Firefox Render Relay (F2R2): High-Fidelity Remote Browsing for Privacy and Security

Patryk Tomlak
University of California, San Diego

Kyle Trinh
University of California, San Diego

Abstract

Modern web browsers expose a large surface for user tracking through browser fingerprinting. While remote browsing can mitigate this, existing solutions based on video streaming suffer from high bandwidth usage and compression artifacts, limiting usability. We propose Firefox Render Relay (F2R2), an innovative method that streams high-fidelity graphics by relaying the internal draw commands from Firefox's WebRender engine. The primary obstacle to this approach was the significant "bitrot" of Mozilla's internal tools for capturing and replaying these commands in (`wr-sequence` and `wrench`). This paper details our progress in ressurecting these tools. Our key contributions are a series of fixes that restore persistent text and dynamic image rendering during sequence playback. We present a working prototype that demonstrates a protoype of (near) perfect rendering fidelity, and discuss the feasibility of F2R2 as a solution for a private and secure remote browsing experience.

1 Introduction

In recent times we've observed a drastic growth of an industry that is using web surveillance and targeted advertising for financial gain. Many internet users feel an increasing craving for sturdy privacy-preserving solutions which tackle the problem by desensitizing the user's browsing session from any identifiable information. A major threat to user privacy is browser fingerprinting, a technique where remote websites collect a wide range of attributes from a user's browser, such as installed fonts, screen resolution, and operating system version. We were inspired by Electronic Frontier Foundation's Cover Your Tracks project [3], which shows that running heavily customized privacy setups often counterintuitively leads to users having an even more unique fingerprint. And, the combination of these attributes can still create a fingerprint that is statistically unique, or has very small overlap with other sessions. This makes users vulnerbale to being identified and tracked across the web, even without traditional tracking mechanisms like cookies [2].

As the issue grew in intensity, we also saw a rise of a wide range of solutions have been developed to mitigate this form of tracking. Client-side solutions, such as the Tor Browser [10] or Brave Browser [1], attempt to create a uniform fingerprint for all users by modifying browser attributes (e.g. stripping identifiable bits from the client to blend in with the most common fingerprints). However, this often comes at the cost of usability, especially in case of Tor Browser which encourages users to use a common window size or disable JavaScript. An alternative approach is remote browsing, where the browser is executed on a remote server and the user interacts with it via a thin client, meaning no local environment details or IP addresses are leaked. However most if not all of the current projects rely on video streaming protocols that transfer browser's graphical output [4] as if we used traditional screen capture / remote connection protocols. This method, while effective at isolation, introduces significant drawbacks, including constant high bandwidth consumption (even when the content is static), noticeable latency, and compression artifacts that degrade user's visual experience and defeat the appeal of such approach.

In this paper, we propose Firefox Render Relay (F2R2), an alternative attempt at remote browsing that targets the goal of high-fidelity rendering without many drawbacks of standard video streaming. We find it likely that beyond privacy, detaching the client from the execution logic and containerizing the browser in an isolated environment makes F2R2 mitigate possible 0-day security exploits, which target the host operating system, that could affect standard browsing environments. The main idea behind our implementation is to intercept the exact rendering state generated by Firefox's internal WebRender engine on the server and relay them to a thin client, which then executes these commands to reconstruct the browser's graphical output. This approach promises perfect fidelity, avoiding compression artifacts and lowering bandwidth usage, very similar in principle to Windows RDP (Remote Desktop Protocol) [6] which heavily inspired our solution. However, while RDP operates at the operating system level by intercepting generic display driver instructions

