Retrofitting Fine Grain Isolation in the Firefox Renderer

Shravan Narayan and Craig Disselkoen, *UC San Diego;* Tal Garfinkel, *Stanford University;* Nathan Froyd and Eric Rahm, *Mozilla;* Sorin Lerner, *UC San Diego;* Hovav Shacham, *UT Austin;* Deian Stefan, *UC San Diego*

Retrofitting Fine Grain Isolation in the Firefox Renderer

Shravan Narayan[†]   Craig Disselkoen[†]   Tal Garfinkel[*]   Nathan Froyd[◇]   Eric Rahm[◇]
Sorin Lerner[†]   Hovav Shacham[⋆,†]   Deian Stefan[†]

[†]UC San Diego   [*]Stanford   [◇]Mozilla   [⋆]UT Austin

Abstract

Firefox and other major browsers rely on dozens of third-party libraries to render audio, video, images, and other content. These libraries are a frequent source of vulnerabilities. To mitigate this threat, we are migrating Firefox to an architecture that isolates these libraries in lightweight sandboxes, dramatically reducing the impact of a compromise.

Retrofitting isolation can be labor-intensive, very prone to security bugs, and requires critical attention to performance. To help, we developed RLBox, a framework that minimizes the burden of converting Firefox to securely and efficiently use untrusted code. To enable this, RLBox employs static information flow enforcement, and lightweight dynamic checks, expressed directly in the C++ type system.

RLBox supports efficient sandboxing through either software-based-fault isolation or multi-core process isolation. Performance overheads are modest and transient, and have only minor impact on page latency. We demonstrate this by sandboxing performance-sensitive image decoding libraries (libjpeg and libpng), video decoding libraries (libtheora and libvpx), the libvorbis audio decoding library, and the zlib decompression library.

RLBox, using a WebAssembly sandbox, has been integrated into production Firefox to sandbox the libGraphite font shaping library.

1   Introduction

All major browsers today employ coarse grain *privilege separation* to limit the impact of vulnerabilities. To wit, they run *renderers*—the portion of the browser that handles untrusted user content from HTML parsing, to JavaScript execution, to image decoding and rendering—in separate sandboxed processes [3, 33, 40]. This stops web attackers that manage to compromise the renderer from abusing local OS resources to, say, install malware.

Unfortunately, this is no longer enough: nearly everything we care about today is done through a website. By compromising the renderer, an attacker gets total control of the current site and, often, any other sites the browser has credentials for [14]. With services like Dropbox and Google Drive, privilege separation is insufficient even to protect local files that sync with the cloud [24].

Browser vendors spend a huge amount of engineering effort trying to find renderer vulnerabilities in their own code [28]. Unfortunately, many remain—frequently in the

dozens of third-party libraries used by the renderer to decode audio, images, fonts, and other content. For example, an out-of-bounds write in libvorbis was used to exploit Firefox at Pwn2Own 2018 [7]. Both Chrome and Firefox were vulnerable to an integer-overflow bug in the libvpx video decoding library [10]. Both also rely on the Skia graphics library, which had four remote code execution bugs until recently [12, 37].

To appreciate the impact of these vulnerabilities and the difficulty of mitigating them, consider a typical web user, Alice, that uses Gmail to read email in her browser. Suppose an intruder, Trudy, sends Alice an email that contains a link to her malicious site, hosted on `sites.google.com`. If Alice clicks on the link, her browser will navigate her to Trudy's site, which can embed an `.ogg` audio track or `.webm` video to exploit vulnerabilities in libvorbis and libvpx and compromise the renderer of Alice's browser. Trudy now has total control of Alice's Gmail account. Trudy can read and send emails as Alice, for example, to respond to password reset requests from other sites Alice belongs to. In most cases, Trudy can also attack cross site [14], i.e., she can access any other site that Alice is logged into (e.g., Alice's `amazon.com` account).

Recent version of Chrome (and upcoming versions of Firefox) support *Site Isolation* [41], which isolates different *sites* from each other (e.g., `*.google.com` from `*.amazon.com`) to prevent such cross-site attacks. Unfortunately, Trudy might still be able to access `{drive,pay,cloud}.google.com`, which manage Alice's files, online payments, and cloud infrastructure—since the renderer that loads the malicious `.ogg` and `.webm` content might still be running in the same process as those origins.

For many sites, Trudy might not even need to upload malicious content to the (trusted) victim origin (`sites.google.com` in our example). Most web applications load content, including images, fonts, and video, from different origins. Of the Alexa top 500 websites, for example, over 93% of the sites load at least one such cross-origin resource (§7.1). And the libraries handling such content are not isolated from the embedding origin, even with Site Isolation [41].

To mitigate these vulnerabilities, we need to harden the renderer itself. To this end, we extend the Firefox renderer to isolate third party libraries in fine grain sandboxes. Using this, we can prevent a compromised library from gaining control of the current origin or any other origin in the browser.

Making this practical poses three significant challenges

---

across three dimensions. First, *engineering effort*—we need to minimize the upfront work required to change the renderer to use sandboxing, especially as this is multiplied across dozens of libraries; minimizing changes to libraries is also important as this can significantly increase the burden of tracking upstream changes. Second, *security*—the renderer was not built to protect itself from libraries; thus, we have to sanitize all data and regulate control flow between the library and renderer to prevent libraries from breaking out of the sandbox. In our experience, bugs at the library-renderer boundary are not only easy to overlook, but can nullify any sandboxing effort—and other developers, not just us, must be able to securely sandbox new libraries. Finally, *efficiency*—the renderer is performance critical, so adding user-visible latency is not acceptable.

To help us address these challenges, we develop a framework called RLBox that makes data- and control-flow at the library-renderer interface explicit, using types. Unlike prior approaches to sandbox automation that rely on extensive custom analysis frameworks (§8), RLBox is simply a library[1] that leverages the C++ type system and is easy to incorporate into Firefox's predominantly C++ codebase.

Using type information, RLBox can identify where security checks are needed, automatically insert dynamic checks when possible, and force compiler errors for any security checks that require additional user intervention. Our type-driven approach enables a systematic way to migrate Firefox's renderer to use sandboxed libraries and allows RLBox to support secure and efficient sharing of data structures between the renderer and library (e.g., by making shared memory operations safe and by lazily copying data out of the sandbox).

To enable efficient sandboxing, we adapt and evaluate two isolation mechanisms for library sandboxing: software-based fault isolation (SFI) leveraging Google's Native Client (NaCl) [49, 65] and a multi-core process-based approach. We also explore applying sandboxing at different granularities (e.g., per-origin and per-library sandboxing) to find the appropriate balance between security and sandboxing overhead.

To evaluate RLBox, we sandbox several libraries in Firefox: the libjpeg and libpng image decoding libraries, the libvpx and libtheora video decoding libraries, the libvorbis audio decoding library, and the zlib decompression library. Browsing a representative sample of both popular and unpopular websites (§7), we find the end-to-end memory overhead of RLBox to be modest—25% with SFI, 18% with process isolation—and transient, appearing only at content load time. The impact on page latency is small: 3% and 13% with SFI and process isolation, respectively. Our sandboxing does not noticeably impact the video frame rates nor audio decoding bitrate.

Our evaluation shows that retrofitting fine grain isolation, especially using SFI, is practical—and we've been integrating

RLBox into production Firefox [51].[2] Since NaCl has been deprecated [17] in favor of WebAssembly (Wasm) [20], our production sandbox also uses Wasm. We used RLBox with this Wasm-based sandbox to isolate the libGraphite font shaping library and are in the process of migrating several others [15, 51]. We describe this effort in Section 9.

Though we developed RLBox to sandbox libraries in Firefox, RLBox is a general library-sandboxing framework that can be used outside Firefox. To demonstrate this, we use RLBox to sandbox libraries in two different contexts: the Apache web server and Node.js runtime. For Apache, we sandbox the libmarkdown library that is used in the mod_markdown module [31]; we find that RLBox with the SFI sandbox increases the tail latency of the Apache server by 10% (4ms) and decreases the throughput by 27% (256 requests/second). For Node.js, we sandbox the C bcrypt library that is used by the JavaScript bcrypt module [38]; we measure RLBox with SFI to impose an overhead of 27% on hashing throughput.

Contributions. We present the case for sandboxing third party libraries in the browser renderer, and potential architectural trade-offs, including our approach (§2). We offer a taxonomy of security pitfalls encountered while migrating the Firefox code base to this architecture that were largely overlooked by previous work (§3), and RLBox, a framework we developed to prevent these pitfalls that leverages the C++ type system to enforce safe data and control flow (§4), and enables an incremental compiler-driven approach to migrating code to a sandboxed architecture (§5). We describe our implementation, including our software fault isolation and multi-core process-based isolation mechanisms (§6), and evaluate the performance of RLBox (§7). We close with a discussion of related work (§8) and our effort upstreaming RLBox into production Firefox (§9).

Availability. All work presented in this paper, including our modified Firefox builds, the RLBox library, and benchmarks are available and open source.[3]

2 Fine grain sandboxing: how and why

Renderers rely on dozens of third-party libraries to support media decoding and other tasks (e.g., decompression, which sites use to optimize page load times and bandwidth consumption). These are written almost exclusively in C and tasked with parsing a wide range of complex inputs. Unsurprisingly, exploitable vulnerabilities in this code are relatively frequent, even after years of scrutiny.

These libraries are a compelling place to employ sandboxing inside the renderer for several reasons. First, media content such as images and video are rich attack vectors, as web applications allow them to be shared pervasively. Over 93% of the Alexa Top 500 websites load such content cross-origin (§7.1). And nearly all forms of social media and peer-to-peer

---

[1] Our only external tooling is a ~100LOC Clang plugin, described in Section 6.1, that makes up for C++'s currently limited support for reflection on structs.

[2] The Tor team is integrating our patches into the Tor Browser [16].

[3] Available at: https://usenix2020-aec.rlbox.dev.

messaging platforms enable the sharing of images and video.

Next, content libraries can be effectively sandboxed, as they require little privilege to operate, i.e., once these libraries are memory isolated, the harm they can inflict is minimal. For example, an attacker that compromises an image decoding library could at worst change how images display. In contrast, sandboxing a highly privileged component like the JavaScript engine is largely ineffectual. An attacker with control over the JavaScript engine can run arbitrary JavaScript code and thus already has complete control of the web application.

Finally, the existing library-renderer interface provides a natural place to partition code. Compared to coarse grain techniques like privilege separation or Site Isolation, which spin up entire new renderer processes, spinning up a sandbox for a library is very cheap (§7). Moreover, because library sandboxes are only needed during content decoding, their memory overhead is transient.

Isolation strategies. A key question remains: *what grain of isolation should be employed?* In particular, different architectures have different implications for performance and security. Prior to RLBox, Firefox was largely exploring a coarse grain approach to library sandboxing, placing certain media libraries into a single sandboxed media process [33]. This approach has some benefits for performance as there is only one sandbox, but trades off security.

First, the assurance of the sandbox is reduced to that of the weakest library. This is less than ideal, especially when we consider the long tail of infrequently used media libraries required to preserve web compatibility (e.g., Theora) which often contain bugs. Next, the attacker gains the power of the most capable library. Some libraries handle *active content*—zlib, for example, is used to decompress HTTP requests that could contain HTML or JavaScript—as opposed to *passive content* such as images or fonts. Thus compromising a passive library like libvorbis, could still enable powerful attacks—e.g., modify the JavaScript decompressed by zlib. When multiple renderers share a common library sandbox, an intruder can attack across tabs, browsing profiles, or sites. Finally, coarse grain sandboxing does not scale to highly performance-critical libraries, such as libjpeg and libpng (§7.5.2).

