Browser history re:`visited`

Michael Smith[†] *Craig Disselkoen*[†] *Shravan Narayan*[†]
Fraser Brown[⋆] *Deian Stefan*[†]

[†]*UC San Diego* [⋆]*Stanford University*

Abstract

We present four new history sniffing attacks. Our attacks fit into two classical categories—visited-link attacks and cache-based attacks—but abuse new, modern browser features (e.g., the CSS Paint API and JavaScript bytecode cache) that do not account for privacy when handling cross-origin URL data. We evaluate the attacks against four major browsers (Chrome, Firefox, Edge, and IE) and several security-focused browsers (ChromeZero, Brave, FuzzyFox, DeterFox, and the Tor Browser). Two of our attacks are effective against all but the Tor Browser, whereas the other two target features specific to Chromium-derived browsers. Moreover, one of our visited-link attacks (CVE-2018-6137) can exfiltrate history at a rate of 3,000 URLs per second, an exfiltration rate that previously led browser vendors to break backwards compatibility in favor of privacy. We hope that this work will lead browser vendors to further reconsider the design of browser features that handle privacy-sensitive data.

1 Introduction

Browsing history can reveal a lot about a person: their age, gender, location, political leanings, preferred adult sites—even who they are in the real world [57, 58]. And one user's browsing history can spill *other* users' secrets, thanks to social networking websites like Facebook and LinkedIn [53]. Anyone who touches a search bar should care about safeguarding this sensitive data.

In principle it should be straightforward; after all, the web platform provides no direct means for JavaScript to read out a user's history. In practice, things get more complicated. Browsers still allow web developers to perform a restricted (and occasionally dangerous) set of computations on history data. For example, using the CSS `:visited` and `:link` selectors, developers can conditionally style a link based on whether its destination URL appears in the user's browsing history. And what a developer can do, an attacker can do too—so browsers must account for all kinds of abuse, like exploiting CSS selectors as *side channels* to "sniff" a URL for visited status.

As early as 2002, attackers discovered ways of detecting whether a `:visited` selector matched a given link element; by pointing the link's destination to a URL of interest, they could leak whether a victim had visited that URL [4–6, 10, 23]. Many popular websites put these attacks into production, actively profiling their visitors; a developer could even purchase off-the-shelf history sniffing "solutions" [24]. Once browsers closed these holes, attackers discovered that they could abuse different browser features, like the `MozAfterPaint` event or the `requestAnimationFrame` API, to steal the same data, or could use `:visited` selectors to trick their victims into giving information away [5, 49, 57]; attackers also learned to leak history information through timing channels based on browsers' caching of embedded resources (e.g., images or scripts) [7, 14, 53].

In response, browser vendors continue to plug leaks ad-hoc, as they are discovered. At the same time, they rush to support new features and new APIs to accommodate new classes of applications, from video games to IoT to virtual and augmented reality. Of course, to ensure that new applications run with reasonable performance, browser vendors also continuously add new caches and optimizations. This increasingly complex piping introduces more joints from which history data may leak—from the CSS Paint API to the JavaScript bytecode cache—and on-demand plumbing won't keep up with the flow forever.

In this paper we present new history sniffing attacks that abuse the complexities of modern browsers. We demonstrate: **(1)** three visited-link attacks, abusing new browser features which give attackers a range of capabilities to operate on sensitive history data, from executing arbitrary JavaScript code in the rendering pipeline with the CSS Paint API [50], to composing complex graphical computations using CSS and SVG; and **(2)** a cache-timing attack that abuses Chrome's new JavaScript bytecode

cache [18]. We evaluate our attacks against four major browsers (Chrome, Firefox, Edge, and Internet Explorer) and five security-focused browsers (ChromeZero, Brave, FuzzyFox, DeterFox, and the Tor Browser). Two of our attacks target features specific to the Chrome family of browsers while the other two are more general. Our attacks can exfiltrate history data on all browsers except the Tor Browser, and our attack on the CSS Paint API even does so at the high rate of 3,000 URLs per second. To our knowledge, this is the fastest visited-link attack since Janc and Olejnik's 2010 attack [23]; Google assigned the new attack CVE-2018-6137 and awarded a $2,000 bounty.

While browser vendors have already begun to plug these individual leaks, new features and caches will continue to allow attackers to steal sensitive information. This need not be the case. Much like browsers enforce the same-origin policy (SOP) in a principled way—ensuring that one *origin*[1] cannot read sensitive data from another origin—they could similarly build architectural protections around history data. As first a step in this direction, we propose to **(1)** associate the referring origin with all persistent URL data, including history and cache entries, and **(2)** only expose this data to code—whether web applications or core browser components—running on behalf of the same origin.

In the next section, we give a brief review of history sniffing and related work. Then, we describe our attacks and the browser features that enable them: the CSS Paint API (Section 3.1), CSS 3D Transforms (Section 3.2), `fill`-coloring SVG images (Section 3.3), and the JavaScript bytecode cache (Section 4). In Section 5 we evaluate these attacks on different browsers, and describe a principled approach to eliminating them altogether.

2 Background and related work

Browsers keep track of the URLs that their users visit in order to **(1)** help those users recognize sites that they have already visited (e.g., by marking familiar links with a different color), and **(2)** speed up browsing by caching resources to avoid network requests. Unfortunately, *web attackers* [1] can exploit this saved state to learn users' private browsing habits. We describe two such history sniffing attacks below.

Visited-link attacks Browsers let developers style links according to history data: a developer can use the CSS `:visited` selector to write style rules that only apply to link elements pointing to previously-visited URLs. For example, the following CSS rules color links blue when unvisited and purple otherwise:

```
/* Default link color to blue: */
a { color: blue; }

/* Turn visited links purple: */
a:visited { color: purple; }
```

Through JavaScript, a developer can query any element's computed style properties by calling its `getComputedStyle` method, which returns data such as `{color: "purple"}`. Previously, calling this on a link element styled as above directly leaked whether or not the user had visited that link's destination URL [10]. Browsers also permitted targeting arbitrary styles to visited links with the `:visited` selector: based on a link's (secret) visited status, an attacker could permute its appearance with CSS in ways the attacker could then observe through JavaScript.

