

A Multi-Layered Approach to Browser Fingerprinting Evasion

Elia Byrne

Abstract

Browser fingerprinting has largely supplanted cookies as the dominant tracking primitive, and the same techniques increasingly underpin anti-fraud, anti-bot, and forensic-evasion detection. This paper presents a two-layer evasion architecture that synchronizes hypervisor-level hardware masking with browser-level JavaScript interception, and documents an iterative implementation of each layer measured against empirical detection instruments (CreepJS, BrowserLeaks). Layer 1 hardens a KVM/QEMU virtual machine’s SMBIOS, CPUID, timing, and peripheral surface into an internally coherent consumer-laptop persona, validated directly against BrowserLeaks’ unmasked WebGL renderer report. Layer 2 develops a Firefox Manifest V3 extension, denoted v0.1 through v0.3.1, that reduces CreepJS’s prototype-tampering lie count from 2,584 lies across 2,507 of 3,395 properties to zero—not through progressively more sophisticated spoofing, but through the reverse: each surface for which Layer 1 already supplies a coherent native value (WebGL parameters, font metrics, layout geometry) is ultimately best served by removing the JavaScript interception entirely, retaining hooks only where no native substitute exists (Canvas, Audio, WebRTC). The result is a case study in the limits of application-layer fingerprint evasion, and an argument that hardware- and OS-level delegation, rather than deeper JavaScript spoofing, is the more sustainable strategy. Implementation, configuration, and version history are available at <https://gitlab.cylab.be/cylab/students/fingerprinting-evasion>.

1 Introduction

Tracking on the web is not inherently adversarial. A substantial share of it is infrastructural: session identifiers keep a user authenticated across page loads, shopping-cart state persists between requests, load balancers pin a client to a consistent backend, and fraud- and bot-detection systems rely on some notion of client continuity to protect accounts [1]. This legitimate, session-scoped tracking is narrow in scope—it identifies a client to the one site it is currently transacting with, for the duration of that transaction.

The tracking this paper is concerned with is a different, broader phenomenon: cross-site behavioral profiling, in which a client is re-identified across unrelated domains over long time horizons, typically to build an advertising or data-broker profile without the user’s informed awareness [1, 2]. Historically, this relied upon stateful mechanisms, primarily HTTP cookies and local storage, alongside a range of persistent respawning techniques (e.g., evercookies) designed to survive a user’s own deletion attempts [5]. The advent of browser fingerprinting marked a paradigm shift toward stateless tracking for exactly this profiling use case [3]. This technique leverages the inherent entropy within a user’s hardware and software configuration to compute a highly unique identifier. By querying JavaScript APIs for graphics rendering capabilities, audio processing nuances, and system hardware specifications, tracking scripts extract a unique signature that persists even when traditional tracking data is purged—and, unlike a session cookie, follows the user across sites rather than being confined to one.

Recent advancements have transitioned from client-side fingerprinting—where the browser computes an exact cryptographic hash—to server-side commercial fingerprinting. Modern solutions deploy machine-learning algorithms that prioritize stable hardware attributes over easily spoofed software metrics. This evolution necessitates a fundamental reevaluation of how evasion environments are constructed.

2 Motivations for Evasion Deployment

The development and deployment of fingerprinting evasion techniques are driven by distinct operational needs, ranging from individual privacy to complex state-sponsored or law enforcement investigations.

2.1 Anti-Profiling and Consumer Privacy

At the consumer level, evasion tools mitigate aggressive behavioral profiling by commercial entities. Users deploy Anti-Fingerprinting Privacy Enhancing Technologies (AFPETs) to prevent the aggregation of their browsing habits, financial data, and personal preferences across disparate web domains. However, poorly configured AFPETs often inadvertently increase a user’s entropy, making them more mathematically unique to trackers. The same tooling is dual-use: browser anti-fingerprinting techniques developed for legitimate privacy purposes have themselves been documented as an enabling pathway for impersonation and payment fraud [19], a tension this paper’s own motivations (Sections 2.2 and 2.3) do not resolve so much as operate alongside.

2.2 Malware Analysis and Reverse Engineering

Cybercriminals frequently deploy cloaking mechanisms within their infrastructure. If a Command and Control (C2) server or a phishing kit detects an automated web crawler, a headless browser, or a known sandbox environment, it will refuse to serve the malicious payload. To successfully extract and preserve forensic evidence, analysts must utilize environments that perfectly mimic vulnerable, genuine endpoints. Furthermore, these environments must be highly dynamic. Because campaign-specific artifacts—such as the active C2 domains, execution workflows, or the specific names of dropped binaries—often rotate depending on the month of execution, the target machine name, or specific victim identifiers, the evasion architecture must be capable of programmatically altering its deep system fingerprint to maintain the illusion of a fresh, plausible victim for every analysis cycle.

2.3 Law Enforcement and Undercover Operations

For Law Enforcement Agencies (LEA) and Open Source Intelligence (OSINT) investigators, evasion is a matter of critical operational security. When infiltrating illicit networks, forums, or adversarial infrastructure, investigators must utilize Managed Attribution systems. Adversaries actively monitor their server logs for anomalous fingerprints or IP ranges associated with government data centers. To avoid operational deanonymization and prevent the burning of investigative infrastructure, agents must utilize environments that perfectly blend into the baseline “grey noise” of the internet, ensuring multiple undercover personas cannot be correlated to a single physical machine.

3 Fingerprinting Parameters and Entropy Sources

Advanced tracking algorithms evaluate a broad spectrum of parameters to formulate an identity [3]. These can be categorized into three primary layers:

- **The Application (JS/Browser) Layer:** This includes high-entropy APIs such as the HTML5 Canvas and WebGL, which rely on the underlying GPU drivers to render images, producing pixel-level variations unique to the specific hardware [4]. It also encompasses the AudioContext API [13], font enumeration techniques [12], and navigator properties (e.g., *hardwareConcurrency* and *deviceMemory*), all of which contribute to the broader device-fingerprinting ecosystem [6, 7].
- **The System/Hardware Layer:** For environments operating within Virtual Machines (VMs), parameters such as SMBIOS strings, MAC Address OUIs, CPUID hypervisor flags, and paravirtualized driver signatures (e.g., QXL graphics or VirtIO controllers) are scrutinized to detect automated or isolated environments [6].

- **The Network Layer:** Before the web server processes an HTTP request, the network stack is analyzed. This includes passive OS fingerprinting (TCP/IP TTL and window sizes) and TLS/SSL fingerprinting (JA3/JA4 hashes) [14], which evaluate the specific order and support of cryptographic ciphers to verify if they match the stated User-Agent. JA4 in particular has recently been adopted directly as a machine-learning feature for web bot detection, rather than merely a passive identifier [24].

4 Existing Solutions

Countermeasures to the entropy sources catalogued in Section 3 fall into three broad camps, distinguished by *where* they intervene and by the strategy they use once there.

4.1 Anonymity-Set Approaches: Making Everyone Look the Same

The Tor Browser is the canonical example of this strategy: rather than varying the fingerprint per user, it fixes nearly every high-entropy parameter (window dimensions via letterboxing, installed-font enumeration, timezone, canvas/audio readback) to one identical, publicly-known value shared by every Tor Browser installation, so that the fingerprinting surface itself carries no distinguishing information [15]. Firefox’s `privacy.resistFingerprinting` preference, inherited from the same Tor Uplift effort, brings a subset of this strategy to mainline Firefox [16]; community-maintained configuration guides such as `arkenfox/user.js` layer additional preference hardening on top of it for users who cannot accept Tor Browser’s stricter usability constraints [18]. The approach is effective in direct proportion to its adoption: an anonymity set is only as large as the population using an identical configuration, and any deviation from that fixed baseline (a non-default window size, a locally installed font) re-introduces the entropy the strategy was designed to eliminate.

4.2 Noise and Randomization-Based Approaches

A second camp perturbs high-entropy outputs directly rather than fixing them to a common value. Brave’s “farbling” technique seeds a per-session pseudo-random noise generator that perturbs Canvas, WebGL, and AudioContext readback on a per-site basis, conceptually close to this paper’s own PRNG-seeding design in Section 8. Academic proposals in the same family include PriVaricator, which injects randomized “little white lies” into fingerprinting-relevant JavaScript properties [9]; FPGuard, which combines runtime detection of fingerprinting scripts with targeted randomization of the properties they query [20]; FP-Block, which controls fingerprintable properties on a per-domain policy basis [10]; and the Disguised Chromium Browser, which patches Canvas and Flash fingerprinting surfaces directly at the browser-engine level rather than via an injected content script [11]. This paper’s own Layer 2 (Section 8) belongs to this camp for the surfaces it does still hook (Canvas, Audio, WebRTC), and Section 5.1 details the specific failure mode—detectable statistical noise signatures—that motivated progressively removing hooks rather than refining them further.

4.3 System- and Hardware-Level Approaches

A third camp, to which this paper’s Layer 1 belongs, intervenes below the browser entirely: rather than controlling what the browser reports, it controls what is true of the underlying machine. Commercial “anti-detect browser” products (e.g., Multilogin, GoLogin) combine a modified browser build with virtualized or containerized hardware personas, marketed primarily at ad-verification, sneaker/ticket botting, and multi-account management use cases. Academic treatment of this specific layer is comparatively sparse relative to the browser-level literature above; most published VM/hypervisor-detection work (e.g., CPUID hypervisor-bit and timing-based checks) is framed from the *detection* side—malware sandbox evasion—rather than the countermeasure side this paper develops, which is itself indicative of the gap this paper’s Layer 1 addresses.