RLBox lets us employ more granular sandboxing policies that can address these shortcomings. Its flexibility lets us explore the performance implications of various sandboxing architectures with different isolation mechanisms (§7.4).

In this paper, we largely employ a unique sandbox per `<renderer, library, content-origin, content-type>`. This mitigates many of the problems noted above, while still offering modest memory overheads. Per-renderer sandboxing prevents attacks across tabs and browsing profiles. Per-library ensures that a weakness in one library does not impact any other library. Per-content-origin sandboxing prevents cross origin (and thus cross site) attacks on content. For example, a compromise on `sites.google.com` as discussed in our example in Section 1, should not impact content from `pay.google.com`. Per-content-type sandboxing addresses the problem of passive content influencing active content.

Both finer and coarser grain policies are practically useful, though. In production Firefox, for example, we create a fresh sandbox for each Graphite font instance (§9). But, we also foresee libraries where, say, same-origin is sufficient.

Attacker model. We assume a web attacker that serves malicious (but passive) content—from an origin they control or by uploading the content to a trusted origin—which leads to code execution (e.g., via a memory safety vulnerability) in a RLBox sandbox. RLBox ensures that such an attacker can only affect (corrupt the rendering of and potentially leak) content of the same type, from the same origin. Per-object (or per-instance) sandboxing can further reduce the damage of such attacks. We, however, only use this policy when sandboxing audio, videos, and font shaping—we found the overheads of doing this for images to be prohibitive.

We consider side channels out-of-scope, orthogonal challenges. With side channels, an attacker doesn't need to exploit renderer vulnerabilities to learn cross-origin information, as browsers like Firefox largely do not prevent cross-origin leaks via side channels. FuzzyFox [27] and cross-origin read blocking [41] are promising ways to tackle these channels.

For the same reason, we consider transient execution attacks (e.g., Spectre [26]) out of scope. We believe that our SFI and our process-based isolation mechanisms make many of these attacks harder to carry out—e.g., by limiting transient reads and control flow to the sandbox memory and code, respectively—much like Site Isolation [41]. But, in general, this is not enough: an attacker could potentially exploit code in the renderer to transiently leak sensitive data. We leave the design of a Spectre-robust sandbox to future work.

The protections offered by RLBox are only valid if the Firefox code that interfaces with the sandboxed library code is retrofitted to account for untrusted code running in the library sandbox. As we discuss next, this is usually notoriously difficult to get right. RLBox precisely reduces this burden to writing a series of validation functions. In our attacker model, we assume these functions to be correct.

3 Pitfalls of retrofitting protection

The Firefox renderer was written assuming libraries are trusted. To benefit from sandboxing requires changing our threat model to assume libraries are untrusted, and modify the renderer-library interface accordingly (e.g, to sanitize untrusted inputs).

While migrating to this model we made numerous mistakes—overlooking attack vectors and discovering many bugs only after building RLBox to help detect them. We present a brief taxonomy of these mistakes, with examples drawn from the code snippet illustrating the interface between the renderer's JPEG decoder and libjpeg[4] shown in

---

[4]We use libjpeg interchangeably with libjpeg-turbo, the faster fork of the

Figure 1. We discuss how RLBox helps prevent these attacks in Section 4.

For the rest of this section, we assume that libjpeg is fully sandboxed or memory isolated, i.e., libjpeg code is restricted from accessing arbitrary memory locations in the renderer process, and may only access memory explicitly dedicated to the sandbox—the *sandbox memory*. The renderer itself can access any memory location including sandbox memory.

3.1 Insecure data flow

Failing to sanitize data. Failing to sanitize data received from libjpeg including function return values, callback parameters, and data read from sandbox shared memory can leave the renderer vulnerable to attack. For example, if the renderer uses the `num_bytes` parameter to the `skip_input_data()` callback on line 25 of Figure 1 without bounds checking it, an attacker-controlled libjpeg could could force it to overflow or underflow the `src->next_input_byte` buffer.

Pointer data is particularly prone to attack, either when pointers are used directly (with C++'s `*` and `->` operators) or indirectly (via memory functions such as `memcpy()` and `memmove()`, array indexing operations, etc.). For example, if the parameter `jd` of `fill_input_buffer()` is not sanitized (line 35), the read of `jd->src` on line 36 becomes an arbitrary-read gadget. In the same callback, if both `jd` and `src` are unsanitized, the write to `src->next_input_byte` on line 39 becomes an arbitrary-write gadget. Similar attacks using the `memmove()` on line 50 are possible.

Missing pointer swizzles. Some sandboxing mechanisms—e.g., NaCl (§6.2) and Wasm (§9)—use alternate pointer representations. Some sandboxing tools e.g., NaCl (§6.2) and Wasm (§9) use alternate pointer representations for efficiency.

Though this is often done for performance, in Wasm's case this is more fundamental: Wasm pointers are 32-bit whereas Firefox pointers are 64-bit. We must translate or *swizzle* pointers to and from these alternate representations when data is transferred between the renderer and the sandboxed libjpeg.

We found that doing this manually is both tedious and extremely error prone. This is largely because pointers can be buried many levels deep in nested data structures. Overlooking a swizzle either breaks things outright, or worse, silently introduces vulnerabilities. For instance, failing to swizzle the nested pointer `mInfo.err` on line 44 prior to dereferencing, can result in a write gadget (whose write-range depends on the precise pointer representation).

Leaking pointers. Leaking pointers from the Firefox renderer to the sandboxed libjpeg can allow an attacker to derandomize ASLR [50] or otherwise learn locations of code pointers (e.g., C++ virtual tables). Together with an arbitrary-write gadget, this can allow an attacker-controlled libjpeg to execute arbitrary code in the renderer.

original libjpeg library which is used in Firefox.

```
1  // InitInternal() registers callbacks
   ↪   for libjpeg to call while decoding an image
2  nsresult nsJPEGDecoder::InitInternal() {
3    ...
4    mInfo.client_data = (void*)this;
5    ...
6    //Callbacks invoked by libjpeg
7    mErr.pub.error_exit = my_error_exit;
8    mSourceMgr.fill_input_buffer = fill_input_buffer;
9    mSourceMgr.skip_input_data = skip_input_data;
10   ...
11 }
12
13 // Adjust output buffers for decoded pixels
14 void nsJPEGDecoder::OutputScanlines(...) {
15   ...
16   while (mInfo.output_scanline < mInfo.output_height) {
17     ...
18     imageRow = ... +
       ↪   (mInfo.output_scanline * mInfo.output_width);
19     ...
20   }
21   ...
22 }
23
24 // Invoked if some input bytes are not needed
25 void skip_input_data (..., long num_bytes) {
26   ...
27   if (num_bytes > (long)src->bytes_in_buffer) {
28     ...
29   } else {
30     src->next_input_byte += num_bytes;
31   }
32 }
33
34 // Invoked repeatedly to get input as it arrives
35 void fill_input_buffer (j_decompress_ptr jd) {
36   struct jpeg_source_mgr* src = jd->src;
37   nsJPEGDecoder* decoder = jd->client_data;
38   ...
39   src->next_input_byte = new_buffer;
40   ...
41   if (/* buffer is too small */) {
42     JOCTET* buf = (JOCTET*) realloc(...);
43     if (!buf) {
44       decoder->mInfo.err->msg_code = JERR_OUT_OF_MEMORY;
45       ...
46     }
47     ...
48   }
49   ...
50   memmove(decoder->mBackBuffer + decoder->mBackBufferLen,
       ↪   src->next_input_byte, src->bytes_in_buffer);
51   ...
52 }
53
54 // Invoked on a decoding error
55 void my_error_exit (j_common_ptr cinfo) {
56   decoder_error_mgr* err = cinfo->err;
57   ...
58   longjmp(err->setjmp_buffer, error_code);
59 }
```

Figure 1: *The renderer-library interface:* this code snippet illustrates the renderer's interface to the JPEG decoder and is used as a running example. The decoder uses libjpeg's streaming interface to decode images one pixel-row at a time, as they are received. Receiving and decoding concurrently is critical for responsiveness.

In our example, the renderer saves pointers to `nsJPEGDecoder` objects in libjpeg **struct**s (line 4), which alone allows an attacker to locate code pointers—the `nsJPEGDecoder` class is derived from the `Decoder` class, which defines virtual methods and thus has a virtual table pointer as the first field. Even initializing callbacks (line 8) could leak pointers to functions and reveal the location of Firefox's code segment[5].

Double fetch bugs. RLBox uses shared memory (§4) to efficiently marshal objects between the renderer and the sandboxed libraries. This, unfortunately, introduces the possibility of double fetch bugs [47, 62, 64].

Consider the `mInfo` object used in Figure 1. Since this object is used by both libjpeg and the renderer, RLBox stores it in shared memory. Now consider the bounds check of `mInfo.output_scanline` on line 16 prior to the assignment of output buffer `imageRow`. In a concurrent libjpeg sandbox thread, an attacker can modify `mInfo.output_scanline` after the check (line 16), and before the value is fetched (again) and used on line 18. This would bypasses the bounds check, leading to an arbitrary-write gadget. While this example is obvious, double-fetch bugs often span function boundaries and are much harder to spot.

3.2 Insecure control flow

Isolation prevents arbitrary control transfers from the sandbox into the renderer. Thus, out of necessity, callbacks for libjpeg must be explicitly exposed. But this alone is not sufficient to prevent attacks.

Corrupted callback state. Callbacks may save state in the sandboxed library. An attacker-controlled libjpeg can abuse the control flow of callbacks by corrupting this state. For example, on line 4 of Figure 1, the renderer stores a pointer to the `nsJPEGDecoder` object into the `client_data` field of `mInfo`. Inside `fill_input_buffer()` this pointer is used to access the `nsJPEGDecoder` object (line 37). Failing to sanitize `client_data` before using it allows an attacker to set the pointer to a maliciously crafted object and hijack control flow when Firefox invokes a virtual method on this object.

Unexpected callback invocation. Security bugs can also occur if an attacker controlled libjpeg invokes a permitted callback at unexpected times. Consider the `my_error_exit()` callback function, which uses `longjmp()` to implement error handling. On line 58, `longjmp()` changes the instruction pointer of the renderer based on information stored in `setjmp_buffer`. If an attacker invokes `my_error_exit()` before `setjmp_buffer` is initialized, they can (again) hijack the renderer control flow.

Callback state exchange attacks. Threading introduces another vector to attack callback state. When Firefox

---

[5]Whether callback locations are leaked depends on the underlying sandboxing mechanism. While both our process isolation and NaCl use jump tables and thus do not leak, other sandbox implementations could leak such information.

decodes two images in parallel, two decoder threads make calls to libjpeg. Firefox expects libjpeg to invoke the `fill_input_buffer()` callback on each thread with the corresponding `nsJPEGDecoder` object. But, an attacker could supply the same `nsJPEGDecoder` object to both threads when calling `fill_input_buffer()`. If the first thread reallocates the source buffer (line 42), while the second thread is using it to get input bytes, this can induce a data race and use-after-free vulnerability in turn.