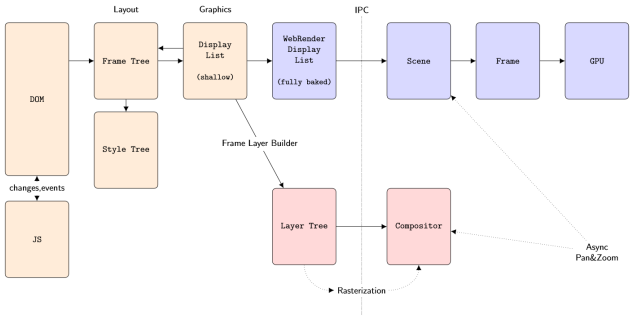

Figure 1: A diagram of the Firefox rendering pipeline [9]. F2R2 intercepts the display list before it is fuly processed by WebRender's hardware pipeline on the server.

and windowing bitmaps [7], F2R2 operates within the application layer by streaming semantic web-rendering primitives. Because it bypasses the OS entirely, and hooks directly into the browser's graphics pipeline, building F2R2 required using advanced internal tooling to capture the browser state. Thankfully, we could avoid a full from-scratch implementation by using Mozilla's internal tools for capturing and replaying WebRender's internal pipeline resources and scenes, namely `wr-sequence` and `wrench`. Beginning our work, we unfortunately discovered that these tools had fallen into a state of disrepair, making the sequence capture and playback feature completely non-functional. The project's primary challenge turned into to resurrecting these tools as a cornerstone on which any further functionality will depend on.

For our contributions we can roughly divide them in three steps. First we analyze and explain the architectural decay of the Firefox's WebRender capture and replay utility. Then we describe a detailed account of the reverse engineering effort and series of patches that allowed as to restore the majority of sequence rendering capabilities within *wrench*, focusing on the challenges of achieving persistent text and dynamic image loading across frames. Finally, we describe the initial results from our prototype that proves the high-fidelity rendering approach of F2R2 is conceptually sound, which paves the way for future work on a complete, privacy-focused remote browsing system.

2 Background and Related Work

To understand the architecture of F2R2, first we need to understand the fundamentals of modern browser rendering. As illustrated, the process is divided into distinct threads and data structures. The core rendering work is offloaded to compositor, which manages a hierarchical frame tree and generates a display list. This display list is then processed by WebRender, which uses GPU-accelerated rasterization to efficiently bake the scene into a frame (see Figure 1).

At the core of F2R2 is WebRender, Firefox's Rust-based GPU-accelerated rendering engine, whose internal architecture and execution is more similar to a 3D game engine than a traditional CPU-based renderer [9]. WebRender takes the display list as input and uses it to build a scene, which is then processed by the GPU to produce the final rendered frame. This process is highly optimized for performance and smoothness.

WebRender's development produced many tools ofr debugging and testing purposes, including tools to capture and replay rendering sessions. `wr-capture` and `wr-sequence` are commands that can be invoked within a browser console of unmodified Firefox to serialize the WebRender display list and all associated resources (like fonts and images) to disk. The `wrench` tool is a standalone application that can then load these captured dumps and replay the rendering steps, allowing WebRender developers to debug any issues without needing to run the full browser. After discovering the existence of these tools it was the sequence capture and viewer that we intended to repurpose for F2R2.

Our work is related to two main types of applications: privacy-preserving browsers and remote desktop technologies. In the first area, our work is an alternative to client-side solutions like Tor Browser [10], which can degrade usability, and pixel-streaming remote browsers like Neko [4], which suffer from high bandwidth and compression artifacts. In the second area, our work can be seen as a new type of remote desktop protocol targeted towards web content. Unlike generic protocols like VNC that streams raw pixels, or Remote DOM protocols [11] that serialize DOM updates but still require the local client to execute the rendering stack (potentially leaking environment capabilities), F2R2 streams high-level semantic drawing commands, which has the potential for greater efficiency and unaffected visual fidelity, while potentially matching the security improvements from privacy focused browsers without many of their annoyances.

