SoK: Taxonomizing the Low-Level Attack Surface of Modern Web Browsers

Han Zheng
EPFL

Qinying Wang
EPFL

Qiang Liu
EPFL

Mathias Payer
EPFL

Abstract*—The web browser remains one of the most exposed remote attack surfaces on end-user systems, and memory-corruption flaws continue to play a central role in real-world browser exploitation. Despite a decade of intensive browser testing and bug-disclosure efforts, the community still lacks an explicit, defense-oriented systematization of the browser's low-level attack surface. Prior SoKs have surveyed browser vulnerabilities and mitigation techniques. However, these perspectives remain fragmented, leaving open a central question: *how is the low-level attack surface of modern web browsers structured, and which parts of this surface remain underexplored by existing security testing?

We approach this primary question through three sub-questions. (RQ1) How is the browser's attack surface structured along input classes and components? (RQ2) Where do memory corruption vulnerabilities arise within this taxonomy? (RQ3) What do these attack-surface patterns imply for existing browser security testing? To answer RQ1, we derive an architecture-grounded Input × Component × Privilege taxonomy that abstracts the architectures of Chrome, Firefox, and Safari into a unified view. To answer RQ2, we map 2,233 memory corruption reports disclosed between 2016 and 2025 onto this taxonomy. To answer RQ3, we overlay a decade of academic browser fuzzers, classified by the targeted input class, onto the bug-density map. Our systematization reveals that current testing concentrates on well-explored components while bug-dense, high-impact surfaces remain insufficiently tested. Moreover, we identify three fuzzer deployment gaps, which are orthogonal to the academic efforts. Our work offers a structured foundation for future browser security research, so that researchers can use this systematization as (i) a map of which inputs reach which components at which privilege, (ii) a measurement of where a decade of bugs and testing efforts concentrate, and (iii) a prioritization guide for where testing is insufficient.

1. Introduction

The web browser is one of the most high-value remote attack target for end-user devices. Memory corruption in the browser codebase is the dominant exploit class, contributing 67% of in-the-wild exploit chains observed against consumer devices [1]–[3]. The dominance of memory corruption motivates the focus on the browser's *low-level attack surface*: the native (C/C++) code running in the browser processes that parses attacker-controlled bytes (such as HTML, JavaScript, images, IPC messages, and UI events). Each path from an attacker-controlled input into the code that parses it forms one *surface*, and the attack surface is the union of these paths (Section 2). To test this surface, browser vendors and academia have responded with a decade of continuous fuzzing of both the browser and the third-party libraries it depends on [4], [5], disclosing tens of thousands of memory-safety bugs [6]–[8]. However, these efforts target individual components in isolation, and the resulting knowledge stays scattered across bug trackers, fuzzer harnesses, and design documents. The community therefore lacks (i) a systematic view of how the low-level attack surface is structured, and (ii) a defense-oriented view of how far existing testing efforts reach into this surface and where future testing should focus. This motivates our overarching research question: *how is the low-level attack surface of modern web browsers structured, and which parts of this surface remain underexplored by existing security testing?*

Prior SoK study browser security through different lenses. Lim et al. [9] provide a broad view of browser bugs, exploitation techniques, and mitigations. These perspectives organize browser security around bug primitives and mitigations. However, their systematization lacks a unified view of the low-level attack surface, *e.g.,* what are the browser inputs, which browser components process these inputs, and what are their privilege. Without this view, it is difficult to assess where memory-corruption risks concentrate, which surfaces are exercised by current testing techniques, and where future testing and hardening should be prioritized. To close this gap, we split our overarching research question into three sub-questions.

RQ1: How is the browser's attack surface structured along input classes and components? We systematize the browser's low-level attack surface along an Input × Component × Privilege taxonomy. Modern browsers follow the principle of least privilege [10], isolating the renderer, GPU, network, and browser code into separate processes at different OS privileges. The taxonomy mirrors this structure: it studies each component, its attacker-controlled inputs, and its privilege tier in isolation across Chrome, Firefox, and Safari, rather than treating the browser as one monolith. We derive it from the public design documents of all three browsers [10]–[13]. The resulting taxonomy distills the three browsers into five attacker-controlled input classes (binary blobs, documents, scripts, UI gestures, and IPC calls), processed by nine components across four privilege tiers.

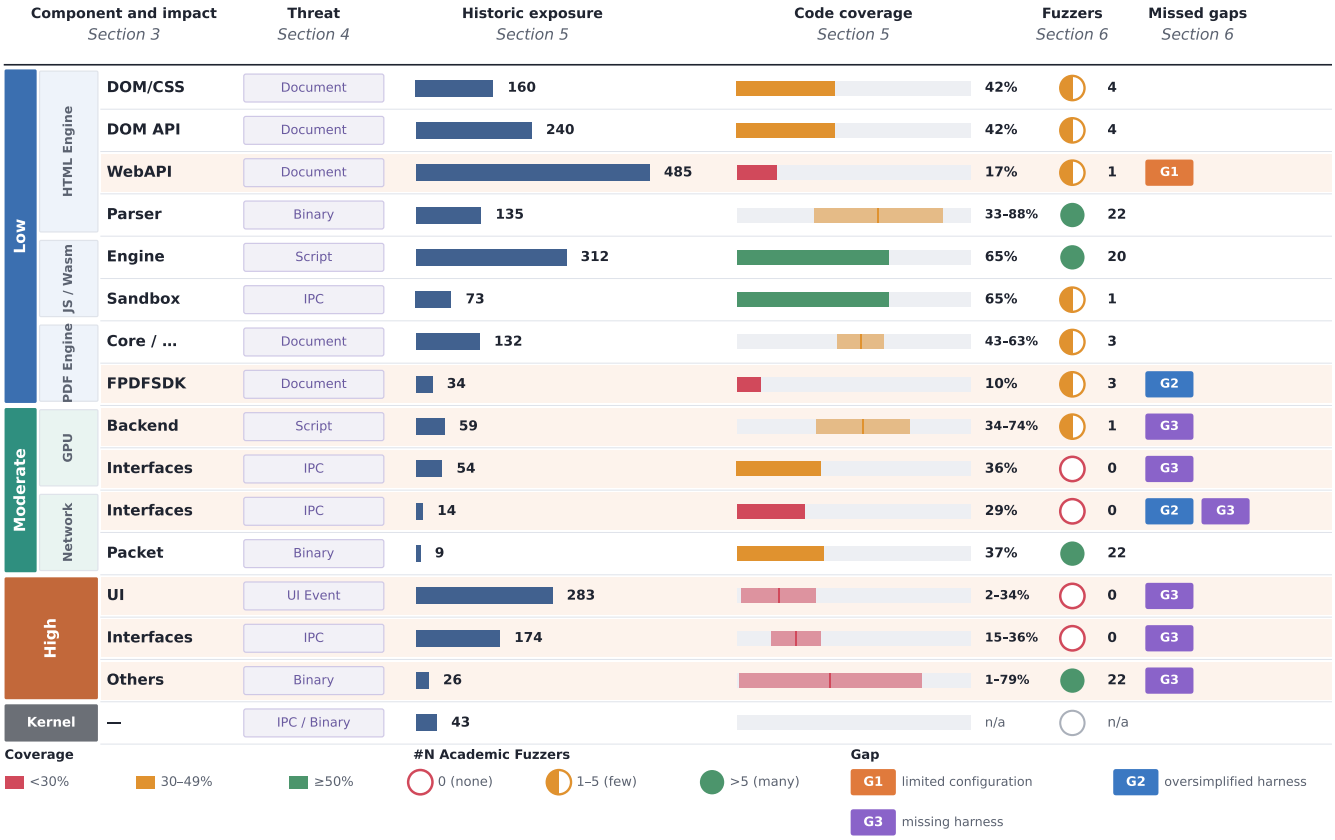

Figure 1. Systematization of the browser attack surface.

RQ2: Where do memory corruption vulnerabilities arise within this taxonomy? We answer RQ2 by collecting 2,233 memory corruption reports disclosed between 2016 and 2025 from the Chromium issue tracker and Firefox security advisories [14], [15], and classifying each against the Input × Component × Privilege matrix. We also pair every component with the line coverage that vendor-deployed fuzzers achieve [4], [16]. Five bug-dense surfaces drive the divergence between vendor testing investment and bug count: the WebAPI bindings, the PDFium SDK boundary, the WebGL backend, the UI input path, and the IPC receivers. Together they hold a large share of high-privilege bugs yet remain insufficiently tested.

RQ3: What do these attack-surface patterns imply for existing browser security testing? We overlay a decade of academic browser fuzzers, classified by their targeted input class, onto the bug-density map from RQ2. The overlay shows that academic fuzzers cluster on the script and document inputs, so the bug-dense surfaces identified in RQ2 stay under-covered. Beyond the testing techniques themselves, we diagnose three recurring deployment gaps that explain part of the shortfall: incorrect configuration, oversimplified harnesses, and missing harnesses.

Overall, this paper presents the following major findings:

- An architecture-grounded systematization of the browser's low-level attack surface: an Input × Component × Privilege taxonomy abstracted from Chrome, Firefox, and Safari.
- An empirical study of historical browser memory corruption vulnerabilities: 2,233 reports from Chrome and Firefox (2016–2025) classified onto the taxonomy to locate bug-dense input-to-component paths.
- A defense-oriented systematization of well-tested and overlooked attack surfaces: vendor coverage and a decade of academic fuzzers overlaid on the bug-density map, exposing under-covered surfaces and three recurring deployment gaps.
- Future directions for browser security research: tighter Principle-of-Least-Privilege enforcement, expanded coverage of insufficiently tested attack surfaces, and low-cost deployment fixes.

2. Scope and Threat Model

Study scope. This paper examines the low-level attack surface of the web browser and the corresponding testing techniques. We define the attack surface as the union of

TABLE 1. COMPONENTS AND INTERFACES OF MODERN BROWSERS. WE DEFINE THE COMPONENT PRIVILEGE FOLLOWING THE VENDOR PRACTICE [17]. **PRIVILEGE:** VL VERY LOW, L LOW, M MODERATE, H HIGH. **SCALE:** WE MEASURE ON CHROME CODEBASE (LOW $<$ 1M, MOD 1–10M LOC, HIGH $\geq$ 10M LOC).

