Site Isolation: Process Separation for Web Sites within the Browser

Charles Reis
Google
creis@google.com

Alexander Moshchuk
Google
alexmos@google.com

Nasko Oskov
Google
nasko@google.com

Abstract

Current production web browsers are multi-process but place different web sites in the same renderer process, which is not sufficient to mitigate threats present on the web today. With the prevalence of private user data stored on web sites, the risk posed by compromised renderer processes, and the advent of transient execution attacks like Spectre and Meltdown that can leak data via microarchitectural state, it is no longer safe to render documents from different web sites in the same process. In this paper, we describe our successful deployment of the *Site Isolation* architecture to all desktop users of Google Chrome as a mitigation for process-wide attacks. Site Isolation locks each renderer process to documents from a single site and filters certain cross-site data from each process. We overcame performance and compatibility challenges to adapt a production browser to this new architecture. We find that this architecture offers the best path to protection against compromised renderer processes and same-process transient execution attacks, despite current limitations. Our performance results indicate it is practical to deploy this level of isolation while sufficiently preserving compatibility with existing web content. Finally, we discuss future directions and how the current limitations of Site Isolation might be addressed.

1 Introduction

Ten years ago, web browsers went through a major architecture shift to adapt to changes in their workload. Web content had become much more active and complex, and monolithic browser implementations were not effective against the security threats of the time. Many browsers shifted to a multi-process architecture that renders untrusted web content within one or more low-privilege sandboxed processes, mitigating attacks that aimed to install malware by exploiting a rendering engine vulnerability [43, 51, 70, 76].

Given recent changes in the security landscape, that multi-process architecture no longer provides sufficient safety for visiting untrustworthy web content, because it does not provide similar mitigation for attacks between different web sites. Browsers load documents from multiple sites within the same renderer process, so many new types of attacks target rendering engines to access cross-site data [5, 10, 11, 33, 53]. This is increasingly common now that the most exploitable targets of older browsers are disappearing from the web (e.g., Java Applets [64], Flash [1], NPAPI plugins [55]).

As others have argued, it is clear that we need stronger isolation between security principals in the browser [23, 33, 53, 62, 63, 68], just as operating systems offer stronger isolation between their own principals. We achieve this in a production setting using *Site Isolation* in Google Chrome, introducing OS process boundaries between web site principals.

While Site Isolation was originally envisioned to mitigate exploits of bugs in the renderer process, the recent discovery of *transient execution attacks* [8] like Spectre [34] and Meltdown [36] raised its urgency. These attacks challenge a fundamental assumption made by prior web browser architectures: that software-based isolation can keep sensitive data protected within an operating system process, despite running untrustworthy code within that process. Transient execution attacks have been demonstrated to work from JavaScript code [25, 34, 37], violating the web security model *without requiring any bugs in the browser*. We show that our long-term investment in Site Isolation also provides a necessary mitigation for these unforeseen attacks, though it is not sufficient: complementary OS and hardware mitigations for such attacks are also required to prevent leaks of information from other processes or the OS kernel.

To deploy Site Isolation to users, we needed to overcome numerous performance and compatibility challenges not addressed by prior research prototypes [23, 62, 63, 68]. Locking each sandboxed renderer process to a single site greatly increases the number of processes; we present process consolidation optimizations that keep memory overhead low while preserving responsiveness. We reduce overhead and latency by consolidating painting and input surfaces of contiguous same-site frames, along with parallelizing process creation with network requests and carefully managing a spare process. Supporting the entirety of the web presented additional compatibility challenges. Full support for *out-of-process iframes* requires proxy objects and replicated state in frame trees, as well as updates to a vast number of browser features. Finally, a privileged process must filter sensitive cross-site data without breaking existing cross-site JavaScript files and other subresources. We show that such filtering requires a new type of confirmation sniffing and can protect not just HTML but also JSON and XML, beyond prior discussions of content filtering [23, 63, 68].

With these changes, the privileged browser process can keep most cross-site sensitive data out of a malicious document's renderer process, making it inconsequential for a web attacker to access and exfiltrate data from its address space. While there are a set of limitations with its current implementation, we argue that Site Isolation offers the best path to mitigating the threats posed by compromised renderer processes and transient execution attacks.

In this paper, Section 2 introduces a new browser threat model covering *renderer exploit attackers* and *memory disclosure attackers*, and it discusses the current limitations of Site Isolation's protection. Section 3 presents the challenges we overcame in fundamentally re-architecting a production browser to adopt Site Isolation, beyond prior research browsers. Section 4 describes our implementation, consisting of almost 450k lines of code, along with critical optimizations that made it feasible to deploy to all desktop and laptop users of Chrome. Section 5 evaluates its effectiveness against compromised renderers as well as Spectre and Meltdown attacks. We also evaluate its practicality, finding that it incurs a total memory overhead of 9-13% in practice and increases page load latency by less than 2.25%, while sufficiently preserving compatibility with actual web content. Given the severity of the new threats, Google Chrome has enabled Site Isolation by default. Section 6 looks at the implications for the web's future and potential ways to address Site Isolation's current limitations. We compare to related work in Section 7 and conclude in Section 8.

Overall, we answer several new research questions:

- Which parts of a web browser's security model can be aligned with OS-level isolation mechanisms, while preserving compatibility with the web?
- What optimizations are needed to make process-level isolation of web sites feasible to deploy, and what is the resulting performance overhead for real users?
- How well does process-level isolation of web sites upgrade existing security practices to protect against compromised renderer processes?
- How effectively does process-level isolation of web sites mitigate Spectre and Meltdown attacks, and where are additional mitigations needed?

2 Threat Model

We assume that a *web attacker* can lure a user into visiting a web site under the attacker's control. Multi-process browsers have traditionally focused on stopping web attackers from compromising a user's computer, by rendering untrusted web content in sandboxed renderer processes, coordinated by a higher-privilege browser process [51]. However, current browsers allow attackers to load victim sites into the same renderer process using iframes or popups, so the browser must trust security checks in the renderer process to keep sites isolated from each other.

In this paper, we move to a stronger threat model emphasizing two different *types* of web attackers that each aim to steal data across web site boundaries. First, we consider a *renderer exploit attacker* who can discover and exploit vulnerabilities to bypass security checks or even achieve arbitrary code execution in the renderer process. This attacker can disclose any data in the renderer process's address space, as well as lie to the privileged browser process. For example, they might forge an IPC message to retrieve sensitive data associated with another web site (e.g., cookies, stored passwords). These attacks imply that the privileged browser process must validate access to all sensitive resources without trusting the renderer process. Prior work has shown that such attacks can be achieved by exploiting bugs in the browser's implementation of the Same-Origin Policy (SOP) [54] (known as universal cross-site scripting bugs, or UXSS), with memory corruption, or with techniques such as data-only attacks [5, 10, 11, 33, 53, 63, 68].

Second, we consider a *memory disclosure attacker* who cannot run arbitrary code or lie to the browser process, but who can disclose arbitrary data within a renderer process's address space, even when the SOP would disallow it. This can be achieved using transient execution attacks [8] like Spectre [34] and Meltdown [36]. Researchers have shown specifically that JavaScript code can manipulate microarchitectural state to leak data from within the renderer process [25, 34, 37].[1] While less powerful than renderer exploit attackers, memory disclosure attackers are not dependent on any bugs in web browser code. Indeed, some transient execution attacks rely on properties of the hardware that are unlikely to change, because speculation and other transient microarchitectural behaviors offer significant performance benefits. Because browser vendors cannot simply fix bugs to mitigate cases of these attacks, memory disclosure attackers pose a more persistent threat to the web security model. It is thus important to reason about their capabilities separately and mitigate these attacks architecturally.

2.1 Scope

We are concerned with isolating sensitive web site data from execution contexts for other web sites within the browser. Execution contexts include both *documents* (in any frame) and *workers*, each of which is associated with a *site* principal [52] and runs in a renderer process. We aim to protect many types of content and state from the attackers described above, including the HTML contents of documents, JSON or XML data files they retrieve, state they keep within the browser (e.g., cookies, storage, saved passwords), and permissions they have been granted (e.g., geolocation, camera).

Site Isolation is also able to strengthen some existing security practices for web application code, such as upgrading clickjacking [30] protections to be robust against com-

[1] In some cases, transient execution attacks may access information across process or user/kernel boundaries. This is outside our threat model.

promised renderers, as discussed in Section 5.1. Not all web security defenses are in scope, such as mitigations for XSS [46].

2.2 Limitations

For both types of attackers we consider, Site Isolation aims to protect as much site data as possible, while preserving compatibility. Because we isolate *sites* (i.e., scheme plus registry-controlled domain name [52]) rather than *origins* (i.e., scheme-host-port tuples [54]) per Section 3.1, cross-origin attacks within a site are not mitigated. We hope to allow some origins to opt into origin-level isolation, as discussed in Section 6.3.

Cross-site subresources (e.g., JavaScript, CSS, images, media) are not protected, since the web allows documents to include them within an execution context. JavaScript and CSS files were already somewhat exposed to web attackers (e.g., via XSSI attacks that could infer their contents [26]); the new threat model re-emphasizes not to store secrets in such files. In contrast, cross-site images and media were sufficiently opaque to documents before, suggesting a need to better protect at least some such files in the future.

The content filtering we describe in Section 3.5 is also a best-effort approach to protect HTML, XML, and JSON files, applying only when it can confirm the responses match the reported content type. This confirmation is necessary to preserve compatibility (e.g., with JavaScript files mislabeled as HTML). Across all content types, we expect this filtering will protect most sensitive data today, but there are opportunities to greatly improve this protection with headers or web platform changes [21, 71, 73], as discussed in Section 6.1.

Finally, we rely on protection domains provided by the operating system. In particular, we assume that the OS's process isolation boundary can be trusted and consider cross-process and kernel attacks out of scope for this paper, though we discuss them further in Sections 5.2 and 6.2.