3 The Bitrot Obstacle

When we wanted to integrate our solution with by immediatelly using `wrench` and `wr-sequence` for F2R2 we encountered a significant obstacle: the tools were effectively non-functional due to bitrot. These internal debugging tools, while once incredibly useful during initial development of WebRender and the process of integrating it in the mainline Firefox years ago, slowly had fallen into a state of disrepair. While `wr-capture` is still used and integrated in the CI-CD pipeline,`wr-sequence` was probably abandoned. Our initial attempts to replay a captured sequence resulted in immediate crashes and suppressing them still resulted in malformed, unusable output of vague shapes and colors. This confirms the Firefox developers statement, who said that the sequence capture functionality was likely unused for a long time and probably broken.

The core technical problem was an aggressive and destructive cache-clearing approach within `wrench`. For every new frame in a captured sequence, the tool would check if the `resource_sequence_id` had changed. If it had, `wrench` would assume that all the resources we loaded so far were invalid, triggering a complete wipe of the resource cache, active documents, and GPU textures. This means that by default all fonts and images are available to be rendered for the first frame, but then are immediately deleted and unavailable for all of the following frames that might still depend on them, resulting in a playback that consisted of mostly colored rectangles and text block outlines.

Unfortunately, that was not the only issue. There was a very sparse, if not to say lack of documentation for these internal tools, forcing us to reverse-engineer their behavior from the source code as the only reliable way to decipher what each of the `wrench` flags does (as well as all usable keyboard shortcuts). Setting up a consistent build environment also proved difficult, as containerized builds lacked the necessary GPU access for WebRender, forcing us to move to a `distrobox`-based solution [5] which turned out to work suprisingly well, with only minor issues of matching the CUDA driver version. Finally, because some of our later fixes required changes to the capture logic within Firefox itself, we could no longer use pre-compiled artifact builds. This sent us on a full, multi-hour recompilation of the entire Firefox browser on each machine, with subsequent compiles being faster yet still very hard to debug, which significantly slowed us down the development and debugging cycle.

4 F2R2: Implementation and Resurrection

Getting past the problems of bitrot required an analytical, robust approach of reverse-engineering, and targeted patches to the WebRender and `wrench` codebases. What we did can be broken down into three main parts: unlocking basic stability, restoring persistent text rendering, and restoring dynamic image rendering.

4.1 Basic Stability

The first priority was to make `wrench` open a sequence viewing window, even if empty. In its initial state, it would panic and crash upon encountering any error, such as a missing frame file in a sequence. This was particularly problematic for live captures, where Firefox might be in the process of writing frame to disk when `wrench` tried to read it. We suppressed most errors and modified the frame loading logic in `wrench` to handle these errors without crashing, suppressing panics and providing fallback debug information for now:

```
// webrender/src/render_backend.rs
// Suppress crashes:
let scene = config.deserialize_for_scene::<Scene, _>(&scene_name)
-   .expect(&format!("Unable to open {}.ron", scene_name));
+   .unwrap_or_else(Scene::new);
```

```
// webrender/src/renderer/mod.rs
// Stop the renderer-backend disagreement panic:
- assert!(old.is_none(), "Renderer and backend disagree!");
+ if old.is_some() { warn!("Renderer and backend disagree!"); }
```

This allowed us to finally process and analyze incomplete or live-written sequences without encountering immediate fatal errors, allowing us to further diagnose visual state of the playback.

4.2 Restoring Persistent Text

The most critical issue was fixing the disappearing text. The fix required creating a new incremental resource loading strategy that replaced the aggresive cache clearing for subsequent frames in a sequence that was currently implemented. We modified the `load_capture` function in `render_backend.rs` to distinguis between the first frame of a sequence and all subsequent frames.