Impact	Component	Function	Privilege			Scale	Process content
			Chrome	Firefox	Safari		
Low	HTML engine	HTML / CSS parsing, DOM, layout, paint	L	L	L	High	HTML Document
	PDF engine	Parse and render PDF documents	L	L	L	Low	PDF document
	JS engine	JS / Wasm execution (in sandbox)	VL	VL	VL	Mod	Script
	Media decoder	Decode images, audio, and video	L	VL	M	Mod	Binary blob
	Third-party parser	Parse fonts, XML, and other formats	L	L	L	Mod	Binary blob
Moderate	GPU	WebGPU/WebGL backend + shader translation	M	H [+]	M	High	Shader script, IPC
	Network	TCP/UDP + TLS + cert validation	M	M	M	Mod	Binary (packet, cert)
	Utility	Privileged helper tasks (print, etc.)	M	M	H	Low	IPC
High	Browser proc.	Profile, navigation, Secure UI	H	H	H	High	UI gesture, IPC

[+] Firefox runs the GPU process unsandboxed except Windows.

paths from each attacker-controlled input class into the native code region that parses it. Each path forms one *surface*, exercised at that code's privilege tier, and Section 4 enumerates these surfaces. For the attack surface, we study the three major web browsers: Chrome, Firefox, and Safari. For the bug study, we explore memory corruption bugs, which are the dominant source of in-the-wild exploits [1]. For the testing technique, we primarily study the fuzzing technique and discuss the rest in Section 7. Figure 1 summarizes our systematization.

Threat model. Following vendor practices [17], we consider a remote attacker who tricks a user into clicking attacker-controlled web pages and into performing specific UI interactions. The attacker's goal is to corrupt the victim's web browser process memory, achieve code execution in one process, elevate privileges and ultimately achieve code execution in the browser process or even in the kernel. We also account for scenarios in which an attacker already has code execution capability inside a compromised low privilege process and sends crafted IPC commands to corrupt higher privilege processes [18].

3. Browser Components and Interfaces

To answer RQ1, we first examine the components and interfaces in the web browsers. Modern browsers follow the principle of least privilege, running each functional component in a separate process at its own OS-level privilege tier, so that a memory bug in one process cannot, by itself, corrupt another [10], [19]. We systematize how Chrome, Firefox, and Safari enforce this partitioning by leveraging their design documents [10]–[13], [19]. The architecture includes two parts: components and interfaces. A *component* is a subsystem that implements a specific browser function and processes a distinct class of attacker-controlled input. Typical browser components include the HTML engine,

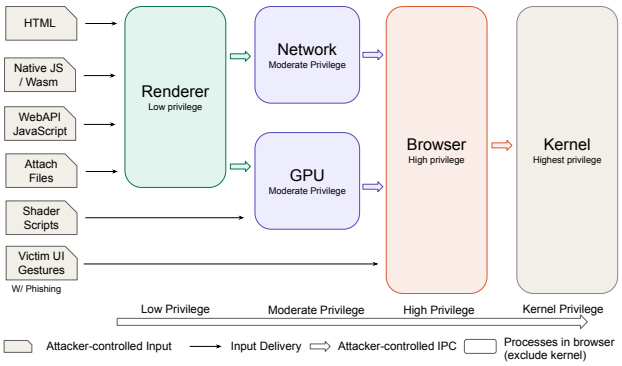

Figure 2. The browser attack surface. We only enumerate the main surfaces and processes. For instance, the network process may receive untrusted packets (binary blobs) from an attacker-controlled website.

PDF engine, JavaScript engine, media decoder, third-party parsers, GPU, network, utility, and browser process, each at a defined privilege tier. Components communicate through inter-process communication (IPC), which we treat as an attacker-controlled channel under the compromised-renderer threat model [18]. Table 1 summarizes each component's role, scale, content, and privilege across the three vendors. Figure 2 illustrates how inputs flow through them.

HTML engine. The HTML engine parses HTML and CSS, builds the DOM tree, and drives layout, paint, and the document API or WebAPI. All three browsers host it in a low-privilege renderer process [20], [21]. When the engine needs a privileged resource (a file dialog, a network request, GPU acceleration), it requests one from a higher-privilege component rather than acting on the resource itself. Because it parses untrusted documents directly from the network, the HTML engine is directly exposed to attacker-controlled input, and at roughly 10M lines of code, which is one of

the largest single-purpose components in the browser.

PDF engine. The PDF engine parses and renders embedded PDF documents. Chrome and Safari ship dedicated native engines (PDFium in Chrome, PDFKit in Safari) and host them inside the low-privilege process [13], [20]. Firefox takes a different route and renders PDFs via PDF.js [22], a JavaScript application that runs inside the JavaScript engine itself. PDF documents are fully attacker-shaped on all three browsers, so the PDF engine is directly exposed to attacker-controlled bytes. At roughly 500K lines of code it is only an small component in the browser.

JavaScript engine. The JavaScript engine (V8 in Chrome, SpiderMonkey in Firefox, JavaScriptCore in Safari [23]–[25]) executes JavaScript and WebAssembly drawn from the loaded document. To bound the impact of the vulnerabilities, JavaScript code runs inside an internal sandbox [26] that confines it to a restricted memory region. This internal sandbox itself lives inside the renderer process and shares its OS-level low-privilege. The engine is fully exposed to attacker-controlled bytes, and spans roughly 3M lines of code.

Media decoder. The media decoder demuxes and decodes the binary image, audio, and video formats that pages reference or embed (*e.g.,* PNG, JPEG, WebP, AVIF, MP4, WebM). Chrome integrates image decoding into the Tab process and offloads hardware-accelerated video decoding to the GPU process when the platform supports it [27]. Firefox runs a dedicated RDD (Remote Data Decoder) process for video and a GMPPlugin process for images at very low privilege [28]. Safari folds both into the GPU process at moderate privilege [29], [30] to improve performance. The media decoder weighs in at roughly 4M lines of code, the bulk of which sits in third-party codec and real-time communication libraries.

Third-party parser. Beyond the core decoders, the browser embeds a long tail of third-party libraries that parse other attacker-supplied formats: FreeType and HarfBuzz for fonts, libxml and expat for XML, and similar parsers for additional formats. All of them live inside the renderer sandbox at low privilege. These parsers see fully attacker-controlled bytes and have a long history of memory corruption disclosures. Together they add roughly 1M lines of code to the renderer.

GPU. The GPU process drives the system's graphics driver, compiles attacker-supplied shader programs, and executes the resulting draw calls [11]–[13]. Because it must call into the closed-source GPU driver, the GPU sandbox has more permission than the renderer. On Firefox, the GPU process runs entirely unsandboxed on non-Windows OSes [28], [31], [32]. The GPU component is by far the largest in the browser at roughly 33M lines of code, the bulk of which is WebGL and WebGPU backends.

Network. The network component owns the TCP stack, SSL/TLS handshake, and certificate validation. All browsers, including Chrome, Firefox, and Safari isolate it into a dedicated network process at moderate privilege [13], [28]. One exception is Firefox, whose NSS cryptographic stack validates certificates in the high-privilege parent process [33]. Although the network process mostly handles IPC, it receives raw bytes from remote servers, which a malicious server can fully control. The networking stack spans roughly 4.5M lines of code.

Utility. Utility processes host functionality that needs more privilege than the renderer but less than the browser [34]. Chrome and Firefox both run multiple utility processes at moderate privilege. Safari instead folds some tasks into the central UIProcess at high privilege. The utility tier accounts for roughly 280K lines of code, the smallest among the components.

Browser process. The browser process is the coordinator: it owns the user-profile database, the cookie store, and the navigation bar, and displays the results to the user. It also receives direct user input events (mouse, keyboard, touch), which become a partially attacker-controlled input under phishing or social-engineering scenarios [17]. All three browsers run at the highest privilege in the browser ecosystem [1]. Counting the reusable browser modules it hosts (autofill, password manager, safe browsing, and similar), the browser process spans roughly 12M lines of code.

Interfaces (IPC). Components in different processes communicate through the IPC: Chrome uses Mojo, Firefox uses IPDL, and Safari uses XPC. By design, every IPC message from a lower-privilege component to a higher-privilege one must be validated by the receiver. In practice, IPC handlers are a recurring source of memory corruption bugs, because they expose privileged components to messages crafted by an attacker who has already compromised the renderer (the canonical sandbox-escape vector). We therefore treat IPC as a first-class attacker-controlled input alongside web content, not as a transport detail (Section 4).

> TAKEAWAY. Modern browsers enforce the principle of least privilege by isolating the HTML, PDF, and JavaScript engines, media and resource parsers, GPU, network, utility, and browser code into separate processes at different OS-level privileges.

4. Browser Attack Surface

To complete the answer to RQ1, we build on the component and privilege structure defined in Section 3 and characterize the threats from the attacker's perspective: the classes of input an attacker can control, and, for each class, the component the input reaches and the privilege it thereby places at risk, following the threat model in Section 2.

We group the browser inputs into five classes, illustrated in Figure 3 and summarised in Table 2. Note that the table only discusses the main components. We further enumerate each class's surfaces: the inputs and the components that parse them.

1. The kernel is outside the ecosystem, which we further discuss in Section 4.

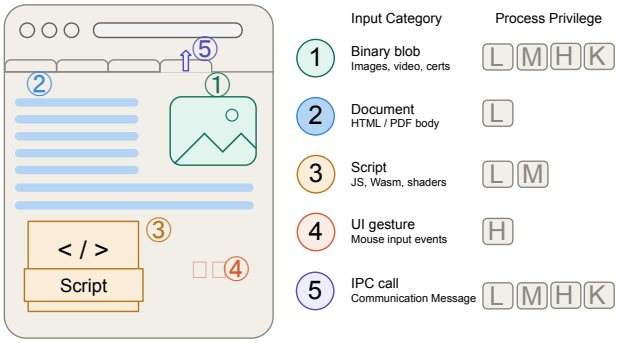

Figure 3. The five categories of input a web browser accepts. The arrow at the top denotes IPC requests issued by the renderer (Tab) process; these become fully attacker-controlled once the attacker achieves renderer code execution.

TABLE 2. THE FIVE INPUT CLASSES, WITH REPRESENTATIVE FORMATS, THE COMPONENT PRIMARILY REACHED, THE PRIVILEGE EXERCISED THERE, AND THE DEGREE OF ATTACKER CONTROL. PRIVILEGE IS COLOUR-CODED: L LOW, M MODERATE, H HIGH AND K KERNEL. ATTACKER CONTROL USES DISTINCT MARKERS: ● FULL CONTROL; ◐ FULL CONTROL ONLY ONCE THE SENDING PROCESS IS COMPROMISED; ○ LIMITED CONTROL, REQUIRING USER PARTICIPATION. *: HTML BODIES ARE PROCESSED BY THE RENDERER PROCESS, BUT ITS DOCUMENT API AND WEBAPI CALLS CAN IMPACT MODERATE/HIGH PRIVILEGE PROCESSES.