3 Site Isolation Browser Architecture

The *Site Isolation* browser architecture treats each web site as a separate security principal requiring a dedicated renderer process. Prior production browsers used rendering engines that predated the security threats in Section 2 and were architecturally incompatible with putting cross-site iframes in a different process. Prior research browsers proposed similar isolation but did not preserve enough compatibility to handle the full web. In this section, we present the challenges we overcame to make the Site Isolation architecture compatible with the web in its entirety.

3.1 Site Principals

Most prior multi-process browsers, including Chrome, Edge, Safari, and Firefox, did not assign site-specific security principals to web renderer processes, and hence they did not enforce isolation boundaries between different sites at the process level. We advance this model in Chrome by partitioning web content into finer-grained principals that correspond to web *sites*. We adopt the *site* definition from [52] rather than *origins* as proposed in research browsers [23, 62, 63, 68]. For example, an origin `https://bar.foo.example.com:8000` corresponds to a site `https://example.com`. This preserves compatibility with up to 13.4% of page loads that change their origin at runtime by assigning to `document.domain` [12]. Site principals ensure that a document's security principal remains constant after `document.domain` modifications.

For each navigation in any frame, the browser process computes the site from the document's URL, determining its security principal. This is straightforward for HTTP(S) URLs, though some web platform features require special treatment, as we discuss in Appendix A (e.g., `about:blank` can inherit its origin and site).

3.2 Dedicated Processes

Site Isolation requires that renderer processes can be dedicated to documents, workers, and sensitive data from only a single site principal. In this paper, we consider only the case where *all* web renderer processes are locked to a single site. It would also be possible for the browser to isolate only some sites and leave other sites in shared renderer processes. In such a model, it is still important to limit a dedicated renderer process to documents and data from its own site, but it is also necessary to prevent a shared process from retrieving data from one of the isolated sites. When isolating all sites, requests for site data can be evaluated solely on the process's site principal and not also a list of which sites are isolated.

The browser's own components and features must be also partitioned in a way that does not leak cross-site data. For example, the network stack cannot run within the renderer process, to protect `HttpOnly` cookies and so that filtering decisions on cross-site data can be made before the bytes from the network enter the renderer process. Similarly, browser features must not proactively leak sensitive data (e.g., the user's stored credit card numbers with autofill) to untrustworthy renderer processes, at least until the user indicates such data should be provided to a site [49]. These additional constraints on browser architecture may increase the amount of logic and state in more privileged processes. This does not necessarily increase the attack surface of the trusted browser process if these components (e.g., network stack) can move to separate sandboxed processes, as in prior microkernel-like browser architectures [23, 62].

3.3 Cross-Process Navigations

When a document in a frame navigates from site A to site B, the browser process must replace the renderer process for site A with one for site B. This requires maintaining state in the browser process, such as session history for the tab, related window references such as openers or parent frames,

and tab-level session storage [74]. Due to web-visible events such as `beforeunload` and `unload` and the fact that a navigation request might complete without creating a new document (e.g., a download or an HTTP "204 No Content" response), the browser process must coordinate with both old and new renderer processes to switch at the appropriate moment: after `beforeunload`, after the network response has proven to be a new document, and at the point that the new process has started rendering the new page. Note that cross-site server redirects may even require selecting a different renderer process before the switch occurs.

Session history is particularly challenging. Each stop in the back/forward history can contain information about multiple cross-site documents in various frames in the page, and it can include sensitive data for each document, such as the contents of partially-filled forms. To meet the security goals of Site Isolation, this site-specific session history state can only be sent to renderer processes locked to the corresponding site. Thus, the browser process must coordinate session history loads at a frame granularity, tracking which data to send to each process as cross-site frames are encountered in the page being loaded.

3.4 Out-of-process iframes

The largest and most disruptive change for Site Isolation is the requirement to load cross-site iframes in a different renderer process than their embedding page. Most widely-used browser rendering engines were designed and built before browsers became multi-process. The shift to multi-process browsers typically required some changes to these existing engines in order to support multiple instances of them. However, many core assumptions remained intact, such as the ability to traverse all frames in a page for tasks like painting, messaging, and various browser features (e.g., find-in-page). Supporting *out-of-process iframes* is a far more intrusive change that requires revisiting such assumptions across the entire browser. Meanwhile, prior research prototypes that proposed this separation [23, 63, 68] did not address many of the challenges in practice, such as how to ensure the iframe's document knows its position in the frame tree. This section describes the challenges we overcame to make out-of-process iframes functional and compatible with the web platform.

Frame Tree. To support out-of-process iframes, multi-process browser architectures must change their general abstraction level from *page* (containing a tree of frames) to *document* (in a single frame). The browser process must track which document, and thus principal, is present in each frame of a page, so that it can create an appropriate renderer process and restrict its access accordingly. The cross-process navigations described in Section 3.3 must be supported at each level of the frame tree to allow iframes to navigate between sites.

Each process must also keep a local representation of documents that are currently rendered in a different process,

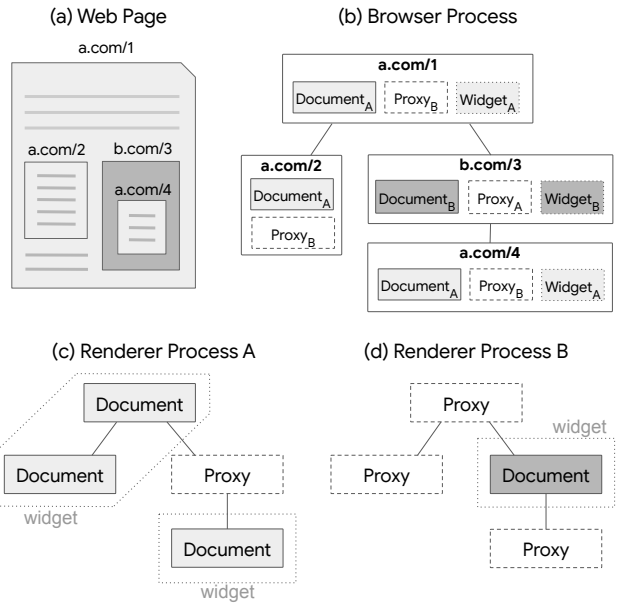

Figure 1: **An example of out-of-process iframes.** To render the web page shown in (a), the browser process (b) coordinates two renderer processes, shown in (c) and (d).

which we call *proxies*. Proxies offer cross-process support for the small set of cross-origin APIs that are permitted by the web platform, as described in [52]. These APIs may be accessed on a frame's `window` object and are used for traversing the frame hierarchy, messaging, focusing or navigating other frames, and closing previously opened windows. Traversing the frame hierarchy must be done synchronously within the process using proxies, but interactions between documents can be handled asynchronously by routing messages. Note that all same-site frames within a frame tree (or other reachable pages) must share a process, allowing them to synchronously script each other.

An example of a page including out-of-process iframes is shown in Figure 1 (a), containing three documents from `a.com` and one from `b.com`, and thus requiring two separate renderer processes. Figure 1 (b) shows the browser process's frame tree, with representations of each document annotated by which site's process they belong to, along with a *set* of proxy objects for each frame (one for each process which might reference the frame). Figure 1 (c-d) shows the corresponding frame trees within the two renderer processes, with proxy objects for any documents rendered in a different process. Note that the actual document and proxy objects live in renderer processes; the corresponding browser-side objects are stubs that track state and route IPC messages between the browser and renderer processes.

For example, suppose the document in `a.com/2` invokes `window.parent.frames["b"].postMessage("msg", "b.com")`. Renderer Process A can traverse its local frame tree to find the parent frame and then its child frame named "`b`", which is a proxy. The renderer process will send the message to the corresponding Proxy_A object for `b.com/3` in

the browser process. The browser process passes it to the current Document$_B$ object in this frame, which sends the message to the corresponding Document object in Renderer Process B. Similar message routing can support other cross-origin APIs, such as focus, navigation, or closing windows.

State Replication. The renderer process may need synchronous access to some types of state about a frame in another process, such as the frame's current name (to find a frame by name, as in the example above) or iframe sandbox flags. As this state changes, the browser process broadcasts it to all proxies for a frame across affected processes. Note that this state should never include sensitive site-specific data (e.g., full URLs, which may have sensitive URL parameters), only what is necessary for the web platform implementation.

Painting and Input. To preserve the Site Isolation security model, the rendered appearance of each document cannot leak to other cross-site renderer processes. Otherwise, an attacker may be able to scrape sensitive information from the visible appearance of frames in other processes. Instead, each renderer process is responsible for the layout and paint operations within each of its frames. These must be sent to a separate process for compositing at the granularity of *surfaces*, to form the combined appearance of the page. The compositing process must support many types of transforms that are possible via CSS, without leaking surface data to a cross-site renderer process.

Often, many frames on a page come from the same site, and separate surfaces for each frame may be unnecessary. To reduce compositing overhead, we use a *widget* abstraction to combine contiguous same-site frames into the same surface. Figure 1 shows how `a.com/1` and `a.com/2` can be rendered in the same widget and surface without requiring compositing. `b.com/3` requires its own widget in Renderer Process B. Since `a.com/4` is not contiguous with the other two `a.com` frames and its layout may depend on properties assigned to it by `b.com/3` (e.g., CSS filters), it has a separate widget within Renderer Process A, and its surface must be composited within `b.com/3`'s surface.

Widgets are also used for input event routing, such as mouse clicks and touch interactions. In most cases, the compositing metadata makes it possible for the browser process to perform sufficient hit testing to route input events to the correct renderer process. In some cases, though, web platform features such as CSS transforms or CSS `pointer-events` and `opacity` properties may make this difficult. Currently, the browser process uses *slow path hit testing* over out-of-process iframes, i.e., asking a parent frame's process to hit-test a specific point to determine which frame should receive the event, without revealing any further details about the event itself. This is only used for mouse and touch events; keyboard events are reliably delivered to the renderer process that currently has focus.

