0.1 Abstract

GrapheneOS advises against Gecko-based browsers like Firefox, citing a security gap compared to Chromium-based alternatives such as Vanadium. Their core claim about the absence of Android's `isolatedProcess` sandboxing in GeckoView is accurate and acknowledged here. Several subsidiary claims mix up distinct mitigation layers (pre-compromise versus post-compromise) and don't account for threat-model dependencies that change the security calculus. This paper evaluates those claims through a multi-layered threat-modeling lens using publicly available sources. It covers the architectural evolution of GeckoView on Android, including the January 2026 shipping and subsequent rollback of Project Fission (Site Isolation) in Firefox 147, which remains disabled on release and beta channels due to unresolved crash bugs (Section 2.3). It covers structural pre-compromise defenses from Firefox's Rust adoption, WebExtension isolation, and declarative content-blocking. It looks at privacy controls intersecting with exploit-chain disruption. And it considers the systemic security risk of a Chromium monoculture. The analysis finds that categorical dismissal of either engine family isn't supported by current evidence. Browser selection is not a binary "secure versus insecure" metric but alignment with a specific threat model.

0.2 1. Introduction and Historical Context

Criticism of Mozilla's Gecko engine on mobile platforms has historically centered on its single-process architecture. Before the completion of multi-process re-architecting, Gecko-based browsers on Android lacked robust site-level process boundaries, making them structurally vulnerable to microarchitectural side-channel attacks such as Spectre and Meltdown [6]. This deficiency was well-documented by Mozilla's own engineering team during the design phase of their multi-process architecture [4].

Security guidance in the mobile ecosystem suffers from persistent documentation latency. The most influential critique of GeckoView on Android is GrapheneOS's published advisory on web browsing [1]. It has remained substantively unchanged while the target architecture has evolved significantly. Criticisms rooted in architectural deficiencies that have since been partially addressed continue to circulate as authoritative guidance without reference to current implementation status.

This paper does two things. One, provide an evidence-based assessment of the security properties of both Chromium-based and Gecko-based browser architectures on Android. Distinguish between verified architectural properties, documented limitations, and areas where public evidence is incomplete. Two, offer a claim-by-claim examination of GrapheneOS's published position. Identify where assertions remain valid, where they've been superseded by developments, and where they reflect divergent threat-model priorities rather than objective security deficits.

Methodology. All claims herein are accompanied by citations to primary or secondary sources accessible as of July 2026. Where evidence is absent or contradictory, this is explicitly noted. The authors have no affiliation with Mozilla, GrapheneOS, the Chromium project, or any commercial browser vendor.

Scope and limitations of this analysis. This paper is a threat-modeling analysis, not an empirical vulnerability comparison. It evaluates architectural security properties, mitigation philosophies, and the alignment of each browser engine with different threat-model priorities. It does not attempt to measure or rank the absolute number of vulnerabilities per browser, and it does not provide a quantitative CVE comparison between Chromium and Firefox. Where vulnerability statistics from published sources (such as Chromium's memory safety data [14]) are cited, they are used to illustrate architec-

tural arguments, not as comparative metrics. Readers seeking a direct CVE-by-CVE comparison should consult the respective vendor security advisory pages [17] and Chromium release notes.

0.3 2. Architectural Analysis of Process Isolation

0.3.1 2.1 Kernel-Level Containment via `android:isolatedProcess`

The Android platform exposes a mechanism for declaring sandboxed service processes through the `android:isolatedProcess="true"` manifest attribute. When set, the Android kernel assigns the child process a heavily restricted, ephemeral User ID (UID). This UID is isolated from the parent application's permissions, file system access, and data streams outside explicit Inter-Process Communication (IPC) channels [9].

Chromium (and by extension Vanadium, GrapheneOS's hardened fork) uses this mechanism as the foundation of its renderer sandbox. Each renderer process runs under an isolated UID with access restricted to its assigned memory region and a narrow, explicitly defined IPC interface to the parent browser process. GrapheneOS explicitly identifies this as the strongest available sandbox implementation on Android [1], [2].

This mechanism provides a **post-compromise containment** guarantee. Even if an attacker achieves arbitrary code execution within a renderer process, the `isolatedProcess` sandbox severely restricts what system resources the compromised process can access. The attacker must then find a separate sandbox-escape vulnerability (typically targeting the browser's IPC layer or a kernel vulnerability) to achieve broader system access.

0.3.2 2.2 GeckoView's Mobile Sandboxing: The Current State

GeckoView does not implement the `isolatedProcess` mechanism for its child processes on Android. This is a substantiated architectural limitation that GrapheneOS correctly identifies [1]. The `isolatedProcess` flag is a declarative Android manifest property, as GrapheneOS describes it, "a very easy to use boolean property for app service processes" [1]. Its absence in GeckoView represents a deliberate engineering choice or resource-allocation decision by Mozilla.

The statement "Firefox does not have internal sandboxing on Android" treats the absence of one specific mechanism as the complete absence of sandboxing. Three developments have altered this assessment:

1. **Multi-process architecture.** GeckoView on Android has operated a multi-process architecture with a privileged parent process and separate content processes since the completion of its multi-process re-engineering. This provides process-level isolation between content and the browser chrome, even if the kernel-level protections differ from Chromium's.
2. **Project Fission (Site Isolation), shipped, then rolled back.** Mozilla shipped Site Isolation for Android in Firefox 147.0 (January 2026), with release notes stating: "Added protection against side-channel attacks such as Spectre using the same Site Isolation safeguards already in use by desktop Firefox" [12]. However, Firefox 147.0.2 (February 2026) disabled Fission on release and beta channels due to content process crashes causing random back-navigation (Bug 2011319). The isolation strategy default was reverted to `ISOLATE_NOTHING` for all channels except nightly and developer [27]. As of Firefox 152 (July 2026), Fission remains disabled on release and beta channels. The root cause, content process crashes when isolating sites with Fission (Bug 2012435), remains open with the fix still in progress [29]. On nightly and developer builds, Fission operates at `ISOLATE_HIGH_VALUE`, which isolates only "high value" sites (e.g., login pages) rather than providing full strict origin isolation. This is different from Chromium's site isolation model and means the cross-origin exfiltration protection described in the paper is currently unavailable on release Firefox for Android.

3. **Memory safety via Rust (Section 3).** An increasing portion of GeckoView's rendering pipeline is written in Rust, a memory-safe language that makes entire classes of vulnerabilities (use-after-free, buffer overflows) structurally impossible in safe code paths. This is a **pre-compromise** defense that reduces the probability of successful initial compromise. It is distinct from and complementary to post-compromise containment.

0.3.3 2.3 Project Fission on Android: Architecture and Caveats

Mozilla announced Fission's stable release for desktop Firefox in version 95 (December 2021) [3], [5]. The desktop implementation assigns each site origin to a dedicated operating system process, with IPC enforcement ensuring that cross-origin data access requires explicit, validated channels.

Fission shipped on Android in Firefox 147.0 (January 2026), but was rolled back in 147.0.2 (February 2026) and remains disabled on release and beta channels as of Firefox 152 (July 2026). The isolation strategy default is `ISOLATE_NOTHING` (0) for release and beta; only nightly and developer channels have `ISOLATE_HIGH_VALUE` (2) [27]. The original release notes cited Spectre-class side-channel protection [12], but this protection is currently not active on the default release configuration.

Rollback timeline.