In response to real-world history sniffing attacks of this kind [12, 24], major browsers adopted a pair of mitigations [4, 5, 46, 57]. First, they addressed *explicit leaks* through `getComputedStyle` by lying about the computed style of a link: the method now always returns the unvisited version of the link's style. Second, they addressed *implicit leaks* by limiting `:visited` link styling to colors—which are supposed to be unobservable by JavaScript—and updating browser layout engines to cache links' computed styles where possible, in place of re-calculating them, in an effort to avoid timing attacks.

Weinberg et al. [57] demonstrated that these mitigations are not enough—that web attackers can still creatively leak history information. They used interactive tasks (e.g., CAPTCHAs) to trick users into disclosing history information, inferred the color of links from screen reflections in webcam images, and used re-paint events—at the time directly exposed to JavaScript—to observe when a link's visited status changed according to an update of its destination URL. After browser vendors responded by removing the functionality exposing re-paint events to JavaScript, Paul Stone showed how attackers could still detect re-paints through a straightforward timing attack [49]. Our work continues in this tradition, using a variety of modern browser features to build new visited-link attacks which are *fast* (leak the visited status of many URLs per second), *reliable* (work across different browsers and operating systems), *invisible* (conceal their presence), and *automated* (require no special interaction from the victim).

Browser-cache attacks Browsers rely on many layers of caching to speed up web applications; by caching a resource like an HTML document or a video, browsers avoid the overhead of re-fetching that resource the second

[1] *Origins* are the security principals of the web, designated by protocol-host-port triples (e.g., `https://www.example.com:443`). For brevity, we elide the protocol and port throughout the paper.

time a user visits a page. Most browsers use only the resource URL to index cache entries, and do not take into account the origin of the page embedding the resource [22]. As Felten and Schneider showed in 2000 [14], this allows a web attacker at `https://evil.com` to perform a cross-origin request to `https://fb.com`, say, and to learn if the user has visited the other site by measuring the duration of that request. The request is faster if the resource from `https://fb.com` is already in the cache, and slower if the browser must fetch it over the network—measurably so, if the target resource is sufficiently large.

Van Goethem et al. [53] show how more recent browser features—the Application Cache [54] and Service Workers APIs [43]—can also leak history data and other private user information. Kim et al. [27] use the Quota Management API [59] for similar attacks; these efforts are part of a broader class of web privacy cache-based attacks [7, 15, 17, 31, 32, 40, 41, 44, 56, 58]. We present our own cache-based attack, with particularly strong reliability, precision, and applicability to a wide range of target sites.

3 Visited-link attacks on history

In this section we describe three related attacks on visited links that reveal user browsing history to an attacker. These "re-paint" attacks each exploit a fundamental vulnerability in how modern browsers handle visited links: by forcing the browser to re-paint according to a link's visited status and measuring when re-paint events occur, an attacker can learn whether or not the URL pointed to by the link has been visited. Attackers can detect the visited status of arbitrary, exact URLs, including path information (so they can distinguish `https://usenix.org/conference/woot-2018` from `https://usenix.org`). Moreover, they can do this without being perceived by the victim.

3.1 Abusing the CSS Paint API

First, we show how an attacker can sniff history data using the CSS Paint API. The CSS Paint API, introduced in 2018, lets websites hook into the browser's rendering pipeline and draw parts of HTML elements themselves—for example, to fill the background of a web page with a repeating checkerboard pattern that adapts to any window size and display resolution. By detecting when these hooks are invoked, an attacker can observe when the browser re-paints a link element on the page. Toggling a link between destination URLs causes the link to be re-painted if its visited status changes, so the attacker can infer whether or not those URLs have been visited.

Background The CSS Paint API allows a developer to plug dynamically-generated graphics into any context where CSS would normally accept a static image [50]. It does so using "paint worklets" (or "paintlets"), small JavaScript programs that run in their own, self-contained execution contexts. Paintlets all contain a `paint` callback, a JavaScript function that accepts as arguments a handle to a drawing canvas, the desired image dimensions, and a read-only set of properties and their values. To create the checkerboard background we mentioned above, the developer makes a paintlet with a `paint` callback that loops through the canvas width and height to draw evenly-spaced squares. Within the paintlet script, they then use the `registerPaint` function to associate their paintlet with a custom identifier like `checkers`. Then, in their CSS file, the developer sets the page's background image to `paint(checkers)`, where `paint` is the CSS command for referencing a paintlet. Now the browser will show the checkerboard pattern when a user navigates to the developer's page, and if the user re-sizes the window, the pattern will automatically adjust to fit. This is because, for each element the `checkers` paintlet is set up to draw, the browser invokes the paintlet's `paint` callback whenever it detects a "paint invalidation" on that element: any event that might change how the canvas is rendered (e.g., when the element is initially created or when its dimensions change).

Attack An attacker can use the CSS Paint API to observe whether a URL was visited by **(1)** crafting a link element that gets re-painted only if its associated URL is visited and **(2)** using a timing channel (leaking information through the timing of browser operations) to determine whether the re-paint took place. Suppose an attacker wants to determine if a victim has visited `https://ashleymadison.com`. First, the attacker chooses a dummy link that they know the victim has *not* visited (e.g., `https://dummy.com`; or a randomly-generated URL). The attacker then creates a link element pointing to the dummy destination and sets the link's background image to be rendered by a paintlet:

```
<a id="target" href="https://dummy.com">link</a>

<style>
#target {background-image: paint(myEvilPainter);}
</style>
```

When the browser initially draws the link, it will invoke `myEvilPainter`'s `paint` callback.

Later, in a normal (non-paintlet) script, the attacker switches the link's destination to the actual target URL (without changing the displayed link text): `target.href = "https://ashleymadison.com"`. If neither the dummy, known-unvisited URL nor the target

URL appear in the victim's history, the link's visited status starts at `false` and stays that way after switching its destination. However, if the target URL *does* appear in the victim's history, then the link's `visited` property changes to `true`—causing a paint invalidation on the link element. The paint invalidation forces the browser to re-paint the link, and thus invoke `myEvilPainter`'s `paint` callback for a second time.

Counting the calls to the `paint` callback tells the attacker whether or not the target URL has been visited by the victim: two calls indicates visited, and one call, unvisited. Counting these invocations is difficult, however, because paintlets run in their own separate context, with a minimal set of capabilities: they cannot make network requests, communicate with other scripts, or use most other APIs typically available to JavaScript. Moreover, browsers ensure that the pixels they draw cannot be read back through JavaScript, and even prevent paintlets from preserving state across multiple executions [47].