Input class	Repre. Format	Target	Priv.	Control
Binary blob	Image, font, Flash	Renderer	L	●
	Video	Renderer, GPU	L M	●
	Network packet	Network, Kernel	M K	●
	Crypto Cert	Network, Browser	M H	●
Document*	HTML, PDF	Renderer	L	●
Script	JavaScript	Renderer	L	●
	Shader	GPU	M	●
UI gesture	Input event	Browser	H	○
IPC call	Sandbox IPC	Renderer	L	◐
	Process IPC	Non-renderer	M H	◐
	Syscall	Kernel	K	◐

4.1. Attack Surface: Binary-Blob Inputs

Web content carries files in diverse formats, including images, fonts, database files, and video [35]–[38]. These inputs reach the browser along two paths: embedded within the HTML document, or delivered as a network payload from an attacker-controlled website [33], [39]. Although each format has its own specification, none imposes the strong structural requirements of a full document format, so we group them together as *binary blobs*.

Because binary blobs arrive directly from untrusted websites, browsers process most of them inside strictly sandboxed, low-privilege processes, typically the renderer, to bound the impact of a parsing-library vulnerability. Some blobs, however, are passed directly to moderate- or high-privilege processes. A hardware-accelerated video stream, for example, is demuxed in the renderer but decoded in the GPU process, which drives the hardware accelerator [27]. Likewise, network packets and cryptographic certificates are processed in the moderate-privilege network service or even high-privilege browser process rather than in the renderer.

> TAKEAWAY. Binary blobs primarily threaten low-privilege components, but certain formats, such as hardware-accelerated video, network packets, and cryptographic certificates, are handled directly in moderate- or high-privilege processes.

4.2. Attack Surface: Document Inputs

Both HTML and PDF documents consist of a body and embedded JavaScript, and all three browsers treat the document as untrusted input. The HTML and PDF engines (Section 3) parse the body inside low-privilege processes in accordance with the principle of least privilege [10].

The threat surface of a document is not confined to its parser, however, since embedded JavaScript can invoke APIs that call into external modules. Here the two formats diverge sharply. HTML scripts have access to both document APIs and WebAPI. Both can reach external modules that run in higher-privilege processes. A document that exercises the WebGPU or WebGL API, for instance, drives graphics code hosted in the moderate-privilege GPU process, so attacker-supplied content can reach and corrupt the moderate-privilege GPU process. PDF, by contrast, exposes a far more restricted JavaScript API that operates only on the document itself. Its scripts cannot reach beyond the renderer, so PDF content remains confined to the low-privilege process.

> TAKEAWAY. Document bodies are parsed in low-privilege processes across all vendors. However, HTML may include WebAPI and document API calls that propagate to higher-privilege processes.

4.3. Attack Surface: Script Inputs

Scripts are embedded in web content to enable web applications and programmable graphics, and are therefore directly under attacker control. Native JavaScript and WebAssembly (Wasm) are delivered straight to the JavaScript engine, which in all modern browsers runs inside a sandboxed, low-privilege process. This placement confines a highly flexible attacker input at an acceptable performance cost, and it reflects the fact that JavaScript engines have historically been a rich source of memory corruption bugs.

JavaScript additionally exposes WebAPI and document APIs, through which a script can interact with other browser subsystems and target higher-privilege components. We account for these paths under other input classes: a WebAPI call that requires no special privilege is folded into the

document class (Section 4.2), while a Mojo WebAPI that presupposes a compromised renderer is classified as an IPC call (Section 4.5). This subsection therefore concerns only native JavaScript.

Beyond JavaScript, browsers also accept native graphics-shader programs as script input. These are passed directly to the shader translator hosted in the GPU process, which translates the shader into a backend-specific language (*e.g.,* HLSL on Windows). The result is then compiled by a vendor-provided compiler (*e.g.,* DXC on Windows) before execution on the GPU. This pipeline allows attacker-controlled input to reach a moderate-privilege process. We do not consider vendor-provided compilers in this work, as they are maintained by the operating-system vendor and are in some cases closed-source [40].

> TAKEAWAY. Native JavaScript only threatens the sandboxed JavaScript engine, whereas graphics shader programs can carry attacker-controlled input into moderate-privilege components such as the GPU process.

4.4. Attack Surface: UI Gestures

UI gestures originate from the user (victim) and are, by default, treated as trusted input. They are mostly processed directly by the browser process, the most privileged component in the architecture. Vendor threat models nonetheless acknowledge that an attacker may persuade a user to perform a specific sequence of UI actions, through a phishing page or other social engineering, so the input is partially, though not fully, attacker-controlled [17]. Unlike inputs that an attacker controls directly, a UI-driven attack typically requires user participation and a specific interaction sequence. When the required sequence is sufficiently complex or implausible, vendors classify the issue as a functional bug rather than a security vulnerability, which bounds the practical reach of this class even though it targets the high-privilege component.

> TAKEAWAY. UI gestures are only indirectly attacker-controlled, yet they are processed by the high-privilege browser process.

4.5. Attack Surface: IPC Calls

Lower-privilege processes or components must request resources from higher-privilege ones, and IPC is the mechanism for doing so. By design, any IPC message originating from a lower-privilege process should be treated as attacker-controlled and validated accordingly. In practice this assumption is not always applied consistently, and an insufficiently-validated IPC message lets an attacker who has compromised a lower-privilege process escape the sandbox and escalate privilege. We restrict attention to the case where a lower-privilege process directs IPC at a higher-privilege one, since the reverse direction yields the attacker nothing, and distinguish three targets.

Native JavaScript engine. The JavaScript engine runs inside a strict sandbox within the renderer. Memory corruption within the engine affects only the engine's own sandboxed memory and cannot, by itself, corrupt the renderer memory outside the sandbox. To escape, an attacker must first achieve code execution inside the engine sandbox, then craft native JavaScript methods that write outside the sandbox region, thereby crossing the first privilege boundary. Only out of the sandbox write primitives are considered harmful in this threat model.

Browser-process IPC. Browser components communicate over IPC. Here an attacker with full control of a lower-privilege process crafts malicious IPC messages aimed at a higher-privilege one (*e.g.,* renderer to GPU, or GPU to browser). If the receiver validates its input insufficiently, the crafted message corrupts its memory, converting code execution in a low-privilege process into memory corruption in a high-privilege one.

System calls. Like any program, browser processes issue system calls to the OS kernel. Lower-privilege processes are restricted to a narrow set, higher-privilege processes to a broader one. An attacker who has compromised the renderer can issue the system calls available to it directly against the kernel, corrupting kernel memory without traversing the intermediate browser-process hierarchy at all [41].

> TAKEAWAY. Attackers with a compromised low privilege process can issue crafted IPC calls to higher privilege components, corrupt their memory, and achieve privilege escalation.

5. Bug Discovery vs. Testing Coverage

To answer RQ2, we identify the components where the discovered vulnerabilities arise and measure how well vendor testing covers them. Specifically, we map 2,233 reported memory corruption vulnerabilities from the past decade (2016–2025) onto the Input × Component × Privilege taxonomy of Section 4, measure the line coverage Chromium in-tree fuzzers achieve, and assess whether bug-manifesting components are well tested. Table 3 presents an overview.

Data sources. We collect vulnerability reports from the Chromium issue tracker and Firefox security advisories [14], [15], from 2016-01-01 to 2025-12-01, including 2,770 Chrome reports and 1329 Firefox reports[2]. We exclude Safari, whose tracker omits bug descriptions. Overall, our dataset of 4099 bugs includes 2,233 memory corruption reports. Moreover, we collect fuzzer code coverage from Chromium coverage dashboard [16], and include the third-party library fuzzing coverage from the OSS-Fuzz coverage [4] for third-party parsers outside the Chromium tree. Line coverage presented are the sum of all fuzzing engines, including libFuzzer, Centipede and Fuzzilli.

2. Firefox Bugzilla has a longer disclosure period, so we fetched reports up to MFSA-2025-51.

TABLE 3. JOINT PRIVILEGE × INPUT VIEW OF THE 2,233 MEMORY-CORRUPTION REPORTS. THE *Gap?* COLUMN FLAGS CLASSES WHOSE BUG-DENSEST SUBTREE SITS BELOW 30% COVERAGE.

	Bug discovery by privilege				*Total*	*High-priv*	*Best-covered subtree*		*Bug-densest subtree*	*Gap?*
Class	Low	Mod.	High	Kern.		share	name	cov. %	name and cov. %	
Binary	**135**	9	25	1	170	15%	OSS-Fuzz parsers	60–90	device brokers 1–12	–
Document	**773**	**126**	**151**	1	1,051	14%	Blink (whole)	36.6	`modules/` 0.1–5.5	**yes**
Script	**317**	54	0	0	371	0%	V8 (engine)	64.7	shader back end (off-tree)	**yes**
UI	1	2	**280**	0	283	99%	whole-browser harness	14–34	no dedicated UI fuzzer	**yes**
IPC	73	68	**174**	43	358	61%	`mojo+ipc`	55	`*/browser/` 15–28	**yes**

Data processing. We collect five fields from the bug report: (i) whether the report describes a *memory corruption* bug; (ii) the *triggering input class*; (iii) the *component name* where the bug manifests; (iv) the *privilege level* of the process whose memory is corrupted; and (v) the *disclosure year*. Among all these fields, (iii) - (v) are processed using string matching and never go through the model. Specifically, we match the shepherd-classified module name to the component, then obtain the process privilege according to the component name. The first two require reading the report context and developer response, so we extract them with `claude-opus-4-7`. To validate the labels, one author (who reported over 10 confirmed Chrome vulnerabilities) manually reviewed both LLM-produced fields on a random sample of 100 reports, and all 100 matched the LLM output.

5.1. Binary Blob: Mostly Least Privilege, Parsers Well Covered

Binary blobs are the only input class whose testing investment is well aligned with the discovered bugs. By design, every binary blob parser should run inside a low-privilege process, following the principle of least privilege [10], [42], and 79% (135/170) of the reports align with this principle. The image, font, video, markup, and compression parsers all are located inside a low privileged renderer process. Coverage mirrors the bug distribution: Most parsers, covered by OSS-Fuzz, achieve 50–90% line-coverage, with only four below 50% (`libwebp` 48, `libavif` 49, `icu` 48, WebRTC 33).