1. **Bug 2003658** (December 2025/January 2026): Fission + SHIP turned on by default for Firefox 147 [30]. The initial implementation shipped with `ISOLATE_HIGH_VALUE`, isolating only "high value" sites (login pages, authentication flows) rather than providing full strict origin isolation.
2. **Firefox 147.0** (January 2026): Ships with Fission enabled.
3. **Bug 2011319** (February 2026): Users report content process crashes causing random back-navigation, a regression directly attributable to Fission [28].
4. **Bug 2011886** (January 23, 2026): Fission switched off in release and beta channels. Commit message: "Set isolation strategy to ISOLATE_NONE in all builds except nightly" [27].
5. **Bug 2012435** (February 2026, **still open**): Root cause identified as content process crashes when isolating sites with Fission. The fix remains in progress [29].

Current state (Firefox 152, July 2026). As reflected in Mozilla's `nimbus.fml.yaml` configuration: - Release and beta channels: `isolationStrategy: 0` (`ISOLATE_NOTHING`) - Nightly and developer channels: `isolationStrategy: 2` (`ISOLATE_HIGH_VALUE`) - Mozilla's Nimbus experiment system may enable Fission for some users in controlled experiments on release channels.

Architectural caveat. A thread on the PrivacyGuides forum noted that "unprivileged user namespaces are not available to apps on Android," meaning the kernel-level isolation mechanisms available on desktop Linux are not directly reproducible on Android [13]. Mozilla's claim of "the same Site Isolation safeguards" should be understood as referring to the same architectural approach (process-per-origin assignment, IPC boundary enforcement, cross-origin data access restriction) rather than identical kernel-level mechanisms. The precise kernel-level implementation differences between desktop and mobile Fission have not been publicly documented by Mozilla as of July 2026, and this remains the most significant gap in publicly verifiable information about the mobile architecture.

Even if Fission were active, it does not replicate the `isolatedProcess`-based sandbox that Chromium employs. It provides origin-level process assignment and IPC enforcement, preventing cross-origin data exfiltration even with side-channel attacks. But it does not provide the same post-exploit kernel-level containment. The relationship between Fission and `isolatedProcess` is complementary rather than substitutive. Fission's current `ISOLATE_HIGH_VALUE` strategy is also narrower than Chromium's full site isolation, it only isolates high-value origins.

0.3.4 2.4 Is `isolatedProcess` the Complete Picture?

Does the absence of `isolatedProcess` sandboxing settle the question of GeckoView's security posture? Three reasons it does not:

1. **Complementarity of pre-compromise and post-compromise defenses.** Fission's site isolation reduces the blast radius of a compromise (you cannot read other origins' data), while `isolatedProcess` reduces the capabilities of a compromised process (you cannot easily access system resources). These are orthogonal security properties, and the absence of one does not eliminate the value of the other.
2. **The Rust advantage (Section 3).** If the renderer process is materially harder to compromise in the first place due to memory-safe code, the relative importance of post-compromise containment is diminished. A sandbox is irrelevant if the renderer is never successfully exploited. However, this calculus must be qualified by the current status of Fission. Because site isolation is not active on release and beta channels (Section 2.3), a compromised renderer on release Firefox for Android does not have origin-level process boundaries. A successful exploit could access data across origins within the same process, increasing the cross-origin exfiltration risk relative to what Fission would provide if active. The Rust memory safety advantage reduces the *probability* of compromise, but the *blast radius* in the event of a successful exploit is larger than would be the case with active site isolation. The paper's original claim that Fission reduces blast radius remains architecturally correct, but the protection is not currently available in the default release configuration.
3. **Threat-model dependence.** For an attacker whose goal is cross-origin data exfiltration (for example, reading your banking session from a malicious ad), Fission provides the relevant defense. For an attacker whose goal is kernel-level persistence after compromising the renderer, `isolatedProcess` provides the relevant defense. These address different attack objectives.

0.3.5 2.5 Capability Expansion vs. Containment Rigor

The security calculus between Chromium and GeckoView is further complicated by each project's strategy for expanding the web platform's native device access APIs. This section examines the trade-off between surface area expansion and sandbox rigor.

Chromium has aggressively pushed the boundaries of what web-exposed interfaces can access. Key examples include:

- **Direct Sockets API.** Allows web applications to establish direct TCP and UDP communications, bypassing HTTP abstractions entirely. While gated behind permissions and secure contexts, this introduces raw networking access into the browser's attack surface. A vulnerability in the Direct Sockets implementation could expose low-level network access to an attacker who has achieved code execution in a renderer.
- **Isolated Web Apps (IWAs).** A new application model that blurs the line between web pages and native applications, granting packaged web apps elevated capabilities including raw socket access, file system write access, and system API exposure [31].
- **HID and USB device access.** APIs that allow web applications to enumerate and communicate directly with hardware peripherals. These represent a significant expansion of the browser's trusted computing base into hardware-interface code traditionally reserved for native applications.
- **File System Access API.** Gives web applications read and write access to the local file system outside the browser's storage sandbox. While user-gated through file pickers, the API expands the blast radius of a compromised renderer to include local file system data.

These capabilities expand the browser's trusted computing base at the API layer. Even with permission gates and sandboxing, each new API represents code paths that must be correct. As the Chromium

security team's own data shows, the majority of severe vulnerabilities are memory-safety bugs in C++ code [14]. Every additional thousand lines of C++ API implementation expands the pool of potential vulnerabilities, regardless of the sandbox quality.

Firefox's approach to these APIs is markedly more conservative. Mozilla has declined to implement Direct Sockets, Isolated Web Apps, and several of the more invasive device APIs. This is a deliberate product philosophy that prioritizes capability gating over capability expansion. The reasoning is straightforward: an API that does not exist in the codebase cannot be exploited.

The result is an asymmetric risk profile. Chromium builds a tighter post-compromise sandbox (through `isolatedProcess` and strict site isolation) but simultaneously expands its pre-compromise attack surface with aggressive device APIs. Firefox scales down sandbox thickness on certain platforms (Android, where `isolatedProcess` is absent and Fission remains disabled on release channels), but balances this by gating extension protocol access [32] and rejecting entire classes of web-exposed attack surface.

Bug 2034168 illustrates this philosophy in the extension context: as of Firefox 153 (July 2026), extensions can no longer access local files by default. The `file:` scheme access is gated behind an explicit "Access local files on your computer" permission, entirely separate from the broader "Access your data for all websites" permission [32]. An extension that can inject content scripts into every website cannot automatically read local files opened in the browser, because those two capabilities are now separated at the permission model level. This is the same principle applied at the extension API layer: capability gating as a structural defense, not merely post-compromise containment.

The comparison cannot be reduced to sandbox thickness alone. Chromium's larger API surface creates a larger vulnerability pool. Even well-contained exploitation of that pool still threatens cross-origin data confidentiality. Firefox's smaller API surface creates a smaller vulnerability pool, partially compensating for weaker kernel-level containment on Android. Neither approach is obviously superior. The two projects made different trade-offs at different layers. A fair comparison must account for both sides.

0.4 3. Memory Safety as a Structural Pre-Compromise Defense

The most significant structural security advantage of GeckoView over Chromium is not in sandboxing architecture but in codebase-level vulnerability resistance. This section examines Firefox's industry-leading adoption of the Rust programming language and its implications for exploit resilience.

0.4.1 3.1 Rust Adoption: Firefox vs. Chromium

Firefox has been systematically rewriting critical components from C++ to Rust. Rust is a memory-safe language that eliminates entire classes of vulnerabilities (use-after-free, buffer overflows, null pointer dereferences) at compile time. Major Rust components shipped in Firefox include:

Component	Function	Shipped	Significance
Stylo	CSS engine	2017	First large-scale Rust component in a major browser; replaced Gecko's C++ CSS system
RLBox	Library sandboxing via WebAssembly	2021	Isolates third-party libraries (font parsers, image decoders, audio codecs); confines exploits even within the same process

Necko	Networking stack	Progressive	Rust-based HTTP/3, DNS over HTTPS, and network protocol implementations
Audio/Video	Media pipeline	Progressive	Rust-based media decoders and processing pipelines
WebRender	GPU rendering	2019 (partial)	Rust-based GPU-accelerated rendering engine
Glyph/Text	Text shaping & rendering	Progressive	Rust-based font shaping and text layout
NSS	Cryptography	Progressive	Rust-based TLS and cryptographic primitives

Google's own security research has consistently found that roughly 70% of critical-severity vulnerabilities in Chromium are memory-safety bugs [14]. Microsoft's Security Response Center has reported comparable figures across its products [15]. By eliminating these vulnerability classes in safe Rust code, Firefox achieves a structural reduction in exploitable vulnerability density that no amount of kernel sandboxing can provide.

