AmbientTemperatureSensor === "undefined" ? "undefined" :
r(AmbientTemperatureSensor)], ["proximity sensor", typeof ProximitySensor ===
"undefined" ? "undefined" : r(ProximitySensor)], ["magnetometer", typeof
Magnetometer === "undefined" ? "undefined" : r(Magnetometer)], ["absolute
orientation sensor", typeof AbsoluteOrientationSensor === "undefined" ?
"undefined" : r(AbsoluteOrientationSensor)], ["geolocation", typeof Geolocation
=== "undefined" ? "undefined" : r(Geolocation)]];
    let str = V["map"](function (t) {
        return t[1] !== "undefined" ? t[0] : void 0;
    })["filter"](Boolean)["join"](",")
    return md5(str);
})(),
```

These identifiers resemble hashed or obfuscated signal keys rather than developer-defined variables. Comparison with reversed Arkose Labs client-side code on GitHub shows structurally similar feature-detection aggregation:

https://github.com/AzureFlow/arkose-fp-docs/blob/7bcbef88a8a20839f5d575f493cd06ce84811a15/arkose_re_docs.html#L255

Screenshots below illustrate the structural overlap.

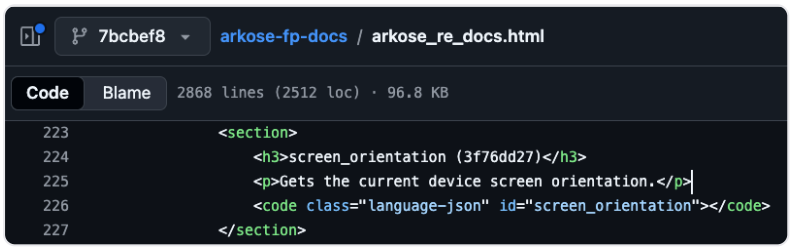


Figure 19: fp-flame iframe — Screen orientation signal stored under a non-semantic/hashed key (`3f76dd27`). The same key is documented in public Arkose reverse-engineering notes as mapping to `screen.orientation.type`, consistent with the fp-flame implementation.

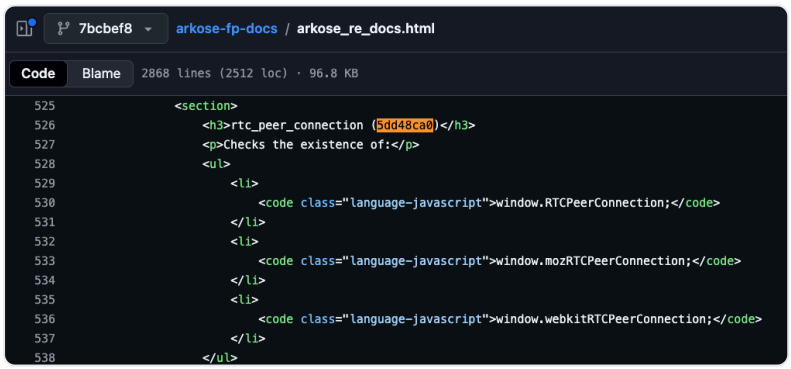


Figure 20: fp-flame iframe — WebRTC feature-probing signal stored under a non-semantic/hashed key (`5dd48ca0`). The same key is documented in public Arkose reverse-engineering notes as a WebRTC capability check, consistent with the fp-flame implementation.

The significance lies in the combination of:

- Non-semantic signal identifiers
- Feature-detection routines aligned with commercial bot-detection systems
- Inclusion of these signals alongside raw device attributes This results in a dual-layer dataset:

1. Raw environmental fingerprints

2. Signals structured to mirror commercial anti-bot telemetry Such a dataset is more consistent with replay, reverse engineering, or anti-detection research workflows than with simple site-level risk scoring.

3.8 Case study 2: Ez CAPTCHA

3.8.1 Operational context

Site analyzed:

- `https://www.ez-captcha[.]com/` Ez-CAPTCHA presents itself as an automated CAPTCHA-solving service (“CAPTCHA farm”) marketed to operators seeking to bypass commercial CAPTCHA and bot-mitigation providers at scale. The screenshot below shows the Ez-CAPTCHA homepage at the time of analysis.

Figure 21: ez-captcha[.]com homepage at time of analysis.

Ez-CAPTCHA advertises support for a broad range of CAPTCHA and anti-bot providers. Castle’s Research Team did not attempt to validate bypass effectiveness against each listed vendor; the objective of this section is to document the finger-print collection mechanisms observed on the service’s web properties.

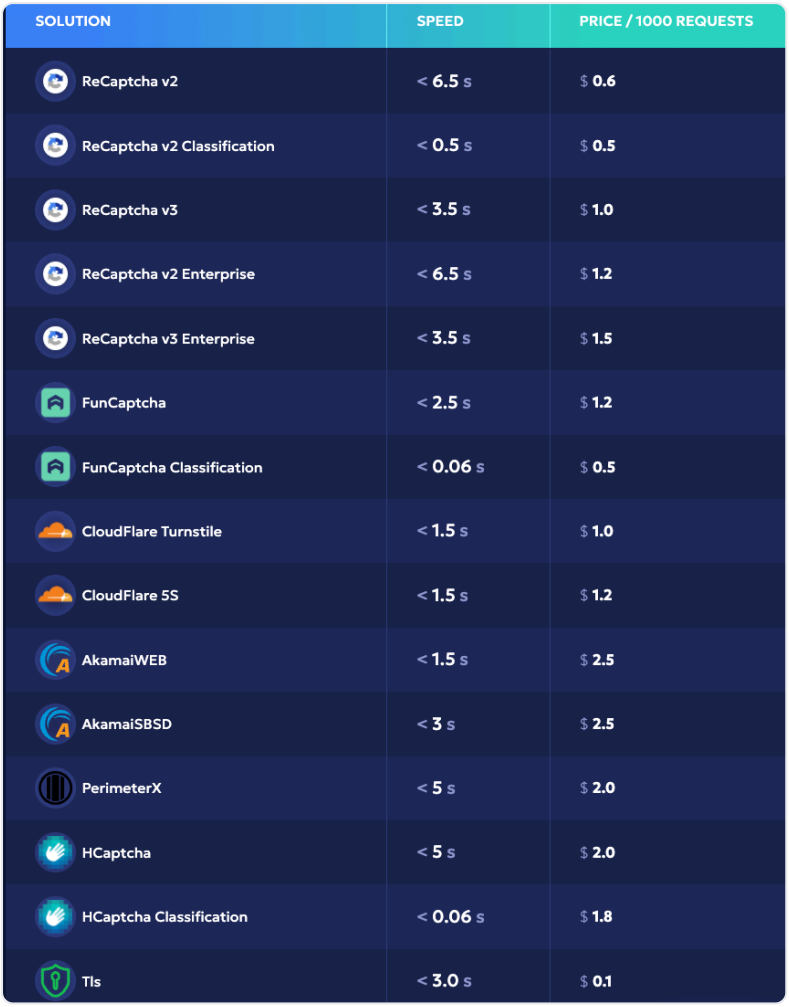


Figure 22: Ez-CAPTCHA marketing page listing supported CAPTCHA / anti-bot providers.

Fingerprinting activity on Ez-CAPTCHA is implemented in both the main execution context and an embedded iframe hosted on a separate collection domain.

- `POST https://collect.bfgcollect[.]com/collect?tag=ez-protal` This endpoint is used by both the main-page script and the embedded iframe.
- Main-page script: `https://collect.bfgcollect[.]com/static/a/a.js?tag=ez-protal`
- Embedded iframe: `https://collect.bfgcollect[.]com/static/h/h.html?tag=ez-protal`

The iframe contains partially obfuscated fingerprinting logic embedded directly within its HTML.

- The main page collects fingerprint signals in the standard JavaScript execution context.
- The embedded iframe performs additional fingerprinting in an isolated context.

- The iframe further leverages a Web Worker to collect signals in an alternative JavaScript environment. This layered execution model enables cross-context validation and increases the internal consistency of collected fingerprint data.

3.8.2 Fingerprint script and payload analysis

Castle's Research Team analyzed the primary fingerprinting script loaded from:

`https://collect.bfgcollect[.]com/static/a/a.js?tag=ez-protal`

This script constructs a structured fingerprint object containing device attributes, rendering signals, automation indicators, and environmental metadata. The resulting object is serialized and transmitted to the collection endpoint.

The payload generated by the main script includes conventional browser fingerprinting attributes. A representative example is shown below.

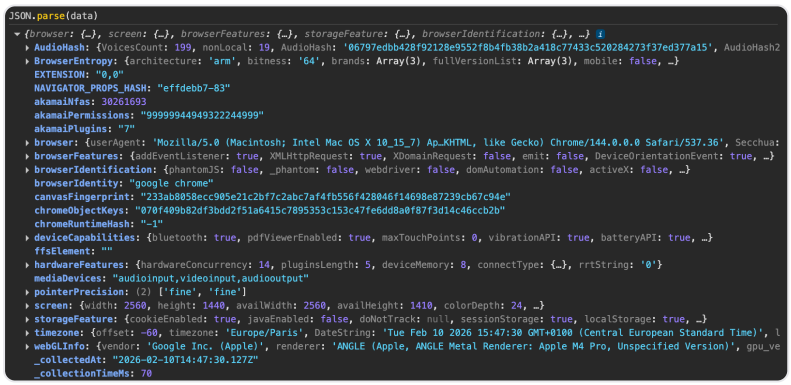


Figure 23: Main-page script (`a.js`) — representative fingerprint payload prior to transmission.

In addition to classical attributes, the script performs browser extension probing by requesting web-accessible resources exposed by specific Chrome extensions. Results are encoded into bitmask-style values.

