This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2025.3538541

Contention-Based Side Channels Enable Faster and Stealthier Browsing History Sniffing

Mojtaba Zaheri, Yossi Oren, and Reza Curtmola
New Jersey Institute of Technology
{*mojtaba.zaheri, yo43, crix*}*@njit.edu*

Abstract—Web browsers are implicitly trusted to handle a large amount of private user information. One such piece of private information, a user's browsing history, is maintained by browsers and is used to provide a popular usability feature: Web links are rendered using different styles depending on whether the URLs they point to have been visited or not. Unfortunately, this feature can be abused by malicious webpages in order to extract users' browsing history.

We present new browsing history sniffing attacks through two contention-based side channels which are new in this context: Last-level CPU cache contention, and GPU execution unit contention. The attacks are robust and can be executed successfully against the popular Chrome browser. Compared to prior work which uses the rendering performance as a side channel, our work achieves an attack rate increase of up to 30x. The new attacks are stealthier, because the side channels we use do not slow down the browser's rendering rate. In addition, we revisit the existing sniffing attacks based on the rendering performance side channel, and show how their attack rate can also be increased by a significant amount. Finally, we discuss the root cause of history sniffing attacks and point out solutions.

1 Introduction

Users share an extraordinary amount of private and potentially sensitive information with their web browsers. One of the sources of this information is the user's browsing history: Each time a user visits a website, the browser records the website's uniform resource locator, or URL, in an internal data structure. Analyzing the list of sites collected in this data structure can provide information about the user's age, gender, income, location, sexual orientation, and even the real identity of a user hiding behind a pseudonym. An attacker who gains knowledge of this information can then use it for nefarious purposes, such as discrimination in access to employment, housing and health opportunities, political persecution, and even the risk of arrest in the case of users in more oppressive regimes. Browser *history sniffing attacks* describe a set of techniques which expose this internal data structure to an external attacker. The field of history sniffing has a long history of attacks and defenses [1], [2], [3], [4], [5], [6].

In the most commonly-considered attacker model, history sniffing is performed through an untrusted *attacker webpage* which the victim is enticed into visiting. One popular browser feature which attackers often exploit in this setting is *visited link styling* – to improve usability, browsers commonly apply different style rules to visited and unvisited links. Whereas browsers assign by default different colors to visited and unvisited links, website designers can further customize the style by using the `:visited` CSS selector. To exploit this feature for history sniffing, the attacker embeds into the attacker web page a link pointing to a potentially-visited website. Due to the visited link styling feature, the browser chooses which style is to be applied to this link by querying its internal data structure storing the user's browsing history. Therefore, the attacker can detect if a certain site is present in the user's browsing history by determining which style was applied by the browser to this attacker-controlled link element – if the browser applied the `:visited` style to the link, this means that the site is in the user's browsing history.

In the early years of web browsers, vendors paid little attention to history sniffing attacks, and there were no restrictions against programmatically querying the style of a link element. This direct approach allowed for quick and reliable history sniffing. Indeed, a 2010 study by Jang *et al*. showed that a considerable amount of popular websites had adopted this approach, and were actively using browsing history sniffing techniques to profile their users [7]. Over time, browser vendors became aware that history sniffing attacks can lead to serious user privacy violations, and started making it more difficult to launch these attacks. For example, browsers began blocking direct programmatic queries to visited link styling, and restricting the selection of styles which may be applied to the `:visited` CSS selector [3], [8]. Nevertheless, all modern browsers still implement visited link styling, meaning that the underlying privacy risk remains. As browsers add new features and new APIs, and browser vendors aggressively optimize their code in order to keep the browsing experience fast and responsive, this added complexity opens the door to new avenues by which browsing history data can be exposed.

One central class of attacks which is used for history sniffing are side-channel attacks. These are attacks which observe indirect effects of the browser's activity, such as its response latency or memory activity, to infer information about the browser's secret internal state. In particular, in the history sniffing context, attackers use side-channel information to learn whether the browser is performing activities caused by the visited link styling feature. The key insight behind many of these attacks is the fact that browsers

attempt to minimize the number of times they repaint the screen in order to improve performance. To use this insight, the attacker creates an HTML link element and makes sure it points to an unvisited link. This can be done, for example, by making the link refer to a randomly-generated invalid URL. Next, the attacker modifies the page such that the link now points to a potentially-visited website, without changing any other element of the page. If this potentially-visited website is, in fact, unvisited, the browser does not need to repaint the screen. If, however, the website is actually visited, the browser has to apply visited link styling to the link element, causing a screen repaint. The attacker attempts to detect this repaint operation using a side channel.