Chromium's Rust adoption remains limited and experimental. As of 2025-2026, Chromium's codebase remains predominantly C++. The V8 JavaScript engine (the single largest source of critical-severity vulnerabilities in Chromium) is written entirely in C++. Google's experimental "Rust in Chromium" initiative has produced limited production deployments, primarily in non-critical paths [16]. The Android-specific Chromium rendering pipeline, including the Blink engine, remains C++-dominant.

The Rust migration is not finished in Firefox either, but the trajectory is what matters. Each component ported from C++ to Rust removes an entire class of memory-safety vulnerabilities from that attack surface. As Mozilla continues porting additional subsystems (networking, media, graphics, cryptography), the potential for further vulnerability reduction grows. The gap between Firefox and Chromium in memory-safe code adoption is widening over time, not narrowing.

0.4.2 3.2 Chromium's Memory Safety and Mitigation Architecture

The Rust-focused comparison in Section 3.1 presents an incomplete picture of Chromium's defense-in-depth. While Chromium's codebase remains predominantly C++, the project deploys multiple structural mitigations that reduce memory corruption vulnerability density and exploitability:

V8 Sandbox (sandboxed execution for JIT-compiled code). V8, Chromium's JavaScript engine, implements a memory sandbox within the address space of the renderer process. The V8 sandbox reserves a virtual address region (typically 1 TB on 64-bit systems) and constrains all JIT-compiled code and V8 heap allocations to this region. Pointers between sandboxed and non-sandboxed memory are encoded as offsets relative to the sandbox base, making it structurally impossible for JIT-generated code to directly reference memory outside the sandbox region even in the presence of a JIT corruption vulnerability [34]. This is a distinct mitigation from the kernel-level `isolatedProcess` sandbox and operates at the memory-safety layer.

Oilpan garbage collection for the DOM. Chromium's DOM objects are managed by Oilpan, a tracing garbage collector integrated into the Blink rendering engine. Oilpan eliminates use-after-free vulnerabilities in DOM-manipulation code paths by providing precise reachability tracking and ensuring that object lifetimes are determined by reference reachability rather than manual reference counting [35]. This is architecturally significant because DOM-manipulation code has historically

been a dense source of use-after-free vulnerabilities in browser engines. Firefox's SpiderMonkey uses a combination of reference counting and cycle collection for DOM object management, which provides memory safety guarantees that are overlapping with but not identical to Oilpan's.

Type-checked IPC (Mojo). Chromium's Mojo IPC system enforces type-safe message passing between processes. Interface definitions are compiled into generated C++ code that validates message structure, bounds, and types at the serialization and deserialization boundaries. This prevents a class of memory corruption vulnerabilities at inter-process communication boundaries, precisely the attack surface that sandbox-escape exploits target [36]. Firefox's IPC layer (similar to Chromium's legacy IPC) does not provide equivalent compile-time type safety guarantees across all channels, though specific interfaces benefit from generated bindings.

PartitionAlloc. Chromium's memory allocator provides hardened allocation with features including: dedicated per-process partitions (array/string/buffer allocations isolated from general-purpose allocations), Bucket-based freelist entropy injection (ASLR within the heap), and MiraclePtr (use-after-free quasi-reference-counting via reference stability) [37]. These mitigations raise the exploitation difficulty for heap corruption vulnerabilities even when the underlying bug is present.

Type-based CFI. Chromium ships with Clang's Cross-DSO CFI (Control Flow Integrity) enabled on supported platforms, validating indirect call targets against their declared types at runtime. This defeats type-confusion-based exploitation techniques that are a common vector for gaining code execution from memory corruption primitives [14].

Memory Tagging Extension (MTE). On ARMv9 hardware, Chromium supports MTE-based heap tagging, which probabilistically detects spatial and temporal memory safety violations at the hardware level. This is noted in Section 3.4 as a shared mitigation, but Chromium's implementation is more mature, with per-partition MTE tags integrated into PartitionAlloc [37].

Chromium's memory safety strategy is multi-layered rather than language-dependent. The absence of widespread Rust adoption is partially compensated by a portfolio of mitigation techniques applied across the C++ attack surface. A fair comparison of pre-compromise defenses must account for both Firefox's Rust adoption and Chromium's mitigation portfolio.

Scope and limitations of Rust-based guarantees. The Rust advantage in Firefox is substantial but not absolute, and its scope deserves precise characterization:

1. **JIT-compiled code is outside Rust's safety guarantees.** The SpiderMonkey JavaScript JIT compiler generates and executes dynamic machine code at runtime. This code is not subject to Rust's compile-time memory-safety checks. A vulnerability in the JIT pipeline (for example, incorrect bounds computation during inline caching) can produce memory corruption regardless of the surrounding code's language. JIT engines are a primary source of critical browser vulnerabilities across both Firefox and Chromium [17]. RLBox, Mozilla's WebAssembly-based sandboxing, mitigates this by isolating certain third-party libraries into sandboxed Wasm compartments [5], but this does not extend to JIT-generated code.

2. **Logic bugs are not prevented by memory safety.** A correctly implemented, memory-safe function can still contain logic errors: incorrect state transitions, confused deputy problems, bypassed access control checks, or mishandled edge cases. These vulnerabilities are not addressed by Rust's safety guarantees and require different mitigation strategies (code review, fuzzing, formal verification).

3. **Cryptographic side channels require language-independent defenses.** Constant-time programming, secret-independent memory access patterns, and mitigation of microarchitectural side channels (cache timing, branch prediction) must be enforced at the implementation level regardless

of the host language. Rust's safety guarantees do not address side-channel leakage. Firefox uses NSS (Network Security Services) for cryptography, with growing Rust components, while Chromium uses BoringSSL (a Google fork of OpenSSL). Both libraries require identical care in side-channel-resistant implementation.

4. **The Rust migration is incomplete.** Significant portions of the GeckoView attack surface remain in C++, including core layout and DOM implementation paths. The Rust migration has made progress in strategically important areas (CSS, networking, GPU rendering), but a complete memory-safe browser engine remains a long-term goal.

0.4.3 3.3 AI-Assisted Hardening Across Both Engines

In mid-2025, Mozilla published details of a collaboration with Anthropic's red team that used AI-assisted techniques to systematically audit Firefox for exploitable bugs [10]. This effort identified and fixed latent security issues across the codebase. Mozilla also published analysis of the dual-use nature of AI in security, noting that the same techniques defenders use to find vulnerabilities can be weaponized by attackers at scale [11].

Google applies comparable techniques across Chromium and Android, including AI-guided fuzzing, automated vulnerability discovery via OSS-Fuzz, and ML-based patch analysis. Both teams operate in the same dual-use environment. Mozilla's Anthropic collaboration represents an investment in hardening methodology that both vendors will need to sustain as AI-assisted offensive capabilities mature.

These AI-assisted approaches are structurally complementary to a Rust migration. AI auditing can identify logic bugs, correctness issues, and complex inter-component vulnerabilities in both Rust and C++ code. Rust's compile-time guarantees address a different and largely orthogonal vulnerability class: memory corruption in safe code paths, regardless of auditing quality.