The 21% of bugs that impact privileged code concentrate in a few components. The Firefox NSS cryptographic stack contributes seven high-privilege reports that corrupt the parent process directly, with no Chrome counterpart. Network-packet parsing (Necko/HTTP and QUIC) contributes another five privileged reports. One ChromeOS Bluetooth-stack report reaches kernel memory through the in-kernel BlueZ driver. The coverage of privileged binary surfaces is uneven: SQLite (79%), libphonenumber (57%), PAK (50%), and NSS (43%) are reasonably fuzzed, while the device brokers remain insufficiently tested (Bluetooth 1%, WebHID 9%, gdk-pixbuf 12%).

TABLE 4. DOCUMENT CLASS: COVERAGE VS. BUG COUNT PER SURFACE.

Surface	Cov %	Bugs
Blink (HTML)		
Document (`renderer/core/`)	41.5	160
DocumentAPI (`renderer/core/`)	41.5	240
WebAPI (`renderer/modules/`)	17.4	485
PDFium (PDF)		
`core`	62.7	22
`xfa`	56.8	74
`fpdfsdk`	9.7	34
others (`third_party/`, ...)	–	36

TAKEAWAY. Binary blobs are the only class where testing investment matches the discovered bugs. Privileged components that violate least privilege are also poorly covered.

5.2. Documents: Bugs in WebAPI, Coverage in the Document Body

Documents are the largest single input class in the dataset: 885 HTML reports (606 Chrome, 279 Firefox) and 166 Chrome PDF reports. The HTML reports partition cleanly into three sub-categories. The *document body* (160 reports: parser, layout, CSS, paint, SVG) is well sandboxed: 91% stays in the renderer. The *document API* (240 reports: `window.*`, `document.*`, navigation, extensions) is also renderer-dominated, with only 16% escaping to UI/browser processes. The *WebAPI bindings* (485 reports: WebGL, WebGPU, WebRTC, Canvas, WebAudio, Media, Payments) is the sub-category that drives bugs out of the renderer: 46% of WebAPI reports land in a moderate-, high-, or kernel-privilege process, with WebGL alone contributing 107 reports against the GPU process. PDF reports follow a different trajectory: all 166 remain at low privilege because PDFium's embedded JavaScript cannot issue IPC.

The coverage profile is the inverse. Blink as a whole sits at 36.6% line coverage. The document body parsers and the document API components (`renderer/core/`) reach 42–57%. The WebAPI surface (`renderer/modules/`)

TABLE 5. SCRIPT CLASS: COVERAGE VS. BUG COUNT PER SURFACE.

Surface	Cov %	Bugs
JavaScript engine (`//v8/`)	64.7	312
WebGL backend (`//third_party/angle/`)	34.0	11
WebGPU backend (`//third_party/dawn/`)	73.6	13

sits at 17% aggregate, with the four bug-densest modules at 0.1% (WebGL), 3.6% (WebGPU) and 5.5% (WebRTC). PDFium tells the same story at smaller scale: the parsers are well covered, while the JavaScript-driven SDK boundary `fpdfsdk` sits at 10% (Table 4).

> TAKEAWAY. The document class shows the sharpest misalignment between bugs and coverage. JavaScript-driven API bindings produce most bugs yet receive the least coverage.

5.3. Scripts: Well Sandboxed, WebGL Backend Overlooked

The script class is the best-sandboxed class in the dataset: no Script bug corrupts a browser-process or the kernel. Among the 371 reports, 312 land at low privilege inside the V8 sandbox region of the renderer (244 native JavaScript and 68 WebAssembly). The remaining 59 shader reports split into 54 in the moderate-privilege GPU process and 5 in the renderer-side shader preprocessing.

Coverage is split cleanly by surface (Table 5): V8 and Dawn (WebGPU) are well covered, while ANGLE (WebGL) only achieves 34%. Beyond the in-tree translators, the bulk of shader bugs sits in stages that are not on the coverage dashboard at all: DXC (the shader compiler that handles the ANGLE/Dawn output) accounts for 14 bugs in the dataset and lives outside the Chromium tree, other vendor compilers contribute 21 bugs and live outside OSS-Fuzz. As these compilers are vendor specific and some are even close-sourced, we cannot measure their actual coverage [40].

> TAKEAWAY. The script class produces no high-privilege bugs, matching the threat model. The JavaScript/Wasm engine and the WebGPU translator are well-fuzzed, while the WebGL translator and the vendor-controlled shader compilers are not.

5.4. UI Gestures: High-Privilege Concentrator, MiraclePtr Mitigated

UI gestures are the most privilege-concentrated class among five input classes: 99% of the 283 UI reports corrupt the high-privilege process directly, and ∼90% of Chrome UI reports are use-after-free. The bugs fall into three surfaces (Table 6): *BrowserChrome* (cross-platform widgets), *PlatformIntegration* (OS-specific toolkit), and *DevToolsWebUI* (internal pages).

TABLE 6. UI CLASS: COVERAGE VS. BUG COUNT PER SURFACE.

Surface (source directory)	Cov %	Bugs
BrowserChrome (`chrome/browser/ui`)	14.3	201
PlatformIntegration (`//ui/`)	33.8	72
DevToolsWebUI (`chrome/browser/devtools`)	2.4	10

TABLE 7. IPC CLASS: COVERAGE VS. BUG COUNT PER SURFACE.

Surface (directory)	Cov %	Bugs
Receiver		
`chrome/browser/`	15.3	45
`content/browser/`	27.7	44
`components/`	20.3	13
`services/`	36.2	10
Adapter		
`mojo/`	55.4	7
`ipc/`	55.1	

The Chrome UI bug count climbs from 2 in 2016 to a peak of 112 in 2022, then collapses to 47, 22, 7, respectively. The reduction matches the deployment of MiraclePtr [43], which keeps freed memory alive while any `raw_ptr<T>` reference exists, so any UAF on a protected object becomes an unexploitable functional bug [17].

Coverage of the UI class is the weakest of the five. The only specialised fuzzers target accessibility features and lift `ui/accessibility` and `content/browser/accessibility` [44]. Everything else is covered as a side effect of whole-browser harnesses (Table 6).

> TAKEAWAY. UI carries the highest percentage of high-privilege bugs and the lowest dedicated coverage. A deployed mitigation reduced the number of exploitable bugs, but the underlying defects persist and the testing surface remains unaddressed.

5.5. IPC: Privilege Concentrator, Adapter Covered, Dispatch Not

IPC is the only class that spans all four privilege tiers, and the only class that targets the privilege boundaries. The 358 reports partition into three sub-types. *V8 sandbox escape* (73 Chrome reports) corrupts memory inside the V8 sandbox region. It is low-privilege at the point of corruption but routes to a sandbox-escape outcome. *Normal IPC* (182 Chrome and 25 Firefox reports) is the canonical sandbox-escape vector: 173 (84%) land in a high-privilege host process and 34 in moderate-privilege utility, GPU, or network processes. *Syscall IPC* (78 reports) reaches the kernel directly: 43 corrupt kernel memory, 34 land in the *virglrenderer* (a GPU process handling VM requests on ChromeOS), and one reaches the ChromeOS parent process, which has higher privilege than the normal browser process.

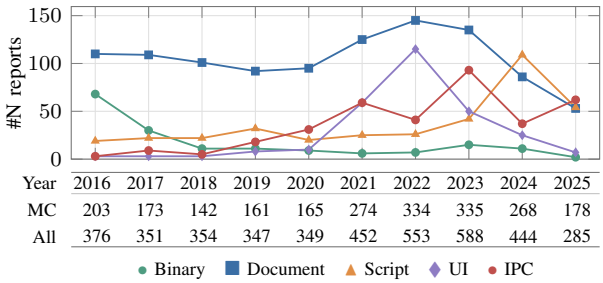

Figure 4. Memory corruption and total vulnerability reports by year and input class. MC: Memory corruption.

Coverage flips between the IPC adapter (where IPC requests are forwarded) and the dispatch targets (where IPC requests are processed). The adapter (`mojo/`, `ipc/`) sits at 55% but accounts for only 7 historical reports. In contrast, the dispatch targets manifest the remaining 112 vulnerabilities at much lower coverage (Table 7): `chrome/browser/` and `content/browser/` together own 75% of the in-tree IPC bugs and run in the high-privilege browser process. `services/<svc>/` is the only receiver tier whose modal privilege is moderate, because each service launches into its own utility process.

> TAKEAWAY. IPC's coverage is the inverse of where its bugs are. The well-covered adapter holds few bugs, while the receiver that hold most bugs are poorly covered.

5.6. Vulnerability Trend In The Past Decade

The per-year decomposition in Figure 4 shows that each class follows a distinct trajectory, from which we conclude four shifts.

Deployed mitigations reshape the bug-discovery distribution. UI disclosures climbed to a peak in 2022 and collapsed after the MiraclePtr deployment (Section 5.4), demonstrating that a single deployed mitigation can collapse a bug class without big code changes.

Upstream features introduce new bugs. WebAssembly (a subclass of script) discoveries remained in the single digits through 2023 and stepped up to 37 in 2024. The step coincides with the shipping of WebAssembly garbage collection, the JavaScript promise integration, exception handling, and relaxed SIMD [45]–[47], a set of new features landing in 2023 and 2024. The implication is direct: under-tested new code introduces new vulnerabilities.

A new harness on a previously-untested surface boosts the bug discovery. Shader bugs spiked in 2024: of the 59 shader reports across the decade, 33 land in that year alone, mostly from one academic work [48] that built new harnesses for the WebGPU translator stack. A new harness on an untested surface boosts a class's bug discovery at a much smaller effort than improving fuzzing across the whole browser.

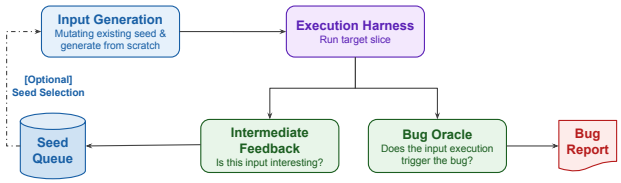

Figure 5. The pipeline of a fuzzer.

Discovered bugs are shifting from in-renderer to cross-privilege surfaces. IPC bugs increase from 3 in 2016 to a peak of 93 in 2023, while binary-blob reports collapse from 68 in 2016 to single digits after 2018 as the Flash adapter retires and OSS-Fuzz improves. The shallow in-renderer surfaces are now well tested, and attack targets are moving from low-privilege to high-privilege code.

> TAKEAWAY. The binary-blob and UI surfaces recede, while the IPC, Wasm, and shader surfaces grow.