Despite these constraints, the attacker can detect browser re-paints using an event-loop timing channel [56]. Specifically, in the `paint` callback, they introduce a loop that runs for twenty milliseconds and blocks the JavaScript event loop. Since the event loop is shared, code running in the page can directly observe this:

```
var start = performance.now();
// ... change link URL & block on paintlet's paint
// callback ...
var delta = performance.now() - start;
if (delta > threshold) {
  alert('Victim visited Ashley Madison!');
}
```

The longer `delta` indicates that the change from dummy URL to target URL caused a re-paint, which in turn means that the victim visited the target URL.

We note that re-paints (e.g., as triggered by an attacker) are not instantaneous—they are queued and handled when the browser renders the "next" frame [38]. Since today's browsers render web pages at a (target) speed of 60 frames per second, this puts an upper bound on the rate of re-paints—and thus the bandwidth (rate of URLs tested per second) of re-paint attacks. Using a single target link, for example, means that our attack can—at best—exfiltrate 60 URLs per second.

Amplified attack We consider an alternative, amplified attack that uses multiple link elements, each pointing to a different target URL.[2] This attack consists of two phases: **(1)** a *record* phase that uses a paintlet to scan target URLs and stow away their visited statuses by abusing the `registerPaint` function; and **(2)** an *exfiltrate* phase that uses another paintlet to "read" the visited status bits and communicate them—via a CSS covert channel—to a normal, non-paintlet attacker script. We describe these two phases below.

Record. In this phase, the attacker first generates a unique identifier string for each target URL (e.g., `ashleyMadison` for `https://ashleymadison.com`). As in our previous attack, the attacker then sets the background of each link element to be rendered by a paintlet. But, in this attack, the `paint` callback does not block the event loop to leak the visited status of the URL. Instead, the callback uses `registerPaint` to associate the paintlet with a new identifier: the old identifier suffixed with `_visited`. This ensures that if the victim visited `https://ashleymadison.com`, for example, the `ashleyMadison_visited` identifier is associated with a paintlet—but not otherwise. In the exfiltration phase we use this to leak the visited status of the URL.

We note that although the attacker could create a paintlet for each URL, with a different identifier baked into the code of each paintlet, our record phase only needs a single paintlet:

```
paint (ctx, geometry, properties) {
  // Get identifier (e.g., ashleyMadison):
  var iden = properties.get('font-family');
  // Associate this paintlet with tweaked
  // identifier (e.g., ashleyMadison_visited):
  registerPaint(`${iden}_visited`, ...);
}
```

By setting the CSS `font-family` style of a link to the URL identifier, the attacker can communicate to the `paint` callback which URL the re-paint is running on behalf of and avoid creating thousands of paintlets.

Exfiltrate. After feeding all the target URL batches through their set of link elements, the attacker creates a new paintlet to check which possible `*_visited` identifiers were registered—corresponding to the set of visited target URLs. To this end, the paintlet calls `registerPaint` with each possible identifier. If the identifier has already been registered (during the record phase), `registerPaint` throws an exception to complain about the duplicate call. The paintlet catches these exceptions and, for each already-registered identifier, registers a *new* unique identifier based on the old:

```
try {
  // Try to associate identifier
  // (e.g., ashleyMadison_visited):
  registerPaint(iden, ...);
} catch (e) {
  // Create new identifier from the old
  // (e.g., ashleyMadison_exfiltrate):
  var newIden = iden.replace('_visited',
                             '_exfiltrate');
```

[2] We find using 1,024 links at a time provides optimal bandwidth. The attacker can feed a longer list of target URLs through their set of link elements in batches of 1,024, allocating each link one target URL per re-paint.

```
  // Associate new identifier with paintlet:
  registerPaint(newIden, ...);
}
```

For example, if the identifier `ashleyMadison_visited` was registered in the record phase, this paintlet will now additionally register `ashleyMadison_exfiltrate`.

The attacker detects these new paintlet identifier registrations using a quirk of the CSS Paint API implementation. Right before creating the "exfiltrate" paintlet, the attacker inserts a series of elements into the page—one per possible `*_exfiltrate` identifier—styling each with CSS of the following form:

```
#ashleyMadison_element::after
{ content: paint(ashleyMadison_exfiltrate); }
```

This CSS code selects the HTML element with identifier `ashleyMadison_element` and inserts a new image element as its last child (via the `::after` "pseudo-element" selector feature [34]). To draw this image, the `content` rule specifies that the browser should invoke the `paint` callback of the paintlet registered to the `ashleyMadison_exfiltrate` identifier; the browser gets around to handling this after the attacker's "exfiltrate" paintlet runs. And now, the quirk: if this paintlet identifier was *just* registered, then the browser calculates a large width value for the child image element. Otherwise—if `ashleyMadison_exfiltrate` was not *just* registered, or was not registered at all—the image element gets a small width. The attacker can loop back through the `*_element` elements and check their widths: a large width corresponds to a visited URL.

Evaluation Our amplified attack can probe a user's browsing history at 3,000 URLs per second without the victim noticing, i.e., we can scan Alexa Internet's list of Top 100,000 Websites [2] in 30-40 seconds—in the background, with no visible effect on the page, and with no interaction required from the victim.[3] This rate is comparable to the original `:visited` attacks that led browser vendors to break backwards compatibility in order to address them [6].

Google Chrome is the only browser that has implemented the CSS Paint API, so it is also the only browser affected by this attack; we successfully performed the attack on Chrome 66.0.3359.117 under Windows, macOS, and Ubuntu Linux. In response, Google assigned CVE-2018-6137 to our report and rewarded a $2,000 bug bounty. Their patch for the Chrome 67 release includes an interim fix that disables the CSS Paint API on link elements and their children.

[3]The Alexa list compresses down to only 650 KB, compared to the median compressed web page size of 1500 KB [21], leaving plenty of room for even longer target URL lists to be downloaded to victims' browsers without arousing suspicion.

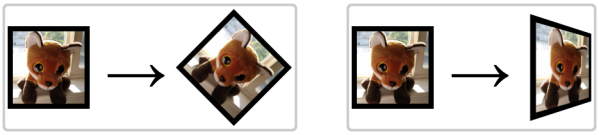

Figure 1: Examples of CSS `transform` on an image element: first, rotating it 45 degrees clockwise in the 2D plane; then, rotating it 45 degrees clockwise around its vertical (Y) axis.