4.4 Measurement and Detection Tooling

A closely related, adversarial category is the audit tooling used to evaluate the three camps above. AmIUnique quantifies how distinguishing a given fingerprint is against a crowd-sourced population [7]. CreepJS goes further, specifically targeting the anonymity-set and randomization camps by cross-examining the API surface for the internal inconsistencies an injected countermeasure tends to leave behind (Section 8.1 develops this in detail) [17]. This paper uses CreepJS as its primary empirical instrument for exactly this reason: it is designed to detect the class of countermeasure Layer 2 implements, making a favorable result against it a meaningful signal rather than a foregone conclusion.

More recent work has moved from manual heuristics toward automated, large-scale detection of fingerprinting *scripts* themselves rather than of the countermeasures against them: FP-Inspector applies syntactic-semantic machine learning to classify fingerprinting behaviour directly from static and dynamic JavaScript analysis [22], while FP-Tracer uses taint-tracking and joint-entropy thresholds to detect fine-grained, even obfuscated, fingerprinting activity at scale [23]. Datta et al. take the complementary evaluator’s perspective this section has been building toward, systematically measuring how well deployed anti-fingerprinting PETs actually perform against real fingerprinting scripts in the wild [21]—the same evaluative posture CreepJS applies to this paper’s own Layer 2, but at web-scale rather than single-target.

5 Limitations of Existing Solutions

Empirical analysis demonstrates that superficial evasion techniques are fundamentally flawed against modern, probabilistic tracking.

5.1 The Failure of Partial Masking and Randomization

Injecting mathematical noise into the Canvas or WebGL APIs (the “Noise Strategy”) succeeds in altering the client-side hash but fails against server-side algorithms. Solutions like Fingerprint Pro have learned to recognize this artificial noise. Because these attributes are heavily targeted by privacy tools, modern algorithms deprioritize them, anchoring the user’s identity to unchangeable hardware metrics (e.g., CPU cores and memory limits). Changing these fundamental attributes in isolation triggers severe “Tampering” or “Bot” heuristics.

5.2 The Virtualization Trap

Deploying a standard Virtual Machine (e.g., QEMU/KVM, VirtualBox) provides a clean state but generates a highly recognizable “virtual” fingerprint. Paravirtualized display adapters (such as the QXL controller or VMware’s SVGA 3D device), generic emulated audio controllers, and characteristic MAC OUIs instantly flag the environment as a non-physical endpoint. The critical insight, developed further in Section 6.2, is that the manner in which such an artifact is concealed matters as much as whether it is concealed: a virtual adapter masked only at the JavaScript layer constitutes a single point of failure, whereas substituting it for a natively plausible device provides defense in depth.

6 Proposed Scenario: A Multi-Tiered Evasion Architecture

To successfully evade advanced tracking and anti-analysis cloaking, the proposed solution is a multi-layered evasion architecture that synchronizes the Hypervisor and the Browser Engine.

6.1 Layer 1: Deep Hypervisor Masking (System Level)

The foundation of the evasion strategy relies on a heavily modified KVM/QEMU hypervisor configuration. To defeat hardware-level introspection, the virtual machine must present hardware identifiers

indistinguishable from a genuine consumer endpoint.

- **CPUID and Timing:** The CPUID hypervisor-present bit is cleared¹, and the KVM signature is hidden from the guest OS. The Time Stamp Counter (TSC) is anchored to a native timer and VM-exits on timing instructions are suppressed², mitigating the microsecond-scale latency jitter associated with VM-exits that characterize timing-based hypervisor detection.
- **SMBIOS Injection:** SMBIOS data modelled on genuine consumer hardware (e.g., a Lenovo IdeaPad chassis) is injected directly into the QEMU/libvirt domain definition. This aligns the system manufacturer, BIOS version and serial, baseboard identifiers, and memory DIMM profiles with a coherent, real-world configuration.
- **Peripheral Scrubbing:** Conspicuous paravirtualized artifacts—such as the VirtIO balloon device and QEMU-default disk model strings—are removed or relabelled to mimic standard consumer SATA/NVMe storage, and the network adapter’s MAC OUI is reassigned to match the emulated controller’s real-world vendor.
- **Persona Coherence:** Critically, because the host-passthrough CPU mode exposes the physical processor’s identity through CPUID, the entire spoofed persona is anchored to the real host architecture rather than contradicting it. An internally consistent “plausible” machine defeats probabilistic trackers more reliably than a higher-specification persona riddled with contradictions.

6.2 Layer 2: Browser and Application Control

Once the hypervisor is hidden, the browser itself must be controlled to project a “Golden Profile”—a fingerprint that looks exactly like millions of normal users, allowing the session to blend into regular internet traffic.

The most secure approach theoretically involves modifying the browser’s actual source code (e.g., compiling a custom Chromium fork) so that it natively reports forged hardware values at the C++ level. However, editing and recompiling a browser engine is highly complex, computationally resource-intensive, and inherently fragile when attempting to maintain parity with upstream zero-day security patches. Consequently, practical and dynamic evasion necessitates reliance on Application-Layer JavaScript interception.

6.2.1 Browser Extensions and JavaScript Proxy Membranes

The primary vector for Application-Layer evasion utilizes standard WebExtension frameworks to inject JavaScript directly into the target webpage’s execution environment. By leveraging ECMAScript 6 features, primarily the `Proxy` and `Reflect` objects, evasion scripts can intercept calls to high-entropy deterministic APIs (such as the HTML5 Canvas or AudioContext) and dynamically manipulate the returned data.

To effectively cloak the session, the injected script must operate synchronously at document initialization (e.g., `document_start`) to ensure hooks are established before tracking scripts execute. Furthermore, because advanced trackers employ sophisticated heuristics to detect the presence of privacy extensions—measuring execution latency, traversing prototype chains³, and verifying WebIDL speci-

¹Leaf 0x1, ECX bit 31—the de facto flag every hypervisor sets to signal virtualization, read by sandboxes and anti-cheat clients as a cheap VM check.

²A VM-exit traps guest execution to the hypervisor to emulate privileged instructions (e.g. RDTSC); each exit costs far more cycles than native execution, exposing a timing signature to a detector bracketing the call.

³Repeated `Object.getPrototypeOf()` calls until reaching `null`; an intercepting `Proxy` can introduce an unexpectedly-shaped link a careful walk exposes.

fication compliance⁴—the JavaScript membrane must be engineered in a way that the information reported is sound, and correctly mimics the original underlying functions. The interception logic cannot merely spoof the data; it must comprehensively emulate the exact invocation semantics, error handling, and object descriptor properties of the native C++ browser engine.

7 Implementation: Layer 1

The masking strategy outlined in Section 6.1 was implemented against a concrete test environment: a Debian Linux host running KVM/QEMU 7.2 (libvirt 9.0) on an AMD Ryzen 7 3700U mobile platform, with a Windows 11 (24H2) guest. This section documents the principal hardening decisions and their reflection in the libvirt domain XML.

7.1 Persona Coherence as a Design Constraint

The single most consequential decision governing Layer 1 was the recognition that the spoofed persona must follow the silicon rather than contradict it. Under the `host-passthrough` CPU mode, the guest reads the physical processor’s brand string directly from CPUID leaves `0x80000002–0x80000004`; this value cannot be rewritten without abandoning passthrough for a hand-constructed CPU model, which in turn reintroduces numerous feature-flag inconsistencies and a substantial performance penalty. Consequently, the entire SMBIOS persona was anchored to the real host architecture: a Lenovo IdeaPad 5 15ARE05, an AMD Ryzen-based consumer laptop that genuinely shipped with the 3700U. This guarantees agreement between the immutable CPUID brand string and every spoofed SMBIOS field, eliminating the class of contradiction that probabilistic trackers exploit.

7.2 SMBIOS and Firmware Reflection in the Domain XML

The firmware identity was redirected away from QEMU defaults via the `<smbios mode='sysinfo'/>` directive, instructing the guest to read the values declared in the `<sysinfo>` block. Within that block, the Type 0 (BIOS), Type 1 (System), Type 2 (Baseboard), and Type 3 (Chassis) tables were populated with values consistent with the chosen persona: vendor `LEN0V0`, product code `81YQ`, board identifier `LNVNB161216`, and a BIOS version string (`E4CN28WW`) whose release date postdates the board’s market availability.

```
1 <sysinfo type='smbios'>
2   <bios>
3     <entry name='vendor'>LEN0V0</entry>
4     <entry name='version'>E4CN28WW</entry>
5     <entry name='date'>03/22/2021</entry>
6   </bios>
7   <system>
8     <entry name='manufacturer'>LEN0V0</entry>
9     <entry name='product'>81YQ</entry>
10    <entry name='version'>IdeaPad 5 15ARE05</entry>
11    <entry name='serial'>PF3N4R2B</entry>
12  </system>
13  <baseBoard>
14    <entry name='manufacturer'>LEN0V0</entry>
15    <entry name='product'>LNVNB161216</entry>
16  </baseBoard>
17 </sysinfo>
```

Listing 1: Excerpt of the SMBIOS `<sysinfo>` block in `win11-hardened.xml`, populating the Type 0/1/2 tables with the Lenovo IdeaPad 5 15ARE05 persona.

⁴The W3C/WHATWG contract every native API follows for receiver/argument validation and exception types; a generic `TypeError` where `Illegal invocation` is mandated betrays a reimplementation.

The processor (Type 4) and memory-device (Type 17) tables, which the libvirt 9.0 schema does not expose declaratively, were injected directly through QEMU command-line arguments under the `qemu:commandline` namespace, declaring two plausible Samsung LPDDR4X DIMM profiles consistent with the model's real memory configuration.

```

1 <qemu:commandline>
2   <qemu:arg value='-smbios' />
3   <qemu:arg value='type=4,sock_pfx=FP6,manufacturer=Advanced Micro Devices Inc
4     .,version=AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx,max-speed=2300,
5     current-speed=2300' />
6   <qemu:arg value='-smbios' />
7   <qemu:arg value='type=17,loc_pfx=ChannelA-DIMM,manufacturer=Samsung,part=
8     K4UBE3D4AA-MGCL,speed=4266' />
9 </qemu:commandline>

```

Listing 2: Type 4 (processor) and Type 17 (memory) SMBIOS tables injected via `qemu:commandline`, since libvirt 9.0's `<sysinfo>` block has no equivalent for either.