Note that images and media from other sites can be included in a document. The Site Isolation architecture does not try to exclude these from the renderer process, for multiple reasons. First, moving cross-origin image handling out of the renderer process and preventing renderers from reading these surfaces would require a great deal of complexity in practice. Second, this would substantially increase the number of surfaces needed for compositing. This decision is consistent with other research browsers [23, 62, 63], including Gazelle's implementation [68]. Thus, we leave cross-site images and media in the renderer process and rely on servers to prevent unwanted inclusion, as discussed in Section 6.1.

Affected Features. In a broad sense, almost all browser features that interact with the frame tree must be updated to support out-of-process iframes. These features could traditionally assume that all frames of a page were in one process, so a feature like find-in-page could traverse each frame in the tree in the renderer process, looking for a string match. With out-of-process iframes, the browser process must coordinate the find-in-page feature, collecting partial results from each frame across multiple renderer processes. Additionally, the feature must be careful to avoid leaking information to renderer processes (e.g., whether there was a match in a cross-site sibling frame), and it must be robust to renderer processes that crash or become unresponsive.

These updates are required for many features that combine data across frames or that perform tasks that span multiple frames: supporting screen readers for accessibility, compositing PDFs for printing, traversing elements across frame boundaries for focus tracking, representations of the full page in developer tools, and many others.[2]

3.5 Cross-Origin Read Blocking

Loading each site's documents in dedicated renderer processes is not sufficient to protect site data: there are many legitimate ways for web documents to request cross-site URLs within their own execution context, such as JavaScript libraries, CSS files, images, and media. However, it is important not to give a renderer process access to cross-site URLs containing sensitive data, such as HTML documents or JSON files. Otherwise, a document could access cross-site data by requesting such a URL from a <`script`>, <`style`>, or <`img`> tag. The response may nominally fail within the requested context (e.g., an HTML file would produce syntax errors in a <`script`> tag), but the data would be present in the renderer process, where a compromised renderer or a transient execution attack could leak it.

Unfortunately, it is non-trivial to perfectly distinguish which cross-site URLs must be allowed into a renderer process and which must be blocked. It is possible to categorize content types into those needed for subresources and those that are not (as in Gazelle [68]), but content types of re-

[2] A list of these features is included in Appendix B.

sponses are often inaccurate in practice. For example, many actual JavaScript libraries have content types of `text/html` rather than `application/javascript` in practice. Changing the browser to block these libraries from cross-site documents would break compatibility with many existing sites.

It may be desirable to require sites to correct their content types or proactively label any resources that need protection (e.g., with a new `Cross-Origin-Resource-Policy` header [21]), but such approaches would leave many existing resources unprotected until developers update their sites.

Until such shifts in web site behavior occur, browsers with Site Isolation can use a best effort approach to protect as many sensitive resources as possible, while preserving compatibility with existing cross-site subresources. We introduce and standardize an approach called *Cross-Origin Read Blocking (CORB)* [17, 20], which prevents a renderer process from receiving a cross-site response when it has a *confirmed* content type likely to contain sensitive information. CORB focuses on content types that, when used properly, cannot be used in a subresource context. Subresource contexts include scripts, CSS, media, fetches, and other ways to include or retrieve data within a document, but exclude iframes and plugins (which can be loaded in separate processes). CORB filters the following content types:

- HTML, which is used for creating new documents with data that should be inaccessible to other sites.
- JSON, which is used for conveying data to a document.
- XML, which is also often used for conveying data to a document. An exception is made for SVG, which is an XML data type permitted within <`img`> tags.

Since many responses have incorrect content types, CORB requires additional confirmation before blocking the response from the renderer process. In other contexts, web browsers perform MIME-type sniffing when a content type is missing, looking at a prefix of the response to guess its type [4]. OP2 and IBOS use such sniffing to confirm a response is HTML [23, 63], but this will block many legitimate JavaScript files, such as those that begin with HTML comments (i.e., "`<!--`"). In contrast, CORB relies on a new type of *confirmation sniffing* that looks at a prefix of the response to confirm that it matches the claimed content type and not a subresource [17]. For example, a response labeled as `text/html` starting with "`<!doctype`" would be blocked, but one starting with JavaScript code would not. (CORB attempts to scan past HTML comments when sniffing.) This is a default-allow policy that attempts to protect resources where possible but prioritizes compatibility with existing sites. For example, CORB allows responses through when they are polyglots which could be either HTML or JavaScript, such as:

```
<!--/*--><html><body><script type="text/javascript"><!--//*/
var x = "This is both valid HTML and valid JavaScript.";
//--></script></body></html>
```

CORB skips confirmation sniffing in the presence of the existing `X-Content-Type-Options: nosniff` response header, which disables the browser's existing MIME sniffing logic. When this header is present, responses with incorrect content types are already not allowed within subresource contexts, making it safe for CORB to block them. Thus, we recommend that web developers use this header for CORB-eligible URLs that contain sensitive data, to ensure protection without relying on confirmation sniffing.

If a cross-site response with one of the above confirmed content types arrives, and if it is not allowed via CORS headers [18], then CORB's logic in the network component prevents the response data from reaching the renderer process.

3.6 Enforcements

The above architecture changes are sufficient to mitigate *memory disclosure attackers* as described in Section 2. For example, transient execution attacks might leak data from any cross-site documents present in the same process, but such attacks cannot send forged messages to the browser process to gain access to additional data. However, a *renderer exploit attacker* that compromises the renderer process or otherwise exploits a logic bug may indeed lie to the browser process, claiming to be a different site to access its data.

The browser process must be robust to such attacks by tracking which renderer processes are locked to which sites, and thus restricting which data the process may access. Requests for site data, actions that require permissions, access to saved passwords, and attempts to fetch data can all be restricted based on the site lock of the renderer process. In normal execution, a renderer process has its own checks to avoid making requests for such data, so illegal requests can be interpreted by the browser process as a sign that the renderer process is compromised or malfunctioning and can thus be terminated before additional harm is caused. The browser process can record such events in the system log, to facilitate audits and forensics within enterprises.

These enforcements may take various forms. If the renderer process sends a message labeled with an origin, the browser process must enforce that the origin is part of the process's site. Alternatively, communication channels can be scoped to a site, such that a renderer process has no means to express a request for data from another site.

The CORB filtering policy in Section 3.5 also requires enforcements against compromised renderers, so that a renderer exploit attacker cannot forge a request's initiator to bypass CORB. One challenge is that extensions had been allowed to request data from extension-specified sites using scripts injected into web documents. Because these requests come from a potentially compromised renderer process, CORB cannot distinguish them from an attacker's requests. This weakens CORB by allowing responses from any site that an active extension can access, which in many cases is all sites. To avoid having extensions weaken the security

of Site Isolation, we are changing the extension system to require these requests to be issued by an extension process instead of by extension scripts in a web renderer process, and we are helping extension developers migrate to the new approach [9].

4 Implementation

With the Chrome team, we implemented the Site Isolation architecture in Chrome's C++ codebase. This was a significant 5-year effort that spanned approximately 4,000 commits from around 350 contributors (with the top 20 contributors responsible for 72% of the commits), changing or adding approximately 450,000 lines of code in 9,000 files.

We needed to re-architect a widely deployed browser without adversely affecting users, both during development and when deploying the new architecture. This section describes the steps we took to minimize the impact on performance and functionality, while Section 5 evaluates that impact in practice.

4.1 Optimizations

Fundamentally, Site Isolation requires the browser to use a larger number of OS processes. For example, a web page with four cross-site iframes, all on different sites, will require five renderer processes versus one in the old architecture. The overhead of additional processes presents a feasibility risk, due to extra memory cost and process creation latency during navigation. To address these challenges, we have implemented several optimizations that help make Site Isolation practical.

4.1.1 Process Consolidation

Our security model dictates that a renderer process may never contain documents hosted at different sites, but a process may still be shared across separate instances of documents from the same site. Fortunately, many users keep several tabs open, which presents an opportunity for process sharing across those tabs.

To reduce the process count, we have implemented a process consolidation policy that looks for an existing *same-site* process when creating an out-of-process iframe. For example, when a document embeds an `example.com` iframe and another browser tab already contains another `example.com` frame (either an iframe or a main frame), we consolidate them in the same process. This policy is a trade-off that avoids process overhead by reducing performance isolation and failure containment: a slow frame could slow down or crash other same-site frames in the process. We found that this trade-off is worthwhile for iframes, which tend to require fewer resources than main frames.

The same policy could also be applied to main frames, but doing this unconditionally is not desirable: when resource-heavy documents from a site are loaded in several tabs, using a single process for all of them leads to bloated processes that perform poorly. Instead, we use process consolidation for same-site main frames only after crossing a *soft process limit* that approximates memory pressure. When the number of processes is below this limit, main frames in independent tabs don't share processes; when above the limit, all new frames start reusing same-site processes when possible. Our threshold is calculated based on performance characteristics of a given machine. Note that Site Isolation cannot support a hard process limit, because the number of sites present in the browser may always exceed it.

4.1.2 Avoiding Non-essential Isolation

Some web content is assigned to an opaque origin [29] without crossing a site boundary, such as iframes with `data:` URLs or sandboxed *same-site* iframes. These could utilize separate processes, but we choose to keep these cases in-process as an optimization, focusing our attention on true cross-site content.

Other design decisions that help reduce process count include isolating at a site granularity rather than origin, keeping cross-site images in-process, and allowing extensions to share processes with each other. Section 6.3 discusses improving isolation in these cases in the future.

4.1.3 Reducing the Cost of Process Swaps

Section 3.3 implies that many more navigations must create a new process. We mask some of this latency by (1) starting the process in parallel with the network request, and (2) running the old document's `unload` handler in the background after the new document is created in the new process.

However, in some cases (e.g., back/forward navigations) documents may load very quickly from the cache. These cases can be significantly slowed by adding process creation latency. To address this, we maintain a warmed-up *spare* renderer process, which may be used immediately by a new navigation to any site. When a spare process is locked to a site and used, a new one is created in the background, similar to process pre-creation optimizations in OP2 [23]. To control memory overhead, we avoid spare processes on low memory devices, when the system experiences memory pressure, or when the browser goes over the soft process limit.