3.2 Abusing CSS 3D transforms

Our attack using paintlets showed how browsers leak history when JavaScript hooks directly into the rendering pipeline, but attackers can also exploit more *indirect* leaks in web page rendering. The next attack takes advantage of CSS 3D transforms, which developers can use to translate, rotate, project, and even animate HTML elements in a three-dimensional space [19, 28, 30]. An attacker stacks these 3D transforms on top of other CSS effects to create a link element that the victim's browser struggles to draw. Then the attacker repeatedly toggles the link element between two different destination URLs and, using the `:visited` selector, forces the browser to complete expensive re-paint operations when the link changes visited status. The presence of these re-paints leaks whether the destination URLs were visited by the victim, information that the attacker harvests by monitoring the page's rendering performance through JavaScript.

Background As of CSS version 3, developers can mark up HTML elements with transformations [36]. To rotate an image 45 degrees clockwise, two-dimensionally, the developer would write this CSS `transform` rule:

```
#photo { transform: rotateZ(45deg); }
```

They can even embed multiple commands in a single `transform` rule by specifying an ordered list of transformations for the browser to apply. Adding the `perspective()` command further enables 3D transformations, like `rotateY()`, which rotates an element around its vertical axis:

```
#photo
{ transform: perspective(100px) rotateY(45deg); }
```

Figure 1 shows the results of these transformations. They can even be combined with other CSS post-processing effects: the `filter` rule, for example, offers contrast, saturation, drop-shadow, and other visual adjustments one would expect to find in photo-editing software [35].

Attack As in Section 3.1, an attacker wishing to detect whether their victim has visited `https://ashleymadison.com` first creates a link pointing to a known-unvisited dummy URL (e.g.,

https://dummy.com). Then the attacker takes advantage of CSS 3D transforms and other post-processing effects to increase the burden on the browser when it re-draws the link:

```
#target {
  transform: perspective(100px) rotateY(37deg);
  filter:
    contrast(200%)
    drop-shadow(16px 16px 10px #fefefe)
    saturate(200%);
  text-shadow: 16px 16px 10px #fefffe;
  outline-width: 24px;
  font-size: 2px; text-align: center;
  display: inline-block;

  color: white;
  background-color: white;
  outline-color: white;
}
```

We experimented with different effects to come up with the formulation above, which includes filter, shadow, and outline styles; in our implementation of the attack, we also fill the link's display text with a long random string of Chinese characters. The specific combination of properties is less important than the idea that the attacker makes the element difficult for the browser to render by layering on computationally intensive effects.

The attacker wants to force the victim's browser to redo these computations when the link's visited status changes. A modern browser will perform them once—albeit relatively slowly—when it initially draws the link, and then reuse the rendered result unless the link's computed style changes. But the attacker can tie the link's computed style to its visited status via the CSS `:visited` and `:link` selectors. This should be the end of the road for this attack: in 2010, browser vendors limited these CSS selectors to only support style rules related to color change, in order to prevent Clover's original visited-link attack. The fix works because simply changing an element's color should be too quick for JavaScript to detect under normal circumstances [46]. In this case, though, changing the link's color means that the browser has to redo all the expensive transformations and post-processing effects that the attacker applied on top of the link element. So the attacker writes a `:visited` style for the link specifying a different set of color values for when its visited status is `true`:

```
a:visited {
  color: #feffff;
  background-color: #fffeff;
  outline-color: #fffffe;
}
```

Next, the attacker toggles the link's destination URL from https://dummy.com to https://ashleymadison.com via JavaScript, as in Section 3.1's attack. Suppose the victim has, in fact, visited https://ashleymadison.com. Changing the link's destination switches its visited status from `false` to `true`, which in turn changes its color values from white to near-white (thanks to the `:visited` styles); as a result, the victim's browser must re-draw the link from scratch with the new colors. The particular color choices here make the link invisible to the victim when placed against a white background, and the change from white to near-white is similarly imperceptible; the link could even be hiding in the background of this paper, and the reader would be none the wiser.

Swapping the link's destination URL once—toggling its visited status—causes the victim's browser to perform a costly computation, holding up the page's rendering cycle while it completes. Doing so *repeatedly*—ping-ponging back and forth between https://dummy.com and https://ashleymadison.com—causes paint performance for the containing page to drop significantly (but not for the rest of the browser, or for scrolling, thanks to modern browsers' parallel, multi-process architectures). If the attacker measures this performance drop, they learn that their victim is an Ashley Madison user. On the other had, if https://ashleymadison.com is unvisited like https://dummy.com, then swapping the link between these destination URLs *doesn't* change its visited status, *doesn't* change its color values, *doesn't* force the victim's browser to repeat the expensive rendering computations—and *doesn't* yield a corresponding drop in the page's paint performance.

To harvest from this leak, the attacker needs to monitor the overall paint performance of the page while they repeatedly toggle their link's destination URL; there are many ways to accomplish this from JavaScript. We chose to use the `requestAnimationFrame` API, an API intended to allow JavaScript code to drive fluid animations. When a developer passes a callback to this function, the browser invokes the callback right before painting the next frame for the page. Browsers aim to invoke the callback about sixty times per second (the same as the page's overall frame rate) but will fall behind if paint performance for the page drops [38]. The attacker can take advantage of this by taking two measurements, each over a fixed time window. First, the attacker takes a control measurement c for oscillating between two different known-unvisited dummy URLs (e.g., https://dummy.com and https://dummy2.com). The attacker gathers c by cyclically registering callbacks to `requestAnimationFrame` and recording the number of times the browser invokes them. Then, using the same procedure, the attacker takes the experimental measurement e for oscillating between the target URL https://ashleymadison.com and the dummy. An e sig-

nificantly lower than c reflects the paint performance drop that the attacker is looking for, signifying that the victim has visited `https://ashleymadison.com`.

One could imagine a mitigation for this attack that simply avoids re-calculating a link element's style based on a change in destination URL alone; however, this mitigation would be ineffective. In an alternate version of this attack, the attacker leaves the link's destination URL constant, pointing to `https://ashleymadison.com`, while rapidly toggling the colors in the `:visited` style rules via JavaScript. If `https://ashleymadison.com` is visited, this technique produces the same effect as before: the link's color values rapidly change, triggering expensive re-draw operations. If it is unvisited, the color values of the link remain set to those specified outside the `:visited` styles. The end result is still the same: the paint performance leaks the user's history.