7.3 Hypervisor and Timing Concealment

Three coordinated modifications⁵ conceal the virtualization boundary at the level the guest can probe. First, the CPUID hypervisor-present bit (leaf 0x1, ECX bit 31) is cleared via `<feature policy='disable' name='hypervisor' />`, since `host-passthrough` alone does not unset it. Second, the `<kvm><hidden state='on' /></kvm>` directive suppresses the KVMKVMKVM signature otherwise readable from the hypervisor CPUID leaf range 0x40000000–0x400000FF, while the Hyper-V enlightenment vendor identifier is overwritten with a neutral string. Third, to address timing-based introspection, the configuration removes the contradictory `hypervclock` timer and anchors the guest to a native TSC timer.

```

1 <features>
2   <hyperv mode='custom'>
3     <vendor_id state='on' value='AuthenticAMD' />
4   </hyperv>
5   <kvm>
6     <hidden state='on' />
7   </kvm>
8 </features>
9 <cpu mode='host-passthrough' check='none' migratable='off'>
10   <feature policy='disable' name='hypervisor' />
11 </cpu>
12 <clock offset='localtime'>
13   <timer name='hpet' present='no' />
14   <timer name='tsc' present='yes' />
15 </clock>

```

Listing 3: Excerpt of the `features/cpu/clock` blocks concealing the hypervisor boundary and normalizing guest timing.

The native-tickpolicy TSC timer is paired with a KVM acceleration argument on the QEMU command line, since libvirt's XML schema has no declarative equivalent for capability-level exit tuning:

```

1 <qemu:arg value='-accel' />
2 <qemu:arg value='kvm,dirty-ring-size=4096' />

```

Listing 4: KVM acceleration argument reducing exit-induced timing jitter, from the same `qemu:commandline` block as Listing 2.

⁵Concretizes Section 6.1's CPUID/TSC concepts and additionally covers the paravirtualization signature leaves and Hyper-V timer enlightenment.

7.4 Peripheral Scrubbing

Obvious paravirtualized and emulated-device artifacts were either removed or relabelled. The display adapter was migrated from QXL—which advertises a conspicuous “Red Hat QXL controller” string through every layer up to and including WebGL—to the standard VGA model, which Windows services with the Microsoft Basic Display Adapter and its WARP software rasterizer. This choice constitutes the architectural pivot described in Section 6.2: rather than masking the QXL string at the JavaScript layer (a single point of failure, trivially bypassed by querying a fresh rendering context inside an `<iframe>`), the underlying device is replaced with one whose native readback behaviour is internally consistent with its reported identity. Should a tracker bypass any higher-layer control, the truth it discovers—a driverless physical machine—remains plausible.

```
1 <video>
2   <model type='vga' vram='16384' heads='1' primary='yes' />
3 </video>
4 <interface type='network'>
5   <mac address='8C:8D:28:4A:B3:1F' />
6   <model type='e1000e' />
7 </interface>
8 <memballoon model='none' />
```

Listing 5: Device XML: QXL replaced with standard VGA, MAC OUI reassigned to Intel, memory ballooning disabled.

The memory-balloon device is disabled outright above, since its VirtIO guest-agent dependency is itself a VM-specific artifact. The SATA disk and optical-drive model strings, meanwhile, were overridden from “QEMU HARDDISK” and “QEMU QEMU DVD-ROM” to plausible Samsung and HL-DT-ST identifiers, with the disk serial populated to a format-correct value. These have no declarative XML equivalent and are set as `-global QEMU` arguments in the same `qemu:commandline` block as Listing 2:

```
1 <qemu:arg value='-global' />
2 <qemu:arg value='ide-hd.model=Samsung SSD 870 EVO 500GB' />
3 <qemu:arg value='-global' />
4 <qemu:arg value='ide-cd.model=HL-DT-ST DVDROM GUE1N' />
```

Listing 6: ATA IDENTIFY model/serial override, correcting Win32_DiskDrive.Model from the QEMU default.

7.5 TPM Emulation and Accepted Residual Risk

Two further decisions round out the device configuration, neither reducible to a single “masked or not” judgment. First, a TPM 2.0 device is emulated:

```
1 <tpm model='tpm-crb'>
2   <backend type='emulator' version='2.0' />
3 </tpm>
```

Listing 7: Emulated TPM 2.0, required by Windows 11 setup and consistent with the near-universal presence of TPM on genuine consumer hardware post-2016.

This is simultaneously a functional requirement (Windows 11 refuses to install without a TPM absent a registry bypass, and that bypass is itself a documented signal of a non-standard installation) and a stealth measure: a real, unmanaged consumer laptop from this era ships with TPM 2.0 enabled by default, so its absence would be the more conspicuous choice.

Second, the SPICE agent channel and its backing `virtio-serial` controller are kept, despite surfacing as “VirtIO Serial Driver” and a SPICE channel in Windows Device Manager—artifacts a sufficiently determined OS-level inspection would flag as virtualization evidence:

```

1 <channel type='spicevmc'>
2   <target type='virtio' name='com.redhat.spice.0' />
3 </channel>

```

Listing 8: SPICE agent channel, kept for host clipboard integration despite being an OS-level VM artifact.

This is a deliberate, scoped risk acceptance rather than an oversight: Device Manager entries are not reachable from any web API, so this artifact cannot surface through the browser-facing attack surface this paper is otherwise concerned with, and removing it would forfeit host clipboard integration for no reduction in browser-observable entropy.

7.6 Validation Outcome

Direct inspection from within the guest browser, using BrowserLeaks’ WebGL report (Figure 1), confirmed that the WebGL surface carries no virtualization artifact. Critically, this holds under the `WEBGL_debug_renderer_info` extension’s *unmasked* vendor/renderer query—the specific channel trackers use to defeat naive spoofing, since it bypasses whatever the plain WebGL context reports and reads the underlying driver identity directly. Both the plain and unmasked renderer strings resolve identically to ANGLE (Microsoft, Microsoft Basic Render Driver (0x0000008C) Direct3D11 vs_5_0 ps_5_0), and the reported HTTP User-Agent (Windows NT 10.0; Win64; x64) is consistent with the declared persona.

WebGL Browser Report

WebGL is a JavaScript API used to render 3D graphics within a web browser by utilizing the device's GPU. This makes it possible for websites to gather detailed information about a user's graphics card, which can be used to create a unique browser fingerprint.

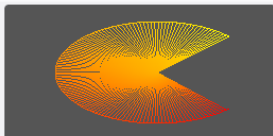
Your Web Browser :

HTTP User-Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:155.0) Gecko/20100101 Firefox/155.0
-----------------	--

WebGL Support Detection :

This browser supports WebGL	✓ True
This browser supports WebGL 2	✗ False (supported, but disabled or unavailable)

WebGL Fingerprint :

WebGL Report Hash	458B5C00986203A2BC4E4473C004A6E0 JSON Dump
WebGL Image Hash	2878045227C46271D55F79DDC98B6C08
WebGL Image	

WebGL Context Info : [raw keys](#)

Supported Context Name(s)	webgl
GL Version	WebGL 1.0
Shading Language Version	WebGL GLSL ES 1.0
Vendor	Mozilla
Renderer	ANGLE (Microsoft, Microsoft Basic Render Driver (0x0000008C) Direct3D11 vs_5_0 ps_5_0), or similar

Debug Renderer Info :

Unmasked Vendor	! Google Inc. (Microsoft)
Unmasked Renderer	! ANGLE (Microsoft, Microsoft Basic Render Driver (0x0000008C) Direct3D11 vs_5_0 ps_5_0), or similar

Figure 1: BrowserLeaks WebGL report from inside the hardened guest. The unmasked renderer—queried via `WEBGL_debug_renderer_info`, the extension trackers use specifically to unmask spoofed WebGL identities—agrees with the plain renderer string, both resolving to the Microsoft Basic Render Driver (WARP) rather than any QEMU/QXL artifact.

The screen properties and audio readback path were not independently re-verified with a separate tool, but are expected to behave identically to WebGL: both are delivered by the same unmodified Win-

dows APIs and WARP rasteriser, with no JavaScript-layer interception involved at Layer 1. Navigator and Canvas, however, were checked directly.

Navigator Object :	
userAgent	Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:155.0) Gecko/20100101 Firefox/155.0
appVersion	5.0 (Windows)
appName	Netscape
appCodeName	Mozilla
product	Gecko
productSub	20100101
vendor	empty
vendorSub	empty
buildID	20181001000000
platform	Win32
oscpu	Windows NT 10.0; Win64; x64
hardwareConcurrency	8
deviceMemory	undefined
devicePosture.type	undefined
language	en-US
languages	["en-US","en"]
onLine	true
doNotTrack	unspecified
cookieEnabled	true
maxTouchPoints	0
webdriver	false
pdfViewerEnabled	true
globalPrivacyControl	false

Figure 2: The navigator object as reported from inside the hardened guest. hardwareConcurrency reads 8, matching the domain XML’s <vcpu placement=’static’>8</vcpu> exactly; oscpu and platform agree with the declared Windows persona; deviceMemory is undefined, which is correct Firefox behaviour (the property is Chromium-only) rather than a gap; webdriver is false.

Figure 2 confirms the claim that navigator fields are “delivered by Layer 1 vCPU topology” (Table 2) is not merely architectural intent: hardwareConcurrency traces directly to the domain XML’s <vcpu placement=’static’>8</vcpu> declaration (Section 7), with zero JavaScript involved, since Layer 2 removed its navigator hooks entirely by v0.3.1.