6. Browser Testing Techniques

To answer RQ3, we examine whether browser testing techniques cover the bug-dense attack surfaces. We primarily focus on the browser fuzzing techniques and discuss other testing techniques in Section 7. This section first decomposes the fuzzing pipeline, then surveys academic browser fuzzers along the attack surfaces, and finally identifies overlooked deployment gaps.

6.1. Fuzzing Pipeline and Deployment

A fuzzer is built from five stages that together determine which inputs it produces, how it judges progress, and how it is wired into the target. These five stages map to the columns of Table 8 and Table 9: **M** mutation, **G** generation, **F** intermediate feedback, **O** bug oracle, and **D** deployment.

(M) Mutation technique. Mutation-based fuzzers derive each new input from an existing seed through two internal stages: *seed selection* (which seed to mutate) and *mutation* (how to mutate the seed). For structured inputs, mutation operates on a high-level representation (*e.g.,* AST or IR) so that syntactic validity is preserved across edits.

(G) Generation technique. Generation-based fuzzers synthesise each input from scratch rather than mutating a seed. Specifically, the fuzzer leverages a template (*e.g.,* grammar, IDL, or custom IR) to generate inputs. The selected template varies for distinct targets: context-free grammars for HTML/CSS, IDL files for browser API bindings, and specifications for WebAssembly modules.

(F) Intermediate feedback. Intermediate feedback is the signal that decides which inputs are interesting enough to keep and re-mutate. The canonical signal is edge coverage produced by instrumentation, but a long line of work refines

TABLE 8. BROWSER-RELEVANT FUZZING TECHNIQUES TARGETING *binary* AND *document* INPUTS. **M** MUTATION TECHNIQUE, **G** GENERATION TECHNIQUE, **F** INTERMEDIATE FEEDBACK, **O** BUG ORACLE, AND **D** DEPLOYMENT. FOUR FURTHER COLUMNS STAND FOR THE PAPER'S EFFECTIVENESS CLAIMS: **COV** – PAPER CLAIMS A COVERAGE IMPROVEMENT; **BUG** – PAPER CLAIMS TO FIND NEW BUGS IN THE TARGET; **BROW** – PAPER CLAIMS TO FIND NEW BROWSER VULNERABILITIES; **$** – PAPER RECEIVES BOUNTY FROM BROWSER VENDORS; **BUG** IS DIFFERENT FROM **BROW**, AS FUZZERS MAY TARGET NON-BROWSER IMPLEMENTATION (*e.g.,* COOPER [49] TESTS ADOBE AND FOXIT PDF).

			Contributions					Effectiveness				
Technique	Year	Venue	M	G	F	O	D	Cov	Bug	Brow	$	One-line summary of core contribution
Binary-blob fuzzers — general-purpose CGFs; target third-party parsers in the browser												
AFLFast [50]	2016	CCS	●	○	○	○	○	✓	✓	○	$	Markov-chain seed scheduling
Driller [51]	2016	NDSS	●	○	○	○	○	✓	✓	○	$	Concolic fallback when CGF stalls
AFLGo [52]	2017	CCS	●	○	●	○	○	○	✓	○	$	Distance-to-target directed fuzzing
Angora [53]	2018	S&P	●	○	○	○	○	✓	✓	○	$	Byte-level taint + gradient descent
CollAFL [54]	2018	S&P	○	○	●	○	○	✓	✓	○	$	Path-sensitive coverage resolves hash collisions
QSYM [55]	2018	USENIX	●	○	○	○	○	✓	✓	○	$	Fast hybrid symbolic execution
Redqueen [56]	2019	NDSS	●	○	●	○	○	✓	✓	○	$	Input-to-state correspondence; register values as feedback
EcoFuzz [57]	2020	USENIX	●	○	○	○	○	✓	✓	○	$	Energy allocation under adversarial bandits
GreyOne [58]	2020	USENIX	●	○	○	○	○	✓	✓	○	$	Taint-aided byte mutation
ParmeSan [59]	2020	USENIX	●	○	●	○	○	✓	✓	○	$	Sanitizer-guided distance-to-target feedback
SymCC [60]	2020	USENIX	●	○	○	○	○	✓	✓	○	$	Compile-time symbolic-execution instrumentation
TortoiseFuzz [61]	2020	NDSS	○	○	●	○	○	○	✓	○	$	Security-instruction-weighted feedback
Weizz [62]	2020	ISSTA	●	○	○	○	○	✓	✓	○	$	Automatic format inference
OptiMin [63]	2021	ISSTA	●	○	○	○	○	○	○	○	$	Empirical study of seed-selection heuristics
Jigsaw [64]	2022	S&P	●	○	○	○	○	✓	✓	○	$	Efficient path-constraint solver for CGF
K-Scheduler [65]	2022	S&P	●	○	●	○	○	✓	✓	○	$	Reachable-node count as feedback for seed scheduling
PATA [66]	2022	S&P	●	○	○	○	○	✓	✓	○	$	Path-aware taint analysis
SymSan [67]	2022	USENIX	●	○	○	○	○	✓	✓	○	$	Sanitizer-style symbolic-execution backend
Truzz [68]	2022	ICSE	●	○	○	○	○	✓	✓	○	$	Runtime program-state guided scheduling
AIFORE [69]	2023	USENIX	●	○	○	○	○	✓	✓	○	$	AI-driven format inference
FishFuzz [70]	2023	USENIX	●	○	●	○	○	✓	✓	○	$	Many-target directed fuzzing with distance-based feedback
MendelFuzz [71]	2025	FSE	●	○	○	○	○	✓	✓	○	$	Selective deterministic stage on critical bytes
ZTaint-Havoc [72]	2025	ISSTA	●	○	○	○	○	✓	✓	○	$	Zero-execution taint inference in havoc
Document fuzzers (HTML / PDF); including the WebAPI and document APIs												
FreeDom [73]	2020	CCS	●	●	○	○	○	✓	✓	✓	$	Context-aware IR; supports both generative and mutation modes
Favocado [74]	2021	NDSS	●	●	○	○	○	○	✓	✓	$	IDL-driven generation + state-aware mutation
FuzzOrigin [75]	2022	USENIX	○	○	○	●	○	○	✓	✓	$	Static origin-tagging oracle
Minerva [76]	2022	FSE	○	●	○	○	○	✓	✓	✓	$	Mod-ref API graph guides API-sequence synthesis
GLeeFuzz [77]	2023	USENIX	●	○	●	○	○	✓	✓	✓	$	GL error messages as lightweight feedback for WebGL
COOPER [49]	2022	NDSS	●	○	○	○	○	✓	✓	○	$	Cooperative mutation of PDF + embedded JS
TypeOracle [78]	2023	ICSE	○	●	○	○	○	✓	✓	○	$	Differential operand-variation type inference

or replaces it with a more domain-specific signal: path-sensitive coverage, memory operation coverage, sandbox-boundary read events, GL error messages, register values, call-graph distance, or reachable-node counts.

(O) Bug oracle. The bug oracle is the predicate that decides whether a particular execution exposed a bug. Most fuzzers rely on memory sanitizers [100], which cannot detect logic bugs. Thus, many works implement differential oracles to find vulnerabilities beyond memory corruption, by comparing the different execution results.

(D) Deployment. Deployment is how the fuzzer is connected to the target in practice. It includes: (i) *harness design*, which decides the program slice the fuzzer actually drives (whole browser, hand-written deep-API driver); and (ii) *fuzzer configuration*, which decides how the available grammars, harnesses, and feedback signals are composed in the running fuzzer. Most academic works inherit the vendor's default deployment (Section 6.3).

6.2. Existing Fuzzers By Attack Surface

Table 8 and Table 9 provide a survey of existing browser fuzzers, grouped by their target attack surfaces. For each surface, we first recall whether there is a gap between bug discovery and testing coverage (Section 5), then discuss whether the surveyed techniques can fill the gap.

Binary-blob fuzzers. Most binary-blob bugs arise in low-privilege parsers, which are already well tested (Section 5.1). As standalone libraries, these parsers are tested by general-purpose fuzzers, which primarily leverage mutation-based input generation. For *seed selection*, existing works model the fuzzing process [50], [57], calculate the seed-target distance [52], [65], [70] and measure the impact of initial corpus [63]. For *mutation*, they introduce concolic execution [51], [55], [60], [64], [67], taint analysis [53],

TABLE 9. BROWSER-RELEVANT FUZZING TECHNIQUES TARGETING *script*, *shader*, AND *sandbox/IPC* INPUTS. LEGEND MATCHES TABLE 8.

			Contributions					Effectiveness				
Technique	Year	Venue	M	G	F	O	D	Cov	Bug	Brow	$	One-line summary of core contribution
JavaScript-engine fuzzers												
CodeAlchemist [79]	2019	NDSS	○	●	○	○	○	●	●	●	$	AST code-bricks with assembly constraints
DIE [80]	2020	S&P	●	○	○	○	○	●	●	●	$	Aspect-preserving mutation of typed AST
Montage [81]	2020	USENIX	○	●	○	○	○	○	●	●	$	LSTM-trained AST fragment generation
Jest [82]	2021	ICSE	○	○	○	●	○	○	●	●	$	N+1-version differential testing of JS engines
SOFI [83]	2021	CCS	●	●	○	○	○	●	●	●	$	Reflection-augmented mutation and generation
JIT-Picking [84]	2022	CCS	○	○	○	●	○	○	●	●	$	Differential test between JIT tiers
Fuzzilli [85]	2023	NDSS	●	●	○	○	○	●	●	●	$	FuzzIL IR with mutation engine and code generators
FuzzJIT [86]	2023	USENIX	●	●	○	●	○	●	●	●	$	JIT-tier-aware mutation atop Fuzzilli's generation
FuzzFlow [87]	2024	CCS	●	○	○	○	○	●	●	●	$	FlowIR graph-based IR mutation with decoupled control/data flow
OptFuzz [88]	2024	USENIX	○	○	●	○	○	●	●	●	$	JIT-optimisation-path feedback
Dumpling [89]	2025	NDSS	○	○	○	●	○	○	●	●	$	Fine-grained differential JIT oracle
WebAssembly fuzzers												
WADIFF [90]	2023	ASE	○	●	○	●	○	○	●	○	$	Symbolic-execution-driven per-instruction generation from spec
Wapplique [91]	2024	ISSTA	●	○	○	○	○	●	●	○	$	Code-fragment appliqué substitution into seed bytecode
Wasmaker [92]	2024	ISSTA	●	○	○	●	○	○	●	○	$	Semantic-aware disassemble-reassemble of real-world binaries
FreeWavm [93]	2025	ISSTA	●	○	○	○	○	●	●	○	$	Parse-tree structure mutation with snapshot guidance
LWDIFF [94]	2025	ICSE	○	●	○	●	○	●	●	○	$	LLM-extracted specification knowledge drives generation
RGFuzz [95]	2025	S&P	○	●	○	●	○	●	●	●	$	Compiler-rule-guided generation with reverse-stack synthesis
Stanley'25 [96]	2025	ASE	○	●	○	●	○	○	●	○	$	Random differential testing of WIT binding generators
Waltzz [97]	2025	USENIX	●	●	○	○	○	●	●	○	$	Stack-invariant mutators with skeleton-based snippet generation
WEST [98]	2025	ASE	○	●	○	○	○	○	●	●	$	SpecTec mechanized-specification-driven test generation
Shader fuzzers												
DarthShader [48]	2024	CCS	●	○	○	○	●	●	●	●	$	Dual AST/IR mutation; WebGPU translator harness
Sandbox / IPC fuzzers												
SbxBrk [99]	2025	CCS	○	○	●	○	○	○	●	●	$	Sandbox-boundary read events as feedback