Two recent results [4], [5] use the *rendering performance side channel* to perform history sniffing. The authors of these works apply a complex CSS style to the text inside the `<a>` HTML tag. As a result, when the browser has to recompute and re-render the text associated with the `<a>` tag, this operation has a heavy impact on the hardware and software, which the attacker can then observe through side channels.

The side channel-based works presented so far had one major limitation – they all relied on the rendering performance side channel, and in particular on the time it takes to render a link, as their sole source of information. The low data rate of this side channel (essentially, one measurement every time the page is refreshed), combined with the continuous reduction in the timer resolution of web-based timers, reduced the effective speed of existing history sniffing attacks, and thus limited the attack's effectiveness and impact. In addition, these methods inherently slow down the browser's rendering speed, degrading the browser's responsiveness and making the attacks easily detectable by the victim.

In this paper, we present new browsing history sniffing attacks through contention-based side channels which are new in this context: The last-level CPU cache side channel and the GPU side channel. These side channels have a higher data rate, allowing the browser's internal state to be probed with a higher sensitivity and, as a result, increasing the effective speed of the attack and reducing its error rate.

As opposed to prior work which uses the rendering performance as a side channel [4], [5], our work has several advantages. First, the attacks we explore can be successfully executed against the popular Chrome browser, which has advanced mitigations against the timing leaks exploited by prior attacks. Second, the attack rate of our attacks is significantly higher: up to 60 URLs/second on the systems we evaluated, compared to 2 URLs/second reported in existing works. Third, since the side channels used in our new attacks do not slow down the browser's rendering rate, the attacks are harder to detect. We also revisit existing sniffing attacks based on the rendering performance side channel, and show how their attack rate can also be increased by a significant amount. A key insight of our approach is that we are able to use the lowest number of calls to the `requestAnimationFrame` API (only 1 or 2 calls per tested URL) in conjunction with leveraging the characteristics of specific side channels.

As a secondary contribution, we perform a deeper investigation of the browsing history attack surface: First, we investigate whether the CPU port contention side channel is robust enough to be used for performing history sniffing. Second, we experiment with our new attacks against the Firefox browser. While our findings suggest that some leakage occurs in both of these cases, developing this leakage into an attack that can be executed in a reliable fashion remains an open challenge.

Specifically, our main contributions are:

- We introduce two new attack methods that leverage micro-architectural side channels to achieve robust, fast and stealthy history sniffing attacks (Section 4.2).
- We introduce a new side channel technique based on WebGL and GPU contention (Section 4.4).
- We successfully execute history sniffing attacks in the Chrome browser using CPU cache-based and GPU-based side channels, which are new in this context. Compared to prior work that relies on the rendering performance side channel, our attacks improve the attack rate by up to 30x and maintain a high attack accuracy while remaining stealthy (Section 4.3, 4.4). We also significantly improve the rate of attacks that rely on the rendering performance side channel (Section 4.5).
- We explore the effectiveness of port contention side-channel attacks for history sniffing attacks, and remark on the challenges of mounting a full attack using this method. (Section 4.6).
- Finally, we discuss the root cause of history sniffing attacks based on `:visited` CSS selector, and point out the interim and the more fundamental solutions against these attacks (Section 5).

Our work shows that side channel-based history sniffing can be performed with accuracy, speed and stealth on modern browsers such as Chrome. This reinforces the need to solve the privacy issue caused by the visited link styling feature in a more systematic way.

Artifact Availability. We provide attack pages that implement the attacks described in this work, together with instructions to reproduce the attacks. Due to the potential security impact, the artifact repository is only accessible through a token: `git clone https://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@github.com/mjz3/HistorySniffing2023.git`

Responsible Disclosure. As part of a responsible disclosure process, we have opened bug reports with browser vendors (Chromium [9], Firefox [10]) and are sharing a draft of this paper with Google and Mozilla. We are currently working with these organizations to address the disclosed vulnerabilities. Until the responsible disclosure process concludes, we plan to embargo the results.