0.4.4 3.4 Implications for the Post-Compromise vs. Pre-Compromise Calculus

The combined effect of Rust migration and AI-assisted hardening shifts the security calculus toward pre-compromise defense. The standard argument for Chromium's sandbox assumes renderer compromise is inevitable and post-compromise containment is critical. But the inevitability of compromise is itself a function of codebase vulnerability density:

$$P(\text{Successful Exploit}) = P(\text{Vulnerability Present}) \times P(\text{Vulnerability Reachable}) \times P(\text{Exploit Successful Given Reachability})$$

By reducing $P(\text{Vulnerability Present})$ through memory-safe language adoption, Firefox reduces the overall exploit probability even before sandboxing is considered. The CVE history of both browsers consistently shows that the majority of critical-severity browser vulnerabilities are memory-safety bugs in C++ code [14]. Mozilla has not published a component-level breakdown of CVEs by language, so a direct Rust-versus-C++ comparison within Firefox cannot be made from public data. However, spot-checking Mozilla security advisories [17] for major Rust components (Stylo, RLBox, WebRender) since their respective shipping dates yields no severity-critical CVEs attributable to memory safety in those components. This observation is consistent with the Rust advantage but should not be treated as a comprehensive audit. Readers can verify this claim by reviewing the same advisories.

This advantage compounds over time. Each new Rust component in Firefox eliminates a vulnerability class from that component permanently. The cumulative effect of Firefox's head start in Rust adoption means the memory-safety gap between the two engines is likely to widen, not shrink, as both codebases evolve.

0.4.5 3.5 Hardware Mitigations and OS-Level Protections

Both Firefox and Chromium benefit from hardware-level exploit mitigations on modern ARM processors. These include Pointer Authentication Codes (PAC) for control-flow integrity, Branch Target Identification (BTI) for indirect branch validation, Memory Tagging Extension (MTE) on ARMv9 hardware for spatial and temporal memory safety, and shadow stacks for return address protection. All major mobile platforms supporting these features apply them to both browser engines equally, as they are enforced at the OS and hardware level, not by the browser vendor.

These mitigations are significant but incomplete. The PACman attack demonstrated that ARM's Pointer Authentication can be bypassed via microarchitectural side channels on M1-series processors, exploiting the limited entropy in unused pointer address bits to forge authenticated pointers without detection [20]. Script-driven JIT engines like SpiderMonkey and V8 can be leveraged to assemble authenticated gadget sequences that defeat PAC at the process level. Hardware mitigations raise the exploitation bar but do not eliminate it, and they do not change the relative assessment between Chromium and GeckoView, as both engines operate on identical hardware.

Post-exploit mitigations (both software-level like `isolatedProcess` and hardware-level like PAC) are important but interdependent. Hardware mitigations make certain classes of sandbox escape more difficult, but they cannot compensate for a structurally higher vulnerability density in the rendering engine itself.

0.4.6 3.6 Empirical Snapshot: Vulnerability Data from Published Sources

This paper does not conduct an independent CVE census (see Scope and limitations, Section 1), but it relies on published aggregate data from the vendors themselves and from independent trackers. The following snapshot contextualizes the architectural arguments in this paper:

Chromium. The Chromium project reports that "around 70% of our serious security bugs are memory safety problems," based on an analysis of 912 high or critical severity security bugs since 2015 affecting the Stable channel [14]. This data is self-published by Google and is the most commonly cited statistic on browser memory safety. Google's Android team reports comparable figures: "memory safety bugs… account for over 60% of high severity security vulnerabilities" on the platform, and the Chrome team's GWP-ASan data confirms the same pattern [22]. Google Project Zero's annual tracking of exploited-in-the-wild 0-days consistently shows that memory corruption (use-after-free, out-of-bounds) constitutes the overwhelming majority of exploitations across all targets, including browsers [23].

Firefox. Mozilla maintains a security advisory page listing all fixed vulnerabilities with severity ratings [17], but does not publish aggregate breakdowns comparable to Chromium's 912-bug analysis. Mozilla has not released a public study classifying its CVE inventory by root cause category (memory safety vs. logic vs. other). This is a gap in the public evidence base. Individual advisory review suggests that Firefox's CVE distribution follows a similar pattern to Chromium's for its C++ components, but this cannot be verified from Mozilla's published data alone. The Rust components shipped in Firefox (Stylo since 2017, RLBox since 2021, WebRender since 2019) have not produced severity-critical memory-safety CVEs in Mozilla's published advisories as of July 2026, which is consistent with the expected benefit of memory-safe language adoption but falls short of a statistically rigorous demonstration.

Summary. The available data supports two conclusions: (a) memory safety bugs dominate the CVE inventories of both engines, and (b) Mozilla's Rust components have a clean track record since shipping, but the sample size and public documentation are insufficient for a quantitative cross-browser comparison. Architectural arguments about Rust's security value (Section 3.1-3.3) should be evaluated in light of this limited empirical basis.

0.5 4. Extension Architecture as Security Infrastructure

GrapheneOS dismisses extension-based security as "privacy theater" and equates content filtering with AntiVirus-style "enumeration of badness" [1]. This characterization mixes up distinct security mechanisms and overlooks structural properties of Firefox's extension architecture.

0.5.1 4.1 Network-Layer Content Interception

The WebExtension content-blocking API allows extensions such as uBlock Origin to parse network requests against declarative filter rules (DNR, Declarative Net Request) and block resources *before* they are fetched or executed. The conceptual flow:

```
[ Network Payload ] -> [ WebExtension Filter ] -> [ Browser Engine ]
                                |
                      (Blocked at network layer)
```

This is structurally distinct from AntiVirus scanning. Key differences:

1. **Timing.** AntiVirus typically scans files after they are written to disk and before execution. WebExtension content blocking intercepts requests at the network layer, before the payload is fetched and before any code execution occurs. The exploit never enters the device's memory.
2. **Attack surface reduction.** Every blocked request reduces the volume of untrusted code processed by the rendering engine. This is a direct reduction in the attack surface exposed to the network, not a detection-after-delivery model.
3. **Determinism.** Declarative filter rules operate on deterministic pattern matching (URL patterns, domain names, resource types), not heuristics or behavioral analysis. There is no false-negative risk from an unrecognized exploit payload. If the payload's delivery infrastructure is blocked, the payload never arrives.

The limitation that GrapheneOS correctly identifies (that filter lists must enumerate known malicious patterns and cannot block novel delivery vectors) is real. But this limitation is not dispositive. Zero-day exploit delivery in practice frequently relies on known malicious infrastructure (command-and-control domains, exploit kit landing pages, compromised ad networks) that filter lists can and do block. The argument that "enumerating badness" is futile assumes that attackers can instantiate novel delivery infrastructure for every target at zero cost. That assumption does not hold for mass-market or spray-and-pray exploitation campaigns.

0.5.2 4.2 Extension-to-Extension Isolation

Firefox's WebExtension architecture enforces strict isolation boundaries between extensions that are more restrictive than Chromium's in several dimensions:

1. **No direct inter-extension communication.** Firefox extensions cannot directly call each other's APIs, access each other's storage, or inspect each other's state. Inter-extension communication is only possible through explicit, user-visible channels (`storage.onChanged` events for same-origin storage, or `runtime.onMessageExternal` with explicit `externally_connectable` manifest declarations).
2. **Storage partition isolation.** Each extension's local storage, IndexedDB, and other persistent storage are cryptographically partitioned by extension ID. Extension A cannot read Extension B's stored data even if both are installed in the same browser profile.
3. **Content script confinement.** Content scripts injected by extensions run in isolated worlds within the page's process. They have no access to the page's JavaScript objects or DOM APIs unless explicitly granted. This prevents a compromised page from leveraging an extension's content script as a privilege escalation vector.