If we encounter a new frame again we trigger a full wipe, otherwise preserve the resources. For all other frames, we also needed to implement a smart loading path mechanism. In this path, we still update the font and image templates, but avoid clearing the GPU glyph and texture caches. This helps us get the rasterized fonts from the first frame to remain in memory and stay available for all following frames. The simplified change to the `load_capture` logic turned out to be:

```
// webrender/src/render_backend.rs
let never_loaded =
    self.loaded_resource_sequence_id == 0;
let resource_changed = !first_load &&
    backend.resource_sequence_id !=
    self.loaded_resource_sequence_id;

if first_load || never_loaded {
    // Original path: clear caches and reload
    // ...
    self.active_documents.clear();
    self.resource_cache.load_capture_full(
        plain_resources, &config
    );
} else if resource_changed {
    // New path: preserve caches, load incrementally
    // ...
    config.resource_id = backend.resource_sequence_id;
    self.loaded_resource_sequence_id =
        backend.resource_sequence_id;
    let plain_externals = self.resource_cache
        .load_capture_incremental(
            plain_resources, &config
        );
    self.result_tx.send(ResultMsg::DebugOutput(
        DebugOutput::LoadCaptureIncremental(
            config.clone(), plain_externals
        )
    )).unwrap();
}
```

This change was the key to restoring persistent text rendering and making sequence playback cohesive, at least when it comes to text. (The images were still missing on subsequent frames).

4.3 Restoring Dynamic Image Rendering

With text rendering done, the next challenge was to fix the missing images. This was a two-part problem. First, we discovered that the `EXTERNAL_RESOURCES` capture bit was not being set for sequence captures at all (likely disabled for optimization reason), causing WebRender to skip writing image data to disk entirely! Enabling this flag was simple but opened many possible routes for future fixes, as we finally could retrieve more image data.

The second, more complex problem was that, like fonts, dynamically loaded images (e.g., those that appear when scrolling) were not being handled correctly. Our incremental loading logic for fonts was not sufficient for images. We had to implement a more robust incremental loading strategy for the entire resource cache. This involved detecting when the resource directory changed (meaning that new images had been captured) and triggering a new incremental load path that would add the new image templates to the cache while evicting any stale image data, all without clearing the existing caches. The logic added to the render backend to detect this change abstracts to:

```
// webrender/src/resource_cache.rs
pub fn load_capture_incremental(
    &mut self,
    resources: PlainResources, ...) {
    //...
    // Preserve existing font templates
    for (key, template) in resources.font_templates {
        self.font_templates.insert(key, template);
    }

    // Incremental Image Loading (simplified code)
    for (key, image) in resources.images {
        // Drop from texture cache if actually stale
        if let Some(mut cached) =
            self.cached_images.resources.remove(&key) {
             cached.drop_from_cache(&mut self.texture_cache);
        }
        self.cached_images.resources.insert(key, image);
    }
}
```

5 Evaluation

The current, tangible, effect of our work is mainly a successful demonstration of a working prototype. By fixing `wrench` and `wr-sequence` to a reasonable degree, we have validated the main architectural requirement of F2R2.

5.1 Rendering Fidelity

The most important result of our work is that the rendering fidelity of the replayed session is perfect. The output of the `wrench` client is a pixel-for-pixel identical reproduction of the original Firefox browsing session. This is the key advantage of the F2R2 approach over traditional video-streaming remote browsers like Neko [4]. There are no compression artifacts, no color bleeding, and no "smearing" during scrolling. The user experience, from a purely visual standpoint, is indistinguishable from local browsing.

5.2 Performance

With overarching `wr-sequence` issues at hand unfortunately we could not focus on performance and prioritized correctness instead. The current prototype is not a real-time streaming system - it reads captured data from disk. Therefore, we cannot meaningfully evaluate latency nor bandwidth savings.

Currently, the captured dumps grow in size rapidly as the user navigates the internet within the Firefox capture session, growing in the order of gigabytes per $\approx$ 10 minutes of browsing. However, we can already see that the vast majority of the frame resources across the adjacent scenes is almost identical. This logically suggest that even with minor optimizations we would be able to minimize the required bandwidth drastically, likely to at least match the order of magnitude required for a video streaming approach of acceptable quality.

Therefore, for the sake of introducing possible bugs before fixing required sequence logic, the `wr-sequence` data is left uncompressed. A future implementation of F2R2 would employ a delta-compression algorithm on the draw call stream, which we expect would significantly reduce the bandwidth requirements, especially for mostly static pages. Furthermore, we want to explore an approach that allows for client-side dynamic scaling of the drawn content, meaning a high-resolution client could be driven by a low-resolution server with minimal extra bandwidth, something impossible with pixel streaming.