Evaluation We find our attack effective in up-to-date versions of Chrome, Firefox, Edge, and Internet Explorer, on Windows, macOS, and Ubuntu Linux; vendors assigned our bug reports high- and medium-priority security classifications. Running the attack on a visited URL (e.g., `https://ashleymadison.com`) causes 70-80% fewer `requestAnimationFrame` callbacks to fire when compared to an unvisited URL like `dummy.com`. We are able to perform the attack with a measurement window of 100ms, producing measurements on the order of 4 callback invocations versus 15. This suggests that an attacker using an intelligent search strategy could quickly test many URLs of interest. As mentioned in the attack description, the right color choices make the HTML elements used by the attack invisible to the victim against the page background, and the drop in paint performance exploited would generally only show up as a slow-down of any animations embedded in the containing page.

The most similar attack to this is Paul Stone's 2013 visited-link attack [49]. His attack follows a more traditional timing attack structure, where the attacker **(1)** either changes a link's destination once or inserts a group of link elements into the page, **(2)** measures the time of the next 1-4 rendered frames, and **(3)** checks for any delay caused by the browser re-painting the link(s) as visited [49]. Testing this attack on the same browsers and operating systems as our attacks, we find it still works in Firefox on macOS and Linux (not Windows)—a testament to the difficulty of plugging these leaks. Our attack makes this challenge yet more difficult: instead of timing the duration of a single re-paint, our attack forces *many* (expensive) re-paints and measures the effect on the page's frame rate.

3.3 Abusing `fill`-coloring of SVGs

Even without CSS 3D transforms, we can still tie the visited status of a link to expensive rendering operations; here, we use SVG images and the CSS `fill` rule. The SVG format describes an image as a series of declarative component parts—rectangles, circles, drawing paths, gradients—that browsers rasterize (render to pixels) as needed. An SVG image embedded in a web page scales to arbitrary sizes and display resolutions automatically, without loss of quality [33]. From the containing page, a developer can use CSS to reach into an SVG image and style elements within it—most notably, change their fill-color with the `fill` style rule [37]. Applying such color updates to complex SVG images becomes very expensive for the rendering engine.

Attack An attacker embeds a complicated SVG image inside a link element and sets a series of CSS `fill` rules under `:visited` selectors, specifying different color values for the image's components depending on the link's visited status:

```
<a href="https://dummy.com">
  <svg xmlns="http://www.w3.org/2000/svg">
  ... embedded SVG image data (verbose XML) ...
  </svg>
</a>
```

In our attack, we construct an SVG image that is extremely expensive to render: the image consists of many (7,000+) complex `path` elements—color-filled polygons—arranged in multiple layers. A more clever attacker can use an SVG image that optimizes the attack bandwidth.

Next, the attacker sets up a series of CSS `fill` rules targeting their SVG image, followed by another series using the `:visited` selector on the containing link—applying one color pattern when the link's visited status is `true` and another when it's `false`:

```
a svg * {fill: #ffffff;}
a svg *:nth-child(3n+1) {fill: #fffffe;}
a svg *:nth-child(3n+2) {fill: #fffeff;}

a:visited svg * {fill: #feffff;}
a:visited svg *:nth-child(3n+1) {fill: #fefffe;}
a:visited svg *:nth-child(3n+2) {fill: #fefeff;}
```

The color values were selected so that the SVG image is invisible on a white background, and so that swapping between the two color patterns is imperceptible to the victim.

The remainder of the attack proceeds exactly as in Section 3.2: the attacker uses JavaScript to rapidly switch either the link's destination URL or the color values of the `:visited` style rules, in order to force many expensive re-paint operations in the case that their chosen target

URL (e.g., `https://ashleymadison.com`) has been visited by the user. Simultaneously, the attacker monitors the overall rendering performance of the page (e.g., with `requestAnimationFrame`) and compares it with a control measurement to infer the visited status of the URL.

Evaluation This attack is successful against Chrome, Firefox, Edge, and Internet Explorer, and was reported to vendors together with the attack in Section 3.2. As in Section 3.2, we can make visited determinations over a measurement period of 100ms (on the order of 2 callback invocations versus 5). The signal grows more and more powerful with longer periods (9 versus 41 at 1000ms, for example). Here, too, the attacker hides elements used in the attack from their victim by picking color values which blend in with the page background.

4 Bytecode-cache attacks on history

Beyond the visited-link mechanism—at the root of the attacks described so far—other modern browser features also leak history data. Browser optimizations that share resources between origins (e.g., caches) may let an attacker probe these resources for traces left behind by pages of different origins. In particular, we examine Chrome's JavaScript bytecode cache (added in 2015). This cache retains the bytecode generated by the JavaScript engine when it compiles and runs a script. If the script must be executed again later, the JavaScript engine can use the cached bytecode instead of re-compiling the script [18].

By probing the bytecode cache, an attacker can reliably determine whether a victim's browser has previously executed a particular script file—therefore inferring details of the victim's browsing history. In fact, the attacker can detect past script executions even after the victim restarts their browser or machine, since Chrome persists its bytecode cache to disk.

Background Say a developer embeds the script `foo.js` on their website. When a user visits the developer's website, their browser downloads `foo.js`, but cannot immediately begin executing it. The browser must first parse the JavaScript code in `foo.js` (since scripts are distributed in source form), and then compile it to bytecode suitable for driving the JavaScript engine. This initial "boot up" phase eats up a big chunk of time thanks to JavaScript's flexible syntax and semantics. Moreover, if the developer embeds `foo.js` across multiple pages of their website, the performance cost gets worse and worse. With every click and page load, the browser repeats the same work to boot up `foo.js`.

Chrome avoid this repeated work with its "bytecode cache" optimization, which targets repeatedly-executed scripts exceeding a predefined size threshold [18, 42]. If `foo.js` fits this description, then upon executing it for the third time, Chrome's JavaScript engine will stow away the generated bytecode to an on-disk cache entry keyed by `foo.js`'s URL. For subsequent executions, Chrome skips the usual boot up phase and reads the bytecode from cache.