```
async function detectExtensions() {
    const urls = [
        "chrome-
extension://aeblfdkhhhdcdjpifhhbdiojplfjncoa/images/icons/app_icon-dark_bg-
color-locked-12.png",
        "chrome-extension://flliilndjeohchalpbbcdekjklbdgfkk/html/blocked.html",
        "chrome-
extension://caljgklbbfbcjjanaijlacgncafpegll/html/dashboard.html",
        // ...
        "chrome-extension://ngghlnfmdgnpegcmbpgehkbhkhkbkjpj/popup.html",
        "chrome-extension://gmagdfpiahgilkjlfgpedjjfpomdlkan/book-ticket.html"
    ];

    let r = 0, o = 0;

    async function check(url, i) {
        try {
            await fetch(url, { method: "HEAD" });
            i < 30 ? r |= 1 << i : o |= 1 << (i - 30);
        } catch (e) {}
    }

    for (let i = 0; i < urls.length; i += 10) {
        await Promise.allSettled(urls.slice(i, i + 10).map((url, idx) =>
check(url, i + idx)));
    }

    return r + "," + o;
}
```

This technique, described more in details in the following blog post, allows the operator to infer the presence of specific extensions, including automation tools or privacy-related plugins.

Once assembled, the fingerprint object is transmitted via:

`POST https://collect.bfgcollect[.]com/collect?tag=ez-protal`

The use of a dedicated collection domain separates fingerprint ingestion infrastructure from the visible Ez-CAPTCHA brand domain.

Ez-CAPTCHA also embeds an iframe loaded from the following URL:

`https://collect.bfgcollect[.]com/static/h/h.html?tag=ez-protal`

The fingerprinting logic within this iframe is partially obfuscated, but the obfuscation is inconsistent. Descriptive variable names such as `webgl_support_extensions` and `css_properties_len` remain intact, and inline comments are preserved.

If the code had been fully obfuscated by the Ez-CAPTCHA operator using a standard obfuscation pipeline, identifiers and comments would typically have been uniformly transformed or removed. The coexistence of obfuscated blocks with clear variable names and retained comments suggests that portions of the script were inserted in pre-obfuscated form — likely originating from commercial anti-bot client code — rather than being fully obfuscated by the operator.

```
// 使用
var font_indices = await getFontIndices();
function ku() {
  var em = Math.floor(Math.random() * 9) + 7;
  var gj = String.fromCharCode(Math.random() * 26 + 97);
  var ku = Math.random().toString(36).slice(-em).replace(".", "");
  return `${gj}${ku}`;
} // 隐私模式检测函数
async function detectPrivateMode() {
  if (!Nf || !("indexedDB" in window)) {
    return null;
  }
  var gj = ku();
  return new Promise(function (ku) {
    if (!("matchAll" in String.prototype)) {
      try {
        localStorage.setItem(gj, gj);
        localStorage.removeItem(gj);
        try {
          if ("openDatabase" in window) {
            openDatabase(null, null, null, null);
          }
          ku(false);
        } catch (cw) {
          ku(true);
        }
      } catch (cw) {
        ku(true);
      }
```

Figure 24: Iframe (`h.html`) — partially obfuscated fingerprinting logic showing inconsistent identifier obfuscation and retained comments.

The iframe-based collector performs fingerprinting in multiple JavaScript execution environments:

1. Standard iframe context
2. Dedicated Web Worker context The worker is dynamically created using a Blob containing embedded JavaScript code:

```
async function web_worker_env_info() {
      var uV;
      uV = new Blob(["!function(){function e(){function e(){try{return
1+e()}catch(e){return 1}}function r(){try{var e=1;return 1+r(e)}catch(e){return
1}}var t=e();var n=r();return[t===n?0:n*8/(t-n),t,n]}var r=e();try{var
t=\"OffscreenCanvas\"in self?new
OffscreenCanvas(1,1).getContext(\"webgl\"):null,n=!1,a=null;if(t){var
s=/Firefox/.test(navigator.userAgent)&&\"hasOwn\"in
Object;if(s||t.getExtension(\"WEBGL_debug_renderer_info\")){var
i=t.getParameter(s?7937:37446);n=/SwiftShader|Basic Render/.test(i),a=
[t.getParameter(s?7936:37445),i]}}var{locale:o,timeZone:u}=\"Intl\"in self?
Intl.DateTimeFormat().resolvedOptions():{},v=[r,navigator.userAgent,
[navigator.language,navigator.languages,o,u],
[navigator.deviceMemory,navigator.hardwareConcurrency],a,null];if(!(\"gpu\"in
navigator)||n)return postMessage(v);navigator.gpu.requestAdapter().then((e=>
{if(!e)return
postMessage(v);var{features:r,limits:t,info:n}=e,a=Array.from(r.values()),s=
[];for(var i in t)\"number\"==typeof t[i]&&s.push(t[i]);return(n?
Promise.resolve(n):e.requestAdapterInfo()).then((e=>
{var{architecture:r,description:t,device:n,vendor:i}=e;return v[5]=
[[i,r,t,n],a,s],postMessage(v)}))})).catch((()=>postMessage(v)))}catch{return
postMessage(void 0)}}();"], {
        type: "application/javascript"
      });
      var gj = URL.createObjectURL(uV);
      var ku = new Worker(gj);
      URL.revokeObjectURL(gj);
      // ...
    }
```

This worker collects additional environmental signals, including:

- WebGL renderer information via `OffscreenCanvas`
- Timezone and locale data
- User agent and language configuration
- Device memory and CPU count
- WebGPU adapter capabilities via `navigator.gpu` Example (WebGL via OffscreenCanvas in worker):