4 RLBox: automating secure sandboxing

Modifying the Firefox JPEG decoder to guard against all the attacks discussed in Section 3 requires substantial code additions and modifications. Doing this manually is extremely error prone. Moreover, it also makes the code exceedingly fragile: anyone making subsequent changes to the decoder must now have an intimate knowledge of all the necessary checks to have any hope of getting it right. Multiply this by the number of libraries Firefox supports and number of developers working on the renderer, and the problem becomes intractable.

We built the RLBox framework to tackle these challenges. RLBox helps developers migrate and maintain code in the Firefox renderer to safely use sandboxed libraries. We designed RLBox with the following goals in mind:

1. *Automate security checks:* Adding security checks un-assisted is labor intensive and error prone, as discussed (§3). However, most of the sandboxing pitfalls can be detected and prevented through static checks, i.e., through compile-time errors indicating where code needs to be manually changed for security, or eliminated with dynamic checks and sanitizations (e.g., pointer swizzling and bounds checking).

2. *No library changes:* We want to avoid making changes to libraries. When libraries come from third parties we do not necessarily understand their internals, nor do we want to. Additionally, any changes we make to libraries increases the effort required to track upstream changes.

3. *Simplify migration:* Firefox uses dozens of libraries, with occasional additions or replacements. Consequently, we want to minimize the per-library effort of using RLBox, and minimize changes to the Firefox renderer source.

In the rest of the section we give an overview of the RLBox framework and describe how RLBox addresses the pitfalls of Section 3 while preserving these goals.

4.1 RLBox overview

RLBox makes data and control flow at the renderer-sandbox interface explicit through its type system and APIs in order to mediate these flows and enforce security checks across the trust boundary.

RLBox mediates *data flow* with `tainted` types that impose a simple static information flow control (IFC) discipline [42].

This ensures that sandbox data is validated before any potentially unsafe use. It also prevents pointer leaks into the sandbox that could break ASLR.

RLBox mediates *control flow* through a combination of `tainted` types and API design. Tainting, for example, allows RLBox to prevent branching on `tainted` values that have not been validated. API design, on the other hand, is used to restrict control transfers between the renderer and sandbox. For instance, the renderer must use `sandbox_invoke()` to invoke functions in the sandbox; any callback into the renderer by the sandbox must first be registered by the renderer using the `sandbox_callback(callback_fn)` API.

Mediating control and data flow allows RLBox to:

▶ **Automate security checks:** Swizzling operations, performing checks that ensure sandbox-supplied pointers point to sandbox memory, and identifying locations *where* tainted data must be validated is done automatically.

▶ **Minimize renderer changes:** `tainted` data validation is enforced only when necessary. Thus, benign operations such as adding or assigning `tainted` values, or writing to tainted pointers (which are checked to ensure they point into the sandbox at creation time) are allowed by RLBox's type system. This eliminates needless changes to the renderer, while still ensuring safety.

▶ **Efficiently share data structures:** Static checks ensure that shared data is allocated in sandbox memory and accessed via `tainted` types. Data structures received by the renderer sandbox are marshaled lazily; this allows code to access a single field in a shared **struct** without serializing a big object graph. Finally, RLBox provides helper APIs to mitigate double fetches [47, 62] when accessing this data.

▶ **Assist with code migration:** Compile-time type and interface errors (§5) guide the developer through the process of migrating a library into a sandbox. Each compile error points to the next required code change—e.g., data that needs to be validated before use, or control transfer code that needs to be changed to use RLBox APIs.

▶ **Bridge machine models:** Sandboxing mechanisms can have a different machine model from the application (e.g., both Native Client and WebAssembly use 32-bit pointers and 32-bit **long**s regardless of the platform); by intercepting all data and control flow we can also automatically translate between the application and sandbox machine models—and for Wasm we do this (§9).

In the rest of this section, we discuss how tainting is used to mediate data flow (§4.2) and control flow (§4.4). We then describe how the renderer can validate and untaint data (§4.3). We detail our implementation as a C++ library later (§6.1).

4.2 Data flow safety

All data originating from a sandbox begins life tainted. Tainting is automatically applied by wrapping data with the `tainted<T>` constructor. Tainting does not change the memory layout of a value, only its type. Once applied, though, tainting cannot be removed. The only way to remove a taint is through explicit validation (§4.3).

In general, RLBox propagates taint following a standard IFC discipline. For example, we propagate taint to any data derived from `tainted` values such as data accessed through a `tainted` pointer, or arithmetic operations when one or more operands are `tainted`. We detail how RLBox implements `tainted` types and taint tracking in the C++ type system in Section 6.1. In the rest of this section we show how RLBox uses tainting to ensure data flow safety.

Data flow into the renderer. To protect the renderer from malicious inputs, all data flows from the sandbox into the renderer are `tainted`. Data primarily flows out of the sandbox through two interfaces. First, `sandbox_invoke()`, the only way to call into the sandbox, taints its return value. Second, the use of `sandbox_callback()`, which permits callbacks into the renderer from the sandbox, statically forces the parameters of callbacks to be `tainted`. Any code failing to follow either of these rules would cause a compilation error.

As an example, consider the JPEG decoder code that calls libjpeg's `jpeg_read_header()` to parse headers shown below:

```
jpeg_decompress_struct mInfo;
int status = jpeg_read_header(&mInfo, TRUE);
```

With RLBox, the second line must be modified to use `sandbox_invoke()`, and `status` must be declared as `tainted`[6]:

```
tainted<int> status = sandbox_invoke(mRLBox,
↪   jpeg_read_header, &mInfo, TRUE);
```

In addition to the invoke and callback interfaces, data can flow into the renderer via pointers to sandboxed memory. RLBox, however, forces both these pointers and any data derived from them (e.g., via pointer arithmetic or pointer dereferencing) to be `tainted`—and, as we discuss shortly, using `tainted` pointers in the renderer is always safe.

Data flow into the sandbox. RLBox requires data flowing into sandbox from the renderer to either have a simple numeric type or a `tainted` type. Untainted pointers, i.e., pointers into renderer memory, are not permitted. This restriction enforces a code correctness requirement—sandboxed code only gets pointers it can access, i.e., pointers into sandbox memory—and, moreover, preserves the renderer's ASLR: any accidental pointer leaks are eliminated by construction.

Compile-time errors are used to guide the code changes necessary to use a sandboxed library. To demonstrate this, we continue with the example of JPEG header parsing shown above. To start, note that the `TRUE` parameter to `jpeg_read_header()` can remain unchanged as it has a simple numeric type (in C++). On the other hand, the parameter **&**mInfo points to a **struct** in renderer memory, which libjpeg cannot access; RLBox thus raises a compile-time error.

---

[6] In this paper, we use full type names such as `tainted<int>` for clarity. In practice, we use C++'s **auto** keyword to make code less verbose.

To address this compilation error, RLBox requires such shared data structures to be allocated in sandbox memory using `sandbox_malloc()`:

```
tainted<jpeg_decompress_struct*> p_mInfo
  ↪  = sandbox_malloc<jpeg_decompress_struct>(mRLBox);
tainted<int> status = sandbox_invoke(mRLBox,
  ↪  jpeg_read_header, p_mInfo, TRUE);
```

Placing shared data structures in sandboxed memory in this way simplifies data marshaling of pointer parameters during function calls—RLBox simply marshals pointers as numeric types, it does not eagerly copy objects. Indeed, this design allows RLBox to automatically generate the marshaling code without any user annotations or pointer bounds information (as required by most RPC-based sandboxing tools). Moreover, RLBox can do all of this without compromising the renderer's safety—renderer code can only access shared sandbox memory via `tainted` pointers.

While RLBox does not allow passing untainted pointers into libjpeg, pointers to callback functions, such as `fill_input_buffer()`, need to be shared with the sandbox—these can be shared either as function call parameters to libjpeg functions, return values from callbacks, or by directly writing to sandbox memory. RLBox permits this without exposing the raw callback pointers to libjpeg, through a level of indirection: *trampoline functions*. Specifically, RLBox automatically replaces each Firefox callback passed to libjpeg with a pointer to a trampoline function and tracks the mapping between the two. When the trampoline function is invoked, RLBox invokes the appropriate Firefox callback on libjpeg's behalf.

Benefits of tainted pointers. By distinguishing pointers to renderer memory from pointers to sandbox memory at the type level with `tainted`, RLBox can automatically enforce several important security requirements and checks. First, RLBox does not permit Firefox to pass untainted pointers to libjpeg. Second, RLBox automatically swizzles and unswizzles pointers appropriately when pointers cross the renderer-library boundary, including pointers in deeply nested data structures. (We give a more detailed treatment of pointer swizzling in Appendix A of [35].) Third, RLBox automatically applies pointer-bounds sanitization checks when `tainted` pointers are created to ensure they always point to sandboxed memory. Together, these properties ensure that we preserve the renderer's ASLR—any accidental pointer leaks are eliminated by construction—and that the renderer cannot be compromised by unsanitized pointers—all `tainted` pointers point to sandbox memory.

4.3 Data validation

RLBox disallows computations (e.g., branching) on `tainted` data that could affect the renderer control and data flow. The Firefox renderer, however, sometimes needs to use data produced by library code. To this end, RLBox allows developers to unwrap `tainted` values, i.e., convert a `tainted<T>` to an untainted `T`, using validation methods. A validation method takes a closure (C++ lambda) that unwraps the tainted type by performing necessary safety checks and returning the untainted result. Unfortunately, it is still up to the user to get these checks correct; RLBox just makes this task easier.

RLBox simplifies the burden on the developer by offering different types of validation functions. The first, `verify(verify_fn)`, validates simple `tainted` value types that have already been copied to renderer memory (e.g., simple values), as shown in this example:

```
tainted<int> status = sandbox_invoke(mRLBox,
  ↪  jpeg_read_header, p_mInfo, TRUE);
int untaintedStatus = status.verify([](int val){
  if (val == JPEG_SUSPENDED ||
      val == JPEG_HEADER_TABLES_ONLY ||
      val == JPEG_HEADER_OK ) { return val; }
  else { /* DIE! */ }
});
if (untaintedStatus == JPEG_SUSPENDED) { ... }
```

Not all `tainted` data lives in renderer memory, though. Validating shared `tainted` data structures that live in sandbox memory is unsafe: a concurrent sandbox thread can modify data after it's checked and before it's used. The `copyAndVerify(verify_fn, arg)` validator addresses this by copying its arguments into renderer memory before invoking the `verify_fn` closure. To prevent subtle bugs where a `verify()` function is accidentally applied to data in shared memory, RLBox issues a compile-time error—notifying the developer that `copyAndVerify()` is needed instead.

The `unsafeUnverified()` function removes tainting without any checks. This obviously requires care, but has several legitimate uses. For example, when migrating a codebase to use sandboxing, using `unsafeUnverified()` allows us to incrementally test our code before all validation closures have been written (§5). Furthermore, `unsafeUnverified()` is sometimes safe and necessary for performance—e.g., passing a buffer of libjpeg-decoded pixel data to the Firefox renderer without copying it out of sandbox memory. This is safe as pixel data is simple byte arrays that do not require complex decoding.

Validation in the presence of double fetches. Though our safe validation functions ensure that sandbox code cannot concurrently alter the data being validated, in practice we must also account for double fetch vulnerabilities.