4.2 Deployment

Shipping Site Isolation in a production browser is challenging. It is a highly disruptive architecture change affecting significant portions of the browser, so enabling it all at once would pose a high risk of functional regressions. Hence, we deployed incrementally along two axes: isolation targets and users. Before launching full Site Isolation, we shipped two milestones to enable process isolation for selective targets:

1. **Extensions.** As the first use of out-of-process iframes from Section 3.4, we isolated *web* iframes embedded inside *extension* pages, and vice versa [50]. This provided a meaningful security improvement, keeping ma-

licious web content out of higher-privileged extension processes. It also affected only about 1% of all page loads, reducing the risk of widespread functional regressions.

2. **Selective isolation.** We created an enterprise policy allowing administrators to optionally isolate a set of manually selected high-value web sites [6].

Deploying these preliminary isolation modes provided a valuable source of bug reports and performance data (e.g., at least 24 early issues reported from enterprise policy users). These modes also show how some form of isolation may be deployed in environments where full Site Isolation may still be prohibitively expensive, such as on mobile devices.

We also deployed each of these milestones incrementally to users. All feature work was developed behind an opt-in flag, and we recruited early adopters who provided bug reports. For each milestone (including full Site Isolation), we also took advantage of Chrome's A/B testing mechanism [13], initially deploying to only a certain percentage of users to monitor performance and stability data.

5 Evaluation

To evaluate the effectiveness and practicality of deploying Site Isolation, we answer the following questions: **(1)** How well does Site Isolation upgrade existing security practices to mitigate renderer exploit attacks? **(2)** How effectively does Site Isolation mitigate transient execution attacks, compared to other web browser mitigation strategies? **(3)** What is the performance impact of Site Isolation in practice? **(4)** How well does Site Isolation preserve compatibility with existing web content? Our findings have allowed us to successfully deploy Site Isolation to all desktop and laptop users of Google Chrome.

5.1 Mitigating Renderer Vulnerabilities

We have added numerous enforcements to Chrome (version 76) to prevent a compromised renderer from accessing cross-site data.[3] This section evaluates these enforcements from the perspective of web developers. Specifically, we ask which existing web security practices have been transparently upgraded to defend against renderer exploit attackers, who have complete control over the renderer process.

New Protections. The following web developer practices were vulnerable to renderer exploit attackers before Site Isolation but are now robust.

- **Authentication.** `HttpOnly` cookies are not delivered to renderer processes, and `document.cookie` is restricted based on a process's site. Similarly, the password manager only reveals passwords based on a process's site.
- **Cross-origin messaging.** Both `postMessage` and `BroadcastChannel` messages are only delivered to processes if their sites match the target origin, ensuring that confidential data in the message does not leak to other compromised renderers. Source origins are also verified so that incoming messages are trustworthy.
- **Anti-clickjacking.** `X-Frame-Options` is enforced in the browser process, and CSP `frame-ancestors` is enforced in the embedded frame's renderer process. In both cases, a compromised renderer process cannot bypass these policies to embed a cross-site document.
- **Keeping data confidential.** Many sites use HTML, XML, and JSON to transfer sensitive data. This data is now protected from cross-site renderer processes if it is filtered by CORB (e.g., has a `nosniff` header or can be sniffed), per Section 3.5.
- **Storage and permissions.** Data stored on the client (e.g., in `localStorage`) and permissions granted to a site (e.g., microphone access) are not available to processes for other sites.

Potential Protections. The Site Isolation architecture should be capable of upgrading the following practices to mitigate compromised renderers as well, but our current implementation does not yet fully cover them.

- **Anti-CSRF.** CSRF [3] tokens remain protected from other renderers if they are only present in responses protected by CORB. `Origin` headers and `SameSite` cookies can also be used for CSRF defenses, but our enforcement implementation is still in progress.
- **Embedding untrusted documents.** The behavioral restrictions of `iframe sandbox` (e.g., creating new windows or dialogs, navigating other frames) and `Feature-Policy` are currently enforced in the renderer process, allowing compromised renderers to bypass them. If sandboxed iframes are given separate processes, many of these restrictions could happen in the browser process.

Renderer Vulnerability Analysis. We also analyzed security bugs reported for Chrome for 2014-2018 (extending the analysis by Moroz et al [41]) and found 94 UXSS-like bugs that allow an attacker to bypass the SOP and access contents of cross-origin documents. Site Isolation mitigates such bugs by construction, subject to the limitations discussed in Section 2.2. Similar analyses in prior studies have also shown that isolating web principals in different processes prevents a significant number of cross-origin bypasses [19, 63, 68].

In the six months after Site Isolation was deployed in mid-2018, Chrome has received only 2 SOP bypass bug reports, also mitigated by Site Isolation (compared to 9 reports in the prior six months). The team continues to welcome and fix such reports, since they still have value on mobile devices where Site Isolation is not yet deployed. We also believe that going forward, attention will shift to other classes of bugs seen during this post-launch period, including:

[3] A list of these enforcements is included in Appendix C.

- **Bypassing Site Isolation.** These bugs exploit flaws in the process assignment or other browser process logic to force cross-site documents to share a process, or to bypass the enforcement logic. For example, we fixed a reported bug where incorrect handling of `blob` URLs created in opaque origins allowed an attacker to share a victim site's renderer process.
- **Targeting non-isolated data.** For example, 14 bugs allowed an attacker to steal cross-site images or media, which are not isolated in our architecture, e.g., by exploiting memory corruption bugs or via timing attacks.
- **Cross-process attacks.** For example, 5 bugs are side channel attacks that rely on timing events that work even across processes, such as a frame's `onload` event, to reveal information about the frame.

In general, we find that Site Isolation significantly improves robustness to renderer exploit attackers, protecting users' web accounts and lowering the severity of renderer vulnerabilities.

5.2 Mitigating Transient Execution Attacks

Transient execution attacks represent *memory disclosure attackers* from Section 2, where lying to the browser process is not possible. Thus, Site Isolation mitigations here depend on process isolation and CORB, but not the enforcements in Section 3.6. This section compares the various web browser mitigation strategies for such attacks, evaluating their effectiveness against known variants.

Strategy Comparison. Web browser vendors have pursued three types of strategies to mitigate transient execution attacks on the web, with varying strengths and weaknesses.

First, most browsers attempted to *reduce the availability of precise timers* that could be used for attacks [14, 39, 48, 67]. This focuses on the most commonly understood exploitation approach for Spectre and Meltdown attacks: a Flush+Reload cache timing attack [75]. This strategy assumes the timing attack will be difficult to perform without precise timers. Most major browsers reduced the granularity of APIs like `performance.now` to 20 microseconds or even 1 millisecond, introduced jitter to timer results, and even removed implicit sources of precise time, such as `SharedArrayBuffers` [59]. This strategy applies whether the attack targets data inside the process or outside of it, but it has a number of weaknesses that limit its effectiveness:

- It is likely incomplete: there are a wide variety of ways to build a precise timer [35, 58], making it difficult to enumerate and adjust all sources of time in the platform.
- It is possible to amplify the cache timing result to the point of being effective even with coarse-grained timers [25, 37, 58].
- Coarsening timers hurts web developers who have a legitimate need for precise time to build powerful web applications. Disabling `SharedArrayBuffers` was a particularly unfortunate consequence of this strategy, since it disrupted web applications that relied on them (e.g., AutoCAD).
- Cache timing attacks are only one of several ways to leak information from transient execution, so this approach may be insufficient for preventing data leaks [8].

As a result, we do not view coarsening timers or disabling `SharedArrayBuffers` as an effective strategy for mitigating transient execution attacks.

Second, browser vendors pursued *modifications to the JavaScript compiler and runtime* to prevent JavaScript code from accessing victim data speculatively [37, 48, 65]. This involved array index masking and pointer poisoning to limit out of bounds access, `lfence` instructions as barriers to speculation, and similar approaches. The motivation for this strategy is to disrupt all "speculation gadgets" to avoid leaking data within and across process boundaries. Unfortunately, there are an increasingly large number of variants of transient execution attacks [8], and it is difficult for a compiler to prevent all the ways an attack might be expressed [37]. This is especially true for variants like Spectre-STL (also known as Variant 4), where store-to-load forwarding can be used to leak data [28], or Meltdown-RW which targets in-process data accessed after a CPU exception [8]. Additionally, some of these mitigations have large performance overheads on certain workloads (up to 15%) [37, 65], which risk slowing down legitimate applications. The difficulty to maintain a complete defense combined with the performance cost led Chrome's JavaScript team to conclude that this approach was ultimately impractical [37, 49].

Site Isolation offers a third strategy. Rather than targeting the cache timing attack or disrupting speculation, Site Isolation assumes that transient execution attacks may be possible within a given OS process and instead attempts to *move data worth stealing outside of the attacker's address space*, much like kernel defenses against Meltdown-US [15, 24].

Variant Mitigation. Canella et al [8] present a systematic evaluation of transient execution attacks and defenses, which we use to evaluate Site Isolation. Spectre attacks rely on branch mispredictions or data dependencies, while Meltdown attacks rely on transient execution after a CPU exception [8]. Table 1 shows how both types of attacks are able to target data inside or outside the attacker's process, and thus both Spectre and Meltdown are relevant to consider when mitigating memory disclosure attacks.