Attack Because Chrome shares each script's bytecode cache entry between pages of different origins, an attacker can infer history information by measuring how long Chrome takes to boot up a given script. Imagine that the attacker wants to know whether their victim has visited `https://ashleymadison.com`. The attacker selects a script embedded by its homepage, e.g., `https://ashleymadison.com/foo.js`. The selected script should be large enough that prior visits to Ashley Madison would have caused the victim's browser to generate a bytecode cache entry for the script. Then, the attacker invisibly embeds a `script` tag pointing to this URL in *their own* page at `https://attacker.com`:

```
<script src="https://ashleymadison.com/foo.js">
</script>
```

When the victim visits `https://attacker.com`, Chrome downloads, compiles, and executes Ashley Madison's `foo.js`—unless it can find an entry for `https://ashleymadison.com/foo.js` in its bytecode cache, in which case it skips the compilation step. In the latter case, `foo.js` goes from downloaded to running in significantly less time than it would otherwise (on the order of tens of milliseconds; see Figure 2). By measuring this time difference, the attacker can infer whether the victim visited Ashley Madison.

To accomplish this *reliably*, the attacker must precisely measure two points in time: **(1)** when the browser finishes downloading the script and **(2)** when the script starts running; these points bookend the compilation step. The attacker must explicitly avoid measuring the time the browser spends on downloading `foo.js` before compilation and the time it spends executing it afterward—these numbers vary based on many factors (e.g., the victim's network connection), and therefore introduce enough noise to obscure the relevant timing signal. To track point 1—when the script is fully downloaded—the attacker can use the Resource Timing API, which provides a timestamp for this event [39]. For point 2—when the script starts running—browsers offer no direct means of measurement. However, the attacker can approximate the script's start time by measuring the time at which the script first sets a global variable.[4]

[4]This is possible because most scripts large enough to trigger bytecode caching tend to contain framework or library code that they intend

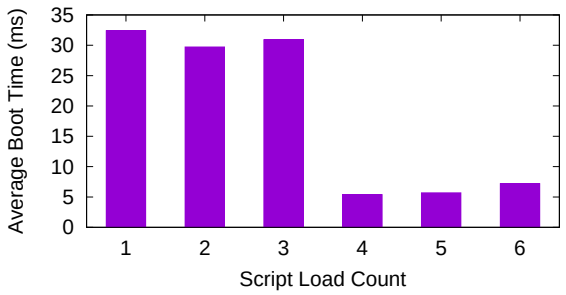

Figure 2: Boot time for a script embedded by Yahoo's homepage based on how many times it has been executed. Chrome's JavaScript engine creates a cache entry following the third execution of the script. Times shown are averages of 10 trials.

Therefore, before carrying out the attack, the attacker must identify the name of the first global variable that `foo.js` sets. To this end, the attacker records a list of all variables in the JavaScript global scope, i.e., the `window` object:

```
var oldGlobals = Object.keys(window);
```

Next, they inject a `script` tag pointing to `foo.js`, which instructs the browser to **(1)** load and execute the target script and then **(2)** call back into their own code, the `loadCallback` function:

```
var scriptTag = document.createElement('script');
scriptTag.async = false;
scriptTag.src = 'https://ashleymadison.com/foo.js';
scriptTag.addEventListener('load', loadCallback);
document.head.appendChild(scriptTag);
```

In the `loadCallback` function, the attacker generates a new list of global variables and compares this list with `oldGlobals` to identify the name of the first global variable that `foo.js` sets.

In the attack phase, the attacker defines a *setter* function that is called whenever this global variable is set by `foo.js`. This function simply records the current time at its first call—the approximate time the script starts running. The attacker uses this timestamp and the download timestamp to compute the script boot time, which they compare against a reference measurement to infer if the script bytecode was previously cached. As Figure 2 shows, the bytecode cache can speed up the boot time by 2.5x-10x, making it easy for the attacker to infer whether or not the victim previously visited Ashley Madison.

Evaluation We find this attack effective against the Windows, macOS, and Linux versions of Chrome. We reported this to Google, who marked our ticket as security-sensitive with low priority to fix. The attack takes around 100ms total to detect a bytecode cache hit or miss for a target script URL, and can be performed multiple times and in parallel to bulk-query the cache for a list of targets (e.g., to scan the Alexa Top Sites [2] and construct a profile of the victim). An attacker can learn more information about their victim with a smarter selection of target scripts—for example, testing scripts which are only loaded for logged-in members of a site. All the while, the victim remains unaware of the running attack, as it involves no visible components.

In an automated scan, we confirm the presence of a suitable target script—of 100 KB or more in size, before compression—on 372 of the top 500 sites (74%). We consider this a strict lower bound for the number of websites vulnerable to our bytecode cache attack, as we only scan for script files embedded statically by each website's HTML source. For performance reasons, some sites dynamically inject their scripts after the initial page load [3]; though bytecode cache entries are still generated for such scripts, and the attack works on them without modification, our scanning tool does not yet detect dynamically injected scripts. Additionally, around 32 of the remaining sites detected as untargetable were either CDN domains which do not serve web pages or had certificate errors that prevented us from testing them.

Compared to cache-timing attacks on other web resource types (e.g., images), our attack is more practical since it cuts out some sources of noise (e.g., from network variability): timing the compilation of a JavaScript file produces a more stable measurement than timing image downloads. Moreover, the required file size threshold to produce a detectable time difference is much smaller, enabling the attacker to target a wide range of websites.

As with most history-sniffing cache-timing attacks, our attack is destructive: the process of querying for a bytecode cache entry forces the creation of said entry if it did not already exist. This means the attack cannot be repeated against the same script URL for the same victim.

5 Consequences and defense

We test in Chrome and Firefox on three operating systems (Windows, macOS, and Ubuntu Linux), and in Edge and Internet Explorer, which are Windows-only. Two of our attacks—the CSS Paint API attack (Section 3.1) and the bytecode cache attack (Section 4)—only affect Chrome, since they target features not yet implemented in other browsers. The other two attacks—involving CSS 3D transforms (Section 3.2) and SVG `fill`-coloring (Section 3.3)—use more traditional features and prove effective across all four of these browsers. Finally, for comparison, we test two separate implementations of Paul Stone's

for *other* scripts to use, so they start by initializing a global variable where those other scripts will expect to find the code they provide (e.g., the popular jQuery library creates a global `jQuery` variable to hold its public functions [25]).

visited-link attack [48, 55], and find it only successful in Firefox on macOS and Linux, as previously mentioned.