Canvas readback is a different case: unlike navigator, it is still actively hooked at Layer 2 (gamma-shift noise), so a canvas fingerprint reflects the full pipeline—WARP rendering physics from Layer 1, then the seeded multiplier from Layer 2—rather than Layer 1 in isolation.

Canvas Fingerprinting

The Canvas API, which is designed for drawing graphics via JavaScript and HTML, can also be used for online tracking via browser fingerprinting. This technique relies on variations in how canvas images are rendered on different web browsers and platforms to create a personalized digital fingerprint of a user's browser.

Canvas Support Detection :

Canvas 2D API	✓ True
Text API for Canvas	✓ True
Canvas toDataURL	✓ True

Canvas Fingerprint :

Signature	11E956FFC77810DCC33CF59DD4AAC37D
Uniqueness	100% (The signature is unique to our database)

Image File Details :

File Size	3936 bytes
Number of Colors	111



Figure 3: BrowserLeaks Canvas report for the same session. The rendered image, signature, and byte size are all well-formed and internally consistent; nothing about the output looks structurally anomalous, despite carrying the Layer 2 gamma-shift perturbation.

The reported “100% uniqueness” in Figure 3 is not itself a red flag: exact canvas hash collisions are rare even across genuine, unmodified physical machines, which is precisely why canvas fingerprinting is used as a tracking vector in the first place. What matters here is the absence of structural anomalies (malformed image data, an inconsistent byte size, a broken signature format) that would betray interception—consistent with the “0 lies” CreepJS result for this same hook (Table 1).

As a final, out-of-scope check, the network-layer fingerprint discussed in Section 3 was also inspected, since neither Layer 1 nor Layer 2 as designed touches the TLS stack at all:

TLS Client Test

This page displays your web browser's SSL/TLS capabilities, including supported TLS protocols, cipher suites, extensions, and key exchange groups. It highlights any weak or insecure options and generates a TLS fingerprint in JA3/JA4 formats. Additionally, it tests how your browser handles insecure mixed content requests.

Your Web Browser :

HTTP User-Agent

Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:155.0) Gecko/20100101 Firefox/155.0

Protocol Support :

TLS 1.3

✔ Enabled

TLS 1.2

✔ Enabled

TLS 1.1

✗ Disabled (Good)

TLS 1.0

✗ Disabled (Good)

Mixed Content Test :

Active Content

✔ Blocked

Passive Content

✔ Upgraded to HTTPS

TLS Fingerprint :

JA4

t13d1517h2_8daaf6152771_3cbfd9057e0d

Expand

JA4_o

t13d1517h2_77608d807651_df135d36b6d5

JA3

424f6d9c8b8928c0a0489a4f1a0f3e89

JA3_n

1d9334692003212ed6c01718554b1b1c

TLS Handshake :

TLS Protocol

0x0304 TLS 1.3

Cipher Suite

0x1301 TLS_AES_128_GCM_SHA256 Recommended

TLS 1.3

Key Exchange

0x11EC X25519MLKEM768

Signature Scheme

0x0403 ecdsa_secp256r1_sha256

Figure 4: TLS client fingerprint from inside the guest. TLS 1.3/1.2 only, a modern post-quantum-hybrid key exchange (X25519MLKEM768), and a JA3/JA4 hash consistent with an unmodified, up-to-date Firefox 155 install—as expected.

The JA3/JA4 hash in Figure 4 is exactly what the unmodified Firefox TLS stack would produce on any machine; it is included not because either layer manipulates it, but to confirm the hypervisor introduces no network-stack anomaly (unusual ClientHello ordering, fragmentation quirks) that a passive network fingerprinter could key on. This is not a purely academic concern: JA4 fingerprints are already being used operationally as a bot-detection feature [24], so a JA4 hash that looked anomalous would itself be a meaningful tell entirely outside the scope of either layer’s controls. The residual artifacts that remain are confined to the OS-introspection layer and are not reachable from the browser:

- The OVMF firmware’s internal ACPI vendor strings (e.g., a BOCHS string in the DSDT/FADT tables).
- The QEMU machine-type description (“QEMU Standard PC (Q35 + ICH9, 2009)”).
- The QEMU firmware-configuration (FwCf) PCI device.
- QEMU’s own PCI vendor IDs on emulated controllers.

This establishes the clean, internally coherent foundation upon which the Layer 2 application-level controls are constructed.

8 Implementation: Layer 2

With the virtualization boundary concealed at Layer 1, attention turns to the application layer. The objective shifts: the browser already reports a coherent, driverless physical persona, so the task is no longer to hide a virtual machine but to address the residual entropy of the rendering-readback surfaces (Canvas, WebGL, AudioContext) without, in the process, introducing artifacts more conspicuous than the entropy being masked. This section documents an iterative implementation across five successive revisions of a Firefox (Manifest V3, `world:"MAIN"`) browser extension, denoted v0.1 through v0.3.1.

Each revision was measured against the open-source CreepJS fingerprinting and lie-detection suite, and the progression of results forms the empirical core of this section. The progression also reveals that a nominally “clean” score conceals a tension between competing detection surfaces, a finding developed in Section 8.9.

8.1 The Self-Incrimination Problem and the Measurement Instrument

A privacy extension that modifies JavaScript APIs faces a paradox already articulated in Section 5.1: the act of interception is itself a high-entropy signal. A naive override changes the value a tracker reads, but simultaneously leaves evidence that *some* value was changed. A sophisticated tracker does not merely record fingerprint values; it cross-examines the API surface for internal inconsistency. The measure of a well-constructed interception layer is therefore not whether it alters the fingerprint, but whether the alteration is distinguishable from a genuine, unmodified browser.

CreepJS operationalises this through what it terms “lie” detection: a battery of consistency probes applied across roughly 3,395 properties of the JavaScript environment. For each function and getter it inspects, it poses several questions whose answers a genuine native implementation keeps mutually consistent. Three probe families are most relevant to this work:

- **Source inspection.** The tracker compares `fn.toString()` against `Function.prototype.toString.call(fn)`, and against a pristine `toString` obtained from a freshly created `<iframe>`. On a genuine native function all three agree; any divergence betrays interception.
- **Invocation semantics.** The tracker calls the function with a deliberately incorrect receiver (e.g., `null`), invokes it as a constructor (`new fn()`), and attempts to subclass it (`class X extends fn`). A native DOM method throws specific, well-defined errors under each; an interception layer that behaves differently is exposed.
- **Structural inspection.** The tracker walks the prototype chain and property descriptors, verifying that the intercepting object is not an exotic type (such as a `Proxy`) where a plain function is expected.

The five revisions constitute a systematic campaign to satisfy each probe family in turn. Table 1 summarises the measured outcomes; the subsections that follow explain the mechanism behind each transition.

8.2 Shared Infrastructure: Deterministic Seeding

All five revisions share a common foundation independent of the concealment strategy: a domain-seeded pseudo-random number generator (PRNG). The core design requirement dictates that a given site must observe a stable, consistent fingerprint across reloads and sessions—naive per-request randomization is itself a tracking signal, since fingerprint-evolution tracking techniques are specifically designed to follow a client across small, incidental changes in an otherwise-stable fingerprint [8], and an unstably morphing fingerprint flags the user as utilizing an evasion tool more reliably than it hides them. Conversely, distinct domains must observe entirely uncorrelated fingerprints, definitively preventing cross-site linkage and tracking networks from aggregating browsing profiles.

Version	Primary change	Lies	Props	DOMRect
v0.1	Global toString swap (WeakMap)	2,584	2,507	25%
v0.2	Per-function Proxy wrapper	40	10	25%
v0.2.1	Probe reconciliation traps	0	0	33%
v0.3	Gamma-shift noise, rect hooks removed	0	0	100%
v0.3.1	WebGL/font hooks removed	0	0	100%

Table 1: CreepJS lie-detection results across the five extension revisions. “Lies” and “Props” report the prototype-tampering probe; “DOMRect” reports a separate geometric-consistency score where higher is cleaner.

This stability is achieved by extracting the site’s effective top-level domain plus one label (eTLD+1). Utilizing the eTLD+1 rather than the full hostname prevents correlation discrepancies across subdomains (e.g., ensuring `auth.tracker.com` and `cdn.tracker.com` resolve to the identical seed). The domain string is concatenated with a global SALT variable and digested using the `cyrb128` algorithm, returning a 128-bit state array that seeds the `sfc32` generator.

```

1 function cyrb128(str) {
2   let h1 = 1779033703, h2 = 3144134277, h3 = 1013904242, h4 = 2773480762;
3   for (let i = 0, k; i < str.length; i++) {
4     k = str.charCodeAt(i);
5     h1 = h2 ^ Math.imul(h1 ^ k, 597399067);
6     h2 = h3 ^ Math.imul(h2 ^ k, 2869860233);
7     h3 = h4 ^ Math.imul(h3 ^ k, 951274213);
8     h4 = h1 ^ Math.imul(h4 ^ k, 2716044179);
9   }
10  h1 = Math.imul(h3 ^ (h1 >>> 18), 597399067);
11  h2 = Math.imul(h4 ^ (h2 >>> 22), 2869860233);
12  h3 = Math.imul(h1 ^ (h3 >>> 17), 951274213);
13  h4 = Math.imul(h2 ^ (h4 >>> 19), 2716044179);
14  h1 ^= (h2 ^ h3 ^ h4); h2 ^= h1; h3 ^= h1; h4 ^= h1;
15  return [h1 >>> 0, h2 >>> 0, h3 >>> 0, h4 >>> 0];
16 }
17
18 function sfc32(a, b, c, d) {
19   return function () {
20     a >>>= 0; b >>>= 0; c >>>= 0; d >>>= 0;
21     var t = (a + b) | 0;
22     a = b ^ (b >>> 9);
23     b = (c + (c << 3)) | 0;
24     c = (c << 21) | (c >>> 11);
25     d = (d + 1) | 0;
26     t = (t + d) | 0;
27     c = (c + t) | 0;
28     return (t >>> 0) / 4294967296;
29   };
30 }
31
32 var domainSeed = getETLD1(location.hostname);
33 var seed = cyrb128(domainSeed + ':' + SALT);
34 var rand = sfc32(seed[0], seed[1], seed[2], seed[3]);

```

Listing 9: Deterministic seeding shared across all revisions. Calculates a secure, domain-locked 128-bit hash state.