[56], [58], [66], [72], and input-format inference [62], [68], [69], [71] to improve constraint solving. For *intermediate feedback*, a line of work replaces the default edge-coverage signal with a domain-specific signal: CollAFL [54] introduces path-sensitive coverage that resolves hash collisions; TortoiseFuzz [61] directs fuzzing toward code that exercises memory-unsafe operations through customized feedback; Redqueen [56] uses register values observed at compare instructions as an additional input-to-state feedback signal; AFLGo [52], ParmeSan [59], and FishFuzz [70] use the call-graph distance from each input to the target sites as feedback to steer mutation; and K-Scheduler [65] uses the number of reachable nodes as feedback to score seeds. The bug oracle relies on memory sanitizers [100], and the harness is typically the OSS-Fuzz library wrapper for each codec or parser library. The remaining gap, the poorly covered privileged parsers, stems not from the techniques but from the deployment (Section 6.3).

Document fuzzers. The document class shows the sharpest gap: the WebAPI bindings and the PDFium SDK boundary hold most bugs yet the least coverage (Section 5.2). For document testing, byte-level mutation breaks syntax validity, so works operate on high-level representations. For *mutation*, existing works propose cooperative mutation of document objects together with their embedded JavaScript [49] and feedback-guided structural mutation of WebGL API call sequences driven by domain-specific error signals [77]. For *generation*, context-free grammars [101] and context-aware IRs [73] synthesise HTML documents, mod-ref API dependency graphs compose call sequences with high inter-API interaction [76], vendor-defined IDLs collect candidate APIs [74], and differential type inference recovers parameter types when the IDL is incomplete [78]. For the *bug oracle*, beyond memory sanitizers, a static origin-tagging oracle has been proposed to detect non-memory-corruption UXSS bugs [75]. The *intermediate feedback* signal is either the edge coverage or the GL error messages [77]. Several document fuzzers already target the bug-dense WebAPI bindings [74], [76], [78], [101]. However, vendor deployments fail to import the available grammars, which leaves these modules under-covered. The PDFium boundary suffers instead from an oversimplified harness (Section 6.3).

Script fuzzers (JavaScript, WebAssembly, shaders). In the script class, the JavaScript/Wasm engines are well fuzzed. The gap sits at the WebGL backend and the vendor shader compilers (Section 5.3). As JavaScript, Wasm, and shaders are all strictly validated before execution, works raise the mutation granularity above bytes to preserve validity. For *mutation*, existing JavaScript and shader works operate on typed ASTs to preserve language aspects [48], [80] or on graph-based IRs that decouple control and data

flow [87]; existing Wasm works substitute code fragments into seed bytecode [91], mutate parse-tree structure with snapshot guidance [93], and synthesise stack-invariant instruction snippets [97]. For *generation*, JavaScript fuzzers compose programs from semantics-aware code bricks [79], learned language models [81], and reflection-discovered runtime types [83], and further specialise an IR to the JIT compiler [85], [86]. Wasm fuzzers synthesise stack-valid modules driven by spec-to-DSL symbolic execution [90], mechanised specifications [98], LLM-extracted spec knowledge [94], production-runtime rewrite rules [95], and skeleton-based snippet templates [97]. For *intermediate feedback*, JS engine fuzzers refine the default edge-coverage signal by tracking JIT optimisation paths [88] and sandbox-boundary memory reads [99]. For the *bug oracle*, differential testing surfaces non-crashing semantic bugs: JIT-vs-interpreter comparison within a single engine [84], [86], [89], specification-as-oracle across multiple engine implementations [82], cross-runtime comparison on real-world Wasm binaries [92], and cross-binding-generator divergence on WIT interfaces [96]. Most works thus further harden the well-fuzzed engines. Only DarthShader [48] builds a dedicated harness for the WebGPU shader-translation pipeline (Dawn), which sits deep in the browser and is hard to reach. The WebGL backend (ANGLE) and the vendor shader compilers remain under-tested.

UI-gesture and IPC fuzzers. Bugs in UI gestures (Section 5.4) and IPC calls (Section 5.5) concentrate in high-privilege processes, yet academic browser fuzzers are largely absent. Only SbxBrk [99] explores sandbox-escape testing, by fuzzing the IPC channel from the V8 sandbox into the JavaScript engine. Both surfaces are otherwise reached only indirectly through whole-browser fuzzing.

Projecting these fuzzers onto the bug-density map of Section 5 answers RQ3. Academic works cluster on the script and document inputs: twenty JavaScript/Wasm and seven document fuzzers, against one shader fuzzer, one IPC fuzzer, and none for UI gestures. Hence, the UI and IPC surfaces lack dedicated fuzzers, while the WebAPI bindings, the PDFium SDK boundary, and the WebGL backend have applicable techniques, but deployment gaps keep their coverage low.

6.3. Overlooked Deployment Gaps

Most techniques in Section 6.2 focus on the Mutation (**M**), Generation (**G**), Intermediate Feedback (**F**), and Bug Oracle (**O**) stages. Researchers inherit the vendor deployment and assume it is correct. However, Section 5 suggests otherwise: the deployment itself leaves gaps. We observe three recurring gaps.

Gap 1: incorrect configuration. Document fuzzers [73], [74], [101] rely on a complete list of the target API interfaces to cover the corresponding components. Even though the required grammars are well defined in the codebase, a harness can still fail to import them, leaving whole modules untested. For instance, in the Chromium tree, the Domato engine [101] ships well-defined grammars (*e.g.,* WebGL, WebGPU), yet the `domato_html_in_process_fuzzer` harness that drives Domato does not import them. As a result, the corresponding WebAPI modules are barely exercised: `modules/webgl` at 0.1% and `modules/webgpu` at 3.6% (Section 5.2). These are not algorithmic gaps but developer configuration mistakes, and they can be resolved by importing the existing grammars into the fuzzing engine.

Gap 2: oversimplified harness. Developers write dedicated fuzzing harnesses to directly expose deeper internal interfaces to the fuzzer. While such hand-written harnesses follow the API semantics, they are not always effective [102]. In the pursuit of simplicity, a developer may omit optional callback fields that the target only invokes when they are supplied. Browsers, by contrast, leverage those fields, so the omission silently elides the code paths that the browser exercises. For instance, we observe that the poor coverage of PDFium's `fpdfsdk` layer originates from null-filled embedder callbacks. Concretely, the shared driver (testing / fuzzers / pdfium_fuzzer_helper . cc) on which every PDFium fuzzer relies leaves the callback tables null, even though Chromium's production PDF integration populates them. Since `fpdfsdk` guards each callback invocation with a non-null check, the form-fill, JavaScript-platform, and named-action dispatch paths are never entered. Similar issues also arise in libpng, media and webcodec fuzzers, suggesting the necessity of fixing such issues.

Gap 3: missing harness. Most academic works fuzz the whole browser or JavaScript engine to stay faithful to the threat model. However, driving the entire target is slow and tends to miss attack surfaces that are hard to reach from the top-level entry point. Dedicated harnesses that expose these components directly are therefore needed for thorough testing. In practice, writing such harnesses costs developer effort, so several bug-dense components still lack any dedicated harness. For instance, IPC receivers (Section 5.5) are not adequately exercised by whole-browser fuzzing alone, leaving their logic untested despite a large volume of historical bugs. Building dedicated harnesses for these components is thus a concrete, one-time investment that recovers the missing coverage.

7. Discussion

Browser testing techniques beyond fuzzing. While fuzzing is the most widely deployed automated testing technique against browsers, vendors and security researchers apply complementary techniques as well, including manual code audits [103], static analysis [104], symbolic execution [105], [106], and LLM-based testing [107]. Different from fuzzing, these techniques are not tied to a specific input class and can in principle reach any component, directly targeting the deeply nested privileged code, such as the UI event path or the IPC dispatch handlers. However, many of them rely either on human expertise or on component-specific heuristics that do not scale to the whole browser. Manual code audits depend on the reviewer's familiarity with the target subsystem and cover only the fraction of

code. Static analysis avoids that bottleneck but suffers from a higher false positive rate. Symbolic and concolic execution reason more precisely about reachable states, but path explosion limits their application. LLM-based testing is a new emerging direction and we further discuss below.

Limitations. Three limitations may threaten the validity of this systematization. The first concerns the precision of our data processing. Most fields in each vulnerability report are extracted by string keyword matching against shepherd-labelled metadata; a small number are derived from LLM-based parsing of the report text. Section 5 describes the pipeline and the manual validation we ran on a sample of 100 reports. The second concerns the interpretation of coverage values. There is no fixed line that separates well-tested code from under-tested code, and the right cut-off can differ across components. We therefore use 30% as a working threshold in this paper. We do not claim this value is precise, but it shows the relative trend across components, which is how our findings should be interpreterd. The third concerns data availability. Our systematization is limited to the data that vendors release publicly. In particular, Safari does not publicly release bug reports, so we cannot include it in the dataset. To avoid overfitting our conclusions to a single vendor, we collect both Chrome and Firefox reports and compare their counts and distributions.

LLM-based browser vulnerability discovery. The improving capability of large language models is now enabling the detection of deeper bugs in real-world software. Early attempts, limited by the model capability, only discovered vulnerabilities in smaller-scale projects such as SQLite [107]. However, recent frontier models have reported vulnerabilities in the JavaScript engine and the browser itself [108]. This shift suggests a fast improvement in capability and points to substantial potential for future LLM-based applications. Compared with traditional fuzzing, LLM-driven testing can generate diverse inputs and resolve complex API dependencies, which allows deeper exploration of the attack surface, though typically at a higher cost per test. These properties make LLM-driven testing a promising direction for the overlooked testing surfaces identified in Section 6.3, where existing public efforts are insufficient.