Consider, for example, migrating the following snippet from the `nsJPEGDecoder::OutputScanlines` function:

```
1  while (mInfo.output_scanline < mInfo.output_height) {
2    ...
3    imageRow = reinterpret_cast<uint32_t*>(mImageData) +
4      ↪(mInfo.output_scanline * mInfo.output_width);
5    ...
6  }
```

Here, `mInfo` is a structure that lives in the sandbox shared memory. Buffer `imageRow` is a pointer to a decoded pixel-row that Firefox hands off to the rendering pipeline and thus must

not be `tainted`. To modify this code, we must validate the results on lines 1 and 4 which are tainted as they rely on `mInfo`. Unfortunately, validation is complicated by the double fetch: a concurrent sandbox thread could change the value of `output_scanline` between its check on line 1 and its use on line 4, for example. Unsafely handling validation would allow the sandbox to control the value of `imageRow` (the destination buffer) and thus perform arbitrary out-of-bounds writes.

We could address this by copying `output_scanline` to a local variable, validating it once, and using the validated value on both lines. But, it's not always this easy—in our port of Firefox we found numerous instances of multiple reads, interspersed with writes, spread across different functions. Using local variables quickly became intractable.

To address this, RLBox provides a `freeze()` method on `tainted` variables and struct fields. Internally, this method makes a copy of the value into renderer memory and ensures that the original value (which lives in sandbox memory), when used, has not changed. To prevent accidental misuse of freezable variables, RLBox disallows the renderer from reading freezable variables and fields until they are frozen. RLBox does, however, allow renderer code to write to frozen variables—an operation that modifies the original value and its copy. Finally, the `unfreeze()` method is used to restore the sandbox's write access.

Unlike most other RLBox features, ensuring that a variable remains frozen imposes some runtime overhead. This is thus a compile-time, opt-in feature that is applied to select variables and **struct** fields.

Writing validators. We identify two primary styles of writing validators in our porting Firefox to use sandboxed libraries: we can focus on either preserving application invariants or on preserving library invariants when crossing the trust boundary. We demonstrate these two alternate styles, using the above `OutputScanlines` example.

1. *Maintaining application invariants:* The first focuses on invariants expected by Firefox. To do this, we observe that `imageRow` is a pointer into the `mImageData` buffer and is used as a destination to write one row of pixels. Thus, it is sufficient to ensure that the result of `output_scanline` * `output_width` is between 0 and `mImageDataSize` − `rowSize`. This means that the `imageRow` pointer has room for at least one row of pixels.

2. *Checking library invariants:* The second option focuses on invariants provided by libjpeg. This option assumes that the Firefox decoder behaves correctly when libjpeg is well-behaved. Hence, we only need to ensure that libjpeg adheres to its specification. In our example, the libjpeg specification states that `output_scanline` is at most the height of the image: we thus only need to freeze `output_scanline` and then validate it accordingly.

4.4 Control flow safety

As discussed in Section 3, a malicious sandbox could attempt to manipulate renderer control flow in several ways. While data attacks on control flow are prevented by tainting (e.g., it's not possible to branch on a `tainted` variable), supporting callbacks requires additional support.

Control transfers via callbacks. It's unsafe to allow sandboxes to callback arbitrary functions in the renderer. It's also important to ensure they can only call functions which use `tainted` appropriately. Thus, RLBox forces application developers—via compile-time errors—to explicitly register callbacks using the `sandbox_callback()` function. For instance, line 8 in Figure 1, must be rewritten to:

```
mSourceMgr.fill_input_buffer
  ↪ = sandbox_callback(mRLBox, fill_input_buffer);
```

Statically whitelisting callbacks alone is insufficient—an attacker-controlled sandbox could still corrupt or hijack the control flow of the renderer by invoking callbacks at unexpected times. To address this class of attacks, RLBox supports unregistering callbacks with the `unregister()` method. Moreover, the framework provides RAII (resource acquisition is initialization) [55] semantics for callback registration, which allows useful code patterns such as automatically unregistering callbacks after completion of an invoked libjpeg function.

To deal with callback state exchange attacks (§3), RLBox raises a compile-time error when renderer pointers leak into the sandbox. For example, the JPEG decoder saves its instance pointer with libjpeg and retrieves it in the `fill_input_buffer()` callback, as shown on line 37 of Figure 1. RLBox requires the application developer to store such callback state in the application's thread local storage (TLS) instead of passing it to libjpeg. Thus, when `fill_input_buffer()` is invoked, it simply retrieves the decoder instance from the TLS, preventing any pointer leaks or callback state modifications.

Non-local control transfers. A final, but related concern is protecting control flow via `setjmp()`/`longjmp()`. These functions are used for exception handling (e.g., `my_error_exit()` in Figure 1). They work like a non-local **goto**, storing various registers and CPU state in a **jmp_buf** on `setjmp()` and restoring them on `longjmp()`.

Naively porting libjpeg and libpng would store **jmp_buf** in sandboxed memory. Unfortunately, this doesn't work—there is no easy way to validate a **jmp_buf** that is portable across different platforms. We thus instead place such sensitive state in the renderer's TLS and avoid validation altogether. With libjpeg this is straightforward since the **jmp_buf** is only used in the `my_error_exit()` callback. libpng, however, calls `longjmp()` itself. Since we can't expose `longjmp()` directly, when sandboxing libpng, we expose a `longjmp()` trampoline function that calls back to the renderer and invokes `longjmp()` on libpng's behalf, using the **jmp_buf** stored in the TLS.

5 Simplifying migration

RLBox simplifies migrating renderer code to use sandboxed libraries while enforcing appropriate security checks. RLBox does this by removing error-prone glue code (e.g., for data marshaling) and by reducing the security-sensitive code required for migration. The resulting reduction in developer effort is evaluated in detail in Section 7.3.

The RLBox framework helps automate porting by (1) allowing Firefox developers to *incrementally* port application code—the entire application compiles and runs with full functionality (passing all tests) between each step of porting—and (2) guiding the porting effort with compiler errors which highlight what the next step should be. We illustrate how RLBox minimizes engineering effort using this incremental, compiler-directed approach using our running example.

To start, we assume a standard Firefox build that uses a statically linked libjpeg.

Step 1 (creating the sandbox). We start using RLBox by creating a sandbox for libjpeg using the `None` sandboxing architecture. As the name suggests, this sandbox does not provide isolation; instead it redirects all function calls back to the statically linked, unsandboxed libjpeg. However, RLBox still fully enforces all of its type-level guarantees such as tainting untrusted data. Thus, we can start using RLBox while still passing functional tests.

Step 2 (splitting data and control flow). Next, we migrate each function call to libjpeg to use the `sandbox_invoke()` API. RLBox flags calls passing pointers to the sandbox as compile-time errors after this conversion, as the sandbox will be unable to access the (application) memory being pointed to. To resolve this, we also convert the allocations of objects being passed to libjpeg to instead use `sandbox_malloc()`. For example, in Section 3, we rewrote:

```
tainted<int> status = sandbox_invoke(mRLBox,
↳   jpeg_read_header, &mInfo, TRUE);
```

to, instead, allocate the `mInfo` object in the sandbox:

```
tainted<jpeg_decompress_struct*> p_mInfo =
  sandbox_malloc<jpeg_decompress_struct>(mRLBox);
tainted<int> status = sandbox_invoke(mRLBox,
↳   jpeg_read_header, p_mInfo, TRUE);
```

At this point, we need to re-factor the rest of this function and several other JPEG decoder functions—`mInfo` is a data member of the `nsJPEGDecoder` class. Doing this in whole is exhausting and error-prone. Instead, we remove the `mInfo` data member and add one extra line of code in each member function before `mInfo` is first used:

```
jpeg_decompress_struct&
↳   mInfo = *(p_mInfo.unsafeUnverified());
```

This *unsafe alias pattern* allows the remainder of the function body to run unmodified, i.e., the alias defined in this pattern can be used everywhere the original `mInfo` variable is

needed, albeit unsafely, as `unsafeUnverified()` temporarily suppresses the need for validation functions.

We also need to deal with return values from `sandbox_invoke()` which are `tainted`, either by writing validation functions to remove the taint or deferring this till the next step and using `unsafeUnverified()` to satisfy the type checker. Again, the entire application should now compile and run as normal.

Step 3 (hardening the boundary). Our next goal is to gradually remove all instances of the unsafe alias pattern, moving Firefox to a point where all data from the sandbox shared memory and all `tainted` return values are handled safely.

We can do this incrementally, checking our work as we go by ensuring the application still compiles and runs without errors. To do this, we simply move each unsafe-alias pattern downwards in the function source; as it moves below a given statement, that statement is no longer able to use the alias and must be converted to use the actual tainted value. This may involve writing validation functions, registering callbacks, or nothing (e.g., for operations which are safe to perform on `tainted` values). We can compile and test the application after any or all such moves. At the end, shared data is allocated appropriately, and all `tainted` values should be validated—no instances of `unsafeUnverified()` should remain.

Step 4 (enabling enforcement). Our final task is to replace the `None` sandbox with one that enforces strong isolation. To start, we remove the statically-linked libjpeg and change the sandbox type from `None` to `None_DynLib`. In contrast to the `None` sandbox, the `None_DynLib` sandbox dynamically loads libjpeg. Any remaining calls to libjpeg made without the `sandbox_invoke()` will fail with a symbol resolution error at compile time. We resolve these and finish by changing the sandbox type to `Process`, `NaCl` sandbox types that enforce isolation. We discuss these isolation mechanisms in more detail in Section 6.2.

6 Implementation

Our implementation consists of two components: (1) a C++ library that exposes the APIs that developers use when sandboxing a third-party library, and (2) two isolation mechanisms that offer different scaling-performance trade-offs. We describe both of these below, and also describe a third approach in Section 9.

6.1 RLBox C++ API and type system

The RLBox API is implemented largely as a pure C++ library. This library consists of functions like `sandbox_invoke()` that are used to safely transfer control between the application and library. These functions return `tainted` values and can only be called with `tainted` values or primitives. The library's wrapped types (e.g., `tainted<T>`) are used to ensure dataflow safety (e.g., when using a value returned by `sandbox_invoke()`). Since the implementation of the

control flow functions is mostly standard, we focus on our implementation of tainted values.

The `tainted<T>` wrapper. We implement `tainted<T>` as a simple wrapper that preserves the memory layout of the unwrapped `T` value, i.e., `tainted<int>` is essentially `struct tainted<int> { int val; }`. The only distinction between tainted and untainted values is at the type-level. In particular, we define methods and operators on `tainted<T>` values that (1) ensure that tainted values cannot be used where untainted values are expected (e.g., branch conditions) without validation and (2) allow certain computations on tainted data by ensuring their results are themselves tainted.

In general, we cannot prevent developers from deliberately abusing unsafe C++ constructs (e.g., `reinterpret_cast`) to circumvent our wrappers. Our implementation, however, guards against common C++ design patterns that could inadvertently break our `tainted` abstraction. For example, we represent tainted pointers as `tainted<T*>` and not `tainted<T> *`. This ensures that developers cannot write code that inadvertently unwraps `tainted` pointers via pointer decay—since all C++ pointers can decay to `void*`. We also use template meta-programming and SFINAE to express more complex type-level policies. For example, we disallow calls to `verify()` on pointer types and ensure that callback functions have wrapped all parameters in `tainted`.