The static SALT constant permits programmatic, global rotation of the operational fingerprint. An automated forensic environment can rewrite the SALT string within the extension source files prior to launching the browser, thereby generating an entirely disparate hardware persona across all tracking vectors simultaneously, ensuring operational continuity in hostile environments.

8.3 Shared Infrastructure: Injection Idempotency

A second piece of infrastructure is shared independently of the concealment strategy. Because every revision runs at `document_start` in every frame (`all_frames: true`), and because Section 8.8.2 below additionally re-injects the script by hand into tracker-created iframes, each revision must detect and skip a redundant execution in a window it has already initialized—otherwise a function hooked twice would wrap its own proxy, doubling every noise calculation and corrupting the PRNG state. Every revision marks a window as initialized with a guarded property, tested and set before any hook is installed. The guard is a non-registered Symbol, recovered across re-injections of the same script by matching its description via `Object.getOwnPropertySymbols` rather than by re-deriving an identical value:

```
1 var _guard = (function () {
2   var DESC = '__cs_hybrid_guard__';
3   var existing = Object.getOwnPropertySymbols(window)
4     .filter(function (s) { return s.description === DESC; })[0];
5   return existing || Symbol(DESC);
6 })();
7 if (window[_guard]) return;
8 window[_guard] = true;
```

Listing 10: The injection guard shared across all revisions.

Using `Symbol()` rather than `Symbol.for(...)` matters: the latter registers its argument in the JavaScript engine’s global symbol registry, so any code that knows or guesses the string can reconstruct the identical symbol and query `window[Symbol.for('__cs_hybrid__')]` directly, without needing this source at all. Recovering the guard by description instead keeps it idempotent across the extension’s own re-injections while remaining unreconstructible by unrelated code.

8.4 Version 0.1: Global toString Substitution and the Cascade

The initial implementation applied deterministic, domain-seeded noise to the rendering readback surfaces (Canvas, AudioContext, Font metrics). To conceal the interception from tracking scripts, the architecture relied on replacing `Function.prototype.toString` globally. A `WeakMap` structure associated each hooked function with the pristine, native C++ source string it was required to report. The replacement `toString` consulted this internal map before deferring to the genuine implementation.

Furthermore, this early iteration relied extensively upon “Counterfeit Objects” to manipulate geometric measurements. Rather than proxying native return objects seamlessly, the script instantiated generic, plain JavaScript objects (`{}`) and manually forced their prototype chains.

```
1 hookFunction(CanvasRenderingContext2D.prototype, 'measureText',
2   function measureText() {
3     var metrics = _apply.call(_measureText, this, arguments);
4
5     // Counterfeit Flaw: Creates a plain object, completely lacking
6     // the internal C++ slots and descriptors of a genuine interface.
7     var proxy = Object.create(TextMetrics.prototype);
8
9     var propNames = ['width', 'actualBoundingBoxLeft', '...'];
10    propNames.forEach(function (prop) {
11      var val = metrics[prop];
12      if (typeof val === 'number') {
13        Object.defineProperty(proxy, prop, {
14          get: function () { return val + randFloat(-0.001, 0.001); },
15          enumerable: true,
16          configurable: true
17        });
18      }
19    });
20  });
```

```

20     return proxy;
21   }
22 );

```

Listing 11: v0.1 Counterfeit Object generation. The script attempts to forge TextMetrics geometry but creates an easily identifiable structural anomaly.

The approach failed comprehensively against empirical evaluation, producing **2,584 lies across 2,507 of 3,395 properties**. The root vector of the failure resides in the first probe family (Source inspection) and the fundamental insecurity of global prototype mutation.

Function.prototype.toString operates as a single, globally shared object within the page execution context. Advanced trackers bypass page-level modifications by dynamically generating an isolated execution environment, historically known as the “Iframe Escape Hatch.” The tracker appends a hidden <iframe> with an empty or pseudo-protocol source (e.g., javascript:”) to the DOM. Because the extension script has not operated within this newly minted context, the tracker can extract a clean, unpolluted reference to toString from the iframe.contentWindow.

By applying this pristine toString reference against the main window’s intercepted functions, the tracker completely circumvents the WeakMap trap, exposing the source code of the evasion script in plain text. More severely, because the global toString method was conclusively identified as compromised, CreepJS’s heuristic algorithms transitively marked every native API reachable through it as corrupt, producing the observed cascade failure from a single contaminated root.

Finally, this revision registered a canvas “16% rgba noise” detection, confirming the pixel perturbation logic was highly visible to spatial statistical analysis.

8.5 Version 0.2: Per-Function Proxy Isolation

The second revision abandoned global mutation entirely, the WeakMap paradigm. Function.prototype.toString was reverted to its native, unhooked state. To replace it, each target API was individually wrapped in an ECMAScript 6 Proxy object whose behaviour was strictly defined by a localized handler.

Property access on the wrapper was routed through a get trap that detected queries specifically for the toString property, returning a pre-calculated native source string exclusively for that specific function instance.

```

1 function hookFunction(obj, prop, implFn) {
2   var original = obj[prop];
3   if (!original) return original;
4
5   var nativeSource = _call.call(_toString, original);
6   var handler = {
7     apply: function (target, thisArg, args) {
8       return _apply.call(implFn, thisArg, args);
9     },
10    get: function (target, prop, receiver) {
11      if (prop === 'toString') {
12        return function toString() { return nativeSource; };
13      }
14      return Reflect.get(target, prop, receiver);
15    }
16  };
17
18  var proxied = new Proxy(original, handler);
19  obj[prop] = proxied;
20  return original;
21 }

```

Listing 12: The v0.2 hook installer. The global toString is never modified; each function is wrapped in an isolated Proxy membrane.

The architectural transition yielded a substantial improvement: **40 lies across 10 properties**, representing a 98% reduction in detection footprint and the absolute elimination of the heuristic cascade. Because no shared global object was mutated, the iframe-reference comparison that previously defeated v0.1 found zero discrepancies—the page’s and the iframe’s `toString` resolved to the same genuine C++ function. The flagged properties were confined precisely to the ten functions the extension had actively hooked, with zero collateral contamination.

The residual forty lies spread uniformly across the hooked functions. These anomalies originated not from source inspection, which now passed flawlessly, but from the second probe family: Invocation semantics and WebIDL specification compliance.

When a tracker invokes a native WebIDL method with a deliberately incorrect receiver (e.g., executing `HTMLCanvasElement.prototype.toDataURL.call(null)`), the native browser engine immediately halts execution, throwing a strict `TypeError: Illegal invocation`. In the v0.2 implementation, the intercepting JavaScript logic evaluated internal requirements *prior* to invoking the native C++ method⁶. For instance, the script attempted to call `this.getContext('2d')` to fetch image data for manipulation. Because the tracking script had deliberately poisoned the `this` receiver by passing `null`, the JavaScript runtime generated a generic `TypeError: null has no properties`. The discrepancy between the expected native C++ exception and the generic JavaScript exception conclusively unmasked the proxy.

8.6 Version 0.2.1: Probe Reconciliation and Native Execution

The third revision systematically breaks down the residual forty lies by extending the Proxy handler to reconcile invocation-semantics and prototype-walking probes.

8.6.1 Constructor Delegation (`Reflect.construct`)

In previous versions, instantiating hooked APIs using the `new` keyword failed. The handler was expanded to include a `construct` trap, utilizing `Reflect.construct` to forward the instantiation transparently to the underlying native engine. This ensured that `new fn()` produced the exact same instantiation behaviour and error signatures as the genuine method, neutralizing the “new instance” and “class extends” tracking probes.

```
1 construct: function (target, args, newTarget) {  
2   return Reflect.construct(target, args, newTarget);  
3 }
```

Listing 13: v0.2.1 Constructor Delegation. `newTarget` is forwarded unmodified so subclassing (`class X extends fn`) sees the identical native construction behaviour.

8.6.2 WebIDL Compliance via Native-First Execution

To resolve the WebIDL illegal invocation discrepancies, the execution ordering within the proxies was architecturally inverted. Rather than performing JavaScript evaluations prior to invoking the C++ engine, v0.2.1 instituted a “Native-First” execution paradigm utilizing `try/catch` exception forwarding.

```
1 hookFunction(HTMLCanvasElement.prototype, 'toDataURL', function toDataURL() {  
2   var nativeResult;  
3   // NATIVE FIRST: Aggressively invoke the C++ engine immediately.  
4   // If the tracker passed 'null' or bad arguments, the C++ engine throws  
5   // the mathematically perfect, genuine WebIDL error here, bypassing our JS.  
6   try {  
7     nativeResult = _apply.call(_toDataURL, this, arguments);  
8   } catch (e) {
```

⁶Native WebIDL operations brand-check this in the C++ binding layer before any JS-visible behaviour runs; whichever side runs first determines whether that check fires.

```

9     throw e; // Forward the genuine Illegal Invocation exception
10 }
11
12 // SAFE EXECUTION: If execution reaches here, the invocation is valid.
13 var w = this.width, h = this.height;
14 var ctx = this.getContext('2d');
15
16 // ... [Canvas image extraction and noise application logic] ...
17 return _apply.call(_toDataURL, tempCanvas, arguments);
18 });

```

Listing 14: v0.2.1 Native-First Execution. C++ validations are processed prior to JavaScript logic, forwarding genuine WebIDL exceptions intact.