8. Lessons From the Systematization

Principle of least privilege is violated. By design, a web browser should follow the principle of least privilege (PoLP): untrusted (web) content must not be handled in privileged code [10]. However, Section 4 suggests this principle is not always applied. For instance, the moderate-privilege GPU process takes graphic shaders (attacker-controlled input) directly, allowing the attacker to corrupt privileged code. The same pattern affects the network process (network packets) and the Firefox browser process (certificate validation). An ideal browser should enforce the PoLP to isolate and minimize the risks.

Testing investment is misaligned with bug yield. Despite extensive fuzzing efforts from industry and academia, the coverage profile of Section 5 shows that several components remain systematically under-tested precisely where they contribute the most memory corruption bugs. The pattern recurs across input classes: bug-heavy surfaces such as Blink's WebAPI bindings (*e.g.,* WebGL, WebGPU), PDFium's SDK boundary (`fpdfsdk`), the UI input path (*BrowserChrome*, *DevToolsWebUI*), and the IPC dispatch targets (`chrome/browser/`, `content/browser/`) are all under-covered. The structural cause is that testing concentrates where mature techniques exist (*e.g.,* binary CGF, V8 engine fuzzing) and largely skips surfaces for which no public harness was ever written.

Beyond the algorithm: fix the deployment mistakes. The fuzzing literature surveyed in Section 6 optimises four algorithmic stages under the implicit assumption that a developer correctly deploys the fuzzer at every reachable surface. The coverage gaps in Section 5 suggest this assumption breaks in three patterns. *Incorrect configuration:* the algorithm and the input specification both exist, but the deployed configuration fails to link them (*e.g.,* Blink's WebAPI IDLs are not imported by Domato's default grammar). *Oversimplified testing harness:* a harness exists for the target surface, but the surrounding environment is stubbed too aggressively (*e.g.,* PDFium's reference fuzzer leaves every form-fill FFI callback null. Mojo grammar fuzzers analogously fix optional message fields to defaults). *Missing harness:* a bug-dense surface has no dedicated harness at all (*e.g.,* the UI input path, where few public Chromium fuzzers are available). Each requires a deployment fix that no algorithmic improvement can substitute for.

9. Conclusion

Despite a decade of intensive browser testing, the community lacks a unified view of the browser's low-level attack surface and of how far that testing reaches into it. This paper systematizes the browser's low-level attack surface along an Input $\times$ Component $\times$ Privilege taxonomy abstracted from the browser design documents. We map 2,233 memory corruption reports disclosed between 2016 and 2025 onto this taxonomy, pair each component with the line coverage that vendor-deployed fuzzers achieve, and overlay a decade of academic browser fuzzers on the same map. The combined view exposes five surfaces where testing investment diverges from bug yield (the WebAPI bindings, the PDFium SDK boundary, the WebGL backend, the UI input path, and the IPC receivers) and three recurring deployment gaps (incorrect configuration, oversimplified harnesses, and missing harnesses).

These findings point to three directions for future browser security work: finer-grained least-privilege separation that converts high-privilege bugs into low-privilege ones; testing-coverage expansion for the five under-tested surfaces; and closing the three deployment gaps. We hope future work can use these findings as a map of the architecture, a measurement of past effort, and a prioritisation guide for where new effort is most needed.

References

[1] M. Stone, "The more you know, the more you know you don't know," https://projectzero.google/2022/04/the-more-you-know-more-you-know-you.html, 2022.

[2] C. Lecigne, "State-backed attackers and commercial surveillance vendors repeatedly use the same exploits," https://blog.google/threat-analysis-group/state-backed-attackers-and-commercial-surveillance-vendors-repeatedly-use-the-same-exploits/, 2024.

[3] S. Huntley, "Buying spying: How the commercial surveillance industry works and what can be done about it," https://blog.google/threat-analysis-group/commercial-surveillance-vendors-google-tag-report/, 2024.

[4] OSS-Fuzz, "Fuzzing introspection of oss-fuzz projects," https://introspector.oss-fuzz.com/, 2026.

[5] Google, "Clusterfuzz," https://google.github.io/clusterfuzz/, 2026.

[6] S. Jacobus, "Vulnerability reward program: 2023 year in review," https://security.googleblog.com/2024/03/vulnerability-reward-program-2023-year.html, 2024.

[7] D. Göhmann, "Vulnerability reward program: 2024 in review," https://blog.google/security/vulnerability-reward-program-2024-in/, 2025.

[8] T. M. Dirk Göhmann, "Vrp 2025 year in review," https://blog.google/security/vrp-2025-year-in-review/, 2026.

[9] J. Lim, Y. Jin, M. Alharthi, X. Zhang, J. Jung, R. Gupta, K. Li, D. Jang, and T. Kim, "Sok: On the analysis of web browser security," *arXiv preprint arXiv:2112.15561*, 2021.

[10] Chrome, "Core principles," https://www.chromium.org/developers/core-principles/, 2026.

[11] Chromium, "Chromium process type," https://source.chromium.org/chromium/chromium/src/+/main:content/common/process_type.cc, 2026.

[12] Firefox, "Firefox process type," https://searchfox.org/firefox-main/source/xpcom/geckoprocesstypes_generator/geckoprocesstypes/__init__.py, 2026.

[13] Safari, "Safari process type," https://github.com/WebKit/WebKit/tree/main/Source/WebKit, 2026.

[14] Chromium, "Chromium issue tracker," https://issues.chromium.org/, 2026.

[15] Firefox, "Security advisories for firefox," https://www.mozilla.org/en-US/security/known-vulnerabilities/firefox/, 2026.

[16] Chromium, "Chromium coverage dashboard," https://analysis.chromium.org/coverage/p/chromium, 2026.

[17] Chrome, "Severity guidelines for security issues," https://chromium.googlesource.com/chromium/src/+/main/docs/security/severity-guidelines.md, 2026.

[18] ——, "Threat model and defenses against compromised renderers," https://chromium.googlesource.com/chromium/src/+/main/docs/security/compromised-renderers.md, 2026.

[19] C. Reis, "Multi-process architecture," https://blog.chromium.org/2008/09/multi-process-architecture.html, 2008.

[20] Chrome, "Sandbox," https://chromium.googlesource.com/chromium/src/+/main/docs/design/sandbox.md, 2026.

[21] Firefox, "Security/sandbox," https://wiki.mozilla.org/Security/Sandbox, 2026.

[22] ——, "Pdf.js," https://mozilla.github.io/pdf.js/, 2026.

[23] Chrome, "What is v8?" https://v8.dev/, 2026.

[24] Apple, "Javascriptcore," https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html, 2026.

[25] Firefox, "Spidermonkey," https://firefox-source-docs.mozilla.org/js/index.html, 2026.

[26] S. Groß, "The v8 sandbox," https://v8.dev/blog/sandbox, 2024.

[27] Chromium, "Deep-dive: Videong," https://developer.chrome.com/docs/chromium/videong, 2026.

[28] Firefox, "Firefox process type," https://firefox-source-docs.mozilla.org/dom/ipc/process_model.html, 2026.

[29] Safari, "Safari image decoding in gpuprocess," https://github.com/WebKit/WebKit/tree/main/Source/WebKit/GPUProcess/graphics, 2026.

[30] ——, "Safari media decoding in gpuprocess," https://github.com/WebKit/WebKit/tree/main/Source/WebKit/GPUProcess/media, 2026.

[31] Chrome, "Why graphics?" https://chromium.googlesource.com/chromium/src/+/main/docs/security/research/graphics/README.md, 2026.

[32] ——, "Unsandboxed processes by platform," https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/security/process-sandboxes-by-platform.md, 2026.

[33] mozillabugs, "Double free in sec_pkcs7_decoder_start_decrypt," https://bugzilla.mozilla.org/show_bug.cgi?id=1899402, 2024.

[34] Chromium, "Ooppd linux unable to display system print dialog from utility process," https://issues.chromium.org/issues/40242360, 2022.

[35] B. Tiszka and D. Manouchehri, "Security: Uaf in imagedecoderexternal due to arraybuffer neuter," https://issues.chromium.org/issues/40053811, 2020.

[36] gserviceaccount, "Cros: Vulnerability reported in media-libs/freetype," https://issues.chromium.org/issues/40059561, 2022.

[37] Z. Chen and N. Wang, "Security: Oob write in sqlite3findinindex," https://issues.chromium.org/issues/40060728, 2022.

[38] E. Hohl, "Buffer overflow (gpu process) in chrome windows media foundation encode accelerator," https://issues.chromium.org/issues/409619251, 2025.

[39] J. Cristau, "Crash in mozilla::net::http2stream::transmitframe," https://bugzilla.mozilla.org/show_bug.cgi?id=1667102, 2021.

[40] Apple, "Metal resources," https://developer.apple.com/metal/resources/, 2026.

[41] J. Horn, "From chrome renderer code exec to kernel with msgoob," https://projectzero.google/2025/08/from-chrome-renderer-code-exec-to-kernel.html, 2025.

[42] Chrome, "The rule of 2," https://chromium.googlesource.com/chromium/src/+/master/docs/security/rule-of-2.md, 2026.

[43] ——, "Miracleptr," https://chromium.googlesource.com/chromium/src/+/main/base/memory/raw_ptr.md, 2026.

[44] A. Taylor, "Using chrome's accessibility apis to find security bugs," https://security.googleblog.com/2024/10/using-chromes-accessibility-apis-to.html, 2024.

[45] E. Ziegler, "Intent to ship: Webassembly garbage collection (wasmgc)," https://groups.google.com/a/chromium.org/g/blink-dev/c/K_GpDF0y5Q8, 2023.

[46] Chromestatus, "Intent to ship: Javascript promise integration," https://groups.google.com/a/chromium.org/g/blink-dev/c/w_jCD4gf7Bc, 2025.

[47] D. Gandluri, "Intent to ship: Webassembly relaxed simd," https://groups.google.com/a/chromium.org/g/blink-dev/c/HzLlEGLSx7E, 2023.

[48] L. Bernhard, N. Schiller, M. Schloegel, N. Bars, and T. Holz, "Darthshader: Fuzzing webgpu shader translators & compilers," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 690–704.

[49] P. Xu, Y. Wang, H. Hu, and P. Su, "Cooper: Testing the binding code of scripting languages with cooperative mutation." in *NDSS*, 2022.

[50] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032–1043.

[51] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[52] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2329–2344.

[53] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[54] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[55] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.

[56] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[57] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "{EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2307–2324.