In addition to testing the stock versions of major browsers, we also evaluate our attacks against these browsers with additional privacy features enabled and against several privacy-oriented browsers:

- *Chrome with Site Isolation.* Site Isolation [51] confines data for each origin in a separate process but does not place any restrictions on `:visited` selectors or the bytecode cache, which remain shared across isolation boundaries. As a result this feature does not prevent our attacks.
- *ChromeZero research extension.* ChromeZero tries to thwart attackers by limiting certain JavaScript APIs [45]. Since our attacks don't rely on these APIs, they still work, even with the extension in its highest protection mode.
- *Brave.* Brave is a security- and privacy-oriented browser. But since Brave is built atop Chromium, we find it to be vulnerable to the same attacks as Chrome. (We only test this browser on macOS.)
- *Firefox with visited links disabled.* Turning off Firefox's `layout.css.visited_links_enabled` configuration flag *should* eliminate visited link styling altogether [5, 46]. Not so: disabling the flag fails to block either our visited-link attacks or Paul Stone's older one; we reported this bug to Mozilla.
- *The Tor Browser.* This Firefox distribution does not keep track of user history [52] and is therefore immune to our attacks.
- *The FuzzyFox and DeterFox research browsers.* Both of these modified versions of Firefox address timing side channels by reducing the resolution of explicit and implicit timers; FuzzyFox also normalizes when browser events are scheduled [8, 29]. Our two Firefox-compatible attacks still work because they don't rely on fine-grained timers, but FuzzyFox imposes a 10x reduction in our attacks' exfiltration bandwidth. Stone's visited-link attack fails in both browsers. (We only compile these two on Ubuntu.)

Appendix A summarizes our results in table form.

Defense To address recurring same-origin policy violations, browser vendors like Mozilla restructured their browser architecture to enforce the SOP by construction [20]. We argue that they should similarly restructure browsers to address history sniffing attacks. To this end, we propose a same-origin-style policy to cover persistent data: browsers should not solely use the URL of a resource when storing it, but should also associate the origin of the page on whose behalf the code is running. For example, any history entry should be labeled with the origin of the referring page, each bytecode cache entry should be labeled with the origin of the page embedding the corresponding script, etc. Accordingly, when fetching stored data, the browser should check the origin of the page performing the lookup—and succeed only if that origin is the same as the "referrer-origin" associated with the stored data. Our proposal is similar to "domain tagging" [13] and "same-origin caching" [22], updated for the modern web.

This defense, of course, incurs some cost. For the bytecode cache, initial loads of a script *on each origin* would pay a cache-miss cost—but subsequent page loads would still benefit from caching; other caches would incur similar costs. To address this, we envision an extension to cross-origin resource sharing (CORS) [26] that allows popular, public resources (e.g., jQuery) to be safely shared across origins.

Our proposed defense also partially breaks web compatibility. Specifically, CSS `:visited` selectors would only accurately represent whether links have been clicked-on from a page of the same origin. But browsers have broken compatibility for privacy before: the fix for the original `:visited` leak changed the `getComputedStyle` API to return incorrect information, for example, and broke existing stylesheets using `:visited`. Our proposed fix would not only entirely replace this previous one but also provide robust protection against future attacks—even attacks that rely on user interaction [57].

6 Conclusion

Protecting browsing history is crucial to user privacy. The four attacks in this paper show that modern browsers fail to systematically safeguard browsing history data from web attackers; various browser features allow attackers to leak this data, in some cases at alarming rates. These attacks are, as a group, effective against all major browsers as well as several privacy-focused designs, across all three major operating systems. We propose a systematic solution to protecting browsing history data with a same-origin-style policy. Although this would incur minor performance costs and a small change to the behavior of visited link styling, we believe that these costs are worth the benefit to user privacy.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and questions. We thank Hovav Shacham, David Kohlbrenner, and Riad Wahby for fruitful discussions on history sniffing. This work was supported by a gift from Cisco.

References

[1] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *CSF*. IEEE, 2010.

[2] Alexa Internet, Inc. Alexa Top 1 Million Global Sites. URL https://www.alexa.com/topsithttp://s3.amazonaws.com/alexa-static/top-1m.csv.zip. Retrieved March 8, 2018.

[3] J. Archibald. Deep dive into the murky waters of script loading, 2013. URL https://www.html5rocks.com/en/tutorials/speed/script-loading/.

[4] D. Baron. Preventing attacks on a user's history through CSS :visited selectors, Apr. 2010. URL https://dbaron.org/mozilla/visited-privacy.

[5] D. Baron. :visited support allows queries into global history, 2010. URL https://bugzilla.mozilla.org/show_bug.cgi?id=147777.

[6] C. Blizzard. Privacy-related changes coming to CSS :visited, 2010. URL https://hacks.mozilla.org/2010/03/privacy-related-changes-coming-to-css-vistited/.

[7] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *WWW*. ACM, 2007.

[8] Y. Cao, Z. Chen, S. Li, and S. Wu. Deterministic browser. In *CCS*. ACM, 2017.

[9] ChromeZero. GitHub at fee8ad, 2018. URL https://github.com/IAIK/ChromeZero.

[10] A. Clover. CSS visited pages disclosure, 2002. URL https://lists.w3.org/Archives/Public/www-style/2002Feb/0039.html.

[11] DeterFox. GitHub at f05a9b, 2018. URL https://github.com/nkdxczh/gecko-dev.

[12] A. Drury. How internet user's identities are being tracked and used. *Tul. J. Tech. & Intell. Prop.*, 15: 219, 2012.

[13] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 25–32, New York, NY, USA, 2000. ACM. ISBN 1-58113-203-4. doi: 10.1145/352600.352606. URL http://doi.acm.org/10.1145/352600.352606.

[14] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *CCS*. ACM, 2000.

[15] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P*. IEEE, May 2018.

[16] FuzzyFox. GitHub at f80d08, 2018. URL https://github.com/dkohlbre/gecko-dev.

[17] D. Gruss, D. Bidner, and S. Mangard. Practical memory deduplication attacks in sandboxed JavaScript. In *ESORICS*. Springer, 2015.

[18] Y. Guo. Code caching, 2015. URL https://v8project.blogspot.com/2015/07/code-caching.html.

[19] C. Heilmann. CSS 3d transformations in Firefox nightly, 2011. URL https://hacks.mozilla.org/2011/10/css-3d-transformations-in-firefox-nightly/.

[20] B. Holley. At long last: Compartment-per-global, 2010. URL https://bholley.wordpress.com/2012/05/04/at-long-last-compartment-per-global/.

[21] HTTP Archive. Report: Page weight. URL https://httparchive.org/reports/page-weight. Retrieved July 22, 2018.