Site Isolation mitigates same-address-space attacks by avoiding putting vulnerable data in the same renderer process as a malicious principal. This targets the most practical variants of transient execution attacks, for which an attacker has a large degree of control over the behavior of the process (relative to attacks that target another process). Site Isolation does not depend on the absence of precise timers for

Attack	Inside Process: Site Isolation	Inside Process: Timers	Inside Process: Compiler	Outside Process: Site Isolation	Outside Process: Timers	Outside Process: Compiler
Spectre-PHT	●	◐	●	○	◐	●
Spectre-BTB	●	◐	●	○	◐	●
Spectre-RSB	●	◐	◐	○	◐	◐
Spectre-STL	●	◐	○	-	-	-
Meltdown-US	-	-	-	○	◐	○
Meltdown-P	-	-	-	○	◐	○
Meltdown-GP	-	-	-	○	◐	○
Meltdown-NM	-	-	-	○	◐	○
Meltdown-RW*	●	◐	○	-	-	-
Meltdown-PK*	●	◐	○	-	-	-
Meltdown-BR*	●	◐	○	-	-	-

Table 1: **Web browser mitigations for Spectre and Meltdown attacks, for targets inside and outside the attacker's process.** Symbols show if an attack is mitigated (●), partially mitigated (◐), not mitigated (○), or not applicable (-). Site Isolation mitigates all applicable same-process attacks, and it depends on other mitigations for cross-process attacks.
** Only affects browsers that use these hardware features.*

mitigating same-process attacks, and it can mitigate attacks like Spectre-STL that are difficult or costly for compilers to prevent [37]. For Meltdown attacks that target same-process data (e.g., Meltdown-RW, which can transiently overwrite read-only data), Site Isolation applies as well. It is less clear whether Meltdown-PK and Meltdown-BR [8] are relevant in the context of the browser, but Site Isolation would mitigate them if browsers used protection keys [38] or hardware-based array bounds checks, respectively.

Site Isolation does not attempt to mitigate attacks targeting data in other processes or the kernel, such as the "Outside Process" variants in Table 1 and Microarchitectural Data Sampling (MDS) attacks [40, 57, 66]. Site Isolation can and must be combined with hardware and OS mitigations for such attacks to prevent web attackers from leaking data across process boundaries or from the kernel. For example, PTI is a widely used mitigation for Meltdown-US, eliminating kernel memory from the address space of each user process [15, 24]. Similarly, microcode updates and avoiding sibling Hyper-Threads for untrustworthy code may be useful for mitigating MDS attacks [40, 57, 66].

Ultimately, cross-process and user/kernel boundaries must fundamentally be preserved by the OS and hardware and cannot be left to applications to enforce. Within a process, however, the OS and hardware have much less visibility into where isolation is needed. Thus, applications that run code from untrustworthy principals (e.g., browsers) must align their architectures with OS-enforced abstractions to isolate these principals. As a result, we have chosen Site Isolation as the most effective mitigation strategy for Chrome. When it is enabled, Chrome re-enables `SharedArrayBuffer` and other precise timers and removes JavaScript compiler mitigations, to restore powerful functionality to the web and regain lost performance.

5.3 Performance

Enabling Site Isolation can affect the browser's performance, so we evaluate its effect on memory overhead, latency, and CPU usage in the wild and in microbenchmarks. We find that the new architecture has low enough overhead to be practical to deploy.

5.3.1 Observed Workload

We first focus on measuring performance in the field, because this more accurately reflects real user workloads (e.g., many tabs, long-tail sites) than microbenchmarks do. The data in this section was collected using pseudonymous metric reporting over a two-week period starting October 1, 2018, from desktop and laptop users of Chrome (version 69) on Windows who have this reporting enabled. We compare results from equal-sized test and control groups within the general user population. (These metrics are enabled by default, but users can opt out during installation or later in settings. Our experimental design and data collection were reviewed under Google's processes.)

Process Count. With Site Isolation, the browser process must create more renderer processes to keep sites isolated from each other: at least as many as unique sites open at a time. Using periodic samples, we found that users had 6.0 unique sites open across the entire browser at the 50th percentile of the distribution, and 41.9 unique sites at the 99th percentile. This only provides a lower bound for the number of renderer processes; each instance of a site might live in a separate process. If this were the case, our metrics give an upper bound estimate of 79.7 processes at the 99th percentile. However, thanks to the process sharing heuristics described in Section 4.1.1, far fewer processes were used in practice, as shown in Figure 2. At the 50th percentile, the number of processes increased 43.5% from 4.4 without Site Isolation to 6.2 with Site Isolation. At the 99th percentile, the process count increased 50.6% from 35.0 to 52.7 processes. This indicates that many more processes are needed for Site Isolation, but also that the process consolidation heuristics greatly reduce the count at the upper percentiles.

Memory Overhead. On its own, the 50% increase in renderer process count is significant, but this does not necessarily translate to an equivalent increase in memory overhead or performance slowdowns. Site Isolation is effectively dividing an existing workload across more processes, so each renderer process is correspondingly smaller and shorter lived. In reported metrics, we found that private memory use per renderer process decreased 51.5% (87.2 MB to 42.3 MB) at

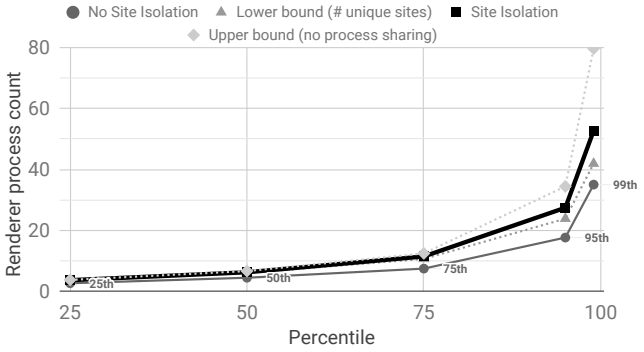

Figure 2: **Renderer process count.** This graph shows the number of renderer processes before and after Site Isolation, as well as an estimated lower and upper bound on process count, controlled by the amount of process sharing for instances of the same site. Site Isolation finds a middle ground between no process sharing and having at most one process per site.

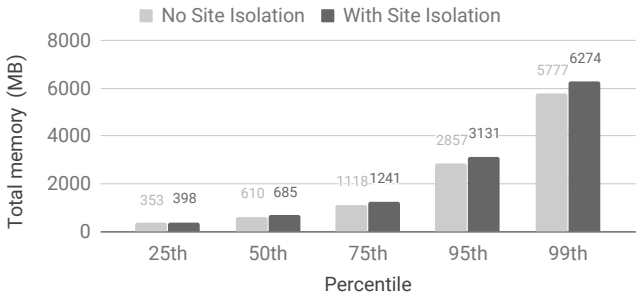

Figure 3: **Total browser memory usage across all processes.** Overall, Site Isolation has a 9-13% overhead.

the 50th percentile and 28.6% (from 714.2 MB to 509.7 MB) at the 99th percentile. Renderer process lifetime decreased 4.3% at the 50th percentile and 55.5% at the 99th percentile.

This leaves an open question about the overhead of each process relative to the workload of the process, which determines the total memory use. Figure 3 compares the total private memory use across all processes (including browser process, renderer processes, and other types of utility processes) with and without Site Isolation. In practice, we see that total memory use increased only 12.6% at the 25th percentile, and only 8.6% at the 99th percentile. This is significantly lower than the 50% increase in process count might suggest, indicating that the large number of extra processes has a relatively small impact on the total memory use of the browser. We confirmed that this is not due to a change in workload size: there were no statistically significant differences in page load count, and we saw at most a 1.5% decrease in the number of open tabs (at the 99th percentile).

Due to the severity of transient execution attacks and the drawbacks of other mitigation strategies in Section 5.2, the Chrome team was willing to accept 9-13% memory overhead for the security benefits of enabling Site Isolation.

Latency. Site Isolation also impacts latency in multiple ways, from the time it takes to load a page to the responsiveness of input events. On one hand, more navigations need to create new processes, which can incur latency due to process startup time. There may also be greater contention for IPC messages and input event routing, leading to some delays. On the other hand, there is a significant amount of new parallelism possible now that the workload for a given page can be split across multiple independent threads of execution. We use observed metrics from the field to study the combined impact of these changes in practice.

Site Isolation significantly increased the percentage of navigations that cross a process boundary, from 5.73% to 56.0%. However, we mask some of the latency of process creation in Chrome by starting the renderer process in parallel with making the network request. Combined with the increased parallelism of loading cross-site iframes in different processes, we see very little change to a key metric for page load time: the time from navigation start to the first paint of page content (e.g., text, images, etc) [22]. Across all navigations, we observe this to increase at most 2.25% at the 25th percentile (457 ms to 467 ms) and 1.58% (14.6 s to 14.8 s) at the 99th percentile. This metric also benefits from the spare process optimization described in Section 4.1.3, which avoids the process startup latency on many navigations. Without the spare process, this "First Contentful Paint" time increases 5.1% at the 25th percentile and 2.4% at the 99th percentile.

If we look closer at various types of navigations, the most significantly affected category is back/forward navigations, which frequently load pages from the cache without waiting for the network. This eliminates most of the benefit of parallelizing process startup with the network request. Here, we see time to First Contentful Paint increase 28.3% (177 ms to 227 ms) at the 25th percentile and 6.8% (4637 ms to 4952 ms) at the 99th percentile. Again, this is better than without using a spare process, in which case we see increases of 40.7% and 12.5% at these percentiles, respectively.

We also looked at the latency impact on input events. The current implementation uses slow path hit testing for mouse and touch events over out-of-process iframes, which results in small increases to input event latency. For key presses, there are no statistically significant differences at the 50th or 99th percentiles, and only a 1.0% latency increase at the 75th percentile (43.6 ms to 44.0 ms). For mouse scroll update events, latency increased 1.3% (21.8 ms to 22.1 ms) at the 50th percentile and 8.6% (228.8 ms to 248.6 ms) at the 99th percentile. For touch scroll update events, latency increased 2.6% (18.4 ms to 18.9 ms) and 10.7% (134.0 ms to 148.3 ms) at these percentiles. We expect to improve these by updating hit testing to avoid the slow path in most cases.