8.6.3 Prototype-Walk Defence (setPrototypeOf)

Advanced fingerprinting libraries detect proxy membranes by traversing the prototype chain of a suspected function via `Object.getPrototypeOf()`. Natively, this traversal terminates elegantly at `null`. Proxies, however, leave distinct structural traces during deep recursion. The handler's `setPrototypeOf` trap was constructed to detect tracking loops and artificially force the JavaScript engine to throw a genuine, native cyclic error.

```

1 setPrototypeOf: function (target, proto) {
2   var current = proto;
3   while (current !== null) {
4     if (current === proxy) {
5       // Create a dummy object and force a cyclic prototype assignment.
6       // This causes the V8/SpiderMonkey engine to throw its native error.
7       var dummy = {};
8       Object.setPrototypeOf(dummy, dummy);
9     }
10    try { current = Object.getPrototypeOf(current); } catch (e) { break; }
11  }
12  return Reflect.setPrototypeOf(target, proto);
13 }

```

Listing 15: v0.2.1 Cyclic Prototype Emulation. Forces a native exception during deep structural introspection.

8.6.4 Pre-computed PRNG State Locking

Dynamic noise execution inside getter traps introduced severe PRNG state desynchronization. If a tracking script queried `rect.width` sequentially, the randomizer advanced continuously, generating contradicting geometric widths across the exact same `DOMRect` instance (the “flickering getter” anomaly). v0.2.1 solved this by pre-computing static noise offsets upon initial script execution, locking the global variables to pass strict equality checks across subsequent geometric reads.

The same native-first ordering was applied throughout v0.2.1's hook set, including the WebGL `getParameter` spoof inherited from v0.2: the native function is invoked unconditionally, in `try/catch`, before any spoofed value is returned, so a poisoned receiver still surfaces the genuine `Illegal invocation` error rather than silently returning a value.

```

1 hookFunction(proto, 'getParameter', function getParameter(pname) {
2   var nativeResult;
3   try { nativeResult = _apply.call(_getParameter, this, arguments); }
4   catch (e) { throw e; }
5
6   if (pname === 0x0D33) return 16384; // MAX_TEXTURE_SIZE
7   if (pname === 0x84E8) return 16384; // MAX_RENDERBUFFER_SIZE
8   return nativeResult;

```

```
9 });
```

Listing 16: v0.2.1 `getParameter` spoof. The native call happens unconditionally, so an illegal receiver throws before any spoofed constant is reached.

Empirical Result for v0.2.1: The implementation of the cyclic membrane, native-first exception forwarding, and state-locking resulted in **0 prototype lies** across all 3,395 properties.

8.7 The DOMMatrix Failure and Version 0.3 Paradigm Shift

However the separate `DOMRect` geometric-consistency score failed to mirror the absolute stealth of the prototype score. In empirical testing, it only nominally *improved*, rising from 25% to 33% (where a score of 100% indicates mathematically clean physics), remaining fundamentally compromised.

Satisfying the prototype-tampering probes on the layout functions did not resolve the underlying geometric anomalies; instead, the independent affine transformation matrix algorithms cross-checking the geometric relationships continued to expose the discrepancies.

The mechanism of this failure relies on the CSS Object Model (CSSOM). A tracker queries a standard 100-pixel element. The v0.2.1 proxy intercepts `getBoundingClientRect` and applies +0.001 static noise, reporting 100.001px. The tracker subsequently applies a CSS affine transformation: `transform: scale(2.5)`. The native C++ browser engine calculates the new dimension internally as exactly 250 pixels. The JS proxy intercepts this new native output, reapplies its static noise, and returns 250.001px.

The tracking algorithm reverses the operation: $250.001 \div 2.5 = 100.0004$. The initial base dimension (100.001) and the derived reverse-matrix dimension (100.0004) mathematically contradict one another, instantly exposing the proxy injection.

8.7.1 Addition by Subtraction and Gamma Shift Evasion

Recognizing that spoofing the `DOMRect` geometry flawlessly in JavaScript requires hooking the entire CSS Object Model (a computational impossibility), the v0.3 architecture adopted a philosophy of “Addition by Subtraction.”

All ‘`DOMRect`’ and layout noise hooks were permanently deleted from the codebase. Layout physics standardization was delegated exclusively to the Layer 1 Virtual Machine, relying on the hypervisor’s underlying font rendering engine to supply uniform, unmodified measurements. This singular deletion returned the CreepJS `DOMRect` trust score to **100%**.

Concurrently, the v0.3 architecture addressed canvas analysis flags. Previous versions manipulated 10 random pixels by exactly ± 1 RGB value. Enterprise models (like Fingerprint Pro) employ spatial-frequency analysis. Real physical GPU variations manifest as global rendering algorithm shifts or sub-pixel edge blurring, not isolated, mathematically perfect “random dots” on solid colour blocks. Consequently, the random dot model was routinely flagged as a **16% artificial rgba noise** anomaly.

v0.3 introduced a Gamma Shift. By computing a microscopic, domain-seeded multiplier (e.g., 1.002), the script scaled every non-transparent pixel in the canvas buffer.

```
1 var gammaShift = 1 + ((rand() * 0.01) - 0.005);
2
3 function applyOrganicCanvasNoise(imageData) {
4   var data = imageData.data;
5   var len = data.length;
6   for (var i = 0; i < len; i += 4) {
7     if (data[i+3] === 0) continue; // Skip transparency
8     // Applies uniform mathematical scaling across the spectrum
9     data[i] = data[i] * gammaShift;
10    data[i+1] = data[i+1] * gammaShift;
11    data[i+2] = data[i+2] * gammaShift;
12  }
13  return imageData;
14 }
```

Listing 17: v0.3 Gamma Shift Canvas Evasion. Scaled pixels organically mimic monitor colour-profile calibration.

8.8 Version 0.3.1: Final Surface Minimisation and Active Iframe Synchronization

The ultimate version of the framework, v0.3.1, extended the “Addition by Subtraction” principle to WebGL and Font enumeration, and fundamentally resolved the Iframe Escape Hatch vulnerability.

8.8.1 Removal of WebGL and Font Enumeration Hooks

In earlier paradigms, scripts attempted to normalize WebGL arrays (e.g., forcing `MAX_TEXTURE_SIZE` to 16384). However, a WebGL context exposes over 75 interconnected constants governing varying vectors, uniform blocks, and shader precision. Spoofing three variables in JavaScript while allowing the remaining 72 to pass through the native VM driver resulted in an impossible architectural state—a “Frankenstein” hardware profile immediately flagged by heuristics. Similarly, injecting random floating-point noise into `measureText` contradicted the strict, deterministic sub-pixel fractions calculated by genuine font engines.

v0.3.1 unequivocally excised both WebGL and Font Enumeration hooks. By delegating these parameters to the Layer 1 WARP software rasteriser, the evasion layer guarantees the hardware configuration remains perfectly cohesive.

8.8.2 Synchronous Iframe Self-Replication

To natively defeat the iframe escape hatch, where trackers extract pristine API references from hidden DOM elements, v0.3.1 deployed active, synchronous DOM replication. By hooking `Node.prototype.appendChild`, the script intercepts the physical insertion of tracker-generated frames. The script executes an `eval()` payload inside the child `contentWindow` prior to yielding execution control back to the tracker.

To address First-Party Isolation, the parent window passes an `__INJECTED_SEED__` variable to the child. In dynamic tracking environments, scripts may load cross-origin iframes. This active synchronization forces every hostile tracking iframe to inherit the exact PRNG state of the parent origin, guaranteeing the identical hardware persona is generated across security boundaries and neutralizing cross-site linkage without triggering prototype lies. The same replication logic hooks `Node.prototype.insertBefore` alongside `appendChild`, since trackers insert frames through either primitive, and tracks already-replicated windows in a closure-local `WeakSet` so a frame moved or scanned twice is never re-initialized.

```
1 var scriptString = 'var __INJECTED_SEED__ = ' + domainSeed + '"; (' +  
    canvasShieldMain.toString() + ')();';  
2 var _injectedWindows = new WeakSet();  
3  
4 function injectFrame(win) {  
5     if (!win || _injectedWindows.has(win)) return;  
6     _injectedWindows.add(win);  
7     try { win.eval(scriptString); } catch (e) { }  
8 }  
9  
10 function scanAndInject(node) {  
11     if (!node) return;  
12     if (node.tagName === 'IFRAME' && node.contentWindow) injectFrame(node.  
        contentWindow);  
13 }  
14  
15 var _appendChild = Node.prototype.appendChild;  
16 hookFunction(Node.prototype, 'appendChild', function appendChild(node) {  
17     var res = _apply.call(_appendChild, this, arguments);  
18     scanAndInject(node); // Synchronously replicate prior to tracker execution
```

```

19     return res;
20 });
21
22 var _insertBefore = Node.prototype.insertBefore;
23 hookFunction(Node.prototype, 'insertBefore', function insertBefore(node, child)
24     {
25         var res = _apply.call(_insertBefore, this, arguments);
26         scanAndInject(node);
27         return res;
28     });

```

Listing 18: v0.3.1 Synchronous Iframe Self-Replication with PRNG Inheritance.

8.8.3 WebRTC IP Leak Protection

v0.3.1 additionally hardens a surface none of the rendering-readback hooks touch: WebRTC ICE candidate gathering can reveal the host’s real local-network or public IP address through a channel entirely outside the HTTP/DNS path Layer 1 controls. This is a network-layer fact no SMBIOS or GPU-driver spoof touches, so by the same “no native substitute” criterion Table 2 applies to Canvas and Audio below, it belongs at Layer 2.