[22] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW*. ACM, 2006.

[23] A. Janc and L. Olejnik. Web browser history detection as a real-world privacy threat. In *European Symposium on Research in Computer Security*, pages 215–231. Springer, 2010.

[24] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *CCS*. ACM, 2010.

[25] JS Foundation. How jQuery works, 2016. URL https://learn.jquery.com/about-jquery/how-jquery-works/.

[26] A. V. Kesteren. Cross-origin resource sharing, 2010. URL https://www.w3.org/TR/cors/.

[27] H. Kim, S. Lee, and J. Kim. Inferring browser activity and status through remote monitoring of storage usage. In *ACSAC*. ACM, 2016.

[28] P. Kinlan. Using CSS 3d transforms, 2010. URL https://webplatform.github.io/docs/tutorials/3d_css/.

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

[30] V. Kokkevis. Accelerated compositing and 3d CSS transforms. chromium-dev@chromium.org, 2010. URL https://groups.google.com/a/chromium.org/forum/#!topic/chromium-dev/lF4PClNYSjc.

[31] S. Lee, H. Kim, and J. Kim. Identifying cross-origin resource status using application cache. In *NDSS*,

2015.

[32] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard. Practical keystroke timing attacks in sandboxed JavaScript. In *ESORICS*. Springer, 2017.

[33] MDN Web Docs. Introduction - SVG: Scalable vector graphics, 2005. URL https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Introduction.

[34] MDN Web Docs. CSS: ::after, 2007. URL https://developer.mozilla.org/en-US/docs/Web/CSS/::after.

[35] MDN Web Docs. CSS: filter, 2011. URL https://developer.mozilla.org/en-US/docs/Web/CSS/filter.

[36] MDN Web Docs. CSS3, 2012. URL https://developer.mozilla.org/en-US/docs/Web/CSS/CSS3.

[37] MDN Web Docs. SVG fill, 2018. URL https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/fill.

[38] MDN Web Docs. window.requestAnimationFrame(), 2018. URL https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame.

[39] MDN Web Docs. Using the resource timing API, 2018. URL https://developer.mozilla.org/en-US/docs/Web/API/Resource_Timing_API/Using_the_Resource_Timing_API.

[40] L. Olejnik, C. Castelluccia, and A. Janc. Why Johnny can't browse in peace: On the uniqueness of web browsing history patterns. In *HotPETs 2012*, 2012.

[41] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*. ACM, 2015.

[42] N. Pierron. JavaScript startup bytecode cache, 2017. URL https://blog.mozilla.org/javascript/2017/12/12/javascript-startup-bytecode-cache/.

[43] A. Russell, J. Song, J. Archibald, and M. Kruisselbrink. Service workers 1. *W3C Working Draft*, Nov. 2017.

[44] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. Fantastic timers and where to find them: high-resolution microarchitectural attacks in JavaScript. In *FC*. Springer, 2017.

[45] M. Schwarz, M. Lipp, and D. Gruss. JavaScript Zero: real JavaScript and zero side-channel attacks. In *NDSS*. Internet Society, 2018.

[46] S. Stamm. Plugging the CSS history leak, 2010. URL https://blog.mozilla.org/security/2010/03/31/plugging-the-css-history-leak.

[47] S. Stephens, I. Kilpatrick, and D. Jackson. CSS painting API level 1, 2018. URL https://www.w3.org/TR/css-paint-api-1/.

[48] P. Stone. Link visitedness can be detected by redraw timing, 2013. URL https://bugzilla.mozilla.org/show_bug.cgi?id=884270#c0.

[49] P. Stone. Pixel perfect timing attacks with HTML5. *Context Information Security (White Paper)*, 2013.

[50] Surma. CSS Paint API, 2018. URL https://developers.google.com/web/updates/2018/01/paintapi.

[51] The Chromium Projects. Site isolation. URL https://www.chromium.org/Home/chromium-security/site-isolation. Retrieved May 30, 2018.

[52] The Tor Project. About Tor browser. URL https://tb-manual.torproject.org/en-US/about-tor-browser.html.

[53] T. Van Goethem, W. Joosen, and N. Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *CCS*. ACM, 2015.

[54] A. van Kesteren and I. Hickson. Offline web applications. *W3C Working Group Note*, May 2008.

[55] A. Vastel. History stealing using timing attack, 2017. URL https://antoinevastel.com/security/privacy/2017/04/09/history-stealing.html.

[56] P. Vila and B. Kopf. Loophole: Timing attacks on shared event loops in Chrome. In *USENIX Security*. USENIX, 2017.

[57] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *S&P*. IEEE, 2011.

[58] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel. A practical attack to de-anonymize social network users. In *S&P*. IEEE, 2010.

[59] K. Yasuda. Quota management API. *W3C Working Group Note*, May 2016.

A Summary of results

Browser	CSS Paint	CSS 3D	SVG	Bytecode Cache	"Pixel Perfect" [48, 55]
Chrome	✓	✓	✓	✓	✗
Chrome (with Site Isolation)	✓	✓	✓	✓	✗
ChromeZero [9, 45]	✓	✓	✓	✓	✗
Brave	✓	✓	✓	✓	✗
Firefox	–	✓	✓	–	✓ Linux, macOS / ✗ Win
Firefox (no visited links)	–	✓	✓	–	✓ Linux, macOS / ✗ Win
FuzzyFox [16, 29]	–	✓	✓	–	✗
DeterFox [8, 11]	–	✓	✓	–	✗
Tor Browser	–	✗	✗	–	✗
Edge	–	✓	✓	–	✗
Internet Explorer	–	✓	✓	–	✗

We evaluate our attacks and two existing attacks against all major browsers and several research prototypes. Checkmarks (✓) indicate an attack was successful, while cross-marks (✗) indicate that an attack failed. We test our attacks across three operating systems: Windows 10 Pro Version 1709 (OS Build 16299.371) with Chrome 66.0.3359.117 (with and without ChromeZero), Firefox 60.0.1, Edge 41.16299.402.0, Internet Explorer 11.431.16299.0, and Tor Browser 7.5.6; macOS 10.10.5 with Chrome 65.0.3325.181, Firefox 60.0.1, Tor Browser 7.5.6, and Brave 0.22.727; and Ubuntu Linux 18.04 with Chromium 66.0.3359.181, Firefox 60.0.1, Tor Browser 7.5.6, FuzzyFox, and DeterFox.