Operators on `tainted` data. For flexibility, we define several operators on `tainted` types. Operations which are always safe, such as the assignment (`operator=`) of a `tainted<int>`, simply forward the operation to the wrapped `int`. Other operators, however, require return types to be `tainted`. Still others require runtime checks. We give a few illustrative examples.

▶ *Wrapping returned values:* We allow arithmetic operators (e.g., `operator+`) on, say, `tainted<int>`s, or a `tainted<int>` and an untainted `int`, but wrap the return value.

▶ *Runtime checks:* We allow array indexing with both `int`s and `tainted<int>`s by defining a custom array indexing operator `operator[]`. This operator performs a runtime check to ensure the array indexing is within sandbox memory.

▶ *Pointer swizzling:* We also allow operations such as `operator=`, `operator*`, and `operator->` on tainted pointers, but ensure that the operators account for swizzling (in addition to the runtime bounds check) when performing these operations. As with the others, these operators return `tainted` values. In Appendix A of [35], we describe the subtle details of type-driven automatic pointer swizzling.

Wrapped structs. Our library can automatically wrap primitive types, pointer types, function pointers, and static array types. It cannot, however, wrap arbitrary user-defined structs without some added boilerplate definitions. This is because C++ (as of C++17) does not yet support reflection on struct field *names*. We thus built a ~100LOC Clang plugin that automatically generates a header file containing the required boilerplate for all struct types defined in the application source.

Other wrapped types. In addition to the `tainted` wrapper type, RLBox also relies on several other types for both safety and convenience. As an example of safety, our framework distinguishes registered callbacks from other function pointers, at the type level. In particular, `sandbox_callback` returns values of type `callback<...>`. This allows us to ensure that functions that expect callbacks as arguments can in fact only be called with callbacks that have been registered with `sandbox_callback`. As an example of convenience, RLBox provides RAII types such as `stack_arr<T>` and `heap_arr<T>` which minimize boilerplate. With these types, developers can for instance invoke a function with an inline string: `sandbox_invoke(sbox, png_error, stack_arr("..."))`.

6.2 Efficient isolation mechanisms

The RLBox API provides a plugin approach to support different, low-level sandboxing mechanisms. We describe two sandboxing mechanisms which allow portable and efficient solutions for isolating libraries in this section. In Section 9 we describe a third mechanism, based on WebAssembly, that we recently integrated in production Firefox.

The first mechanism uses software-based fault isolation (SFI) [60] extending Google's Native Client (NaCl) [49, 65], while the second uses OS processes with a combination of mutexes and spinlocks to achieve performance. These approaches and trade-offs are described in detail below.

SFI using NaCl. SFI uses inline dynamic checks to restrict the memory addressable by a library to a subset of the address space, in effect isolating a library from the rest of an application within a single process. SFI scales to many sandboxes, has predictable performance, and incurs low overhead for context switching (as isolation occurs within a single process). The low context-switch overhead (about $10\times$ a normal function call) is critical for the performance of streaming libraries such as libjpeg, which tend to have frequent control transfers. We explore this in more detail later (§7).

To support library sandboxing with SFI, we extend the NaCl compiler toolchain [49, 65]. NaCl was originally designed for sandboxing mobile code in the browser, not library sandboxing. Hence, we made significant changes to the compiler, optimization passes, ELF loader, machine model and runtime; we give a thorough description of these changes in Appendix B of [35]. To ensure that our changes do not affect security, we always verify the code produced by our toolchain with the *unmodified* NaCl binary code verifier.

We use our modified NaCl compiler toolchain to compile libraries like libjpeg along with a custom runtime component. This runtime component provides symbol resolution and facilitates communication with the renderer.

Process sandboxing. Process sandboxing works by isolating

a library in a separate *sandbox process* whose access to the system call interface is restricted using seccomp-bpf [33]. We use shared memory between the two processes to pass function arguments and to allocate shared objects. Compared to SFI, process sandboxing is simpler and does not need custom compiler toolchains. When used carefully, it can even provide performance comparable to SFI (§7).

As with SFI, process sandboxing also includes a custom runtime that handles communication between the library and renderer. Unlike SFI, though, this communication is a control transfer that requires inter-process synchronization. Unfortunately, using a standard synchronization mechanism—notably, condition variables—is not practical: a simple cross-process function is over 300× slower than a normal function call.

Our process sandbox uses both spinlocks and condition variables, allowing users to switch between to address application needs. Spinlocks offer low control-transfer latency (20× a normal function call), at the cost of contention and thus scalability. Condition variables have higher latency (over 300× a normal function call), but minimize contention and are thus more scalable. In the next section we detail our Firefox integration and describe how we switch between these two process sandboxing modes.

6.3   Integrating RLBox with Firefox

To use the SFI or Process sandbox mechanisms efficiently in Firefox, we must make several policy decisions about when to create and destroy sandboxes, how many sandboxes to keep alive, and for the Process sandbox when to switch synchronization modes. We describe these below.

Creating and destroying sandboxes. We build Firefox with RLBox sandboxing web page decompression, image decoding, and audio and video playback. We apply a simple policy of creating a sandbox on demand—a fresh sandbox is required when decoding a resource with a unique `<renderer, library, content-origin, content-type>` as discussed in Section 2. We lazily destroy unused sandboxes once we exceed a fixed threshold. We determine this threshold experimentally. Most webpages have a large number of compressed media files as compared to the number of images (§7.1). Since we can only reasonably scale to 250 sandboxes (§7.5.3), we conservatively use a threshold 10 sandboxes for JPEG and PNG image decoding, and 50 sandboxes for webpage decompression. Browsers do not typically play multiple audio or video content simultaneously—we thus simply create a fresh sandbox for each audio and video file that must be decoded and destroy the sandbox immediately after.

Switching synchronization modes. For the Process sandbox, we switch between spinlocks and conditional variables according to two policies. First, we use spinlocks when the renderer performs a latency sensitive task, such as decoding an image using a series of synchronous libjpeg function calls and callbacks. But, when the renderer requests more input data, we switch to the condition variables; spinlocks

would create needless CPU contention while waiting for this data (often from the network). Second, for large media decoding such as 4K images, we use condition variables: each function call to the sandbox process takes a large amount of time (relative to the context switch) to perform part of the decoding. Though we can use more complex policies (e.g., that take load into effect), we find these two simple policies to perform relatively well (§7).

Leveraging multiple cores. Since 95% of devices that run Firefox have more than 1 core, we can use multiple cores to optimize our sandbox performance.[7] In particular, our process sandbox implementation pins the sandboxed process on a separate CPU core from the renderer process to avoid unnecessary context switches between the renderer and sandboxed process. This is particularly important when using spinlocks since the sandbox process's spinlock takes CPU cycles even when "not in use"; pinning this process to a separate core ensures that the sandbox process does not degrade the renderer performance. In our evaluation (§7), we reserve one of the system's cores exclusively for the processes created by our sandboxing mechanism, when comparing against the stock and SFI builds (which use all cores for the renderer process' threads).

7   Evaluation

We present the following results:

▶ *Cross origin resources that could compromise the renderer are pervasive*, and could even be used to compromise websites like Gmail. We present measurements of their prevalence in the Alexa top 500 websites in section 7.1.

▶ *Library sandboxing overheads are modest*: section 7.2 breaks down the individual sources of sandboxing overhead, section 7.4, shows end-to-end page latencies and memory overheads for popular websites are modest with both isolation mechanisms(§6.2)–even on media heavy websites, section 7.5, shows that the CPU overhead of web page decompression and image decoding in the renderer process are modest, and that CPU and memory overheads scale well up to our current maximum of 250 concurrent sandboxes.

▶ *Migrating a library into RLBox typically takes a few days with modest effort*, as shown in Section 7.3.

▶ *RLBox is broadly useful for library sandboxing beyond Firefox*, as demonstrated in   7.6, where we discuss our experience apply RLBox to sandboxing native libraries in Apache and Node.js modules.

Machine Setup. All benchmarks run on an Intel i7-6700K (4 GHz) machine with 64 GB of RAM, and hyperthreading disabled, running 64-bit Ubuntu 18.04.1. Firefox builds run pinned on two isolated CPU cores (i.e., no other process is allowed to run on these CPUs), to reduce noise. As discussed in Section 6.3, with the process sandbox build, the renderer

---

[7]See `https://data.firefox.com/dashboard/hardware`, last visited May 15, 2019.
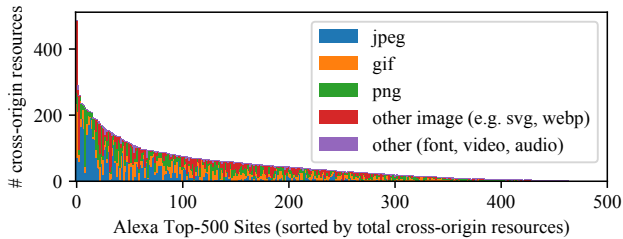
Figure 2: Cross-origin resource inclusion in the Alexa Top-500.

process is pinned to one core, while the sandbox process uses the other core.

7.1 Cross-origin content inclusion

To evaluate how often web sites include content (e.g., images, audio, and videos) cross-origin, we crawled the Alexa top 500 websites. Our crawler—a simple Firefox extension—logs all cross-origin resource requests made by the website for 10 seconds after page load (allowing some dynamic content to be loaded). Figure 2 shows our measurements, categorized by the resource MIME type.

We find that 93% of the sites load at least one cross-origin media resource (primarily images), with mean of 48 and median of 30, cross-origin media resources loaded. Many of the resource loads (median 35 and mean 17) are not just cross-origin but also *cross-site*. In the presence of media parsing library bugs, such loads would undermine Site Isolation protections.

The pervasive use of cross-origin resources inclusion indicates that sandboxing libraries at the `<renderer, library, content-origin, content-type>` granularity can significantly reduce the renderer attack surface. Although not all cross-origin content is necessarily untrusted, the origin is nevertheless an important trust boundary in practice—and many websites do consider cross-origin media untrusted. For instance, Google allows users to freely upload content to `sites.google.com`, but serves such media content from `googleusercontent.com`. Google even re-encodes images, another sign that such images are untrusted.

Unfortunately, they do not re-encode video or audio files. To test this, we created a page on `sites.google.com` in which we embedded both the VPX video proof of concept exploit of CVE-2015-4506 [10] and the OGG audio proof of concept exploit of CVE-2018-5148 [11]. In both cases the files were unmodified. For VPX, we modified Firefox and Chrome (with Site Isolation) to re-introduce the VPX bug and visited our malicious website: in both browsers the video successfully triggered the bug.

We found we could include such malicious content as part of an email to a Gmail address. Gmail re-encodes images, but does not re-encode the video and audio files. The Gmail preview feature even allows us to play the audio track—which surprisingly, we found was hosted on `mail.google.com`.

7.2 Baseline RLBox overhead

To understand the overhead of different parts of the RLBox framework in isolation we perform several micro-benchmarks.

Sandbox creation overhead. The overhead of sandbox creation is ≈1ms for SFI sandboxes and ≈2ms for process sandboxes. These overheads do not affect page latency for any realistic website, and can be hidden by pre-allocating sandboxes in a pool. For this reason, we don't include sandbox creation time in the remaining measurements.