```
var t = "OffscreenCanvas" in self ? new OffscreenCanvas(1,
1).getContext("webgl") : null;
    var n = false;
    var a = null;
    if (t) {
      var s = /Firefox/.test(navigator.userAgent) && "hasOwn" in Object;
      if (s || t.getExtension("WEBGL_debug_renderer_info")) {
        var i = t.getParameter(s ? 7937 : 37446);
        n = /SwiftShader|Basic Render/.test(i);
        a = [t.getParameter(s ? 7936 : 37445), i];
      }
    }
```

By collecting fingerprint attributes across multiple execution contexts (main page, iframe, worker), the system enables cross-environment consistency validation. This approach is commonly used in advanced bot detection systems to identify discrepancies introduced by browser spoofing or incomplete environment patching.

The final iframe-generated payload is significantly larger than the main-page payload and contains dozens of structured attributes. For readability, the payload has been divided into three representative excerpts below.

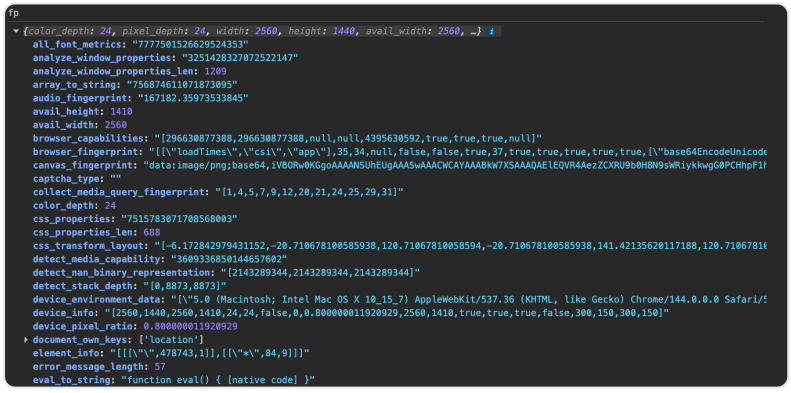


Figure 25: Iframe-generated payload excerpt — Part 1 (representative fields).

Figure 26: Iframe-generated payload excerpt — Part 2 (representative fields).

Figure 27: Iframe-generated payload excerpt — Part 3 (representative fields).

What we observed on Ez-CAPTCHA goes beyond typical analytics or basic fraud checks. The implementation collects a wide fingerprint dataset across multiple execution contexts (main page, embedded iframe, and worker) and includes signals explicitly named or structured after commercial anti-bot vendors. Taken together, this is more consistent with collecting fingerprints for reuse or emulation than with site-local measurement. The next section details the specific indicators.** **

3.8.3 Indicators of fingerprint harvesting

Several technical elements observed in Ez-CAPTCHA's implementation are consistent with fingerprint harvesting rather than site-local fraud detection or analytics.

Explicit references to Akamai-derived signals

Within `https://collect.bfgcollect[.]com/static/a/a.js?tag=ez-protal`, Castle's Research Team identified functions explicitly referencing Akamai:

```
let akamaiPermissions=await getAkmPermissions();
q.akamaiPermissions=akamaiPermissions;

let akamaiPlugins=getPlugins();
q.akamaiPlugins=akamaiPlugins;

let akamaiNfas={};
akamaiNfas=getAKMNfas();
q.akamaiNfas=akamaiNfas;
```

These are not generic variable names. They explicitly include the `akamai` prefix, strongly suggesting that the logic was derived from, or modeled after, Akamai's client-side bot detection code.

The `getAKMNfas` function computes a bitmask-style value representing the presence or absence of numerous browser features:

```
function getAKMNfas()
{
  try {
    let EM2 = window.Boolean(window.navigator.credentials) +
              (window.Boolean(window.navigator.appMinorVersion) << 1) +
              (window.Boolean(window.navigator.bluetooth) << 2) +
              // ...
              (window.Boolean(window.Math.hypot) << 24);
    return EM2;
  } catch (xM2) {
    return 0;
  }
}
```

Similarly, `getAkmPermissions` queries a structured list of permission names and encodes the resulting states into a formatted numeric string:

```
async function getAkmPermissions(resolve) {
  let permissionResult = "";
  let permissionValues = [];
  let permissionList = [
    "speaker",
    "device-info",
    // ...
    "clipboard",
    "accessibility-events"
  ];
  ...
}
```

The naming conventions and structure strongly suggest derivation from Akamai's commercial bot-detection scripts rather than generic capability probing. Collecting Akamai-aligned signals on a CAPTCHA-solving service is operationally significant: such telemetry is primarily useful for emulating or bypassing Akamai's detection logic.

The iframe loaded from:

`https://collect.bfgcollect[.]com/static/h/h.html?tag=ez-protal`

contains partially obfuscated JavaScript that appears to embed code derived from commercial bot-detection challenges.

One indicator is the iframe title referencing hCaptcha challenge content:

```
// 创建iframe元素
const iframe = document.createElement("iframe");

// 设置属性
iframe.id = "mainIframe";
iframe.src = "https://collect.bfgcollect.com/static/h/h.html?tag=ez-protal";
iframe.frameBorder = "0";
iframe.scrolling = "no";
iframe.allow =
  "private-state-token-issuance 'src'; private-state-token-redemption 'src'";
iframe.title = "Main content of the hCaptcha challenge";
```

Figure 28: Ez-CAPTCHA iframe — distinctive font stack extracted from the embedded logic.

Castle's Research Team identified highly specific strings within the fingerprinting logic that match reversed hCaptcha client-side code.

For example, the following font stack (truncated for readability):

```
'Segoe Fluent Icons','Ink Free','Bahnschrift','Segoe MDL2 Assets',
'HoloLens MDL2 Assets','Leelawadee UI','Javanese Text','Segoe UI Emoji',
// ..,
'Droid Sans Mono','Roboto','Ubuntu','Noto Color Emoji'
```

matches content documented in reversed hCaptcha scripts, cf screenshot below:

https://github.com/Implex-ltd/hcaptcha-reverse/blob/352e9d98ee1354cb741492425f7b109fdafc9a89/src/assets/stub.js#L3654

Figure 29: Reversed hCaptcha implementation — matching font stack used for comparison.

Similarly, a long and highly distinctive sequence of Unicode code points used in canvas rendering appears in both Ez-CAPTCHA's iframe script and reversed hCaptcha implementations (cf screenshot below):

```
var nC = [[55357, 56832], [9786], [55358, 56629, 8205, 9794, 65039], ...
].map(function (cw) {
```

Figure 30: Ez-CAPTCHA iframe — distinctive Unicode/canvas rendering sequence aligned with reversed hCaptcha logic.

The probability of such long, highly specific sequences appearing independently is low. This strongly suggests that portions of the fingerprinting logic were extracted from production hCaptcha challenge scripts.

Ez-CAPTCHA collects:

- Low-level device attributes (canvas, WebGL, audio, fonts, hardwareConcurrency, deviceMemory)
- Akamai-style feature bitmasks and permission encodings
- hCaptcha-derived rendering and font detection logic
- Cross-context consistency signals via workers

This produces a dual-layer dataset:

Such a dataset materially lowers the barrier to:

- Reverse engineering anti-bot decision inputs
- Crafting forged payloads
- Injecting realistic telemetry into automated clients While fingerprint collection alone does not prove harvesting intent, the presence of vendor-specific logic within a CAPTCHA-solving service is a strong indicator that the collected data may be intended for reuse beyond site-local security purposes.

3.9 Case study 3: Cybersole

3.9.1 Operational context

Site analyzed:

- `https://cybersole[.]io/` Cybersole is a commercial sneaker and retail automation bot marketed to individuals seeking to automate purchases of high-demand inventory (limited-edition sneakers and streetwear, collectibles, and high-traffic ticket drops). In practice, tools in this category operate against checkout flows protected by multiple commercial anti-bot and fraud providers, which incentivizes continuous adaptation to client-side detection and telemetry requirements. The screenshot below shows the Cybersole homepage at the time of analysis.

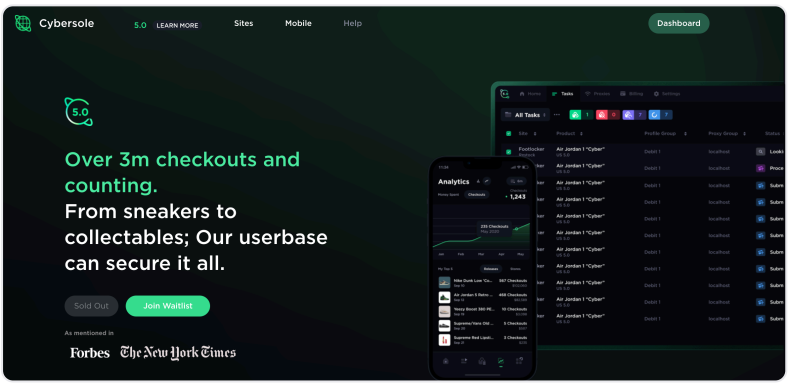


Figure 31: cybersole[.]io homepage at time of analysis.

Fingerprint collection on Cybersole's website is implemented via a first-party JavaScript collector executed in the main page context.

Observed collection endpoint:

- `POST https://cybersole[.]io/api/collector` **Primary script location:**
- `https://cybersole[.]io/js/collector.js?v=FZdAwaZh8LRj6wflMULomeDDcc-IonqNgNwXbRCRu20w` Fingerprint data is gathered in-page and submitted to the internal API endpoint as a structured JSON payload.

3.9.2 Fingerprint script and payload analysis

Castle's Research Team analyzed the fingerprinting script loaded from:

`https://cybersole[.]io/js/collector.js?v=FZdAwaZh8LRj6wflMULomeDDccIonqNgNwXbRCRu20w`

This script constructs a structured fingerprint object and transmits it via:

`POST https://cybersole[.]io/api/collector`

A representative overview of the generated payload is shown below.

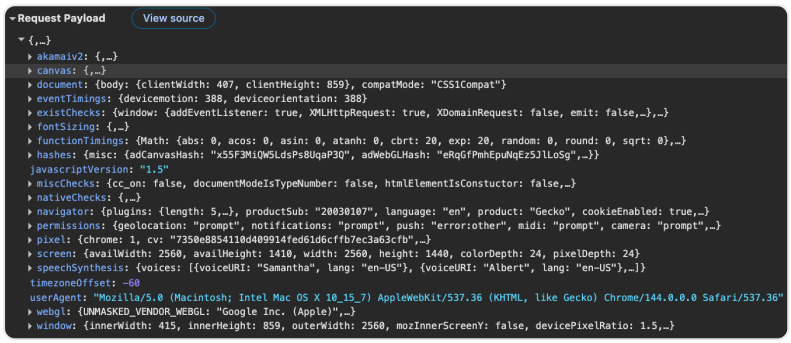


Figure 32: Cybersole collector — representative fingerprint payload prior to transmission.

The collector includes standard device and browser attributes, but also implements additional signal categories that are particularly relevant in bot/anti-detect contexts: (1) timing baselines and (2) integrity checks on native APIs. These classes of signals are commonly used by commercial anti-bot systems to detect instrumented runtimes, virtualization, and environment patching.

The script measures execution timing for a set of built-in primitives and records median timings. One representative function is shown below:

```
function mathTimings() {
  try {
    const _0x421410 = [];
    const _0x441d38 = 1000;
    const _0x1f79cc = [Math.abs, Math.acos, Math.asin, Math.atanh, Math.cbrt,
Math.exp, Math.random, Math.round, Math.sqrt, isFinite, isNaN, parseFloat,
parseInt, JSON.parse];
    let _0x178214 = 0;
    for (; _0x178214 < _0x1f79cc.length; _0x178214++) {
      const _0x1f851f = [];
      let _0x35008b = 0;
      const _0x389650 = performance.now();
      let _0x56b39b = 0;
      let _0x501471 = 0;
      if (_0x1f79cc[_0x178214] !== undefined) {
        _0x56b39b = 0;
        for (; _0x56b39b < _0x441d38 && _0x35008b < 0.6; _0x56b39b++) {
          const _0x2aeb48 = performance.now();
          let _0x37650e = 0;
          for (; _0x37650e < 4000; _0x37650e++) {
            _0x1f79cc[_0x178214](3.14);
          }
          const _0x4b307c = performance.now();
          let _0x143648 = Math.round((_0x4b307c - _0x2aeb48) * 1000);
          _0x1f851f.push(_0x143648);
          _0x35008b = _0x4b307c - _0x389650;
        }
        const _0x9540a9 = _0x1f851f.sort();
        _0x501471 = _0x9540a9[Math.floor(_0x9540a9.length / 2)] / 5;
      }
      _0x421410.push(_0x501471);
    }
    return _0x421410;
  } catch (_0x1af78e) {
    return "exception";
  }
}
```

Execution timing measurements like these are frequently used in commercial anti-bot systems to identify emulation and instrumentation artifacts (for example, abnormal timing distributions introduced by patched engines, headless modes, or automation hooks). In the context of a bot product, collecting timing baselines is operationally useful because it helps characterize what “normal” looks like across real client environments.

Cybersole's collector also includes integrity checks intended to detect whether native browser functions have been overridden. The helper below checks whether a function's `toString()` output contains the standard `[native code]` marker:

```
function isNative(_0xe56c7) {
  return typeof _0xe56c7 == "function" && /\{\s*\[native code\]\s*\}/.test("" +
_0xe56c7);
}}
```

This integrity check is applied to several native APIs, including:

```
window: {
    RunPerfTest: isNative(window.RunPerfTest),
    openDatabase: isNative(window.openDatabase),
    BatteryManager: isNative(window.BatteryManager),
    setTimeout: isNative(window.setTimeout),
    setInterval: isNative(window.setInterval),
    EventSource: isNative(window.EventSource)
  }
```

These checks are designed to detect browser-level patching commonly performed by anti-detect browsers and automation frameworks (wrapping, proxying, or replacing native APIs). Their presence suggests the payload contains "environment integrity" signals in addition to raw device and browser characteristics.

The next section analyzes vendor-specific structures embedded in the payload and documents indicators that portions of the logic are derived from commercial anti-bot and payment-provider telemetry.

3.9.3 Indicators of fingerprint harvesting

Several elements observed in Cybersole's collector are consistent with fingerprint harvesting and anti-detection research rather than site-local analytics.

Within the payload structure, Cybersole includes an object named `hashes.misc` containing fields explicitly prefixed with `ad`:

```
hashes: {
  misc: {
    adCanvasHash: adyenHash(adyCanvasFP()),
    adWebGLHash: adyenHash(adyWebFP()),
    adFontHash: adyenHash(adJSFont())
  }
},
```

The naming convention (`adyCanvasFP` , `adyWebFP` , `adyJSFont` , `adyenHash`) directly references Adyen, a widely used payment processor. The `adyCanvasFP` function implements a canvas fingerprint challenge:

```
function adyCanvasFP() {
  let _0x54cd9d = document.createElement("canvas");
  if (!!_0x54cd9d.getContext && !!_0x54cd9d.getContext("2d")) {
    var _0x4a8c91 = document.createElement("canvas");
    var _0x4464f3 = _0x4a8c91.getContext("2d");
    var _0x19d009 = "#&*(sdfjlSDFkjls28270(";
    _0x4464f3.font = "14px 'Arial'";
    _0x4464f3.textBaseline = "alphabetic";
    _0x4464f3.fillStyle = "#f61";
    _0x4464f3.fillRect(138, 2, 63, 20);
    _0x4464f3.fillStyle = "#068";
    _0x4464f3.fillText(_0x19d009, 3, 16);
    _0x4464f3.fillStyle = "rgba(105, 194, 1, 0.6)";
    _0x4464f3.fillText(_0x19d009, 5, 18);
    return _0x4a8c91.toDataURL();
  }
  return "exception";
}
```

The distinctive seed string:

```
"#&*(sdfjlSDFkjls28270("
```

also appears in Adyen-related client code documented on GitHub (used here as a comparison reference):

https://github.com/StemboltHQ/solidus-adyen/blob/master/app/assets/javascripts/test-adyen-encrypt.js.erb

The screenshot below illustrates the overlap.

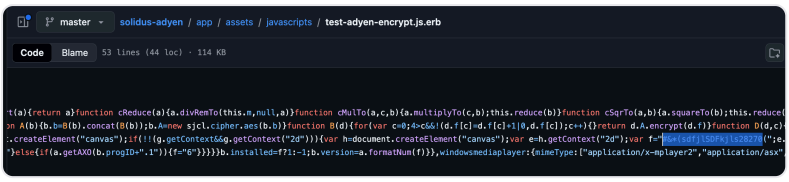


Figure 33: String-level overlap — Cybersole canvas challenge string matching Adyen-related client code.

The combination of explicit naming and string-level overlap is consistent with Adyen-derived (or Adyen-modeled) fingerprint logic being incorporated into Cybersole's collector.

The fingerprint payload also contains an `akamaiv2` object:

```
akamaiv2: await async function () {
  var _0x3d6df8 = await getSpeechSynthesisData();
  return {
    ...akaHeadlessFp(),
    voiceHash: function () {
      let _0x10ca49 = "";
      for (let _0x2d812b of _0x3d6df8) {
        _0x10ca49 += _0x2d812b.voiceURI + "_" + _0x2d812b.lang;
      }
      return hex(sha256(_0x10ca49));
    }(),
    webgl: akaWebGl()
  };
}(),
```

Function names such as `akaHeadlessFp`, `akaWebGl`, and `akamaiv2` explicitly reference Akamai.

In addition, the canvas challenge structure closely mirrors Akamai-style implementations:

```
canvas: function () {
      let _0xec0bb5 = {
        fingerprints: {
          "<@nv45. F1n63r,Pr1n71n6!": akaCanvasFp1("<@nv45. F1n63r,Pr1n71n6!"),
          "m,Ev!xV67BaU> eh2m<f3AG3@": akaCanvasFp1("m,Ev!xV67BaU> eh2m<f3AG3@")
        }
      };
      for (let _0x25faff = 0; _0x25faff < 999; _0x25faff++) {
        _0xec0bb5.fingerprints[String(_0x25faff)] = akaCanvasFp2(_0x25faff);
      }
      return _0xec0bb5;
    }(),
```

The screenshot below shows similar logic in reversed Akamai scripts.

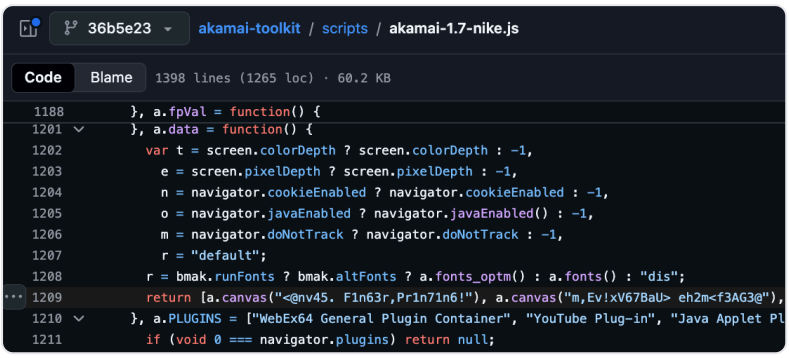


Figure 34: Reversed Akamai client-side implementation — structurally similar seeded canvas challenge logic (iterative fingerprint generation and deterministic key naming) used for comparison.

The structure includes seeded canvas challenges, iterative generation, and deterministic key naming — patterns that align with commercial bot-detection challenge logic rather than generic “off-the-shelf” fingerprinting.

Cybersole’s collector aggregates:

- Low-level device attributes (screen, WebGL, fonts, `hardwareConcurrency`, `deviceMemory`)
- Timing measurements of native primitives
- Integrity checks for overridden native APIs
- Adyen-derived canvas/WebGL/font hashes
- Akamai-style headless and canvas rendering signals This produces a dual-layer dataset:

1. Raw environmental characteristics
2. Outputs structured to mirror commercial anti-bot and payment-provider signals Access to real-world vendor-aligned challenge outputs collected from genuine environments can materially reduce the cost of building realistic anti-detection tooling (for example, by enabling payload modeling and replay in custom clients). While fingerprinting alone does not establish harvesting intent, the inclusion of both Adyen- and Akamai-aligned logic within a sneaker automation ecosystem is consistent with fingerprint collection for reuse in anti-detection workflows rather than purely for site-local telemetry.

3.10 Case study 4: StellarAIO

3.10.1 Operational context

Site analyzed:

- `https://stellaraio[.]com/` StellarAIO is a commercial retail automation bot designed to automate purchases of high-demand products, including sneakers, trading cards (e.g., Pokémon), gaming consoles, and other limited-release items. The screenshot below shows the StellarAIO homepage at the time of analysis.

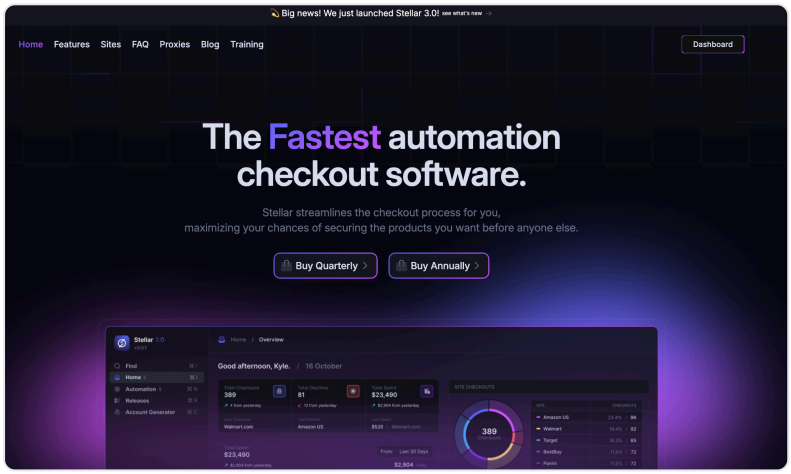


Figure 35: stellaraio[.]com homepage at time of analysis.

StellarAIO advertises support for over 70 e-commerce platforms, including Amazon, Walmart, Pokémon Center, and numerous EU sneaker retailers. These platforms are typically protected by commercial anti-bot and anti-fraud vendors. Automation tools operating in this ecosystem must therefore continuously interact with, analyze, and adapt to sophisticated commercial bot detection mechanisms.

Fingerprint collection on `stellaraio[.]com` is implemented through two first-party JavaScript files:

- `https://stellaraio[.]com/a.js?v=3`
- `https://stellaraio[.]com/i.js?v=4` These scripts transmit structured fingerprint payloads to three internal API endpoints:
- `POST https://ab-api.stellaraio[.]com/px/c`
- `POST https://ab-api.stellaraio[.]com/incapsula/c`

- `POST https://ab-api.stellaraio[.]com/td/c` The endpoint naming and routing structure are operationally significant and are analyzed in detail below.

Castle's Research Team analyzed the fingerprinting scripts loaded from:

- `https://stellaraio[.]com/a.js?v=3`
- `https://stellaraio[.]com/i.js?v=4` Both scripts execute in the main page context and construct structured fingerprint objects that are encrypted prior to transmission. We inspect the payload structure by setting a breakpoint before encryption routines are applied.

Representative excerpts of the decrypted payloads are shown below.

Figure 36: Decrypted payload to `ab-api.stellaraio.com/px/c` observed immediately before encryption.

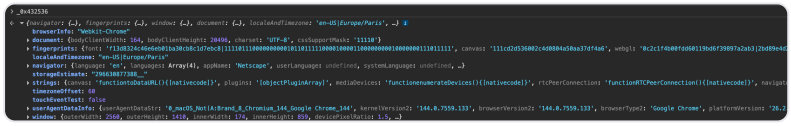


Figure 37: Decrypted payload to `ab-api.stellaraio.com/td/c` observed immediately before encryption.

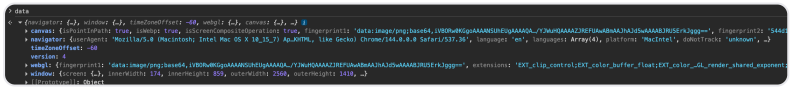


Figure 38: Decrypted payload to `ab-api.stellaraio.com/incapsula/c` observed immediately before encryption.

As in previous case studies, StellarAIO collects a standard set of browser and device attributes.

Signals derived from the `navigator` object include:

```
navigator: {
    brave: !!navigator.brave,
    webdriver: navigator.webdriver,
    share: typeof navigator.share === "function" && navigator.share.toString(),
    plugins: getBrowserPlugins(),
    connection: getNavigatorConnection(),
    language: navigator.language,
    languages: navigator.languages,
    platform: navigator.platform,
    userAgent: navigator.userAgent,
    doNotTrack: navigator.doNotTrack,
    deviceMemory: navigator.deviceMemory,
    hardwareConcurrency: navigator.hardwareConcurrency,
    product: navigator.product,
```

Rendering and media-related signals include:

```
fingerprints: {
      font: generateFontFingerprint(),
      canvas: captureCanvasDrawing(),
      webgl: captureWebGLFingerprint(),
      unmaskedWebgl: getUnmaskedWebGLInfo(),
      mimeTypes: getMimeTypesFingerprint(),
      audio: await captureAudioFingerprint()
    }
```

These primitives are consistent with standard browser fingerprinting practices. On their own, they do not distinguish StellarAIO from defensive fraud-prevention implementations. The distinguishing characteristic lies not in the presence of fingerprinting, but in how the collected telemetry is segmented and structured.

Rather than transmitting a single monolithic fingerprint object, StellarAIO routes fingerprint data into three distinct payload streams:

- `/px/c`
- `/incapsula/c`
- `/td/c` The first two paths are directly aligned with known commercial anti-bot vendors:
- `px` → PerimeterX (HUMAN)

- `incapsula` → Imperva Incapsula Instead of collecting generic site analytics, the implementation appears to model or replicate vendor-specific telemetry streams. Each endpoint receives a differently structured payload, suggesting per-vendor schema alignment.

3.10.2 Indicators of fingerprint harvesting

Several elements in StellarAIO's implementation are consistent with vendor-specific telemetry modeling rather than generic site protection.

StellarAIO transmits fingerprint payloads to:

- `POST https://ab-api.stellaraio[.]com/px/c`
- `POST https://ab-api.stellaraio[.]com/incapsula/c`
- `POST https://ab-api.stellaraio[.]com/td/c` The naming convention itself signals intentional alignment with commercial anti-bot ecosystems.

While `/td/c` cannot be conclusively attributed to a specific vendor based on available artifacts, its structural isolation reinforces the broader segmentation pattern. The existence of multiple vendor-labeled telemetry streams suggests that the system is designed to generate or store vendor-aligned fingerprint datasets rather than simple marketing-site analytics.

Within the `/px/c` payload, Castle's Research Team identified canvas fingerprinting logic containing explicit `PX`-prefixed markers:

```
(() => {
        var _0x5c3332 = "no_fp";
        try {
          var _0x474971 = createCanvasElement(650, 12);
          if (_0x474971) {
            var _0x3f3553 = Fh(_0x474971);
            _0x5c3332 = "PX11982";
            if (_0x3f3553) {
              _0x3f3553.font = "8px sans-serif";
              var _0x302a84 = 1;
              for (var _0x1c2fd4 = 128512; _0x1c2fd4 < 128591; _0x1c2fd4++) {
                _0x3f3553.fillText(N("0x" + _0x1c2fd4.toString(16)), _0x302a84 *
8, 8);
                _0x302a84++;
              }
              _0x5c3332 = xMD5(_0x3f3553.canvas.toDataURL());
            }
          } else {
            _0x5c3332 = "PX12423";
          }
        } catch (_0xa190fe) {
          console.log(_0xa190fe);
          _0x5c3332 = "PX11474";
        }
        return _0x5c3332;
      })()
```

The presence of strings such as:

- `PX11982`
- `PX12423`
- `PX11474` is operationally significant. A GitHub search for `PX11982` links this string to PerimeterX client-side fingerprinting scripts.

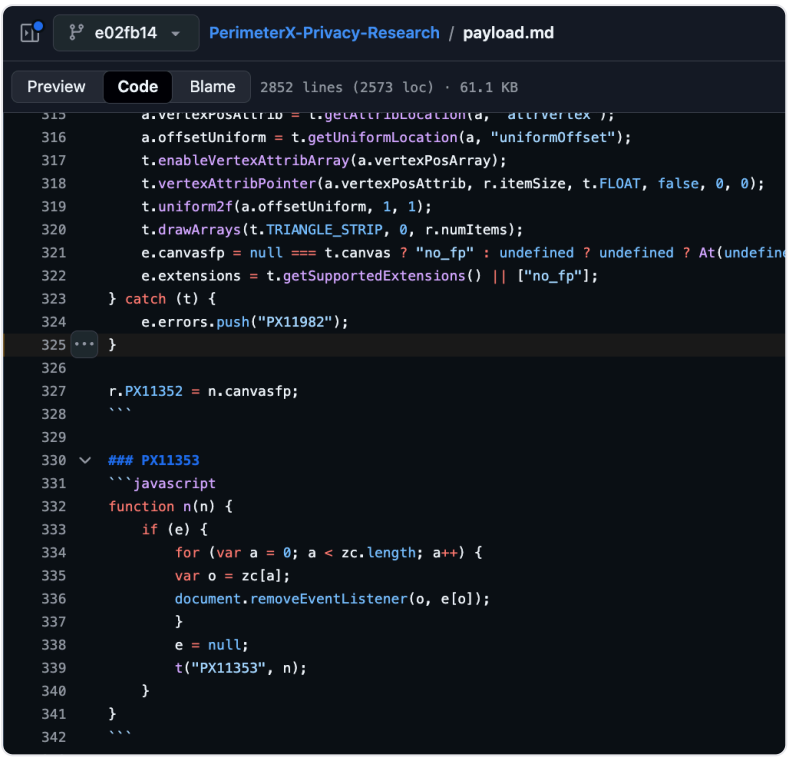


Figure 39: Public reference showing `PX11982` marker associated with PerimeterX client-side sensor logic.

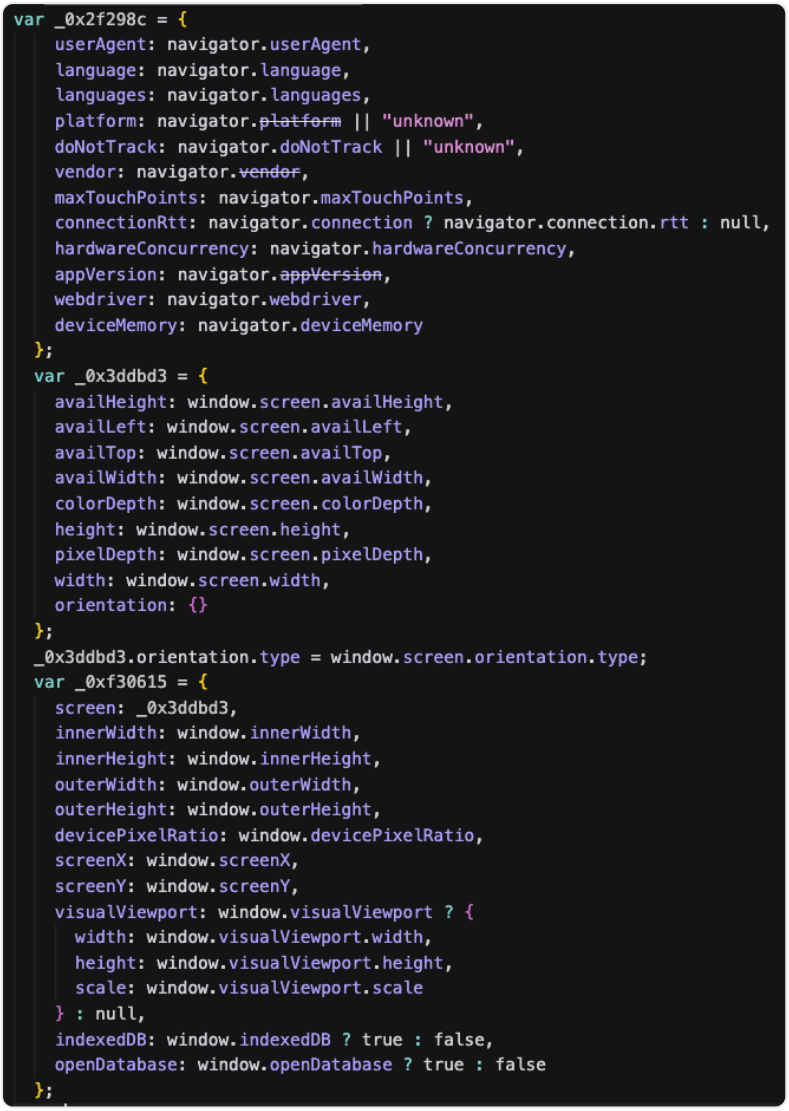
The reuse of distinctive PerimeterX markers strongly suggests derivation from, or modeling of, production PerimeterX client-side challenge code. On a sneaker automation platform, collecting PerimeterX-style challenge outputs aligns more closely with reverse engineering and replay research than with protecting the marketing site itself.

Fingerprint data is also routed to:

- `POST https://ab-api.stellaraio[.]com/incapsula/c` Although this payload does not contain immediately distinctive string markers comparable to the `PX` identifiers, its isolation into a dedicated endpoint remains notable.

The decrypted payload includes:

- Device and browser attributes
- Canvas fingerprint outputs
- WebGL entropy signals
- Feature-detection values