The security implication is that a compromised extension cannot easily pivot to compromise other extensions. A compromised content-blocking extension like uBlock Origin cannot exfiltrate data from a password manager extension's storage. This blast radius containment is a structural security property of the extension architecture.

In Chromium's extension architecture, similar isolation principles apply, but the broader API surface for inter-extension messaging and the availability of native messaging hosts (which can bridge extensions to system-level processes) create a larger lateral-movement surface for a compromised extension [18].

0.5.3 4.3 The Limitations of an Extension-Based Approach

Extension-based defenses have well-documented limitations:

1. **Reactive enumeration.** Filter lists must be maintained and updated. A novel exploit delivery vector that does not match known infrastructure patterns will not be blocked.
2. **Fingerprinting risk.** Custom extensions and configurations increase browser distinctiveness. The marginal fingerprinting cost of a widely-used extension with default settings is lower than GrapheneOS's framing suggests.
3. **Performance overhead.** Content filtering and script blocking consume CPU and memory resources.

Extension-based defenses and kernel-level sandboxing are complementary layers operating at different points in the exploit chain. Dismissing one layer as "privacy theater" overlooks its genuine security value.

0.5.4 4.4 Manifest V3 and Extension API Surface

The transition from Manifest V2 to Manifest V3 in Chromium has security implications that intersect with the content-blocking discussion. Manifest V3 restricts certain extension APIs that content blockers rely on: the `webRequest` blocking API is replaced by the more limited `declarativeNetRequest` API, which imposes caps on dynamic filter rule counts and restricts the timing of rule evaluation. These changes were justified by Google on security and performance grounds, specifically citing the principle of least privilege [21].

Firefox continues to support Manifest V2 extension APIs, including full `webRequest` blocking. This has two relevant effects for this analysis:

1. **More effective content blocking.** uBlock Origin on Firefox can enforce dynamic, user-created filter rules and larger block lists without hitting API-imposed limits. On Chromium-based browsers (including Vanadium), the same extension is restricted to a subset of its filtering capabilities.
2. **Larger extension API surface.** Maintaining support for the V2 API surface means Firefox exposes a broader set of extension capabilities that, if compromised, could be leveraged by a malicious extension. This is a real trade-off.

The net assessment depends on whether one views the extension API surface primarily as attack surface or as defense infrastructure. GrapheneOS's position falls firmly in the former camp [1]. For users who deploy extensions, network-layer content interception as a pre-compromise defense justifies the API surface exposure. For users who do not use extensions, the API surface difference is irrelevant.

0.6 5. Privacy Architecture Overlap with Security Models

Security and privacy intersect at the reconnaissance phase of targeted exploitation. Privacy controls that disrupt device fingerprinting, cross-site tracking, and behavioral profiling directly impede an attacker's ability to identify and target specific individuals.

0.6.1 5.1 Firefox's Structural Privacy Defenses

Firefox on Android deploys multiple structural privacy protections as built-in defaults. Several operate at the network stack or browser engine level, not as extension-based configurations:

Total Cookie Protection (dynamic first-party isolation). Firefox partitions cookies and site storage by the top-level site. This prevents cross-site tracking via cookie synchronization, storage access, and state sharing. A tracker embedded in `site-a.com` cannot read the cookie it set when embedded in `site-b.com`, because the cookie jar is partitioned by the top-level origin. This is implemented at the network stack level and is enabled by default [19].

Enhanced Tracking Protection (ETP). Firefox blocks known tracking resources, fingerprinting scripts, and cryptominers by default using a combination of the Disconnect list and built-in heuristic detection. ETP operates at the network level before resources are loaded or executed.

Multi-Account Containers. Tabs can be assigned to isolated containers, each with separate cookie jars, storage, browsing history, and site state. This provides user-level identity separation (for example, work versus personal browsing) that operates orthogonally to site-level isolation. A tracking script in a "work" container cannot access data from a "personal" container, even if both are open simultaneously.

Anti-fingerprinting protections. Firefox includes fingerprinting-resistant APIs derived from the Tor Browser project, covering canvas fingerprinting, WebGL, audio context fingerprinting, font enumeration, and battery status. These are less extensive than Tor Browser's full fingerprinting resistance, but they provide baseline defense against passive fingerprinting without requiring user configuration.

DNS-over-HTTPS (DoH) with strict mode. Firefox can encrypt DNS queries, preventing DNS-level surveillance, tampering, and redirection. This is a network-level privacy protection that also prevents certain classes of DNS-based tracking.

0.6.2 5.2 Vanadium's Privacy Roadmap

GrapheneOS's Vanadium project has historically prioritized security hardening over privacy features. The stated privacy roadmap includes [1]:

- **Always-incognito mode** (no persistent browsing state)
- **Improved state partitioning** beyond current cookie isolation
- **Network Isolation Keys**, dividing connection pools, caches, and other network state based on site origin

GrapheneOS acknowledges that this work is "currently in a very early stage" and that "at the moment, the only browser with any semblance of privacy is the Tor Browser" [1]. As of July 2026, Vanadium's structural privacy protections remain less mature than what Firefox ships as defaults.

0.6.3 5.3 The Reconnaissance Disruption Argument

The attack chain for a targeted exploitation campaign often begins with data collection:

```
[ Data Brokers / Ad Networks ] -> [ Fingerprinting Profiles ] -> [ Targeted
Phishing ] -> [ Exploit Delivery ]
```

Disrupting the left side of this chain is a legitimate security strategy. An attacker who cannot reliably identify and profile a target cannot deliver a tailored exploit. For users whose primary risk is targeted

exploitation enabled by data-broker profiling (rather than state-sponsored zero-click attacks), privacy controls serve a real security function.

GrapheneOS's position does not dispute this connection but questions the efficacy of client-side anti-fingerprinting: "Most privacy features for browsers are privacy theater without a clear threat model and these features often reduce privacy by aiding fingerprinting and adding more state shared between sites" [1]. This critique has merit for poorly implemented anti-fingerprinting measures, but it does not apply equally to all privacy features. Total Cookie Protection, for example, does not increase fingerprinting surface. It partitions state, which is a strictly additive privacy gain with no fingerprinting cost.

0.7 6. The Systemic Risk of Engine Monoculture

Beyond the architectural comparison between individual engines lies a systemic security question: what are the aggregate security properties of a browser ecosystem dominated by a single engine?

0.7.1 6.1 Market Concentration and Attack Incentives

As of 2025-2026, Chromium-based browsers account for the vast majority of global mobile browser usage. This market concentration creates a structural security risk. A single vulnerability in the Chromium rendering engine, V8 JavaScript engine, or Mojo IPC layer potentially threatens billions of devices simultaneously.

The concept of software monoculture as a systemic vulnerability was articulated by Geer et al. [7], who argued that homogeneity in critical software infrastructure concentrates attack value and reduces the diversity required for ecosystem resilience. In the browser context, a zero-day vulnerability in Chromium's V8 engine simultaneously affects Google Chrome, Microsoft Edge, Samsung Internet, Brave, Opera, and hardened forks like Vanadium, across every platform they run on. This concentration of value creates powerful incentives for exploit developers (both commercial zero-day brokers and advanced persistent threats) to invest in Chromium-specific research.

0.7.2 6.2 Codebase Diversity as Macro-Level Circuit Breaker

GeckoView operates on an entirely independent codebase: the Gecko rendering engine and SpiderMonkey JavaScript engine. An exploitation payload engineered for a Chromium-specific memory-corruption vulnerability or sandbox-escape technique is structurally inert when processed by GeckoView. This independence provides what this paper terms a **macro-level circuit breaker**. Localized environments running GeckoView are automatically insulated from exploit campaigns targeting the Chromium codebase.