2 Background

2.1 CSS and Visited Link Styling

A web page contains multiple HTML elements, including headings, paragraphs, images, links and tables. The Cascading Style Sheet (CSS) language allows the web page to describe how these elements should be visually presented to the user. CSS statements consist of a series of *rules*,

where each rule contains a *selector* and a *style block*. The selector determines which elements need to be formatted in a given style, and under which conditions. Selectors may be used to pick elements based on one of the element's properties, such as its type (image, link, heading, etc.), or its programmer-defined class. Selectors may also be used to refer to the relationship between elements, for example selecting the first row in every table, or all links which are inside a specific area of the page. Finally, and of interest to this work, selectors can be used to choose elements based on their dynamic state. One of these last class of selectors, referred to as *pseudo-class* selectors in CSS notation, is the `:visited` selector, which can be used to define CSS rules which should only be applied to links which were previously visited by the user. As we show below, the *visited link styling* functionality forms the basis of the attacks we describe in this work.

Listing 1: HTML page incorporating a style for visited links.

```
<html>
  <head>
    <title>My HTML Page</title>
    <style>
      a:visited {
        color: pink;
      }
      a:visited {
        color: red;
      }
      a:not(:visited) {
        color: green;
      }
    </style>
  </head>
  <body>
    <a href="https://www.example.com/">
        link to Example.com</a>
  </body>
</html>
```

After the selector statement, which chooses which elements are to receive a certain style, CSS rules contain a style block. The style block defines the actual formatting which is to be applied to the element matching the selector. Style blocks can modify the element's color, size, positioning and other similar attributes. Some CSS styles are highly resource-intensive. For example, CSS styles can incorporate SVG filters, which apply complex image processing algorithms to elements on the page, as well as box shadows, gradients, and animations. The HTML specification allows multiple rules to apply to a single element. If the rules are in conflict, the specification gives priority to the most specific rule, or otherwise to the last rule which was defined. Listing 1 shows a sample HTML page demonstrating visited link styling applied to `<a>` elements, which represent links. In this example, visited links are painted red, whereas unvisited links are painted green. The first rule, specifying that visited links will be painted pink, is superseded by an identical rule which is defined later.

JavaScript code has the ability to modify web pages after they are loaded. This includes making changes not only to the web page's HTML content, but also to the CSS rules which define its appearance. These CSS rules can be modified by accessing the `document.styleSheets[].cssRules[]` object space. Every time a change to the page is made, the browser checks if the change has a user-visible effect. If so, the browser must repaint the page to the screen, in order to reflect this change to the user. In modern browsers, the repaint operation is not performed immediately after the page is modified. Instead, browsers attempt to queue multiple changes to the web page and handle them all at once during a single repaint event. These repaint events are generally synchronized to the physical refresh rate of the user's screen, which is 60 Hz on most systems. This corresponds to one refresh event every 16.66 milliseconds. Web pages can use a browser-provided API called `requestAnimationFrame` to make sure that animations and other visual updates are synchronized with the browser's rendering activity. As demonstrated in Listing 3, the web page passes a function pointer to the `requestAnimationFrame` API, and the browser calls this user-supplied function immediately after the previous repaint event has finished. The web page can then use this function to queue any changes it would like to be applied to the page the next time it is repainted. As observed by [11] and others, an adversary can gain an insight into the rendering complexity of the current web page by measuring the time between consecutive calls to this user-supplied function. If two consecutive calls are 16.66 ms apart, this indicates the page was rendered quickly, while larger time differences suggest that the browser took a longer time to render the page, causing it to miss at least one physical refresh event.

2.2 Existing Privacy Protections for Visited Link Styling

As noted in Section 1, the combination of the fact that the browser consults its internal history data structure when choosing how to style links, and the fact that JavaScript was originally allowed to directly read out the style applied to any link element, made it simple for malicious websites to perform history sniffing at a rate of thousands of potentially-visited websites per second. Browsers have evolved several mitigations to prevent this threat. The first set of mitigations, introduced into Firefox in 2010 [3], includes three main components: First, if JavaScript code attempts to query an element's computed style, the browser will always return style values corresponding to an unvisited status, even if the element is actually visited. Second, the types of styling which can be applied to a visited link are severely restricted; In particular, it is not allowed to use any styles which load an external resource, or change the position or size of an element. This prevents history sniffing from being performed using a collaborating malicious server, which would log accesses to these external resources, or by querying the position of other elements of the page, which are potentially affected by the changes to the link element. Finally, the Firefox rendering code was modified to make sure the code paths for visited and unvisited links are as similar as possible. This third mitigation is an initial step towards preventing rendering performance side channel-

based history sniffing attacks, which are described in more detail below.