Control transfer overhead. To understand the overhead of a control transfer, we measure the elapsed time for an empty function call with different isolation mechanisms. For reference, an empty function call without sandboxing takes about 0.02μs in our setup. With sandboxing, we measured this to be 0.22μs for SFI sandboxes, 0.47μs for Process sandboxes using spinlocks, and 7.4μs for Process sandboxes using conditional variables. SFI and Process sandboxes using spinlocks are an order of magnitude slower than a normal function call, but over $10\times$ faster than Processes using condition variables, and are thus better suited for workloads with frequent control transfers.

Overhead of RLBox dynamic checks. The RLBox API introduces small overheads with its safety checks (e.g., pointer bounds checks, swizzling, and data validation functions). To measure these, we compare the overhead of rendering `.jpeg` and `.png` images on Firefox with sandboxed libraries with and without RLBox enabled. We find the difference to be negligible ($< 1\%$). This is unsurprising: most of RLBox's checks are static and our dynamic checks are lightweight masks.

Overhead of SFI dynamic checks. Unlike process-based sandboxing, SFI incurs a baseline overhead (e.g., due to inserted dynamic checks, padding etc.) [65]. To understand the overhead of our NaCl SFI implementation, we implement a small `.jpeg` image decoding program and measure its slowdown when using a sandboxed libjpeg. We find the overhead to be roughly 22%.

7.3 Migrating Firefox to use RLBox

In this section, we evaluate the developer effort required to migrate Firefox to use sandboxed libraries. In particular, we report the manual effort required by RLBox developers and the manual effort saved by using the RLBox API. Figure 3 gives a breakdown of the effort. On average, we find that it takes a bit over two days to sandbox a library with RLBox and roughly 180 LOC (25% code increase); much of this effort is mechanical (following Section 5).

Figure 3 also shows the tasks that RLBox automates away. First, RLBox eliminates 607 lines of glue-code needed by both the Process and SFI sandboxes to marshal function parameters and return values for each cross-boundary function call; RLBox automatically generates this boilerplate through

	Task	JPEG Decoder	PNG Decoder	GZIP Decompress	Theora Decoder	VPX Decoder	OGG-Vorbis Decoder
Effort saved by RLBox automation	Generated marshaling code	133 LOC	278 LOC	38 LOC	39 LOC	60 LOC	59 LOC
	Automatic pointer swizzles for function calls	30	96	5	36	46	34
	Automatic nested pointer swizzles	17	5	6	8	9	5
	Automatic pointer bounds checks	64 checks	25 checks	8 checks	12 checks	15 checks	14 checks
	Number of validator sites found	28	51	10	5	2	4
Manual effort	Number of person-days porting to RLBox	–	–	1 day	3 days	3 days	2 days
	Application LOC before/after port	720 / 1058	847 / 1317	649 / 757	220 / 297	286 / 368	328 / 395
	Number of unique validators needed	11	14	3	3	2	2
	Average LOC of validators	3 LOC	4 LOC	2 LOC	3 LOC	2 LOC	2 LOC

Figure 3: Manual effort required to retrofit Firefox with fine grain isolation, including the effort saved by RLBox's automation. We do not report the number of days it took to port the JPEG and PNG decoders since we ported them in sync with building RLBox.
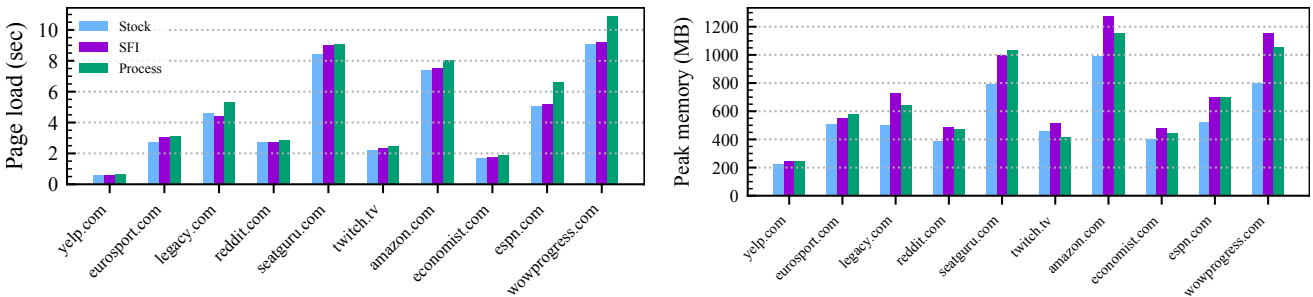

Figure 4: Impact of sandboxing on page load latencies and *peak* memory usage overheads. Firefox with SFI sandboxes incurs a 3% page latency and 25% memory overhead while Firefox with process isolation incurs a 13% page latency and a 18% memory overhead.

C++ templates and meta-programming. Second, RLBox automatically swizzles pointers. This is necessary for any sandbox functions or callbacks that accept pointers; it's also necessary when handling data-structures with nested pointers that are shared between the application and the sandbox. This is a particularly challenging task without RLBox, as manually identifying the 297 locations where the application interacts with such pointers would have been tedious and error-prone. Third, RLBox automatically performs bounds checks (§4.2); the number of required pointer bounds checks that were automatically performed by RLBox are again in the hundreds (138).

Finally, RLBox identifies the (100) sites where we must validate tainted data (§4.3). Though RLBox cannot automate the validators, we find that we only need 35 unique validators—all less than 4 lines of code. In practice, we found this to be the hardest part of migration since it requires understanding the domain-specific invariants.

7.4 RLBox overhead in Firefox

We report the end-to-end overheads of Firefox with library sandboxing by measuring page latencies of webpages, memory overheads in Firefox as well as audio video playback rates.

Experimental setup. We evaluate end-to-end performance with six sandboxed libraries: libjpeg-turbo 1.4.3, libpng 1.6.3, zlib 1.2.11, libvpx 1.6.1, libtheora 1.2, and libvorbis as used in Firefox 57.0.4. We report performance for two Firefox builds that use libraries sandboxed with SFI and Process mechanisms respectively. Both builds create fresh sandboxes

for each `<renderer, library, origin, content-type>` combination as described in §2. We measure impact on page load times for both these builds.

7.4.1 End-to-end impact on real-world websites

Benchmark. We report the overhead of page load latency and memory overheads in Firefox with the six sandboxed libraries by measuring latencies of the 11 websites used to measure the overhead of Site Isolation [41]. These websites are a representative sample of both popular and slightly-less-popular websites, and many of them make heavy use of media resources. We measure page load latency using Firefox's Talos test harness [32]. We measure memory overheads with `cgmemtime` [46]—in particular, the peak resident memory and cache usage during a run of Firefox. We run the test 10 times and report the median values of page latency and memory overheads.

Results. As shown in Figure 4, the page latency and CPU utilization overheads are modest. Our SFI build incurs a 3% overhead in page latency while the Process sandbox incurs an overhead of 13%. As a comparison point, the overhead of naively using process sandboxes (using only conditional variables without any CPU pinning) incurs an overhead of 167%.

We find the average peak renderer memory overhead to be 25% and 18% for the SFI and Process sandboxes, respectively. This overhead is modest and, more importantly, transient: we can destroy a sandbox after the media has been decoded during page load.

7.4.2 Sandboxing video and audio decoding

To understand the performance overhead of RLBox on video and audio decoding, we measure the performance of Firefox when decoding media from two video formats (Theora and VPX) and one audio format (OGG-Vorbis).

Benchmark. Our benchmark measures decoding performance on the Sintel sequence in Xiph.org's media test suite[8]. As this sequence is saved as lossless video, we setup the benchmark by first converting this to ultra HD videos of 4K resolution in the Theora and VP9 formats; we similarly convert the lossless audio to a high resolution OGG-vorbis audio file with a sample rate of 96 kHz and a bit rate of 500 Kb/s using the suggested settings for these formats [57, 59]. Our benchmark then measures the frame-rate and bit-rate of the video and audio playback—by instrumenting the decoders with timers—when using the sandboxed libraries. We run the test 5 times and report the median values of frame-rate and bit-rate.

Results. We find that neither the SFI nor the Process sandboxing mechanism visibly degrades performance. In particular, our sandboxed Firefox are able to maintain the same frame-rate (24 fps for the VPX video and 60 fps for the Theora video) and bit-rate (478 bits per second) as stock Firefox for these media files.

7.5 Microbenchmarks of RLBox in Firefox

To understand the performance impact of RLBox on the different libraries, we perform several microbenchmarks that specifically measure the impact of sandboxing webpage decompression, image decoding and sandbox scaling in Firefox.

7.5.1 Sandboxing webpage decompression

Firefox uses zlib to decompress webpages. Since webpage decompression is done entirely before the page is rendered, we report the overhead of sandboxing zlib by measuring the slowdown in page load time.

Benchmark. We create a webpage whose HTML content (excluding media and scripts) is 1.8 MB, the size of an average web page[9], and measure the page load time with Talos. We use the median page load time from 1000 runs of this test.

Results. For both SFI and Process sandboxing mechanisms, the overhead of sandboxing zlib is under 1%. In other words, the overhead of sandboxing zlib is largely offset by other computations needed to render a page.

7.5.2 Sandboxing image decoding

To understand performance impact of sandboxing on image rendering, we measure per-image execution time for the .jpeg and .png decoders, with different forms of sandboxing, and compare our results to stock Firefox. Decoder execution time is a better metric for image rendering performance than page

load time because Firefox decodes and renders images asynchronously; the usual test harness would notify us that a page has been loaded before images have actually been decoded and displayed in full—and this might be visible to the user.

Benchmarks. We use the open Image Compression benchmark suite[10] to measure sandboxed image decoding overheads. To capture the full range of possibilities for performance overhead, we measure overheads of images at 3 sizes (135p, 320p, and 1280p) and three compression levels (best, default, and none) for each image in the benchmark suite. We run this test 4000 times for each image and compare the median decoder code execution times.

Results. Since all images in the suite produce similar results, we give the results of one 8-bit image in Figure 5. We start with three high-level observations. First, both SFI and Process based sandboxes have reasonable overheads—23-32% and $< 41\%$ respectively for most JPEGs, 2-49% and $< 15\%$ respectively for PNGs. Second, Process sandbox sometimes has negative overheads. This is because the Process sandbox dedicates one of the two available cores exclusively for execution of sandboxed code (§6.3) including the sandboxed image rendering code, while the stock and SFI Firefox builds use all cores evenly. Third, for JPEGs at the best compression, the overhead relative to stock Firefox is high—roughly 80% for SFI and 140% for Process sandboxes. This is because decoding high compression images have low absolute decode times (~650μs), and thus have larger overheads as control transfer overheads between Firefox and the sandbox image libraries cost are not effectively amortized. However, in absolute terms, the differences are less than 1.5ms and have no impact on end-user experience.

7.5.3 Sandbox scaling characteristics

Web pages often contain tens of images of different types from multiple origins. Thus, the scaling properties of different isolation mechanisms are an important consideration.

Benchmark. We evaluate sandbox scaling by rendering pages with an increasing number of JPEG images from unique origins. Each image thus creates a new sandbox which incurs both CPU and memory costs. CPU costs are measured by measuring the total amount of time executing image decoding functions. We measure memory overhead as before, but don't destroy any sandbox; this allows us to estimate the worst case scenario where memory usage is *not* transient. As before (§7.5.2), we measure the decoder execution time for 4000 image loads at each scale, and report the median overhead.