The security literature on diversity as a defense mechanism supports this principle. The use of diverse, functionally equivalent implementations reduces the likelihood that a single attack technique compromises all targets [7], [8]. This is an operational security principle employed in critical infrastructure, cryptographic implementations, and defense-in-depth architectures. The browser ecosystem's effective monoculture is an anomaly in security engineering, not a best practice.

0.7.3 6.3 Engine Diversity vs. Aggregate Attack Surface

GrapheneOS objects that GeckoView is not a WebView implementation, so Firefox on Android must be deployed alongside the platform's Chromium-based WebView, creating "the remote attack surface of two separate browser engines instead of only one" [1]. This objection is addressed in detail in the claim-by-claim analysis (Section 7.5). In the context of monoculture risk, the trade-off can be stated concisely: the dual-engine state is an Android platform constraint, not a Firefox deficiency. The marginal attack surface of adding GeckoView must be weighed against the monoculture risk reduction that engine diversity provides.

0.8 7. Critical Examination of GrapheneOS's Claims Against Gecko-Based

Browsers GrapheneOS maintains a well-documented position advising against Gecko-based browsers, rooted in specific architectural and operational concerns [1]. This section examines each of those claims against current evidence (as of July 2026), identifying where they are substantiated, where they have aged, and where they reflect divergent threat-model priorities rather than objective security deficits.

0.8.1 7.1 Claim: "Firefox does not have internal sandboxing on Android" Status: Substantiated, requires clarification of terminology.

Firefox on Android does not implement Android's `isolatedProcess` mechanism for its child processes. This claim is accurate and is not in dispute [1]. The `isolatedProcess` attribute is a declarative manifest flag, and its absence represents a deliberate or resource-constrained decision by Mozilla. The earlier version of this paper categorized this claim as "partially outdated," which overstated the degree to which the architecture had changed. **Correction: the core `isolatedProcess` claim remains fully substantiated.**

The definitional question is whether "internal sandboxing" requires `isolatedProcess` specifically, or whether other forms of process isolation (multi-process architecture, Fission's origin-level process assignment on nightly/developer channels) qualify as sandboxing. This paper takes the broader definition, which GrapheneOS disputes. The reader should evaluate which definition aligns with their threat model:

- If sandboxing is defined as kernel-level UID isolation via `isolatedProcess`, GrapheneOS's claim is correct and has not aged.
- If sandboxing is defined more broadly to include any form of process-level privilege separation, GeckoView provides process isolation without kernel-level UID sandboxing, and the two projects are using different definitions.

The first version of this paper did not make this definitional distinction clearly, which created an impression of dismissing GrapheneOS's accurate claim. The claim is correct within GrapheneOS's definitional framework, and this paper should have stated that explicitly.

0.8.2 7.2 Claim: "Gecko-based browsers like Firefox are much more vulnerable to exploitation" Status: No longer supported by current evidence.

This claim requires separate evaluation for two components: (a) vulnerability density in the codebase, and (b) the difficulty of exploiting residual vulnerabilities given available mitigations.

On component (a), Firefox's industry-leading Rust adoption (Section 3.1) provides compile-time memory-safety guarantees that Chromium's predominantly C++ codebase does not have. Given that roughly 70% of critical-severity browser vulnerabilities are memory-safety bugs [14], a codebase that eliminates these vulnerability classes in critical paths has a structurally lower vulnerability density. Mozilla's 2026 AI-assisted hardening effort [10] has further reduced the residual vulnerability count.

On component (b), the absence of `isolatedProcess` on Android means that a successfully exploited memory corruption vulnerability faces weaker post-exploit containment than on Chromium. This is a real limitation, but its overall risk contribution is attenuated by the lower probability of initial compromise (due to memory safety). The overall exploit probability ($P(\text{Compromise}) \times P(\text{Successful Exploitation Given Compromise})$) is not obviously higher for GeckoView, because the two factors move in opposite directions.

GrapheneOS's framing that Firefox is "much more vulnerable" mixes up a difference in post-compromise architecture with a difference in overall exploit risk. These are distinct metrics, and the available evidence does not support the categorical claim.

0.8.3 7.3 Claim: "Even in the desktop version, Firefox's sandbox is still

substantially weaker (especially on Linux) and lacks full support for isolating sites" **Status: Requires platform-specific assessment; partially outdated.**

This claim reflected an accurate description of the desktop state when it was written [1]. Desktop sandboxing is platform-specific, and the validity of GrapheneOS's claim now varies by operating system.

Windows. Firefox on Windows has reached Content Process Sandbox Level 9 across all release channels as of early 2026 [24]. This is the highest sandbox level defined. Level 9 includes total Win32k system call lockdown (closing a historically major sandbox-escape vector), zero-trust file system access (deny-by-default, with explicit whitelists only for required resources), and third-party DLL load blocking. The GPU process sandbox operates at Level 2, isolating graphics driver access from the OS. Mozilla's sandbox architecture on Windows uses Job objects, restricted access tokens, integrity levels, and Win32k lockdown, the same primitives that Chromium's Windows sandbox uses. The gap on this platform has effectively closed. Users can verify their sandbox level by checking the Content Process Sandbox Level entry in `about:support`.

Linux. Firefox on Linux ships Content Process Sandbox Level 6, which provides seccomp-BPF syscall filtering with default-deny for ioctl, filesystem read/write brokering via a separate broker process, network and socket restrictions, chroot jail, and unprivileged user namespaces when available [24]. The Linux sandbox uses a different mechanism than Windows because the platform provides different primitives (namespace isolation, seccomp-BPF, AppArmor/SELinux). Chromium's Linux sandbox also uses seccomp-BPF and namespaces, but has a more finely restricted syscall policy in some areas. Without an updated side-by-side technical comparison, the claim that Firefox's Linux sandbox is "substantially weaker" cannot be verified or refuted from public documentation.

macOS. Firefox on macOS uses a whitelist-based sandbox policy at Level 3, which denies all system access by default and explicitly permits only required resources (specific file system paths, WindowServer, microphone, named sysctls, IOKit properties) [24]. Write access to the entire file system is blocked, along with inbound/outbound network I/O, exec, fork, printing, and camera access. This is architecturally comparable to Chromium's macOS sandbox, which also uses the macOS Sandbox framework with a deny-by-default policy.

Site isolation (Fission). The claim that Firefox "lacks full support for isolating sites" is clearly outdated. Fission has been shipping on desktop Firefox since version 95 (December 2021) with continuous refinements since [3], [5].

Without an updated, platform-specific technical comparison from GrapheneOS or an independent researcher, the blanket claim that Firefox's desktop sandbox is "substantially weaker" cannot be sustained. On Windows, the evidence suggests parity. On Linux and macOS, the comparison requires more detailed analysis than either party has published.

0.8.4 7.4 Claim: "Achieving browser privacy through piling on extensions

is privacy theater" **Status: A legitimate philosophical disagreement with important nuance.**

GrapheneOS argues that "most privacy features for browsers are privacy theater without a clear threat model" and that "every change you make results in you standing out from the crowd" [1]. This position contains internal tensions that require examination:

1. **The AntiVirus analogy is inapt.** GrapheneOS equates content-filtering extensions with AntiVirus software, arguing that both involve "enumerating badness." The timing and mechanism differ. AntiVirus scans files on disk after delivery. WebExtension content blocking intercepts requests at the network layer *before* any payload reaches the rendering engine.
2. **Fingerprinting distinctiveness is a continuous, not binary, property.** GrapheneOS's claim that any extension-based change increases fingerprinting surface is theoretically correct, but the practical fingerprinting cost of deploying widely-used extensions with default settings is lower than the framing suggests. A user running Firefox with uBlock Origin's default filter lists, Total Cookie Protection, and ETP is not uniquely fingerprintable. They belong to a substantial population of similarly configured users.
3. **Content blocking reduces attack surface directly, independent of privacy.** Blocking known exploit-delivery domains at the network layer reduces the volume of untrusted code reaching the rendering engine. This operates as a security measure regardless of the browser's anti-fingerprinting quality.