2.3 Rendering Performance-Based Side Channel Attacks

As stated previously, web browsers typically attempt to render the contents of a webpage to the user's screen at a rate of 60 frames per second. If, however, the webpage contains computation-intensive elements, such as complex styles or animations, the browser will not be able to update the screen at this frame rate, and will instead resort to a lower rate. The browser's effective refresh rate can be observed in JavaScript using the `requestAnimationFrame` API. This creates a side-channel leak which can let a malicious web page learn about the page's current rendering complexity. To exploit this side channel for history sniffing, the attacker creates a page which is simple to render if a link is not visited, but hard to render if a link is visited. This can be achieved, for example, by defining a computationally-heavy styling for visited links. Next, the attacker measures the effective refresh rate of the browser as it displays the link. A reduced frame rate indicates that the browser is struggling to render complex content, suggesting that the link element it is trying to render is already visited.

Prior privacy attacks based on measuring the effective frame rate relied on a common approach, in which the attacker counts the number of times the `requestAnimationFrame` API is called in a specified period of time, *e.g.*, 500ms. During each call to the API, the `href` value of an `<a>` tag is swapped between a target URL and a URL that the attacker knows is not visited. The page records the number of times the API is called. A low number of calls means the system is busy with a style recomputation caused by the constant changes to the element's `:visited` status. This means the target URL is visited. In contrast, a high number of calls means the target URL is not visited. Prior work on history sniffing which used this approach achieved an attack rate of 2 URLs/second [5].

One major disadvantage of this approach is its lack of stealth. Because of the attack's very design, any time the browser attempts to render a visited link, its performance will degrade. Besides from negatively affecting the user, this allows the attack to be detected. In this paper, we show how using more refined side-channel approaches can allow history sniffing without noticeably degrading the performance of the browser.

2.4 CPU Cache-Based Side-Channel Attacks

Modern computers typically contain one or more central processing unit (CPU) cores, which are connected to a large amount of dynamic random-access memory (DRAM). The access speed of DRAM is considerably slower than the potential speed of the CPU cores. Thus, connecting the two modules directly can cause a severe degradation in the system's overall performance. To overcome this speed gap, a series of memory elements, called *caches*, are placed between the cores and the DRAM. The cache memory is typically much faster than DRAM, allowing the CPU cores to carry out most of its processing tasks without being slowed down by DRAM. They are, however, smaller in size than the DRAM, and can only store a subset of the entire system memory. Caches are organized in a *cache hierarchy*, consisting of a series of progressively larger and slower cache elements bridging the gap between the CPU and the DRAM. The largest cache memory on most PCs is called the *last-level cache*, or LLC. It is typically several megabytes in size, and is shared among all of the cores in the CPU. Due to the limited size of the cache, compared to the overall size of the DRAM, caches experience *contention*, which causes active and recently-accessed code and data to *evict* some of the existing contents of the cache when they are loaded into the cache. Modern CPUs have intricate and highly-optimized schemes for mapping physical memory into the cache, and for determining the *eviction and replacement policies*, which determine which existing elements of the cache are evicted to make room for new entries [12].

Cache-based side-channel attacks allow one process to spy on another process by exploiting contention on the cache. One building block for these attacks is the Prime+Probe method [13], [14]. To use Prime+Probe, the spy process first *primes* the cache, bringing it to a known state. Next, the spy waits for the victim to perform a task which potentially accesses the cache. Finally, the spy *probes* the cache, checking whether the victim's activity changed the cache state. High-resolution Prime+Probe attacks can be used to monitor accesses to an area of memory as small as 64 bytes, with a temporal accuracy of nanoseconds. Unfortunately, this type of attack requires the attacker to understand the virtual and physical address space layouts of the system, and to have access to high-resolution timers. Both of these capabilities are generally not available in a web context without complex and time-consuming preprocessing steps [15].