[58] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "{GREYONE}: Data flow sensitive fuzzing," in *29th USENIX security symposium (USENIX Security 20)*, 2020, pp. 2577–2594.

[59] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "{ParmeSan}: Sanitizer-guided greybox fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2289–2306.

[60] S. Poeplau and A. Francillon, "Symbolic execution with {SymCC}: Don't interpret, compile!" in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 181–198.

[61] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization." in *NDSS*, 2020.

[62] A. Fioraldi, D. C. D'Elia, and E. Coppa, "Weizz: Automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 1–13.

[63] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, 2021, pp. 230–243.

[64] J. Chen, J. Wang, C. Song, and H. Yin, "Jigsaw: Efficient and scalable path constraints fuzzing," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 18–35.

[65] D. She, A. Shah, and S. Jana, "Effective seed scheduling for fuzzing with graph centrality analysis," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2194–2211.

[66] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun, "Pata: Fuzzing with path aware taint analysis," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1–17.

[67] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, "{SYMSAN}: Time and space efficient concolic execution via dynamic data-flow analysis," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2531–2548.

[68] K. Zhang, X. Xiao, X. Zhu, R. Sun, M. Xue, and S. Wen, "Path transitions tell more: Optimizing fuzzing schedules via runtime program states," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1658–1668.

[69] J. Shi, Z. Wang, Z. Feng, Y. Lan, S. Qin, W. You, W. Zou, M. Payer, and C. Zhang, "{AIFORE}: Smart fuzzing based on automatic input format reverse engineering," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4967–4984.

[70] H. Zheng, J. Zhang, Y. Huang, Z. Ren, H. Wang, C. Cao, Y. Zhang, F. Toffalini, and M. Payer, "{FISHFUZZ}: Catch deeper bugs by throwing larger nets," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1343–1360.

[71] H. Zheng, F. Toffalini, M. Böhme, and M. Payer, "Mendelfuzz: The return of the deterministic stage," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 44–64, 2025.

[72] Y. Xie, W. Zhang, and D. She, "Ztaint-havoc: From havoc mode to zero-execution fuzzing-driven taint inference," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 917–939, 2025.

[73] W. Xu, S. Park, and T. Kim, "Freedom: Engineering a state-of-the-art dom fuzzer," in *Proceedings of the 2020 acm sigsac conference on computer and communications security*, 2020, pp. 971–986.

[74] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé *et al.*, "Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases." in *NDSS*, 2021.

[75] S. Kim, Y. M. Kim, J. Hur, S. Song, G. Lee, and B. Lee, "{FuzzOrigin}: Detecting {UXSS} vulnerabilities in browsers through origin fuzzing," in *31st usenix security symposium (usenix security 22)*, 2022, pp. 1008–1023.

[76] C. Zhou, Q. Zhang, M. Wang, L. Guo, J. Liang, Z. Liu, M. Payer, and Y. Jiang, "Minerva: browser api fuzzing with dynamic mod-ref analysis," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1135–1147.

[77] H. Peng, Z. Yao, A. A. Sani, D. J. Tian, and M. Payer, "{GLeeFuzz}: Fuzzing {WebGL} through error message guided mutation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1883–1899.

[78] S. Guo, X. Wan, W. You, B. Liang, W. Shi, Y. Zhang, J. Huang, and J. Zhang, "Operand-variation-oriented differential analysis for fuzzing binding calls in pdf readers," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 95–107.

[79] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines." in *NDSS*, 2019.

[80] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1629–1642.

[81] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language {Model-Guided}{JavaScript} engine fuzzer," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2613–2630.

[82] J. Park, S. An, D. Youn, G. Kim, and S. Ryu, "Jest: N+ 1-version differential testing of both javascript engines and specification," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 13–24.

[83] X. He, X. Xie, Y. Li, J. Sun, F. Li, W. Zou, Y. Liu, L. Yu, J. Zhou, W. Shi *et al.*, "Sofi: Reflection-augmented fuzzing for javascript engines," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2229–2242.

[84] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "Jit-picking: Differential fuzzing of javascript engines," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 351–364.

[85] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, "Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities." in *NDSS*, 2023.

[86] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "{FuzzJIT}:{Oracle-Enhanced} fuzzing for {JavaScript} engine {JIT} compiler," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1865–1882.

[87] H. Xu, Z. Jiang, Y. Wang, S. Fan, S. Xu, P. Xie, S. Fu, and M. Payer, "Fuzzing javascript engines with a graph-based ir," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3734–3748.

[88] J. Wang, Y. Kang, C. Wu, Y. Hu, Y. Sun, J. Ren, Y. Lai, M. Xie, C. Zhang, T. Li *et al.*, "{OptFuzz}: Optimization path guided fuzzing for {JavaScript}{JIT} compilers," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 865–882.

[89] L. Wachter, J. Gremminger, C. Wressnegger, M. Payer, and F. Toffalini, "Dumpling: Fine-grained differential javascript engine fuzzing," in *NDSS*, 2025.

[90] S. Zhou, M. Jiang, W. Chen, H. Zhou, H. Wang, and X. Luo, "Wadiff: A differential testing framework for webassembly runtimes," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 939–950.

[91] W. Zhao, R. Zeng, and Y. Zhou, "Wapplique: Testing webassembly runtime via execution context-aware bytecode mutation," in *Proceedings of the 33rd ACM SIGSOFT international symposium on software testing and analysis*, 2024, pp. 1035–1047.

[92] S. Cao, N. He, X. She, Y. Zhang, M. Zhang, and H. Wang, "Wasmaker: Differential testing of webassembly runtimes via semantic-aware binary generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1262–1273.

[93] P. Qian, X. Ying, J. Wang, L. Liu, L. Zhang, J. Chen, and Q. He, "Freewavm: Enhanced webassembly runtime fuzzing guided by parse tree mutation and snapshot," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 159–181, 2025.

[94] S. Zhou, J. Wang, H. Ye, H. Zhou, C. Le Goues, and X. Luo, "Lwdiff: An llm-assisted differential testing framework for webassembly runtimes," in *2025 IEEE/ACM 47TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE*. IEEE, 2025, pp. 153–164.

[95] J. Park, Y. Kim, and I. Yun, "Rgfuzz: Rule-guided fuzzer for webassembly runtimes," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 920–938.

[96] E. Stanley and E. Eide, "Finding bugs in webassembly interface type binding generators," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2025, pp. 204–216.

[97] L. Zhang, B. Zhao, J. Xu, P. Liu, Q. Xie, Y. Tian, J. Chen, and S. Ji, "Waltzz:{WebAssembly} runtime fuzzing with {Stack-Invariant} transformation," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 6159–6178.

[98] D. Youn, W. Shin, and S. Ryu, "West: Specification-based test generation for webassembly," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2025, pp. 1403–1414.

[99] N. Bars, L. Bernhard, M. Schloegel, and T. Holz, "Empirical security analysis of software-based fault isolation through controlled fault injection," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 2639–2652.

[100] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "{AddressSanitizer}: A fast address sanity checker," in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.

[101] P. Zero, "Domato: A dom fuzzer," https://github.com/googleprojectzero/domato, 2017.

[102] P. Görz, J. Schilling, T. Holz, and M. Böhme, "An empirical study of fuzz harness degradation," *arXiv preprint arXiv:2505.06177*, 2025.

[103] Chrome, "Code reviews," https://chromium.googlesource.com/chromium/src/+/main/docs/code_reviews.md, 2026.

[104] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in javascript bindings," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 559–578.

[105] F. Brown, D. Stefan, and D. Engler, "Sys: A {Static/Symbolic} tool for finding good bugs in good (browser) code," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 199–216.

[106] H. Han, A. Wesie, and B. Pak, "Precise and scalable detection of {Use-after-Compacting-Garbage-Collection} bugs," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2059–2074.

[107] the Big Sleep team, "From naptime to big sleep: Using large language models to catch vulnerabilities in real-world code," https://projectzero.google/2024/10/from-naptime-to-big-sleep.html, 2024.

[108] A. F. R. Team, "Partnering with mozilla to improve firefox's security," https://www.anthropic.com/news/mozilla-firefox-security, 2026.

[109] J. Drescher, S. Mirzaei, S. Khodayari, D. Klein, T. Barber, M. Johns, and G. Pellegrino, "In the dom we trust: Exploring the hidden dangers of reading from the dom on the web," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 3042–3056.

[110] Z. Liu, T. Lee, J. Yu, Z. Kang, and Y. Cao, "The {DOMino} effect: Detecting and exploiting {DOM} clobbering gadgets via concolic execution with symbolic {DOM}," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 8293–8312.

[111] S. Song, J. Hur, S. Kim, P. Rogers, and B. Lee, "R2z2: Detecting rendering regressions in web browsers through differential fuzz testing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1818–1829.

[112] S. Song and B. Lee, "Metamong: Detecting render-update bugs in web browsers through fuzzing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1075–1087.

[113] F. Xiao, Z. Su, G. Yang, and W. Lee, "Jasmine: Scale up javascript static security analysis with computation-based semantic explanation," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 296–311.

[114] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1059–1076.

Appendix

1. Ethics Considerations

This paper studies disclosed vulnerabilities in the browser ecosystem. While we point out future directions for vulnerability research in the browser, the paper itself does not disclose any new vulnerability and therefore does not threaten the browser system. Moreover, we collect only public reports from vendor issue trackers. The entire data collection and processing pipeline involves no work with live systems or human-subject studies. We therefore consider that this work does not involve any ethics considerations.

2. Paper Selection Criteria

We collect papers published at top-tier security and software-engineering venues, including IEEE S&P, USENIX Security, NDSS, ACM CCS, ICSE, ISSTA, ASE, and FSE.

For documents, we search the keywords "DOM", "Browser", and "Binding" to find HTML testing papers, and further search "PDF" to identify PDF engine testing papers. We exclude server-side bug-finding works, such as those targeting SSRF and XSS detection [109], [110], and works that target only functional bugs [111], [112].

For scripts, we search the keywords "JavaScript", "WebAssembly", and "Wasm". We exclude papers that target web applications or Node.js packages [113], [114], keeping only works that target vulnerabilities in JavaScript or Wasm engines.

For graphics, we search the keywords "WebGPU" and "WebGL", the two major graphics APIs exposed by modern browsers.

For UI and IPC, we search the keywords "GUI" and "IPC". We find no academic works that specifically target browser UI or IPC vulnerabilities.

For binary inputs, we search the keyword "Fuzz" to cover general greybox fuzzing techniques.