The core critique (that enumeration-based approaches cannot prevent novel zero-day delivery vectors) remains valid. But this is a limitation shared by all detection-based security mechanisms, including the `isolatedProcess` sandbox (which cannot prevent a novel renderer exploit, only contain it after exploitation). The two approaches are complementary. Content blocking reduces the volume of exploit attempts reaching the renderer, while sandboxing contains those that succeed.

0.8.5 7.5 Claim: "Firefox on Android must be deployed alongside

Chromium-based WebView, creating dual-engine attack surface" **Status: Describes a platform constraint, not a Firefox deficiency.**

This claim is factually correct as a description of Android's current platform architecture. Its framing as a Firefox security deficiency is logically flawed:

- The dual-engine state is enforced by Android's platform design, which mandates a system WebView independent of the browser. It is not a design choice by Mozilla.
- On GrapheneOS, the WebView is Vanadium (Chromium-based) regardless of the user's browser choice. This is a design decision by GrapheneOS.
- The dual-engine concern would be eliminated if (a) Android's platform architecture permitted a user-selectable WebView engine, or (b) GrapheneOS substituted a GeckoView-based WebView for Vanadium.
- The dual-engine state is symmetric in an important sense. A Vanadium-only user still has two copies of the Chromium engine (browser + WebView), which is not flagged as a security concern.

Does adding GeckoView increase aggregate attack surface beyond what the platform already requires? Yes, unavoidably. But this incremental increase must be weighed against the monoculture risk reduction and architectural diversity that GeckoView provides (Section 6). A user who values diversity as a defense-in-depth measure may rationally accept the incremental attack surface of a second engine in exchange for the structural protection against Chromium-specific zero-day campaigns.

0.8.6 7.6 Summary Assessment

The following table summarizes the status of each major GrapheneOS claim:

Claim	Assessment	Rationale
"No `isolatedProcess` sandboxing"	**Substantiated**	GeckoView does not use `isolatedProcess` [1]

"No internal sandboxing on Android"	**Partially outdated**	Fission shipped Jan 2026, rolled back; active nightly/dev only [27]-[30]; multi-process architecture exists
"Much more vulnerable to exploitation"	**Not supported by current evidence**	Rust memory safety, AI hardening, Fission landing; post-compromise vs. pre-compromise conflation
"Desktop sandbox substantially weaker"	**Platform-dependent; Windows at parity, Linux/macOS unverified**	Windows Level 9 [24], [25]; Linux Level 6; Fission shipped 2021; no current platform-specific comparison from GrapheneOS
"Extension privacy is theater"	**Philosophical disagreement**	Content blocking provides genuine pre-delivery interception; anti-AV analogy is structurally inapt
"Dual-engine attack surface"	**Platform constraint, not Firefox deficiency**	Enforced by Android; symmetric concern; design choice by GrapheneOS

0.9 8. Conclusion and Threat Model Matrix

The security properties of mobile browser engines cannot be reduced to a single metric or a categorical "secure vs. insecure" classification. Each architecture makes tradeoffs that align with different threat-model priorities. The categorical assertion that Gecko-based browsers are non-viable on Android is not supported by current evidence. But the counter-assertion that GeckoView has achieved parity with Chromium's sandboxing on mobile is also not supported.

The following matrix summarizes the evaluated properties across the full threat landscape:

Threat Profile Vector	Chromium Architecture (Vanadium)	GeckoView Architecture (Firefox)
Primary Mitigation Philosophy	Post-compromise kernel containment	Pre-compromise memory safety + content interception
Sandboxing Mechanism	Kernel-level (`isolatedProcess`) UID isolation	Application-level process spawning + Fission origin isolation
Site Isolation (Mobile)	Strict site isolation with kernel enforcement [1]	Fission shipped Firefox 147 (Jan 2026), rolled back in 147.0.2; disabled on release/beta channels as of Firefox 152; active on nightly and developer channels only [27]-[30]; kernel mechanism differs from desktop [12]

Post-Exploit Containment	Strong (UID isolation, CFI, SSP, seccomp-BPF on desktop)	Limited by absence of `isolatedProcess` [1]
Memory Safety	Predominantly C++ (V8, Blink); Rust experimental [14], [16]	Extensive Rust adoption (Stylo, RLBox, Necko, WebRender); C++ legacy paths remain
Memory Safety Trajectory	Rust adoption nascent; gap with Firefox widening	Continued porting of subsystems; compounding vulnerability reduction over time
Pre-Delivery Interception	Optional via extensions; built-in content filtering available	Built-in ETP + WebExtension content blocking (uBlock Origin)
Extension Isolation	Broader inter-extension messaging; native messaging hosts [18]	Stricter isolation; no direct inter-extension access
Monoculture Risk Exposure	High (primary target of global exploit pipelines)	Low (immune to Chromium-specific exploits)
Privacy Architecture (Shipped)	Basic; privacy roadmap in early stages [1]	Total Cookie Protection, ETP, Containers, anti-fingerprinting, DoH [19]
Privacy Architecture (Maturity)	Roadmap only; "very early stage" [1]	Mature defaults; enabled by default
Dual-Engine Requirement	Single engine (WebView + browser both Chromium)	Dual engine (GeckoView + Chromium WebView required by platform)

0.9.1 8.1 Threat Model Alignment

- **For users facing targeted, state-sponsored zero-click exploitation** where kernel-level containment after compromise is critical, the Chromium/Vanadium architecture (with its `isolatedProcess` sandboxing, site isolation, and CFI) provides objectively stronger post-exploit defenses. This remains the strongest use case for a Chromium-based browser on Android.
- **For users facing mass surveillance, corporate data mining, and widespread exploit campaigns targeting the Chromium monoculture**, GeckoView's independence offers structural protections that Chromium derivatives cannot provide, even with hardening. The Rust-based memory safety and network-layer content interception provide pre-compromise defenses that operate regardless of the kernel sandbox quality.
- **For privacy-conscious users concerned with reconnaissance-phase disruption**, Firefox's structural privacy protections (Total Cookie Protection, ETP, Containers) are materially more mature than Vanadium's privacy roadmap, which remains in early development [1].
- **For security-conscious users on GrapheneOS specifically**, the OS-level hardening applies to all applications. GeckoView does not benefit from `isolatedProcess` regardless of the host OS, and the dual-engine deployment increases aggregate attack surface. Users in this category should evaluate whether the monoculture risk reduction and memory safety benefits of GeckoView outweigh the weaker post-exploit containment.

0.10 Methodology Note

This analysis is based on publicly available documentation accessed in July 2026. Primary sources include GrapheneOS's official usage guide and features documentation [1], [2]; Mozilla's architecture documentation and engineering blog posts [3], [4], [5], [10]; peer-reviewed security literature [6], [7], [8]; Android platform documentation [9]; community release announcements [12]; and technical community discussion [13].

Verified claims are those confirmed by at least one primary source. **Uncertain claims** are noted explicitly. The most significant remaining gap is the absence of detailed technical documentation from Mozilla on how Fission's kernel-level isolation mechanisms on Android differ from the desktop implementation. Mozilla's engineering team has not published an architecture document comparable to Chromium's Site Isolation paper [6].

This paper does not assess the iOS versions of either browser, as iOS WebKit requirements fundamentally alter the sandboxing landscape. It also does not provide a comprehensive assessment of desktop sandboxing, except where desktop architectures are referenced for comparison.