```
var _0x2f298c = {
    userAgent: navigator.userAgent,
    language: navigator.language,
    languages: navigator.languages,
    platform: navigator.platform || "unknown",
    doNotTrack: navigator.doNotTrack || "unknown",
    vendor: navigator.vendor,
    maxTouchPoints: navigator.maxTouchPoints,
    connectionRtt: navigator.connection ? navigator.connection.rtt : null,
    hardwareConcurrency: navigator.hardwareConcurrency,
    appVersion: navigator.appVersion,
    webdriver: navigator.webdriver,
    deviceMemory: navigator.deviceMemory
  };
  var _0x3ddbd3 = {
    availHeight: window.screen.availHeight,
    availLeft: window.screen.availLeft,
    availTop: window.screen.availTop,
    availWidth: window.screen.availWidth,
    colorDepth: window.screen.colorDepth,
    height: window.screen.height,
    pixelDepth: window.screen.pixelDepth,
    width: window.screen.width,
    orientation: {}
  };
  _0x3ddbd3.orientation.type = window.screen.orientation.type;
  var _0xf30615 = {
    screen: _0x3ddbd3,
    innerWidth: window.innerWidth,
    innerHeight: window.innerHeight,
    outerWidth: window.outerWidth,
    outerHeight: window.outerHeight,
    devicePixelRatio: window.devicePixelRatio,
    screenX: window.screenX,
    screenY: window.screenY,
    visualViewport: window.visualViewport ? {
      width: window.visualViewport.width,
      height: window.visualViewport.height,
      scale: window.visualViewport.scale
    } : null,
    indexedDB: window.indexedDB ? true : false,
    openDatabase: window.openDatabase ? true : false
  };
```

Figure 40: Representative Incapsula-aligned payload excerpt (device + rendering + feature detection fields) observed prior to encryption.

Even without explicit string markers, the endpoint naming and structural separation indicate alignment with Imperva Incapsula's client-side telemetry model.

StellarAIO aggregates:

- Baseline device fingerprint attributes
- Canvas, WebGL, audio, and font entropy signals
- PerimeterX-style seeded canvas challenge logic
- Segmented payloads aligned with multiple vendor ecosystems This design suggests that fingerprint data is organized per vendor model rather than collected as a generic analytics artifact.

In sneaker automation communities, operators frequently reverse engineer anti-bot client-side scripts and reproduce expected telemetry using custom HTTP clients instead of full browser automation.

Access to:

- Real device-derived vendor challenge outputs
- Correctly structured, vendor-aligned payload formats
- Multi-vendor telemetry datasets materially reduces the complexity of forging compliant anti-bot requests.

While fingerprint collection alone does not establish harvesting intent, the combination of:

- Explicit vendor endpoint naming
- Reuse of distinctive PerimeterX markers
- Structured multi-vendor telemetry segmentation within an automation product is consistent with collection designed to support anti-detection research and replay workflows rather than purely marketing-site telemetry.

3.11 Indicators of structured fingerprint harvesting activity

Across the four case studies analyzed in this report, Castle's Research Team identified recurring architectural and implementation patterns that are more consistent with structured fingerprint harvesting than with site-local analytics or standard defensive telemetry.

The findings below summarize the cross-case indicators observed.

3.11.1 Multi-vendor telemetry segmentation

Several services route fingerprint telemetry into **distinct payload streams aligned to commercial anti-bot ecosystems**.

Examples include:

- StellarAIO transmitting payloads to:
  - `/px/c`
  - `/incapsula/c`
  - `/td/c`

- Cybersole embedding `akamaiv2` and Adyen-aligned structures
- Ez-CAPTCHA collecting Akamai-style signals and embedding hCaptcha-derived logic This segmentation is operationally significant. Typical analytics or site-local risk scoring pipelines generally consolidate client telemetry into a unified schema. In contrast, vendor-aligned separation suggests the payloads are being structured to resemble (or interoperate with) specific commercial anti-bot telemetry models.

3.11.2 Reuse of vendor-specific client-side artifacts

Multiple scripts contain identifiers, constants, and structural patterns that match reversed commercial anti-bot implementations, including:

- `PX11982` and related PerimeterX-style markers
- Adyen-linked canvas challenge strings such as `"#&*(sdfjlSDFkjls28270("`
- Akamai-aligned function naming (`akaHeadlessFp` , `akaWebGl` , `akamaiv2`)
- hCaptcha-derived font stacks and Unicode rendering sequences
- Arkose-style feature keys using non-semantic / hashed identifiers These are not generic fingerprinting primitives. They reflect vendor-specific challenge logic and telemetry conventions. The recurring presence of distinctive challenge seeds, identifiers, and aggregation patterns across unrelated contexts is more consistent with extraction, replication, or modeling of commercial client-side logic than independent implementation.

3.11.3 Collection of both raw and vendor-derived signals

Across cases, implementations collect **two complementary layers** of fingerprint telemetry:

- **Raw environmental attributes**, such as `hardwareConcurrency` , `deviceMemory` , WebGL renderer strings, canvas output, font availability, codec support, and related device/browser characteristics.
- **Vendor-derived or vendor-shaped outputs**, including hashed challenge artifacts, feature bitmasks, permission encodings, integrity checks, and other derived fields designed to mirror how commercial systems structure or transform client signals. This dual-layer approach materially increases the operational value of the dataset. Raw attributes support device-profile replay; vendor-shaped outputs support emulation of commercial telemetry expectations and reduce the effort required to craft structurally valid payloads.

3.11.4 Operational alignment with botting ecosystems

All analyzed services operate in ecosystems that are operationally adjacent to automation and evasion:

- Disposable email and account-creation tooling
- CAPTCHA-solving (“CAPTCHA farm”) services
- Sneaker/retail automation bots
- Anti-detection and bypass-oriented workflows In these environments, vendor-aligned fingerprint telemetry has clear utility: it supports reverse engineering, reduces the cost of crafting “realistic” client profiles, and enables replay of structured telemetry in automated clients.

While fingerprint collection alone does not prove harvesting intent, the consistent presence of:

- vendor-aligned telemetry segmentation,
- reuse of commercial challenge artifacts,
- combined raw + vendor-shaped signal collection,
- and cross-context enrichment, is consistent with structured fingerprint harvesting activity rather than standard analytics deployments or purely site-local defensive telemetry.

4. Mitigation Strategies

This section outlines defensive measures to reduce the operational value of harvested fingerprints and make replay materially harder at scale.

Fingerprint harvesting only becomes valuable if collected data can be reliably reused. If attackers cannot replay harvested fingerprints, reproduce vendor-aligned signal structures, or generate valid payloads outside a real browser, the economic incentive for large-scale collection decreases.

Recommendations are structured by stakeholder.

4.1 For end users and operators

While this report primarily targets fraud and bot detection teams, individual operators should recognize that visiting untrusted or bot-adjacent services can expose device fingerprints to harvesting scripts.

Users should:

- Avoid interacting with disposable email services, CAPTCHA-solving platforms, and automation tooling sites from primary personal or corporate devices.
- Use isolated environments when analyzing suspicious websites.
- Assume that client-side JavaScript executed in the browser can collect high-entropy fingerprint data.
- Use an ad blocker or privacy extension to reduce exposure to third-party fingerprinting scripts. During our analysis, some collectors (for example, the `fp-flame` scripts) were blocked by common filter lists in tools such as uBlock Origin.
- Prefer browsers with anti-fingerprinting protections when investigating untrusted services (for example, Firefox and Safari's built-in tracking protections, or Chromium-based browsers with anti-fingerprinting countermeasures). This reduces the risk of contributing clean fingerprint data to harvesting operations.

4.2 For organizations and anti-bot vendors

4.2.1 Rotate client-side fingerprinting logic

Static client-side bundles materially lower attacker cost. Once reverse engineered, stable scripts allow:

- Offline payload generation.
- Replay of previously harvested fingerprints.
- Emulation of vendor-specific telemetry formats. To reduce replay feasibility:
- Rotate fingerprinting JavaScript scripts frequently.
- Avoid long-lived, identical script bundles.
- Treat fingerprinting code as part of the security surface, not a static asset. Rotation forces continuous reverse engineering and invalidates older harvested datasets.

4.2.2 Make payload formats and signal definitions dynamic

Fixed JSON schemas and predictable field structures enable attackers to generate compliant payloads outside a browser.