The mitigation avoids wrapping or replacing a tracker’s own `onicecandidate` handler, since doing so would break `pc.onicecandidate === handler` reference equality. Instead, a listener is registered on each `RTCPeerConnection` at construction time, before the caller can attach one of their own; because event-handler IDL attributes and `addEventListener` callbacks share a single listener list invoked in registration order, this listener runs first and calls `stopImmediatePropagation()` on any non-relay candidate, silencing every listener registered afterwards without touching the caller’s function.

```

1 function guardIceEvents(pc) {
2     _call.call(_addEventListener, pc, 'icecandidate', function (event) {
3         var c = event && event.candidate;
4         if (c && c.candidate && !/\btyp relay\b/.test(c.candidate)) {
5             event.stopImmediatePropagation();
6         }
7     });
8 }
9
10 var _RTCPeerConnection = RTCPeerConnection;
11 window.RTCPeerConnection = new Proxy(_RTCPeerConnection, {
12     construct: function (target, args, newTarget) {
13         var pc = Reflect.construct(target, args, newTarget);
14         guardIceEvents(pc);
15         return pc;
16     }
17 });

```

Listing 19: v0.3.1 WebRTC IP leak protection. The guard listener is attached before the constructor returns, so it always precedes any listener the caller registers.

The constructor itself is replaced with a `Proxy` rather than the individual prototype methods hooked elsewhere in this section, since the property being intercepted is the global binding, not a single method call; the absence of a `get` trap relies on the same `Function.prototype.toString` forwarding behaviour for `Proxy` exotic objects that lets every other hook in this section omit an explicit `toString` trap.

8.9 The Surface-Versus-Surface Tradeoff

The zero-lie state was achieved not by making the browser *more* like a generic real machine, but by making the interception layer’s machinery invisible to a specific suite’s probes. The prototype-lie count

is a measure of one detector’s tamper-detection battery, not a measure of absolute, undeniable indistinguishability from silicon.

Second, the evasion surfaces did not progress linearly. A perfect score was obtained only at v0.3 by *removing* the layout interception entirely. In other words, on the one surface where JS interception was pursued to its logical mathematical end, the correct architectural move proved to be absolute abandonment. This indicates that detection surfaces impose partially opposed constraints, and that neutralizing one heuristic (e.g., prototype stealth) does not automatically resolve underlying physics contradictions (e.g., geometric matrices).

The synthesis reveals that in-context interception possesses no absolute zero. The tracking network holds an insurmountable structural advantage: it need only locate a single contradiction among thousands of probed behaviours across an open-ended set of WebIDL interfaces. The interception script must reproduce native C++ semantics perfectly across all of them, against every future probe. The trajectory of this research points toward Application-Layer minimalism and native Hardware/OS-level delivery as the only sustainable evasion strategy.

8.10 Summary

Table 2 consolidates the disposition of every interception surface across the final design. Every entropy surface that a lower layer (the VM hypervisor) could supply natively was ultimately best served by removing the JavaScript interception. Only the surfaces possessing no native substitute—such as explicit canvas serialization, audio buffer processing, and WebRTC candidate gathering—retain a hook in the final v0.3.1 revision.

Surface	Final disposition	Reason
Canvas 2D readback	Hooked (gamma noise)	No native substitute
OffscreenCanvas	Hooked (gamma noise)	No native substitute; worker reach
Audio rendering	Hooked (mult. scale)	No native substitute
WebRTC ICE candidates	Hooked (relay-only filter)	No native substitute; network-layer fact
WebGL parameters	Removed	WARP reports coherent native values
WebGL readback	Removed	Redundant with Canvas path
Layout / DOMRect	Removed	Degraded geometric score (Matrix Lie)
Font / measureText	Removed	Population bucketing natively robust
navigator fields	Removed	Delivered by Layer 1 vCPU topology
screen fields	Removed	Delivered by Layer 1 display config

Table 2: Final disposition of each interception surface. Only surfaces without a native lower-layer substitute retain a hook in the finalized architecture.

9 Test Results and Empirical Validation

Table 1 tracked the extension’s lie-detection score across its five architectural revisions during development. This section validates a different property of the deployed system: that its identity is stable to a script probing the *same* origin, decorrelated across *different* origins, and can be deliberately rotated end-to-end on demand, as the seeding design of Section 8.2 claims. It also cross-checks the extension against a commercial fraud/bot detector operating independently of CreepJS’s lie-detection methodology.

9.1 Cross-Session Identity Rotation Under a Salt and Network Change

To exercise the rotation property directly, CreepJS was run twice against the same site: once under the extension’s default SALT over a direct network connection, and once after reloading the extension with

a changed SALT value and routing the outbound connection through a VPN, changing the visible IP address. Figure 5 shows the resulting fingerprint identifiers from both runs.

```
FP ID: 83c3c0512372ae4d4546db4f66bb9e087b415e4dcaee4a577db12687c88b75a9
Fuzzy: 1c4a158a6be1d6ac316948e34825fd0c5ff594e1bf1aaff895b6000000000000
```

(a) Before: default SALT, direct network connection.

```
FP ID: 012248461d6fb7a856066e571b09878d827e4adc3cc03f9de28c0be4adb8bc3b
Fuzzy: f49a158a6be1d6ac316938e34825f4092ff594e1ff1aaff895b6000000000000
```

(b) After: extension reloaded with a rotated SALT, outbound traffic routed through a VPN.

Figure 5: CreepJS fingerprint identifiers before and after a deliberate identity rotation.

The two identifiers CreepJS reports behave exactly as the architecture predicts, but for different reasons:

- **FP ID** (the exact fingerprint hash) changes completely: comparing the two 64-character hex strings position-by-position, only 7 of 64 characters coincide—within the $\sim 6\%$ (1-in-16) rate expected of two unrelated random hex strings by chance alone. The tracking identifier a site would key a profile on is fully unlinkable across the rotation.
- **Fuzzy** (CreepJS’s similarity hash over browser/hardware-class properties, tolerant of small legitimate variation) remains largely unchanged: 56 of 64 characters coincide (87.5% positional agreement; 89.1% under sequence-alignment matching), and the shared substrings—158a6be1d6ac316, 4825f, 594e1, f1aaff895b6—span most of the string. The browser continues to present as the *same class* of genuine, non-automated machine before and after rotation; only the seeded, domain-specific noise layered on top of that class changed.

This is the concrete behavior the SALT-rotation design of Section 8.2 is intended to produce: a burned identifier can be discarded without the underlying persona ever looking like a different, newly-suspicious machine.

9.2 Commercial Detector Cross-Check: Fingerprint Pro

CreepJS’s lie-detection methodology is specific to internal API consistency (Section 8.1); it is not a fraud- or bot-scoring engine. As an independent check against a commercial detector built for that purpose, the same browser and VPN configuration was evaluated against fingerprint.com’s live visitor-identification demo (Figure 6). Fingerprint Pro is a deliberate choice of target here beyond convenience: a recent grey-box study of the service found that its client-side fingerprint is only one input to a server-side identifier that also incorporates cookies and IP address, and that this combination is considerably more robust to purely client-side countermeasures than the attribute-uniqueness literature alone would suggest [25]. A clean result against Fingerprint Pro is therefore a stronger claim than a clean result against a client-side-only measurement tool like CreepJS, precisely because the server-side state Luo et al. describe is outside either layer’s reach.

Two results are notable together. First, Fingerprint Pro correctly flags **VPN: Detected**—this work does not attempt to hide VPN usage itself, only the browser/hardware-level tampering signals a VPN alone does not touch, so this is an accurate, expected result rather than a failure. Second, and more importantly, the overall **Suspect Score is 4** (low) with the verdict “You look like a trusted user: No signs of fraud, bots, or spoofing”—meaning the Canvas/Audio/WebGL/WebRTC surface this paper’s Layer 2 controls did not itself trip Fingerprint Pro’s fraud/tampering heuristics, even though the tool is fully aware the session is running through a VPN and evidently does not treat that alone as disqualifying.

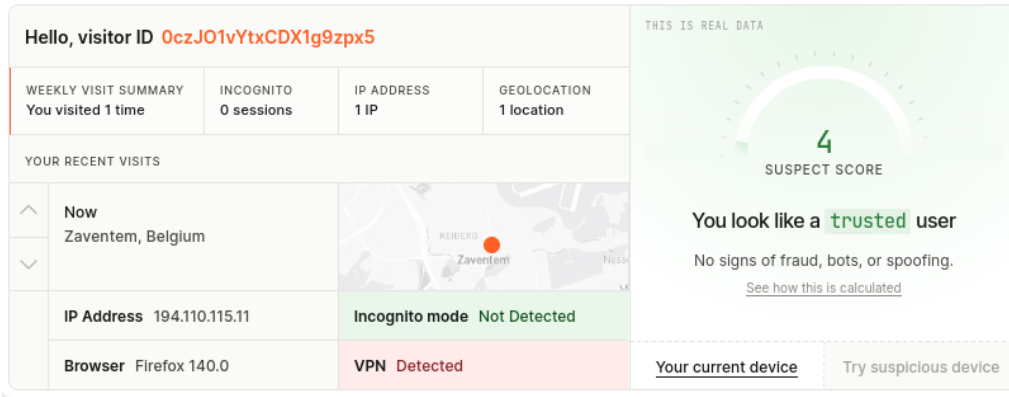


Figure 6: Fingerprint Pro visitor report for the same browser/VPN configuration. The VPN is correctly detected and disclosed rather than hidden; despite this, the session is scored as trusted with no fraud, bot, or spoofing signal raised.

9.3 Additional Validation Tools

The extension and hardened VM were also exercised against AmIUnique, EFF’s Cover Your Tracks, and BrowserLeaks.com as part of the same validation pass; Section 7 (Figure 1) already presents the BrowserLeaks WebGL check for Layer 1 specifically. AmIUnique and Cover Your Tracks were used qualitatively, to confirm no single reported property (canvas hash, font list, plugin list, header ordering) stood out as an outlier against each tool’s respective reference population, rather than to produce a numeric score comparable across the different measurement methodologies discussed in this section. This qualitative posture is consistent with recent comparative measurements of anonymity techniques: an exploratory study of incognito mode, proxies, VPNs, and Tor against fingerprinting scripts found that VPN and proxy use alone left the browser fingerprint itself untouched and still detected, with Tor the only technique to substantially suppress fingerprint recording [26]—reinforcing why this paper pairs a VPN with an actively-hardened browser and VM rather than relying on the VPN’s IP substitution alone.