Results. Figure 6 shows the CPU overhead of image rendering as we increase the number of sandboxes for both large (1280p) and small (135p) JPEG images using default compression. This experiment allows us to make several observations. We can run up to 250 concurrent SFI sandboxes

---

[8]Online: https://media.xiph.org/. Last visited November 15, 2019.
[9]See the HTTP Archive page weight report, https://httparchive.org/reports/page-weight. Last visited May 15, 2019.

[10]Online: https://imagecompression.info/test_images/. Last visited May 15, 2019.

(a) JPEG rendering overhead

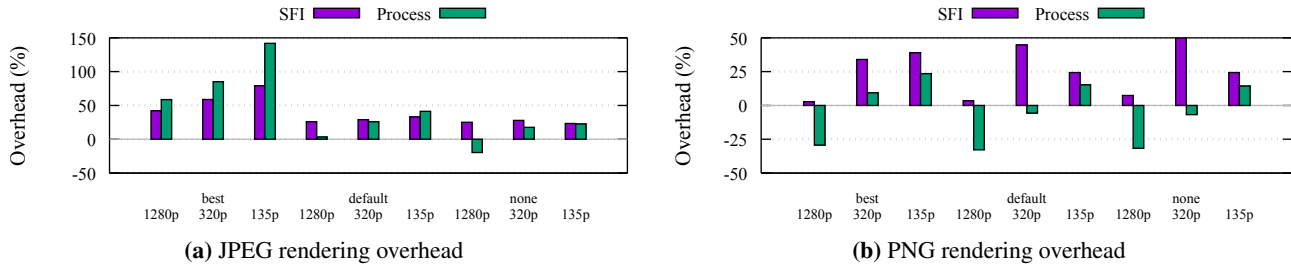**(b)** PNG rendering overhead

Figure 5: Per-image decoding overhead for images at 3 compression levels and 3 resolutions, normalized against stock Firefox.
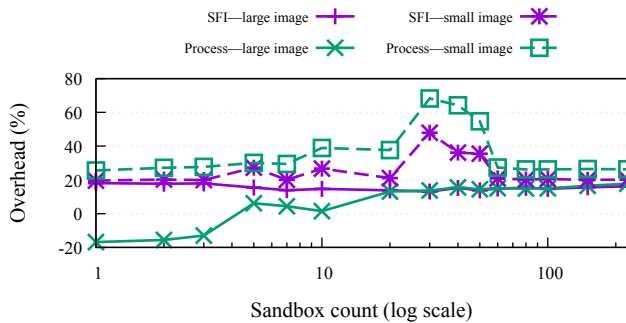

Figure 6: Performance overhead of image decoding with increasing the number of sandboxes (each image is rendered in a fresh sandbox).

before we run into limitations like exhausting pre-allocated thread local storage or finding aligned free virtual memory. These limitations can be overcome with more engineering effort. We never came close to these limits browsing real websites, including those of Section 7.4.1. Both the SFI and the Process sandbox similarly scale well on both small and large images, with CPU overheads between 20% and 40% for most sandbox counts. The process sandbox, however, scales only because we use multiple synchronization modes described (§6).

Extra sandboxes add memory overhead for two reasons. First, each sandbox uses a private copy of code (e.g., libjpeg and libc for each libjpeg sandbox). Second, each sandbox has its own stack and heap. In this experiment, we observed that memory consumption increases linearly with the number of images (which corresponds to the number of sandboxes created). On average, an SFI sandbox consumes 1.6 MB, while Process sandboxing consumes 2.4 MB for each sandbox. Several optimizations to reduce memory consumption exist that we have not yet implemented. For example, the SFI sandbox currently loads a fresh copy of the code for each sandbox instance. We could optimize this by sharing code pages between sandboxes—and, indeed, we do this in production for our Wasm sandbox.

7.6 RLBox outside Firefox

RLBox is a general-purpose sandboxing framework that can be used in any C++ application. To demonstrate its utility

beyond Firefox, we applied it in two different contexts: the Apache web server and Node.js runtime.

Apache allows developers to write C modules that extend its base functionality. These modules often depend on third-party libraries. For example, the mod_markdown [31] module uses the libmarkdown library to transform Markdown to HTML on the fly to support serving Markdown files.

To protect Apache from bugs in libmarkdown we modify mod_markdown to run libmarkdown in an RLBox SFI sandbox. The change required a single person-day, and added or modified roughly 300 lines of code. We measured the average and tail latency as well as throughput of the webserver using the autocannon 4.4.0 benchmarking tool [2] with default configurations (1 minute runtime, 10 parallel connections) serving a 16K markdown file. The unmodified webserver's average latency, tail latency and throughput were 10.5ms, 36ms and 940 requests/second, respectively; the sandboxed server's average latency, tail latency and throughput were 14ms, 40ms and 684 requests/second. Though the average latency and throughput overhead is modest, we observe that the tail latency—arguably the most important metric—is within 10% of baseline.

Node.js is a JavaScript runtime system written in C++, largely used for web applications. Like Apache, Node.js allows developers to expose new functionality implemented in native plugins to the JavaScript code. For example, the bcrypt [38] password hashing library relies on native code—indeed the JavaScript code largely wraps Provos' C bcrypt library. To protect the runtime from memory-safety bugs in the C library, we modify the C++ code in bcrypt to run the bcrypt C library in an RLBox SFI sandbox—a change that required roughly 2 person hours, adding or modifying 75 lines of code. We measured—using the benchmark.js library—the overhead in average hashing throughput (hashing a random 32-byte password) to be modest: 27%.

8 Related work

Isolation in the browser. Modern browsers since Chrome [3] rely on coarse grain *privilege separation* [39] to prevent browser compromises from impacting the local OS [40]. However, a compromised renderer process can still use any credentials the browser has for other sites, enabling extremely

powerful *universal cross-site scripting* (UXSS) attacks [14].

In response to UXSS attacks and recent Spectre attacks, Chrome introduced *Site Isolation* [41]. Site Isolation puts pages and iframes of different sites into separate processes. Unfortunately, as discussed in Section 1, this does not prevent UXSS attacks across related sites (e.g., `mail.google.com` and `pay.google.com`). Firefox's Project Fission [34] proposes to go further and isolate at the origin boundary, similar to previous research browsers [18, 44, 61], but this still does not protect the renderer when loading cross-origin resources such as images.

An unpublished prototype using SFI called MinSFI [48] was developed at Google in 2013 to protect the Chrome renderer from compromise of zlib library; however, it was missing several features necessary for compatibility and efficiency, including threading and callback support. Additionally, the project was primarily focused on improving the efficiency of SFI rather than the integration challenges tackled by RLBox including handling `tainted` data, migration of code bases, etc.

In some parts of the renderer, there is no substitute for strong memory safety to prevent attacks. Servo [1] is an ongoing project to rewrite much of the Firefox rendering stack in Rust. However, for the foreseeable future, Firefox and other browsers will continue to rely on libraries written in C/++. This makes sandboxing the most viable approach to containing vulnerabilities.

Sandboxing. There has been some related work on providing APIs to simplify using sandboxed libraries (e.g., Codejail [63] and Google Sandboxing APIs [5]). However, these efforts do not provide the type-driven automation of RLBox (e.g., pointer swizzling and migration assistance) nor the safety of `tainted` types—leaving developers to manually deal with attacks of Section 3. Sammler et al. [43] formal model addresses some of these attacks using a type-direct approach, but require applications to be formally verified correct (in contrast to our local validators) to give meaningful guarantees.

There is a long line of work on sandboxing mechanisms with different performance trade-offs [20, 48, 49, 58, 60]. Recent, excellent surveys [52, 56] present a comprehensive overview of these mechanisms. RLBox makes it easy for developers to use such mechanisms without modifying the application or library (§6.2). In production we use WebAssembly; WebAssembly stands out as a principled approach with wide adoption [20].

Data sanitization and leaks. There is a large body of work on static and dynamic approaches to preventing or mitigating missed sanitization errors; see the survey by Song et al. [53]. These tools are orthogonal to RLBox. Developers could use them to check their data validation functions for bugs.

DUI Detector [22] uses runtime trace analysis to identify missing pointer sanitizations in user code. Other work has looked at sanitizing user pointers in the kernel [9]. For example, type annotations [25] have been used to distinguish between untrusted user pointers and trusted pointers in OS

kernel code. In contrast, RLBox automatically applies such pointer sanitizations by leveraging the C++ type system.

One approach to avoid double-fetch bugs is to marshal all shared data before using it. But, this comes at a cost. Marshaling tools and APIs typically require array bounds annotations, which is tedious and demands in-depth knowledge of the sandboxed library's internal data structures. Automatic marshaling tools like C-strider [45] and PtrSplit [29] address this limitation; however, these tools either impose a significant overhead or lack support for multi-threading. RLBox uses shared memory and statically enforces that shared data is placed in shared memory, avoiding the need for custom marshaling tools or annotations. The use of shared memory, however, introduces possible double-fetch bugs. While RLBox provides APIs to assist with double-fetches, the possibility of unhandled double-fetch bugs still remain. Several recent techniques detect double-fetches from shared memory [47, 62, 64] and can be used alongside RLBox.

Previous efforts have also sought to prevent leaking pointers that could compromise ASLR [6, 13, 30]. RLBox prevents pointer leaks by disallowing pointers to renderer memory to pass into the sandboxed library via the type system.

Porting assistance. Several privilege separation tools provide assistance when migrating to a sandboxed architecture. Wedge (Crowbar) [4] uses dynamic analysis to guide the migration, and also supports an incremental porting mode that disables isolation so that developers can test the partial port and identify the next step. SOAAP [19] uses code annotations and a custom compiler to guide the developer. PrivTrans [8] uses code annotations and automatic source code rewriting to separate code and data into separate components. In contrast, RLBox assists with porting without any custom tooling, purely through the use of compile-time errors, by identifying code that must be modified for security and shared data that must be migrated to sandbox memory (§5).

9   Using RLBox in production

Over the last 6 months we've been integrating RLBox into production Firefox. In this section, we describe the difference between our research prototype and the production RLBox, and our migration of the libGraphite font shaping library to use RLBox. We are in the process of migrating several other libraries and adding support for Firefox on Windows [15, 51].

9.1   Making RLBox production-ready

To make RLBox production-ready we adapt a new isolation mechanism based on WebAssembly and rewrite the RLBox API, using our prototype implementation as a reference.

SFI using WebAssembly. In production, we use Wasm instead of NaCl to isolate library code from the rest of Firefox within a single process. Though Wasm's performance and feature-set is not yet on par with NaCl's [23], NaCl has been deprecated in favor of Wasm [17] and maintaining a separate toolchain for library sandboxing is prohibitive. Moreover,

these limitations are likely to disappear soon: as part of the Bytecode Alliance, multiple companies are working together to build robust Wasm compilation toolchains, a standard syscall interface, SIMD extensions, etc.

Since our goal is to reap the benefits of these efforts, we need to minimize the changes to these toolchains. In particular, this means that we cannot adjust for differences between the Firefox and Wasm machine model as we did for NaCl [36]—by intrusively modifying the compiler, loader, runtime, etc. (§6.2). We, instead, take advantage of the fact that RLBox intercepts all data and control flow to automatically translate between the Firefox and Wasm machine models in the RLBox API.