Scope classification. This analysis is primarily an architectural threat-modeling comparison, not an empirical vulnerability census. Where aggregate CVE data is cited (Section 3.5), it is drawn from vendor-published statistics and independent trackers, with limitations explicitly noted. Readers who need a direct quantitative comparison of CVE counts between Chromium and Firefox should consult the respective vendor advisory pages [14], [17]. The authors have made no attempt to produce an independent CVE inventory, as the methodological challenges (differential disclosure practices, severity rating inconsistencies, and the absence of Mozilla-published root-cause classifications) would produce results of limited reliability.

0.11 References

[1] GrapheneOS, "Usage: Web Browsing." [Online]. Available: https://grapheneos.org/usage#web-browsing. Accessed: Jul. 2026.

[2] GrapheneOS, "Features." [Online]. Available: https://grapheneos.org/features. Accessed: Jul. 2026.

[3] A. Gakhokidze, "Introducing Firefox's new Site Isolation Security Architecture," Mozilla Hacks, May 2021. [Online]. Available: https://hacks.mozilla.org/2021/05/introducing-firefox-new-site-isolation-security-architecture/

[4] R. Jesup, "Process Isolation Architecture in the Gecko Rendering Engine," Mozilla Wiki. [Online]. Available: https://mozilla.github.io/firefox-browser-architecture/text/0012-process-isolation-in-firefox.html

[5] Mozilla Wiki, "Project Fission." [Online]. Available: https://wiki.mozilla.org/Project_Fission

[6] C. Reis, G. Moteva, and S. Gribble, "Site Isolation: Process separation for web sites within the browser," in *Proc. 28th USENIX Security Symp.*, Santa Clara, CA, USA, 2019, pp. 1461-1478.

[7] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. P. Pfleeger, J. S. Quarterman, and B. Schneier, "CyberInsecurity: The Cost of Monopoly," Computer & Communications Industry Association (CCIA), 2003.

[8] K. Thompson, "Reflections on Trusting Trust," *Commun. ACM*, vol. 27, no. 8, pp. 761-763, Aug. 1984.

[9] Android Open Source Project, "Isolated Process," Android Developers. [Online]. Available: https://developer.android.com/guide/topics/manifest/service-element#isolatedProcess

[10] Mozilla, "Hardening Firefox Together with Anthropic's Red Team," Mozilla Blog, 2025. [Online]. Available: https://blog.mozilla.org/en/firefox/hardening-firefox-anthropic-red-team/

[11] Mozilla, “AI Security and Zero-Day Vulnerabilities,” Mozilla Blog, 2025. [Online]. Available: https://blog.mozilla.org/en/privacy-security/ai-security-zero-day-vulnerabilities/

[12] Mozilla, “Firefox for Android 147.0 Release Notes,” Jan. 2026. (Current as of Firefox 152, Jul. 2026). [Online]. Available: https://www.mozilla.org/en-US/firefox/android/147.0/releasenotes/

[13] PrivacyGuides Community, “Site Isolation (Fission) now appears to be active in Firefox on Android,” PrivacyGuides Discourse, Jan. 2026. [Online]. Available: https://discuss.privacyguides.net/t/site-isolation-fission-now-appears-to-be-active-in-firefox-on-android/34899

[14] Chromium Project, “Memory Safety.” [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/ . Accessed: Jul. 2026.

[15] M. Miller, “Trends and Challenges in the Vulnerability Mitigation Landscape,” USENIX Enigma 2019. [Online]. Available: https://www.youtube.com/watch?v=fxR7qEa0hVI

[16] Chromium Project, “Rust in Chromium.” [Online]. Available: https://security.googleblog.com/2023/01/supporting-use-of-rust-in-chromium.html . Accessed: Jul. 2026.

[17] Mozilla Security Blog, “Security Advisories,” various years. [Online]. Available: https://www.mozilla.org/en-US/security/advisories/ . Accessed: Jul. 2026.

[18] Chrome Developer Documentation, “Native Messaging.” [Online]. Available: https://developer.chrome.com/docs/extensions/mv3/nativeMessaging/ . Accessed: Jul. 2026.

[19] Mozilla, “Total Cookie Protection,” Mozilla Security Blog. [Online]. Available: https://blog.mozilla.org/security/2021/08/10/firefox-91-introduces-enhanced-cookie-protection/ . Accessed: Jul. 2026.

[20] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, “PACMAN: Attacking ARM Pointer Authentication with Speculative Execution,” in *Proc. 49th ACM/IEEE Int. Symp. Computer Architecture (ISCA)*, New York, NY, USA, 2022, pp. 685-698. [Online]. Available: https://pacmanattack.com/

[21] Google, “Overview of Manifest V3,” Chrome Developers. [Online]. Available: https://developer.chrome.com/docs/extensions/mv3/intro/ . Accessed: Jul. 2026.

[22] Google, “Memory Safety,” Android Open Source Project. [Online]. Available: https://source.android.com/docs/security/test/memory-safety . Accessed: Jul. 2026.

[23] Google Project Zero, “0-days In-the-Wild.” [Online]. Available: https://googleprojectzero.github.io/0days-in-the-wild/ . Accessed: Jul. 2026.

[24] Mozilla Wiki, “Security/Sandbox,” Oct. 2024. [Online]. Available: https://wiki.mozilla.org/Security/Sandbox . Accessed: Jul. 2026.

[25] r/firefox, “Firefox Sandbox Isolation Hits Level 9, The Gap with Chrome Has Closed,” Reddit, Jan. 2026. [Online]. Available: https://old.reddit.com/r/firefox/comments/1qkqfcx/firefox_sandbox_isolation_hits_level_9_the_gap/ . Accessed: Jul. 2026.

[26] Mozilla, “Firefox for Android 152.0 Release Notes,” Jul. 2026. [Online]. Available: https://www.mozilla.org/en-US/firefox/android/152.0/releasenotes/

[27] Bug 2011886 - Switch off isolated processes by default. Mozilla Bugzilla. https://bugzilla.mozilla.org/show_bug.cgi?id=2011886

[28] Bug 2011319 - Firefox for Android frequently and randomly going back to the previous page. Mozilla Bugzilla. https://bugzilla.mozilla.org/show_bug.cgi?id=2011319

[29] Bug 2012435 - Content process crashes when isolating a site with Fission. Mozilla Bugzilla. https://bugzilla.mozilla.org/show_bug.cgi?id=2012435

[30] Bug 2003658 - Make Fission + SHIP default in 147. Mozilla Bugzilla. https://bugzilla.mozilla.org/show_bug.cgi?id=2003658

[31] Chrome Developers, “Isolated Web Apps.” [Online]. Available: https://developer.chrome.com/docs/extensions/reference/manifest/isolated-app/ . Accessed: Jul. 2026.

[32] Bug 2034168 - Restrict extensions from reading local file data without specific permission. Mozilla Bugzilla. https://bugzilla.mozilla.org/show_bug.cgi?id=2034168

[33] Chromium Security Architecture. https://chromium.googlesource.com/chromium/src/+/master/docs/security/overview.md. Accessed: Jul. 2026.

[34] V8 Sandbox Design Document. https://docs.google.com/document/d/1FM4fQmIhEqPG8uGp5o9A-mnPB5BOeScZYpkHjo0KKA8.

[35] Oilpan Design Document. Chromium. https://chromium.googlesource.com/chromium/src/+/main/third_party/blink/renderer/platform/heap/BlinkGCDesign.md

[36] Mojo IPC Documentation. Chromium. https://chromium.googlesource.com/chromium/src/+/main/mojo/README.md

[37] PartitionAlloc Design Document. Chromium. https://chromium.googlesource.com/chromium/src/+/main/base/allocator/partition_allocator/PA_README.md