CPU Usage. Finally, we study the impact of Site Isolation on CPU usage. Average CPU usage in the browser process increased 8.2% (32.0% to 34.6%) at the 99th percentile,

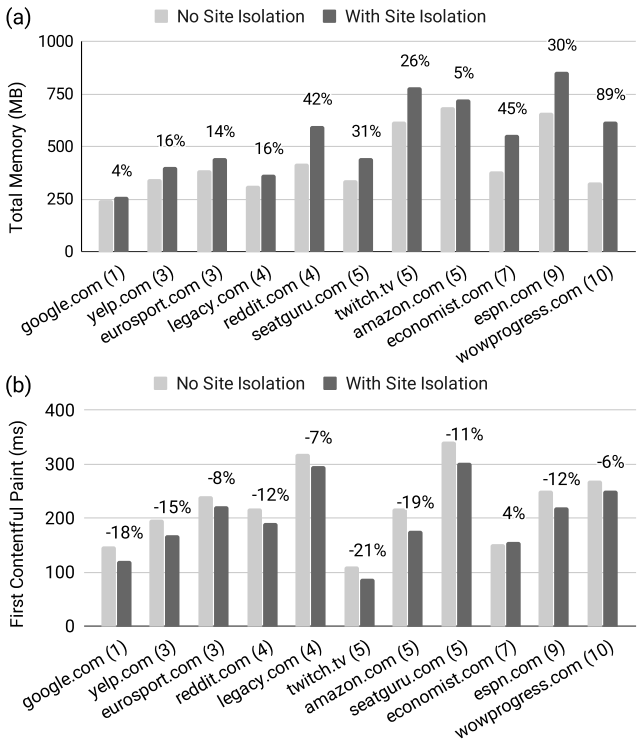

Figure 4: **(a) Total browser memory usage and (b) Time to First Contentful Paint for individual sites.** Parentheses denote the number of renderer processes required to load each site with Site Isolation. Without Site Isolation, each site requires one renderer process.

due to additional IPC messages and coordination across processes. While there were more renderer processes, each renderer's average CPU usage dropped 33.5% (47.7% to 31.8%) at the 99th percentile, since the workload was distributed across more processes.

Overall, we found that enabling Site Isolation had a much smaller performance impact than expected due to the properties of the workload. Given the importance of mitigating the attacks in the threat model described in Section 2, the Chrome team has chosen to keep Site Isolation enabled for all users on desktop and laptop devices.

5.3.2 Microbenchmarks

We also report microbenchmark results showing the overhead of Site Isolation on individual web pages when loaded in a single tab, with nothing else running in the browser. This setup does not benefit from process consolidation across multiple tabs as discussed in Section 4.1.1, and hence it is not representative of the real-world workloads used in the previous section. However, these measurements establish a baseline and provide a reproducible reference point for future research.

To study a mix of the most popular (likely highly optimized) and slightly less popular sites, we selected the top site as well as the 50th-ranked site in Alexa categories for news, sports, games, shopping, and home, as well as `google.com` as the top overall URL.[4] This set provides pages with a range of cross-site iframe counts, showing how the browser scales with more processes per page.

Next, we started Chrome version 69.0.3497.100 with a clean profile, and we loaded each site in a single tab, both with and without Site Isolation. We report the median of five trials for each data point to reduce variability, and we replayed recorded network data for all runs using WprGo [69]. Our experiments were performed on a Windows 10 desktop with an Intel Core i7-8700K 3.7 GHz 6-core CPU and 16 GB RAM. Our data collection script is available online [45].

Figure 4 (a) shows the total browser memory use for each site, sorted by the number of renderer processes (shown in parentheses) that each site utilizes when loaded with Site Isolation. As expected, the relative memory overhead generally increases with the number of processes, peaking at 89% for `wowprogress.com` with 10 processes. Sites that use more memory tend to have smaller relative overhead, as their memory usage outweighs the cost of extra processes. For example, a heavier `amazon.com` site has a 5% overhead compared to `seatguru.com`'s 31%, even though both require five processes. `google.com` does not have any cross-site iframes and requires no extra processes, but it shows a 4% increase in memory use due to the spare process that we maintain with Site Isolation, as explained in Section 4.1.3.

The overhead seen in these results is significantly higher than the 9-13% overhead we reported from real-world user workloads in the previous section. This underscores the limitations of microbenchmarks: users tend to have multiple tabs (four at 50th percentile) and a variety of open URLs. In practice, this helps reduce memory overhead via process consolidation, while iframe-heavy sites like `wowprogress.com` may represent only a small part of users' browsing sessions.

Figure 4 (b) shows time to First Contentful Paint [22] for each site, to gauge impact on page load time. Most paint times improve with Site Isolation because the spare process helps mask process startup costs, which play a larger role than network latency due to the benchmark's use of recorded network traffic. The speedups are not correlated with process counts; Site Isolation offloads some of the work from the main frame into iframe renderers, which may make the main frame more responsive regardless of process count.

5.4 Compatibility

Site Isolation strives to avoid web-visible changes. For example, we found that CORB blocks less than 1% of responses, most of which are not observable; if it only relied on content type and not confirmation sniffing, it would block 20% of responses [17]. Also, since cross-origin frame interactions had been mostly asynchronous prior to our work, making these interactions cross-process is largely transpar-

[4] If a site's main content required logging in, we picked the next highest-ranked site.

ent to web pages. During deployment, we closely monitored bug reports for several months to judge the impact on actual users and content. We have received around 20 implementation bugs, most of which are now fixed. We did uncover some behavior changes, described below. Overall, however, none of the bug reports warranted turning Site Isolation off, indicating that our design does not result in major compatibility problems when deployed widely.

Asynchronous Full-page Layout. With Site Isolation, full-page layout is no longer synchronous, since the frames of a page may be spread across multiple processes. For example, if a page changes the size of a frame and then sends it a `postMessage`, the receiving frame may not yet know its new size when receiving the message. We found that this disrupted behavior for some pages, but since the HTML spec does not guarantee this behavior and relatively few sites were affected, we chose not to preserve the old ordering. Instead, we provided guidance for web developers to fix the few affected pages [7] and are pursuing specification changes to explicitly note that full-page layout is asynchronous [27].

Partial Failures. Site Isolation can expose new failure modes to web pages, because out-of-process iframes may crash or become unresponsive independently from their embedder, after having been loaded. Although this may lead to unexpected behavior in the page, it happens rarely enough to avoid being a problem in practice, and for users, losing an iframe is usually preferable to losing the entire page.

Detecting Site Isolation. A web page should not know if it is rendered with or without Site Isolation, and we have avoided introducing APIs for doing so: a browser's process model is an implementation detail that developers should not depend on. We did encounter and fix some bugs that allowed detection of Site Isolation, such as differing JavaScript exception behavior for in-process and out-of-process frames. Fundamentally, though, it is possible to detect Site Isolation via timing attacks. For example, a cross-process `postMessage` will take longer than a same-process `postMessage`, due to an extra IPC hop through the browser process; a web page could perform a timing analysis to detect whether a frame is in a different process. However, such timing differences are unlikely to affect compatibility, and we have not received any such reports.

6 Future Directions

Site Isolation protects a great deal of site data against renderer exploit attackers and memory disclosure attackers, but there is a strong incentive to address the limitations outlined in Section 2.2.

It is worth noting that web browsers are not alone in facing a new security landscape. Other software systems that isolate untrustworthy code may require architecture changes to avoid leaking data via microarchitectural state. For example, SQL queries in databases might pose similar risks [47]. Applications that download and render untrustworthy content from the web, such as document editors, should likewise leverage OS abstractions to isolate their own principals [42].

6.1 Protecting More Data

CORB currently only protects HTML, XML, and JSON responses, and only when the browser can confirm them using sniffing or headers. There are several options for protecting additional content, from using headers to protect particular responses, to expanding CORB to cover more types, to changing how browsers request subresources.

First, web developers can explicitly protect sensitive resources without relying on CORB, using a `Cross-Origin-Resource-Policy` response header [21] or refusing to serve cross-site requests based on the `Sec-Fetch-Site` request header [71].

Second, the Chrome team is working to isolate cross-site PDFs and other types [2, 60]. Developer outreach may also cut down on mislabeled subresources, eliminating the need for CORB confirmation sniffing.

Third, recent proposals call for browsers to make cross-origin subresource requests without credentials by default [73]. This would prevent almost all sensitive cross-site data from entering a renderer process, apart from cases of ambient authority (e.g., intranet URLs which require no credentials).

These options may close the gaps to ensure essentially all sensitive web data is protected by Site Isolation.

6.2 Additional Layers of Mitigation

Because Site Isolation uses OS process boundaries as an isolation mechanism, it is straightforward to combine it with additional OS-level mitigations for attacks. This may include other sandboxing mechanisms (e.g., treating different sites as different user accounts) or mitigations for additional types of transient execution attacks. For example, microcode updates and OS mitigations (e.g., PTI or disabling HyperThreading) may be needed for cross-process or user/kernel attacks [15, 24, 40, 57, 66]. These are complementary to the mitigations Site Isolation offers for same-process attacks, where the OS and hardware have less visibility.

6.3 Practical Next Steps

Mobile Devices. This paper has described deploying Site Isolation to users on desktop and laptop devices, but the new web attackers are important to consider for mobile phone browsers as well. Site Isolation faces greater challenges on mobile devices due to fewer device resources (e.g., memory, CPU cores) and a different workload: there are fewer renderer processes in the working set due to proactive discarding by the mobile OS, and thus fewer opportunities for process sharing. We are investigating options for deploying similar mitigations on mobile browsers, such as isolating a subset of sites that need the protection the most.

Isolation in Other Browsers. There are opportunities for other browsers to provide a limited form of process isolation without the significant implementation requirements of out-of-process iframes. For example, sites might adopt headers like `Cross-Origin-Opener-Policy` to opt into a mode that can place a top-level document in a new process by disrupting some cross-window scripting [44].

Origin Isolation. Within browsers with Site Isolation, further isolation may be practical by selectively moving from a site granularity to a finer origin granularity. Too many web sites rely on modifying `document.domain` to deploy origin isolation by default, but browsers may allow sites to opt out of this feature and thus become eligible for origin isolation [72]. Making this optional may reduce the impact on the process count. Similarly, we plan to evaluate the overhead impact of isolating opaque origins, especially to improve security for sandboxed same-site iframes.