Our only modification to the Lucet Wasm runtime is an optimized trampoline, which we are working on upstreaming [67]. Since Wasm is well-typed and has deterministic semantics, our trampolines safely eliminate the context-and stack-switching code, reducing the cost of a cross-boundary crossing to a function call. This optimization was key to our shipping the sandboxed libGraphite (§5)—it reduced the overhead of RLBox by 800%. The details and formalization of these *zero-cost trampolines* will be presented in a separate paper.

Meaningful migration error-messages. We re-implement RLBox in C++ 17 and use new features—in particular `if constexpr`—to customize the error messages that guide developers during migration (§5). Meaningful error messages (as opposed to a wall of generic, template failures) is key to making RLBox usable to other developers. Although implementing custom error messages in our C++ 11 prototype is possible, it would make the implementation drastically more complex; C++ 17 allows us to keep the RLBox API implementation concise (under 3K lines of code) and give meaningful error messages.

9.2 Isolating libGraphite

We use RLBox to isolate the libGraphite font shaping library, creating a fresh sandbox for each Graphite font instance. We choose libGraphite largely because the Graphite fonts are not widely used on the web, but nevertheless Firefox needs to support it for web compatibility. This means that the library is part of Firefox attack surface—and thus memory safety bugs in libGraphite are security vulnerabilities in Firefox [54, 66].

Evaluation To measure the overhead of our sandboxing, we use a micro-benchmark that measures the page render time when reflowing text in a Graphite font ten times, adjusting the font size each time, so font caches aren't used.[11] We find that Wasm sandboxing imposes a 85% overhead on the libGraphite code, which in turn slows down Firefox's font rendering component (which uses libGraphite internally) by 50%. We attribute this slowdown largely to the nascent Wasm toolchains, which don't yet support performance optimization on par with, say LLVM [21, 23]. Nevertheless, this overhead

---

[11]Available at: https://jfkthame.github.io/test/udhr_urd.html.

is not user-perceptible; in practice page rendering is slowed down due to the network and heavy media content, not fonts.

To measure memory overhead, we use `cgmemtime` to capture the peak resident memory and cache used by Firefox on the same micro-benchmark. We find the memory overhead to be negligible—the median peak memory overhead when loading the micro-benchmark ten times is 0.68% (peak memory use went from 431460 KB to 434426 KB).

Deployment. The rewritten RLBox library as well as the modifications to Firefox to use a sandbox libGraphite have been merged into the Firefox code base [51]. Our retrofitted Firefox successfully tested on both the Firefox Nightly and Beta channels, and ships in stock Firefox 74 to Linux users and in Firefox 75 to Mac users [15].

10 Conclusion

Third party libraries are likely to remain a significant source of critical browser vulnerabilities. Our approach to sandboxing code at the library-renderer interface offers a practical path to mitigating this threat in Firefox, and other browsers as well.

RLBox shows how a type-driven approach can significantly ease the burden of securely sandboxing libraries in existing code, through a combination of static information flow enforcement, dynamic checks, and validations. RLBox is not dependent on Firefox and is useful as a general purpose sandboxing framework for other C++ applications.

Acknowledgements

References

[1] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin. Engineering the Servo Web browser engine using rust. In *ICS 2016: SEIP*. ACM, 2016.

[2] autocannon: fast HTTP/1.1 benchmarking tool written in Node.js. https://github.com/mcollina/autocannon, 2016.

[3] A. Barth, C. Jackson, C. Reis, and the Google Chrome Team. The security architecture of the Chromium browser. Technical report, 2008.

[4] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI*. USENIX, 2008.

[5] C. Blichmann, R. Swiecki, and ISE Sandboxing team. Open-sourcing sandboxed API. https://security.googleblog.com/2019/03/open-sourcing-sandboxed-api.html, 2019.

[6] K. Braden, L. Davi, C. Liebchen, A.-R. Sadeghi, S. Crane, M. Franz, and P. Larsen. Leakage-resilient layout randomization for mobile devices. In *NDSS*. Internet Society, 2016.

[7] C. Brook. Firefox, Safari, Edge all fall at Pwn2Own 2018. https://digitalguardian.com/blog/firefox-safari-edge-all-fall-pwn2own-2018, 2018.

[8] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Security*. USENIX, 2004.

[9] S. Bugrara and A. Aiken. Verifying the safety of user pointer dereferences. In *S&P*. IEEE, 2008.

[10] Bug 1192226 (CVE-2015-4506) vp9 init context buffers. https://bugzilla.mozilla.org/show_bug.cgi?id=1192226, 2015.

[11] Bug 1446062 (CVE-2018-5146) ZDI-CAN-5822 - Mozilla Firefox Audio Driver Out of Bounds. https://bugzilla.mozilla.org/show_bug.cgi?id=1446062, 2018.

[12] Security: stack-buffer-overflow in break. https://bugs.chromium.org/p/chromium/issues/detail?id=850350, 2019.

[13] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *S&P*. IEEE, 2015.

[14] S. Di Paola and G. Fedon. Subverting Ajax. Presented at 23C3, 2006.

[15] N. Froyd. Securing Firefox With WebAssembly. https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly/, 2020.

[16] Georg Koppen. Use RLBox for sandboxing third-party libraries. https://trac.torproject.org/projects/tor/ticket/32379.

[17] Google Chrome Team. (P)NaCl Deprecation Announcements. https://developer.chrome.com/native-client/migration#p-nacl-deprecation-announcements, 2017.

[18] C. Grier, S. Tang, and S. T. King. Designing and implementing the OP and OP2 Web browsers. *ACM Transactions on the Web*, 5(2), 2011.

[19] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean application compartmentalization with soaap. In *CCS*. ACM, 2015.

[20] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *PLDI*. ACM, 2017.

[21] L. T. Hansen. Cranelift: Performance parity with Baldr on x86-64. https://bugzilla.mozilla.org/show_bug.cgi?id=1539399, 2019.

[22] H. Hu, Z. L. Chua, Z. Liang, and P. Saxena. Identifying arbitrary memory access vulnerabilities in privilege-separated software. In *ESORICS*, volume 9326 of *LNCS*. Springer, 2015.

[23] A. Jangda, B. Powers, E. D. Berger, and A. Guha. Not so fast: analyzing the performance of webassembly vs. native code. In *ATC*. USENIX, 2019.

[24] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang. The "Web/local" boundary is fuzzy: A security study of Chrome's process-based sandboxing. In *CCS*. ACM, 2016.

[25] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Security*. USENIX, 2004.

[26] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *S&P*. IEEE, 2019.

[27] D. Kohlbrenner and H. Shacham. Trusted browsers for uncertain times. In *Security*. USENIX, 2016.

[28] G. Kwong. JavaScript fuzzing in Mozilla, 2017. Presented at COSCUP 2017. https://nth10sd.github.io/js-fuzzing-in-mozilla/, 2017.

[29] S. Liu, G. Tan, and T. Jaeger. PtrSplit: Supporting general pointers in automatic program partitioning. In *CCS*. ACM, 2017.

[30] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *CCS*. ACM, 2015.

[31] Markdown filter module for apache httpd server. https://github.com/hamano/apache-mod-markdown, 2011.

[32] Performance sheriffing/Talos. https://wiki.mozilla.org/Performance_sheriffing/Talos, 2018.

[33] Security/sandbox. https://wiki.mozilla.org/Security/Sandbox, 2018.

[34] Project Fission. https://wiki.mozilla.org/Project_Fission, 2019.

[35] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the Firefox renderer: Extended version. https://arxiv.org/abs/2003.00572, 2020.

[36] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan. Gobi: Webassembly as a practical path to library sandboxing, 2019.

[37] S. Nichols. It's 2019, and a PNG file can pwn your Android smartphone or tablet: Patch me if you can. https://www.theregister.co.uk/2019/02/07/android_january_patches/, 2019.

[38] bcrypt for nodejs. https://github.com/kelektiv/node.bcrypt.js, 2010.

[39] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Security*. USENIX, 2003.

[40] C. Reis and S. D. Gribble. Isolating Web programs in modern browser architectures. In *EuroSys*. ACM, 2009.

[41] C. Reis, A. Moshchuk, and N. Oskov. Site isolation: Process separation for web sites within the browser. In *USENIX Security Symposium*. USENIX Association, 2019.

[42] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[43] M. Sammler, D. Garg, D. Dreyer, and T. Litak. The high-level benefits of low-level sandboxing. In *POPL*. ACM, 2019.

[44] R. Sasse, S. T. King, J. Meseguer, and S. Tang. IBOS: A correct-by-construction modular browser. In *FACS*, volume 7684. Springer, 2013.

[45] K. Saur, M. Hicks, and J. S. Foster. C-strider: type-aware heap traversal for C. *Software: Practice and Experience*, 46(6), 2016.

[46] G. Sauthoff. cgmemtime. https://github.com/gsauthof/cgmemtime, 2012.

[47] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *AsiaCCS*. ACM, 2018.

[48] M. Seaborn. Sandboxing libraries in Chrome using SFI: zlib proof-of-concept. https://docs.google.com/presentation/d/1RD3bxsBfTZOIfrlq7HzGMsygPHgb61A1eTdelIYOurs/, 2013.

[49] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *Security*. USENIX, 2010.

[50] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS*. ACM, 2004.

[51] Shravan Narayan. Use Wasm sandboxed libraries in Firefox to reduce attack surface. https://bugzilla.mozilla.org/show_bug.cgi?id=1562797.

[52] R. Shu, P. Wang, S. A. Gorski, III, B. Andow, A. Nadkarni, L. Deshotels, J. Gionta, W. Enck, and X. Gu. A study of security isolation techniques. *ACM Computing Surveys*, 49(3), 2016.

[53] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for security. In *IEEE Security and Privacy 2019*, 2019.

[54] Stefan Marsiske, Pierre Pronchery, Marcus Bointon. Penetration Test Report: Graphite font system. https://wiki.mozilla.org/images/9/98/Graphite-report.pdf, 2017.

[55] B. Stroustrup. Exception handling (and RAII). In *The C++ Programming Language*, chapter 13. Addison-Wesley, 4th edition, 2013.

[56] G. Tan. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends in Privacy and Security*, 1(3), 2017.

[57] A brief Theora and Vorbis encoding guide. https://trac.ffmpeg.org/wiki/TheoraVorbisEncodingGuide, 2019.

[58] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Security*. USENIX, 2019.

[59] VP9 bitrate modes in detail. https://developers.google.com/media/vp9/bitrate-modes/, 2019.

[60] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*. ACM, 1993.

[61] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Security*. USENIX, 2009.

[62] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Security*. USENIX, 2017.

[63] Y. Wu, S. Sathyanarayan, R. H. Yap, and Z. Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *ESORICS*, volume 7459 of *LNCS*. Springer, 2012.

[64] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and scalable detection of double-fetch bugs in OS kernels. In *S&P*. IEEE, 2018.

[65] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *S&P*. IEEE, 2009.

[66] Yves Younan. Vulnerability Spotlight: Libgraphite Font Processing Vulnerabilities. https://blog.talosintelligence.com/2016/02/vulnerability-spotlight-libgraphite.html, 2016.

[67] Optimized transitions for Lucet compiler. https://github.com/bytecodealliance/cranelift/issues/1083, 2019.