10 Conclusion and Absolute Limitations

This paper set out to synchronize two camps that Section 4 treats as largely separate in existing work: browser-level countermeasures (anonymity-set fixation, noise injection) and system-level countermeasures (VM/hardware personas). The resulting architecture—a hardened KVM/QEMU hypervisor presenting an internally coherent consumer-laptop persona (Layer 1, Section 7), paired with a browser extension that hooks only the surfaces Layer 1 cannot itself deliver natively (Layer 2, Section 8)—establishes a formidable “Grey Noise” evasion environment without incurring the structural fingerprint common to conventional, single-layer privacy tools.

The empirical record across this paper supports that claim from three independent angles. Table 1 tracks the architectural progression of Layer 2 against CreepJS, from 2,584 lies across 2,507 of 3,395 properties at v0.1 to zero at v0.3.1. Section 7’s BrowserLeaks check confirms Layer 1’s WebGL surface resolves identically under both plain and *unmasked* renderer queries—the specific channel trackers use to defeat naive spoofing. Section 9 then validates the deployed system as a whole rather than as an architectural exercise: a deliberate SALT-and-VPN identity rotation changed CreepJS’s exact fingerprint hash completely (7/64 characters coincident, chance-level) while its structural similarity hash remained largely intact (56/64, 87.5%), and an independent commercial detector, Fingerprint Pro, scored the same configuration as a trusted user with no fraud, bot, or spoofing signal raised despite correctly detecting the VPN.

10.1 The Central Finding: Delegation, Not Deeper Interception

The most consequential result of this work is not any single lie count but the trajectory in Section 8.9: the zero-lie state at v0.3.1 was reached by *removing* JavaScript interception from layout, WebGL, and font enumeration, not by refining it further. Table 2’s disposition column makes the pattern explicit—every surface Layer 1 could supply a coherent native value for was best served by deleting the hook entirely; only Canvas, Audio, and WebRTC, which have no native substitute, retain one. Zero lies against CreepJS is consequently a statement about invisibility to *that suite’s* tamper-detection battery, not an absolute, undeniable indistinguishability from silicon: a tracking network need only locate a single contradiction among thousands of probed behaviours across an open-ended and continually growing set of WebIDL interfaces, while the interception layer must reproduce native C++ semantics perfectly across all of them, against every future probe. Architecturally, this favors minimalism and native hardware/OS-level delivery over ever more sophisticated JavaScript spoofing, and it is the reason this paper treats Layer 1 as foundational rather than as an optional hardening pass alongside Layer 2.

10.2 Absolute Limitations

Several limitations remain, of different characters:

- **Execution timing.** Native C++ execution of operations such as `getImageData` resolves in approximately 0.2 milliseconds; the proxy membrane, regardless of algorithmic optimization, inherently introduces overhead. Undetected by standard behavioral trackers, an enterprise-grade detector deploying precise `performance.now()` or `SharedArrayBuffer`-based timing attacks may theoretically discern proxy presence purely through latency profiling. This concern is not hypothetical: browser-executable JavaScript microbenchmarks have already been shown to infer specific CPU microarchitectural properties, including vendor and model, purely from timing side channels, unaffected by existing fingerprinting mitigations [27].
- **Residual OS-level artifacts.** Section 7 enumerates firmware- and machine-type strings (OVMF’s BOCHS ACPI vendor string, the QEMU machine-type description, the `FwCfg` PCI device, QEMU’s own PCI vendor IDs) that remain detectable to OS-level introspection though not to any web API, plus the SPICE/virtio-serial channel kept as a deliberate, scoped risk acceptance for clipboard integration.
- **Uncovered application-layer surfaces.** `Intl.DateTimeFormat().resolvedOptions()` locale/timezone fingerprinting and CSS/paint-timing fingerprinting are identified but left unaddressed (Section 8.10).
- **Structural asymmetry.** Independent of any specific gap, the tracker’s advantage is combinatorial and the defender’s is not: closing today’s known probes does not bound tomorrow’s, and this asymmetry is not something either layer of this architecture—or any application-layer countermeasure surveyed in Section 4—can resolve outright.

10.3 Identity Rotation

Within these constraints, the design supports a programmatically sound form of identity rotation, demonstrated empirically in Section 9. Because the noise applied to the canvas and audio surfaces is derived from a single global seed (SALT), changing that constant regenerates the seeded portion of the fingerprint deterministically, allowing a burned noise profile to be replaced with a distinct one without reconfiguring the underlying system. This rotation applies only to the surfaces the extension actively perturbs; the structural attributes delivered natively by Layer 1—the renderer string, processor topology, and display geometry—are unaffected by the seed and remain constant across rotations, which is itself the point: a genuine machine does not change its hardware identity between browsing sessions, and neither should a well-constructed persona.

References

- [1] J. R. Mayer and J. C. Mitchell, “Third-Party Web Tracking: Policy and Technology,” in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2012.
- [2] S. Englehardt and A. Narayanan, “Online Tracking: A 1-million-site Measurement and Analysis,” in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.
- [3] P. Eckersley, “How Unique Is Your Web Browser?” in *Proc. Privacy Enhancing Technologies Symposium (PETS)*, 2010.
- [4] K. Mowery and H. Shacham, “Pixel Perfect: Fingerprinting Canvas in HTML5,” in *Proc. Web 2.0 Security and Privacy Workshop (W2SP)*, 2012.
- [5] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, “The Web Never Forgets: Persistent Tracking Mechanisms in the Wild,” in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.
- [6] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, “Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting,” in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2013.
- [7] P. Laperdrix, W. Rudametkin, and B. Baudry, “Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints,” in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2016.
- [8] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, “FP-STALKER: Tracking Browser Fingerprint Evolutions,” in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2018.
- [9] N. Nikiforakis, W. Joosen, and B. Livshits, “PriVaricator: Deceiving Fingerprinters with Little White Lies,” in *Proc. International Conference on World Wide Web (WWW)*, 2015.
- [10] C. F. Torres, H. Jonker, and S. Mauw, “FP-Block: Usable Web Privacy by Controlling Browser Fingerprinting,” in *Proc. European Symposium on Research in Computer Security (ESORICS)*, 2015.
- [11] P. Baumann, S. Katzenbeisser, M. Stopczynski, and E. Tews, “Disguised Chromium Browser: Robust Browser Fingerprinting Protection,” in *Proc. ACM Workshop on Privacy in the Electronic Society (WPES)*, 2016.
- [12] D. Fifield and S. Egelman, “Fingerprinting Web Users Through Font Metrics,” in *Proc. Financial Cryptography and Data Security (FC)*, 2015.
- [13] Y. Cao, S. Li, and E. Wijmans, “(Cross-)Browser Fingerprinting via OS and Hardware Level Features,” in *Proc. Network and Distributed System Security Symposium (NDSS)*, 2017.
- [14] J. B. Althouse, J. Atkinson, and J. Atkins, “JA3 – A Method for Profiling SSL/TLS Clients,” Salesforce Engineering Blog, 2017.
- [15] M. Perry, E. Clark, and S. Murdoch, “The Design and Implementation of the Tor Browser,” The Tor Project, Technical Report.
- [16] Mozilla, “`privacy.resistFingerprinting`,” Firefox Privacy Documentation / Tor Uplift Project.
- [17] A. Juliot, “CreepJS,” <https://github.com/abrahamjuliot/creepjs>.

- [18] arkenfox contributors, “arkenfox/user.js: A Firefox Configuration Hardening Guide,” <https://github.com/arkenfox/user.js>.
- [19] O. N. R. Haalstra, “Browser Anti-Fingerprinting as a Pathway to Impersonation and Fraud,” M.S. thesis, University of Twente, 2020.
- [20] A. Faiz Khademi, “Browser Fingerprinting: Analysis, Detection, and Prevention at Runtime,” M.S. thesis, Queen’s University, 2014.
- [21] A. Datta, J. Lu, and M. C. Tschantz, “Evaluating Anti-Fingerprinting Privacy Enhancing Technologies,” in *Proc. The Web Conference (WWW)*, 2019.
- [22] U. Iqbal, S. Englehardt, and Z. Shafiq, “Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors,” in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2021.
- [23] S. Boussaha, L. Hock, M. Bermejo, R. Cuevas Rumin, A. Cuevas Rumin, D. Klein, M. Johns, L. Compagna, D. Antonioli, and T. Barber, “FP-Tracer: Fine-grained Browser Fingerprinting Detection via Taint-tracking and Entropy-based Thresholds,” *Proceedings on Privacy Enhancing Technologies*, vol. 2024, no. 3, pp. 540–560, 2024.
- [24] G. Jarad and K. Bıçakcı, “When Handshakes Tell the Truth: Detecting Web Bad Bots via TLS Fingerprints,” arXiv:2602.09606, 2026.
- [25] E. Luo, T. Ritter, S. Savage, and G. M. Voelker, “Understanding Server-side Commercial Fingerprinting,” in *Proc. ACM Web Conference (WWW)*, 2026.
- [26] B. S. Husodo, J. Sidabutar, S. Cahyono, A. Amiruddin, and S. S. Zen, “Exploratory Analysis of Browser Fingerprint in Online Anonymity Internet Browser, Proxy, VPN, and TOR,” in *Proc. IEEE International Conference on Cryptography, Informatics, and Cybersecurity (ICoCICs)*, 2025.
- [27] L. Trampert, C. Rossow, and M. Schwarz, “Browser-based CPU Fingerprinting,” in *Proc. European Symposium on Research in Computer Security (ESORICS)*, 2022.