Performance. Finally, there are performance opportunities to explore to reduce overhead and take advantage of the new architecture. More aggressive renderer discarding may be possible with less cross-site sharing of renderer processes. Isolating cross-origin iframes from some web applications may also provide performance benefits by parallelizing the workload, moving slower frames to a different process than the primary user interface to keep the latter more responsive.

7 Related Work

Prior to this work, all major production browsers, including IE/Edge [76], Chrome [52], Safari [70], and Firefox [43], had multi-process architectures that rendered untrustworthy web content in sandboxed renderer processes, but they did not enforce process isolation between web security principals, and they lacked architectural support for rendering embedded content such as iframes out-of-process. Site Isolation makes Chrome the first widely-adopted browser to add such support. Other research demonstrated a need for an architecture like Site Isolation by showing how existing browsers are vulnerable to cross-site data leaks, local file system access via sync from cloud services, and transient execution attacks [25, 33, 53].

Several research browsers have proposed isolating web principals in different OS processes, including Gazelle [68], OP and its successor OP2 [23, 62], and IBOS [63]. Compared to these proposals, Site Isolation is the first to support the web platform in its entirety, with practical performance and compatibility. First, these proposals all define principals as origins, but this cannot support pages that change `document.domain` [12]. Other research browsers isolate web applications with principals that are similarly incompatible: Tahoma [16] uses custom manifests, while SubOS [31, 32] uses full URLs that include path in addition to origin. To preserve compatibility, we adopt the *site* principal proposed in [52]; this also helps reduce process count compared to origins. Second, we describe new optimizations that make Site Isolation practical, and we evaluate our architecture on a real workload of Chrome users. This shows that Site Isolation introduces almost no additional page load latency and only 9-13% memory overhead, lower than expected from microbenchmark evaluations. Third, we comprehensively evaluate the implications of new transient execution attacks [8] for browser security. Fourth, we show that protecting cross-origin network responses requires new forms of confirmation sniffing to preserve compatibility; content types and even traditional MIME sniffing are insufficient. Finally, while Gazelle, OP2, and IBOS have out-of-process iframes, our work overcomes many challenges to support these in a production browser, such as supporting the full set of cross-process JavaScript interactions, challenges with painting and input event routing, and updating affected features (e.g., find-in-page, printing).

The OP and OP2 browsers [23, 62] also use OS processes to isolate other browser components, including the network stack, storage, and display. Such additional process separation is orthogonal to Site Isolation and offers complementary benefits, such as making the browser more modular, reducing the size of the browser process, and keeping crashes in one component isolated from the rest of the browser.

Dong et al [19] argued that practical browser designs will require a trade-off between finer-grained isolation and performance. Our experience echoes this finding, and we indeed make trade-offs to reduce memory overhead, such as isolating sites rather than origins. Dong et al's evaluation relied on sequentially browsing top Alexa sites; we additionally collect measurements from browsing workloads in the wild, providing a more realistic performance evaluation. For example, this factors in process sharing across multiple tabs, which significantly reduces overhead in practice.

Other researchers propose disabling risky JavaScript features unless user-defined policies indicate they are safe for a desired site [56, 61]. These approaches aim to disrupt a wide variety of attacks (including microarchitectural), but they impose barriers to adoption of powerful web features, and they rely on users or third parties to know when features are safe to enable. Site Isolation's scope is more limited by compatibility, but it does not require actions from users or disabling powerful features.

8 Conclusion

The web browser threat model has changed significantly. Web sites face greater threats of data leaks within the browser due to compromised renderer processes and transient execution attacks. Site Isolation offers the best path to mitigating these attacks in the browser, protecting a significant amount of site data today with future opportunities to expand the coverage. We have shown that Site Isolation is practical to deploy in a production desktop web browser, incurring a 9-13% total memory overhead on real-world work-

loads. We recommend that web developers and browser vendors continue down this path, protecting additional sensitive resources, adding more mitigations, and pursuing similar isolation in environments like mobile browsers.

9 Acknowledgements

We would like to thank Łukasz Anforowicz, Jann Horn, Ken Buchanan, Chris Palmer, Adrienne Porter Felt, Franziska Roesner, Tadayoshi Kohno, Antoine Labour, Artur Janc, our shepherd Adam Doupé, and the anonymous reviewers for their input on this paper. We also thank the many Chrome team members who made this work possible.

References

[1] Adobe. Flash & The Future of Interactive Content. https://theblog.adobe.com/adobe-flash-update/, 2017.

[2] L. Anforowicz. More CORB-protected MIME types - adding protected types one-by-one. https://github.com/whatwg/fetch/issues/860, Jan. 2019.

[3] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *CCS*, 2008.

[4] A. Barth, D. Song, and J. Caballero. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. In *IEEE Symposium on Security and Privacy*, 2009.

[5] A. Barth, J. Weinberger, and D. Song. Cross-origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *USENIX Security*, 2009.

[6] M. Blumberg. Security enhancements and more for enterprise Chrome browser customers. https://www.blog.google/products/chrome-enterprise/security-enhancements-and-more-enterprise-chrome-browser-customers/, Dec. 2017.

[7] M. Bynens. Site Isolation for web developers. https://developers.google.com/web/updates/2018/07/site-isolation, July 2018.

[8] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*, 2019.

[9] Changes to Cross-Origin Requests in Chrome Extension Content Scripts. https://www.chromium.org/Home/chromium-security/extension-content-script-fetches, Jan. 2019.

[10] S. Chen, H. Chen, and M. Caballero. Residue Objects: A Challenge to Web Browser Security. In *EuroSys*, 2010.

[11] S. Chen, D. Ross, and Y.-M. Wang. An Analysis of Browser Domain-Isolation Bugs and A Light-Weight Transparent Defense Mechanism. In *CCS*, 2007.

[12] Chrome Platform Status: DocumentSetDomain. https://www.chromestatus.com/metrics/feature/popularity#DocumentSetDomain, Dec. 2018.

[13] Chromium Blog: Changes to the Field Trials infrastructure. https://blog.chromium.org/2012/05/changes-to-field-trials-infrastructure.html, May 2012.

[14] Chromium Security: Mitigating Side-Channel Attacks. https://www.chromium.org/Home/chromium-security/ssca, Jan. 2018.

[15] J. Corbet. The current state of kernel page-table isolation. https://lwn.net/Articles/741878/, Dec. 2017.

[16] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A Safety-Oriented Platform for Web Applications. In *IEEE Symposium on Security and Privacy*, 2006.

[17] Cross-Origin Read Blocking (CORB). https://chromium.googlesource.com/chromium/src/+/master/services/network/cross_origin_read_blocking_explainer.md, Mar. 2018.

[18] Cross-Origin Resource Sharing (CORS). https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS, 2019.

[19] X. Dong, H. Hu, P. Saxena, and Z. Liang. A Quantitative Evaluation of Privilege Separation in Web Browser Designs. In *ESORICS*, 2013.

[20] Fetch Standard: CORB. https://fetch.spec.whatwg.org/#corb, May 2018.

[21] Fetch Standard: Cross-Origin-Resource-Policy header. https://fetch.spec.whatwg.org/#cross-origin-resource-policy-header, Jan. 2019.

[22] First Contentful Paint. https://developers.google.com/web/tools/lighthouse/audits/first-contentful-paint, 2019.

[23] C. Grier, S. Tang, and S. T. King. Designing and Implementing the OP and OP2 Web Browsers. *TWEB*, 5:11, May 2011.

[24] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. KASLR is Dead: Long Live KASLR. In *ESSoS*, 2017.

[25] N. Hadad and J. Afek. Overcoming (some) Spectre browser mitigations. https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/, June 2018.

[26] V. Hailperin. Cross-Site Script Inclusion. https://www.scip.ch/en/?labs.20160414, Apr. 2016.

[27] C. Harrelson. Adjust event loop processing model to allow asynchronous layout of frames. https://github.com/whatwg/html/issues/3727, May 2018.

[28] J. Horn. Speculative Execution, Variant 4: Speculative Store Bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[29] HTML Living Standard: opaque origin. https://html.spec.whatwg.org/multipage/origin.html#concept-origin-opaque, Jan. 2019.

[30] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson. Clickjacking: Attacks and Defenses. In *USENIX Security*, 2012.

[31] S. Ioannidis and S. M. Bellovin. Building a secure web browser. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[32] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-operating systems: a new approach to application security. In *Proceed-*

ings of the 10th SIGOPS European workshop, 2002.

[33] 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*, 2016.

[34] 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 *IEEE Symposium on Security and Privacy*, 2019.

[35] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *ESORICS*, 2017.

[36] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[37] R. McIlroy, J. Sevcík, T. Tebbi, B. L. Titzer, and T. Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019.

[38] Memory Protection Keys for Userspace. `https://www.kernel.org/doc/Documentation/x86/protection-keys.txt`, Jan. 2019.

[39] Microsoft Edge Team. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer. `https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/`, Jan. 2018.

[40] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. V. Bulck, D. Genkin, D. Gruss, B. Sunar, F. Piessens, and Y. Yarom. Fallout: Reading Kernel Writes From User Space. `https://mdsattacks.com`, 2019.

[41] M. Moroz and S. Glazunov. Analysis of UXSS exploits and mitigations in Chromium. Technical report, Google, 2019. `https://ai.google/research/pubs/pub48028`.

[42] A. Moshchuk, H. J. Wang, and Y. Liu. Content-based Isolation: Rethinking Isolation Policy Design on Client Systems. In *CCS*, 2013.

[43] N. Nguyen. The Best Firefox Ever. `https://blog.mozilla.org/blog/2017/06/13/faster-better-firefox/`, 2017.

[44] R. Niwa. Restricting cross-origin WindowProxy access (Cross-Origin-Opener-Policy). `https://github.com/whatwg/html/issues/3740`, June 2018.

[45] N. Oskov. Site Isolation Benchmark Script. `https://github.com/naskooskov/site-isolation-benchmark`, May 2019.

[46] OWASP. XSS (Cross Site Scripting) Prevention Cheat Sheet. `https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.md`, Feb. 2019.