Reduce predictability by introducing controlled variability in:

- Field names and nesting structure.
- Attribute ordering.
- Optional field inclusion.
- Encoding and compression formats.
- Signal representation (for example, bucketed values instead of raw values). The goal is to make replay dependent on continuous validation against the current production implementation.

4.2.3 Bind payloads to execution context

Fingerprint payloads should not be valid in isolation.

Increase replay resistance by:

- Binding payloads to per-session or per-request nonces.
- Tying payload validity to short-lived server-issued tokens.
- Incorporating request-specific context into client-side computation. This increases the cost of offline payload forgery.

4.2.4 Collect context signals, not only device signals

Harvested fingerprints can contain realistic hardware and rendering attributes. Context signals help detect replay outside expected workflows.

In addition to browser attributes, collect lightweight context signals such as:

- Page route or high-level URL identifier.
- Expected DOM structure indicators.
- Execution context (main window vs iframe vs worker).
- Script load ordering. Replay performed via headless clients or partial script emulation often fails to reproduce correct contextual state.

4.2.5 Incorporate timing and integrity signals

Timing and integrity measurements increase the difficulty of high-fidelity spoofing.

Recommended categories include:

- Macro timing, such as timestamp and time since navigation start.
- Micro timing, such as duration of specific fingerprint collection steps.
- Integrity checks for native functions using `Function.prototype.toString()`.
- Cross-context consistency validation across main window, iframe, and worker execution. The objective is not to fingerprint users by timing. It is to increase the difficulty of reproducing internally consistent telemetry outside a real browser environment.

4.2.6 Cross-check redundant signals

Attackers frequently patch a subset of APIs. Redundant signal collection exposes inconsistencies.

Examples:

- Validate WebGL and WebGPU consistency.
- Cross-check reported user agent against feature availability.
- Compare navigator attributes across execution contexts. Multiple weak signals that must agree are harder to forge than a single strong signal.

4.2.7 Treat fingerprint data as untrusted input

Client-side fingerprint data must be treated as attacker-controlled input.

Operational model:

- If signals are clearly inconsistent or malicious, challenge or block.
- If signals appear normal, treat them as probabilistic indicators, not proof of legitimacy. Fingerprinting should be combined with:
- Network reputation and proxy detection.
- Behavioral telemetry.
- Account history and velocity controls.

5. Observables

5.1 Network Observables

IP Address	Domains	Hosting Provider	Country	Description
	cybertemp[.]xyz	Vercel		Disposable email service embedding dual fingerprint collectors
	fp-flame.vercel.app	Vercel		Third-party iframe fingerprint collector embedded on cybertemp and related sites
	monitor[.]ac	Unknown/behind cloudflare		Fingerprint collection endpoint used by fp-flame iframe (`/collector`)
	noinspect[.]top	Vercel		Site loading fp-flame fingerprinting infrastructure
	webora[.]cc	Vercel		Site loading fp-flame fingerprinting infrastructure
35[.]157[.]26[.]135	cyberious[.]xyz	AWS		Site loading `preferencenail[.]com/sfp.js` fingerprint script
185[.]196[.]197[.]71	preferencenail[.]com	DATAWEB GLOBAL LP.	NL	Hosted `sfp.js` fingerprinting script (blocked by EasyList at time of analysis)
	collect.bfgcollect[.]com	Unknown/behind cloudflare		Ez-CAPTCHA fingerprint collection infrastructure (`/collect?tag=ez-protal`)
	ez-captcha[.]com	Unknown/behind cloudflare		CAPTCHA-solving service embedding multi-context fingerprint collection
	cybersole[.]io	Unknown/behind cloudflare		Sneaker automation bot with vendor-aligned fingerprint collector (`/api/collector`)
	stellaraio[.]com	Vercel		Sneaker automation bot with segmented vendor-aligned fingerprint telemetry
66[.]33[.]22[.]1	ab-api.stellaraio[.]com	Railway		StellarAIO fingerprint ingestion endpoints (`/px/c` ,

IP Address	Domains	Hosting Provider	Country	Description
				`/incapsula/c`, `/td/c`)

5.2 File Observables

Filename (Full URL – Defanged)	Description
`https://www.cybertemp[.]xyz/_next/static/chunks/881.35be087306d117a4.js`	Cybertemp first-party fingerprint bundle (injects `X-Client-Meta` header)
`https://fp-flame[.]vercel.app/a/f.js`	fp-flame font, canvas, plugin, and screen fingerprinting module
`https://fp-flame[.]vercel.app/a/g.js`	fp-flame feature detection and Arkose-style hashed signal module
`https://fp-flame[.]vercel.app/a/w.js`	fp-flame WebGL fingerprint module
`https://fp-flame[.]vercel.app/a/c.js`	fp-flame aggregation, encryption, and transmission logic
`https://fp-flame[.]vercel.app/_next/static/chunks/app/page-ce580fedf644df13.js`	fp-flame Next.js bundled collector observed on multiple domains
`https://preferencenail[.]com/sfp.js`	Fingerprinting script observed on cyberious[.]xyz (blocked by EasyList at time of analysis)
`https://collect[.]bfgcollect.com/static/a/a.js?tag=ez-protal`	Ez-CAPTCHA main-page fingerprint collector
`https://collect[.]bfgcollect.com/static/h/h.html?tag=ez-protal`	Ez-CAPTCHA iframe fingerprint collector with embedded Web Worker logic
`https://cybersole[.]io/js/collector.js?v=FZdAwaZh8LRj6wflMULomeDDccIonqNgNwXbRCRu20w`	Cybersole first-party fingerprint collector
`https://stellaraio[.]com/a.js?v=3`	StellarAIO fingerprint collector (segmented telemetry routing)
`https://stellaraio[.]com/i.js?v=4`	StellarAIO fingerprint collector (vendor-aligned payload construction)
`https://api[.]sb.rayobyte.com/fingernaut/pixel.deps.js`	Rayobyte “Fingernaut” structured fingerprinting script

6. Conclusion

This report examined how browser fingerprints are collected and operationalized within bot ecosystems. The findings demonstrate that realistic fingerprint collection and replay are not theoretical risks, but observable operational behaviors embedded within automation-adjacent infrastructure.

Across multiple case studies, we identified:

- Public discussion of collecting fingerprints from real user traffic.
- Deployment of high-entropy fingerprinting scripts on bot-adjacent services.

- Reuse of vendor-derived client-side logic associated with commercial anti-bot and payment providers.
- Segmented payloads explicitly aligned with specific vendor ecosystems.

Taken together, these patterns indicate that browser fingerprints are being treated not merely as session identifiers, but as reusable operational assets.

For fraud and bot detection teams, this reframes the threat model. The relevant question is no longer limited to whether individual fingerprint attributes can be spoofed. It is whether attackers can access and replay real-world fingerprint datasets — including signals structurally aligned with commercial anti-bot systems — in a way that survives server-side validation.

Controls that rely on static JavaScript bundles, stable payload schemas, predictable field names, or long-lived client-side logic are more exposed to reverse engineering and replay. As automation tooling matures and fingerprint harvesting becomes more structured, the marginal cost of operationalizing realistic fingerprints is likely to decrease.

At the same time, this research highlights an often-overlooked upstream factor: harvesting requires supply. Bot-adjacent services rely on real user traffic to generate clean, high-entropy fingerprint datasets. End users who interact with disposable email platforms, CAPTCHA-solving services, proxy tooling, or other automation ecosystems may unknowingly contribute production-quality fingerprint data. Isolating analysis environments, using privacy-focused browsers with anti-fingerprinting protections, and employing script-blocking or filtering tools can reduce the quality and volume of fingerprints available for harvesting.

For defenders, the priority should be replay resilience rather than static signal strength. Effective defensive strategies:

- Reduce the standalone replay value of collected fingerprints.
- Introduce controlled variability in client-side logic and payload structure.
- Bind fingerprint payloads to short-lived server-issued context.
- Validate consistency across multiple execution environments.
- Incorporate timing, integrity, and contextual signals.
- Combine fingerprint-derived signals with behavioral telemetry, network intelligence, and server-side risk indicators. Fingerprinting should be treated as an adaptive, probabilistic risk signal — not a static identifier or standalone verdict.

As attackers incorporate vendor-aligned telemetry and real-world fingerprint datasets into their tooling, defenders should expect continued iteration. Systems designed to withstand replay pressure, schema modeling, and vendor-specific emulation will remain more resilient than those that depend on the stability of any single client-side attribute.

The core shift is conceptual: browser fingerprints are no longer just something to collect — they are something adversaries can collect too. Defensive design must account for that symmetry.