5.3 Privacy

A full evaluation of the privacy guarantees of F2R2, for example by using the Electronic Frontier Foundation's Cover Your Tracks tool, is also a future work. It would require the implementation of the full network relay server and client. The focus of the current work, despite optimistic initial plans, turned into proving the feasibility of the underlying rendering and capture technology.

6 Discussion: A Critical Analysis of Trade-offs

The F2R2 architecture prioritizes high-fidelity rendering and session portability. However, achieving these benefits requires significant compromises. Evaluating F2R2 requires staying impartial and critically evaluating how it intertwines traditional security and privacy traits with rendering performance.

6.1 Privacy: Trusting the Server

F2R2 is not an anonymity network. Its privacy model relies entirely on trusting the server operator. This is a far weaker guarantee than Tor's onion routing, which provides cryptographic protection from the relays themselves. The server has

unencrypted access to the user's entire session, including passwords and history. F2R2 successfully hides the user's unique hardware from websites by blending them into a shared fingerprint, but it offers zero privacy from the relay provider, likely requiring the user to host this solution themselves.

Compared to a locally-run, heavily hardened Firefox instance (using security-focused 'about:config' flags and privacy extensions), F2R2 may even offer less protection against sophisticated trackers, as it relies solely on a single, shared fingerprint. F2R2 trades the strong, cryptographic anonymity of other systems for usability and rendering fidelity. The only case where it becomes a strong security measure is during wide adoption, in effect diluting the unique fingerprint (or rotating set of fingerprints).

6.2 Security: Protecting the Hardware, Not the Data

F2R2 isolates the user's physical machine from web-based malware, but it does not protect the browsing data. If a zero-day exploit compromises the remote Firefox container, the attacker gains full access to that session's cookies and sensitive information. Completed solution would trap the threat at the isolated remote server resource, protecting the user's local hardware, but the active session data still remains vulnerable.

A secure production F2R2 server must improve upon a default Firefox installation. This includes (1) running a hardened Firefox instance with aggressive security configurations, (2) implementing process-level sandboxing beyond containers using Mandatory Access Control systems like SELinux or AppArmor to strictly limit the browser's capabilities, and (3) enforcing strict session volatility to ensure no data persists between uses, or an alternative solution for persistent data.

6.3 Isolating Execution vs Syncing Data

F2R2 differs conceptually from features like Firefox Sync [8]. Sync tools seamlessly replicate tabs, history, and passwords across devices but still execute the browser locally, exposing the user's physical hardware, operating system, and screen resolution to tracking scripts. F2R2 offloads the execution entirely. It provides session continuity from any device without bringing the physical execution context onto the client, ensuring websites only ever see a generic hardware footprint of the remote server.

F2R2 fulfills a different objective: rendering fidelity and session portability without the compromise of bringing the physical execution context onto the client. The user maintains untraceable continuity via the relay network since the visual state relies only on the external server's homogeneous footprint. Additionally, this approach obfuscates any network changes the user might leak otherwise through their IP address.

6.4 Semantic Streaming

Perhaps the most novel and promising application of F2R2 is in remote work. Traditional remote browsers stream a video feed of the screen. F2R2 streams the underlying drawing instructions. This (in theory) reduces bandwidth for text-heavy pages and allows the client device to render the page at its native resolution. For example, a user on a 4K monitor receives perfectly scaled, crisp text without the blurriness or compression artifacts inherent to video streaming. This makes F2R2 a superior architecture for remote access to text-heavy or data-dense web applications, even if it cannot match the low-latency performance sometimes required for highly interactive content.

7 Future Work

The resurrection of `wr-sequence` is a good foundation for F2R2, but several core components are required to build a fully deployable and secure system.