[47] C. Palmer. Isolating Application-Defined Principals. `https://noncombatant.org/application-principals/`, July 2018.

[48] F. Pizlo. What Spectre and Meltdown Mean For WebKit. `https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/`, Jan. 2018.

[49] Post-Spectre Threat Model Re-Think. `https://chromium.googlesource.com/chromium/src/+/master/docs/security/side-channel-threat-model.md`, May 2018.

[50] C. Reis. Improving extension security with out-of-process iframes. `https://blog.chromium.org/2017/05/improving-extension-security-with-out.html`, May 2017.

[51] C. Reis, A. Barth, and C. Pizano. Browser Security: Lessons from Google Chrome. *Commun. ACM*, 52(8):45–49, Aug. 2009.

[52] C. Reis and S. D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *EuroSys*, 2009.

[53] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis. Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses. In *IEEE European Symposium on Security and Privacy*, 2017.

[54] J. Ruderman. The Same Origin Policy. `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`, 2019.

[55] J. Schuh. The Final Countdown for NPAPI. `https://blog.chromium.org/2014/11/the-final-countdown-for-npapi.html`, 2014.

[56] M. Schwarz, M. Lipp, and D. Gruss. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *NDSS*, 2018.

[57] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. `https://zombieloadattack.com`, 2019.

[58] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security*, Jan 2017.

[59] SharedArrayBuffer. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer`, 2019.

[60] Site Isolate PDFium. `https://crbug.com/809614`, Jan. 2019.

[61] P. Snyder, C. Taylor, and C. Kanich. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *CCS*, 2017.

[62] S. Tang, S. T. King, and C. Grier. Secure Web Browsing with the OP Web Browser. In *IEEE Symposium on Security and Privacy*, 2008.

[63] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *OSDI*, 2010.

[64] D. Topic. Moving to a Plugin-Free Web. `https://blogs.oracle.com/java-platform-group/moving-to-a-plugin-free-web`, Jan. 2016.

[65] Untrusted code mitigations. `https://v8.dev/docs/untrusted-code-mitigations`, Jan. 2018.

[66] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue In-flight Data Load. In *IEEE Symposium on Security and Privacy*, 2019.

[67] L. Wagner. Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/, Jan. 2018.

[68] 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 *USENIX Security*, 2009.

[69] Web Page Replay. https://github.com/catapult-project/catapult/blob/master/web_page_replay_go/README.md, Sept. 2017.

[70] WebKit2. https://trac.webkit.org/wiki/WebKit2, July 2011.

[71] M. West. Fetch Metadata Request Headers. https://mikewest.github.io/sec-metadata, 2018.

[72] M. West. Proposal: Control over 'document.domain'. https://github.com/w3c/webappsec-feature-policy/issues/241, Nov. 2018.

[73] M. West. Incrementally Better Cookies. https://mikewest.github.io/cookie-incrementalism/draft-west-cookie-incrementalism.html, May 2019.

[74] Window.sessionStorage. https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage, 2019.

[75] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[76] A. Zeigler. IE8 and Loosely-Coupled IE (LCIE). https://blogs.msdn.microsoft.com/ie/2008/03/11/ie8-and-loosely-coupled-ie-lcie/, 2008.

A Determining Site Principals

This appendix provides additional details on how we define principals used in Site Isolation. Figure 5 compares principal definitions in monolithic browsers, multi-process browsers that isolate coarser-grained groups of principals, Site Isolation, and Origin Isolation. Origin Isolation, where principals are defined as origins, offers stronger security guarantees at the cost of breaking `document.domain` compatibility and performance challenges due to a larger number of principals.

As noted in Section 3.1, computing site URL for most HTTP(S) URLs is straightforward, but some web platform features require special treatment. For example, frames may be navigated to `about:blank`, a special URL which must inherit the security origin, and hence the site, from the frame initiating the navigation. The web also supports *nested URLs* such as `blob:` URLs. These URLs embed an origin; e.g., `blob:http://example.com/UUID` addresses an in-memory blob of data controlled by the `http://example.com` origin. In these cases, we extract the inner origin from the URL and then convert it to a site.

A document may also embed a frame and specify its HTML content *inline* rather than from the network, either using the `srcdoc` attribute (e.g., `<iframe srcdoc="<html>content</html>">`) or a `data:` URL (e.g., `data:text/html,<html>content</html>`). `Srcdoc` frames inherit their creator's origin and must stay in the principal of their embedding document. In contrast, `data:` URLs load in an opaque origin [29], which cannot be accessed from any other origin. Browsers may choose to load each `data:` URL in its own separate principal and process, but our current implementation uses the creator's principal (which typically controls the content) to reduce the number of processes required. Similarly, our current implementation keeps same-site iframes with the `sandbox` attribute, which typically load in an opaque origin, in the principal of their URL's site. In practice, sites often use sandboxed iframes for untrustworthy content that they wish to isolate from the rest of the site; we discuss opportunities for finer-grained isolation within a site in Section 6.3.

Non-web Principals. Many browsers can load documents that do not originate from the web, including content from local files, extensions, browser UI pages, and error pages. These forms of content utilize the web platform for rendering, so the browser must define principals for them. Each local URL (e.g., `file:///homes/foo/a.html`) is typically treated as its own origin by the browser, so each path could use a separate principal and process. Our current implementation treats all local files as part of the same *file* principal to reduce the process count, since they ultimately belong to a local user. We may revise this to isolate each file in the future, since this group of local files may contain less trustworthy pages saved from the web.

We assign content from extensions to a separate shared principal, and we isolate all browser UI pages, such as settings or download manager, from one another. These pages require vastly different permissions and privileges, and a compromise of one page (e.g., a buggy extension) should not be able to take advantage of permissions granted to a more powerful page (e.g., a download management page that can download and open files). We do allow extensions to share processes with each other to reduce the process count; thus, Figure 5 (c) shows extensions in a shared principal. However, extensions never share processes with other types of pages.

B Features Updated to Support Out-of-process iframes

This appendix lists a subset of Chrome features that needed to be updated to support out-of-process iframes, beyond those discussed in Section 3.4.

- Accessibility (e.g., screen readers).
- Developer tools.
- Drag and drop.
- Extensions (e.g., injecting scripts into frames of a page).
- Find-in-page.

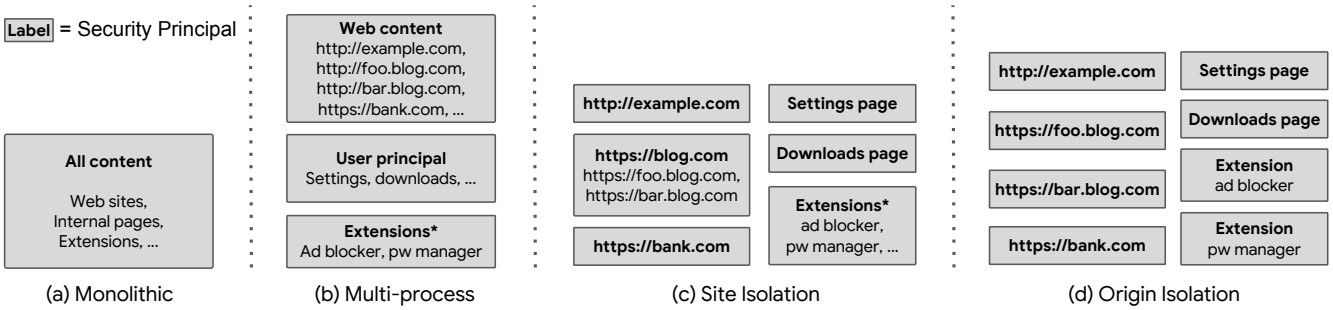

Figure 5: **Evolution of security principals in browser architectures.** Compared to prior browser architectures, Site Isolation defines finer-grained principals that correspond to sites. Origin Isolation (d) further refines sites to origins and is the most desirable principal model in the long term, but backward compatibility and performance challenges currently limit its practicality.
** In pre-Site-Isolation browsers (b), extensions were isolated in higher-privileged processes, but with a caveat: extensions could embed web URL iframes which would stay in the extension's process. With Site Isolation (c), process sharing across the web/extension boundary is no longer possible, though extensions may still share a process with one another.*

- Focus (e.g., tracking focused page and frame, focus traversal when pressing Tab).
- Form autofill.
- Fullscreen.
- IME (Input Method Editor).
- Input gestures.
- JavaScript dialogs.
- Mixed content handling.
- Multiple monitor and device scale factor support.
- Password manager.
- Pointer Lock API.
- Printing.
- Task manager.
- Resource optimizations (e.g., deprioritizing offscreen content).
- Malware and phishing detection.
- Save page to disk.
- Screen Orientation API.
- Scroll bubbling.
- Session restore.
- Spellcheck.
- Tooltips.
- Unresponsive renderer detector and dialog.
- User gesture tracking.
- View source.
- Visibility APIs.
- Webdriver automation.
- Zoom.

C Compromised Renderer Enforcements

This appendix lists the current places that privileged browser components in Chrome (version 76) limit the behavior of a renderer process based on its associated site, to mitigate compromised renderers.

- Cookie reads and writes (`document.cookie`, `HttpOnly` cookies).
- Cross-Origin Read Blocking implementation [20].
- `Cross-Origin-Resource-Policy` blocking [21].
- Frame embedding (`X-Frame-Options`).
- JavaScript code cache.
- Messaging (`postMessage`, `BroadcastChannel`).
- Password manager, Credential Management API.
- Storage (`localStorage`, `sessionStorage`, `indexedDB`, `blob` storage, Cache API, WebSQL).
- Preventing web page access to `file://` URLs.
- Web permissions (e.g., geolocation, camera).

We expect the following enforcements to be possible as well, with additional implementation effort.

- Address bar origin.
- Custom HTTP headers requiring CORS.
- Feature Policy.
- iframe sandbox behaviors.
- `Origin` Header and CORS implementation.
- SameSite cookies.
- `Sec-Fetch-Site` [71].
- User gestures.