Containerization and Advanced Anonymity The immediate next step is standardizing the F2R2 server deployment. By packaging the relay server in a standardized container, we can guarantee a completely uniform browser fingerprint (e.g., identical fonts, OS environment, and WebGL footprint) for all users. Beyond this baseline, future work could explore integrating Tor-like privacy features. Possibly routing the relay connection through the Tor network and applying the Tor Browser's aggressive anti-fingerprinting patches directly to the server container. This would harden the system against advanced traffic analysis, all while the user connects via a lightweight, native client.

At the same time, it is also entirely possible that this fingerprint may be too specific, so that someone could identify a user as an F2R2 user. Further mitigations could include modifications to the unified container, which would happen during runtime, so that even if someone were to fingerprint the F2R2 container, they would lose continuity as a different fingerprint would appear elsewhere or the current one get shuffled from a pool. (This also could be used to avoid fingerprint-based restrictions/bans).

Delta Compression and Bandwidth Optimization To make F2R2 highly efficient, we must implement delta compression. Instead of transmitting the entire display list and all resources every frame, the server should only compute and send the drawing commands that have changed. Pairing this with a persistent client-side cache for fonts, images, and layout templates will drastically reduce bandwidth and latency. A key open research question is whether streaming these compressed, high-level drawing primitives can empirically outperform the bandwidth efficiency of highly optimized video codecs like H.264 for interactive web browsing.

Dynamic Scaling and Upstreaming Because F2R2 streams underlying drawing instructions rather than fixed pixels, clients can theoretically re-render the web page at any window size or native resolution. This means a lightweight server could drive a client on a 4K monitor with fractional scaling without the blurring or artifacting penalties of stretching a video feed. Future work will focus on implementing and testing this dynamic, client-side scaling. Finally, we plan to package our fixes for `wr-sequence` and `wrench` into a formal pull request to the main Mozilla Firefox repository, restoring this debugging functionality for future browser development.

8 Conclusion

This paper introduced F2R2, a remote browsing architecture that streams internal WebRender drawing commands instead of traditional video. Building this system required repairing Mozilla's abandoned `wr-sequence` and `wrench` tools to correctly handle the persistent rendering of text and dynamic images.

Our working prototype proves that this semantic streaming approach is technically feasible and delivers pixel-perfect rendering fidelity. While relying on a remote execution environment introduces strict privacy and security trade-offs, F2R2 offers distinct advantages over existing remote desktop protocols. By transmitting rendering instructions rather than raw pixels, it enables significant bandwidth efficiency and allows the client device to scale the web page natively. Ultimately, resurrecting these tools provides the necessary engineering foundation to develop a new class of high-performance, visually lossless remote browsers.

References

[1] Brave Software. Brave browser. https://brave.com/.

[2] Peter Eckersley. How unique is your web browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, PETS '10, 2010.

[3] Electronic Frontier Foundation. Cover your tracks. https://coveryourtracks.eff.org/.

[4] m1k1o. Neko. https://github.com/m1k1o/neko.

[5] Luca Di Maio. Distrobox. https://github.com/89luca89/distrobox. Accessed: March 21, 2026.

[6] Microsoft. Remote desktop protocol (rdp). https://learn.microsoft.com/en-us/troubleshoot/windows-server/remote/understanding-remote-desktop-protocol.

[7] Microsoft Remote Desktop Services Team. Top 10 rdp protocol misconceptions - part 1. https://techcommunity.microsoft.com/t5/enterprise-mobility-security/top-10-rdp-protocol-misconceptions-part-1/ba-p/246707, 2018. Accessed: March 21, 2026.

[8] Mozilla. How do i set up sync on my computer? https://support.mozilla.org/en-US/kb/how-do-i-set-sync-my-computer.

[9] Mozilla Gfx Team. Firefox rendering overview. https://firefox-source-docs.mozilla.org/gfx/RenderingOverview.html.

[10] The Tor Project. Tor browser. https://www.torproject.org/.

[11] Wikipedia. Remote browser isolation (history). https://en.wikipedia.org/wiki/Remote_browser.