To make cache attacks more feasible in a web setting, Shusterman *et al*. introduced in 2019 the *cache occupancy channel*, a coarse-grained method which measures the time required to access the entire cache [16], based on the work of Maurice *et al*. [17]. Since the cache occupancy channel operates over the entire cache, it does not require that the attacker understand the system's cache mapping strategies, nor does it require high-resolution timers. Its drawback is a dramatically reduced temporal and spatial accuracy, which makes it less useful for fine-grained attacks, such as attacks on cryptographic algorithms. In settings where the timer resolution is further reduced, Shusterman *et al*. proposed an additional method, called *sweep counting*. Instead of measuring the time needed to access the cache once, the attacker counts the number of times the cache can be accessed in a specified time interval.

2.5 GPU-Based Side Channel Attacks

Modern computers offload their rendering tasks to a dedicated graphics processing unit, or GPU, which is either part of the system on chip (SOC) or located in a discrete external chip. The GPU contains multiple execution units, or cores, connected to a high-speed shared memory. This architecture lets the GPU perform multiple rendering tasks in parallel. Web pages may interface with the GPU using WebGL [18], a portable variant of the native code OpenGL API. WebGL allows web developers to write image rendering code fragments, called *shaders*, and submit them to the

GPU. These shaders are written in the GL Shader Language (GLSL) format. When the web page calls a WebGL API specifying tasks which use these shaders, they are dispatched to the multiple execution units on the GPU and executed in parallel, asynchronously of the tasks running on the main CPU. The code running on the CPU can then observe the execution time of these tasks, either directly by using a timer query, or indirectly by measuring the time it takes to render them to a CPU-side data structure.

Just like the CPU's cache, the collection of execution units on the GPU is a limited resource, which is shared among multiple competing processes. Naghibijouybari *et al*. [19] showed how native OpenGL code, with access to the GPU's memory allocation and performance counting API, can use this shared resource to perform several privacy-violating attacks, including website fingerprinting, user activity tracking and keystroke timing inference. In the web context, Laor *et al*. [20] showed how WebGL can be used to perform individual device fingerprinting, by measuring the performance of individual execution units in the GPU. Laor *et al*.'s attack used multiple execution rounds, where at each round, one execution unit is tasked with a relatively heavy operation through shaders, while the other execution units remain idle. Since all the execution units run in parallel, the total time it takes to complete all the tasks corresponds directly to the time it takes to complete the task in the unit with the heavier operation. As Laor *et al*. showed, the GPU execution units may have different performance due to the hardware manufacturing process, and as a result, these measurements for a set of execution units can be used to generate a unique profile for each device. In this work, we show how to construct a contention-based GPU side channel using WebGL, allowing malicious websites to perform history sniffing. By employing a simple GPU task to measure ongoing contention, our approach eliminates the need to utilize shaders for the attack.

3 Threat Model

The attacks presented in this work assume that the attacker has partial or full control over a malicious website, and that the victim can be induced to visit this website. This model can be realized through embedded malicious advertisements injected into otherwise-innocent web pages, through compromise of a web server in active use by the user, or by inducing the user to click on a link sent through a phishing campaign.

The adversary is not allowed to install any software on the victim's machine, nor is the adversary capable of observing network traffic entering and exiting the victim's machine. For clarity of presentation, we assume that the adversary has some prior knowledge of the victim's hardware configuration, and in particular of the size of the victim's last-level cache. Existing works show how this parameter can be detected remotely, through the use of side-channel attacks [21]. All of the attacks presented in this work target a recent version (v109) of the Google Chrome browser. Although we found evidence which suggests that the Firefox browser leaks some information about a user's browsing history, developing this leakage into an attack that can be executed in a reliable fashion remains an open challenge. We did not attempt the attacks against other browsers such as Safari or Edge.

The attacker is interested in mounting a closed-world browsing history sniffing attack against the victim. In this setting, we assume that there is a finite number of websites that are of interest to the attacker, denoted as $W = \{W_1 \cdots W_n\}$. The victim has previously visited a certain subset of these websites, denoted as $S \subseteq W$. The victim may have also visited an arbitrary number of websites outside of W, but these cannot be detected by the attacker. The attacker, using one of the side-channel methods described below, attempts to discover which of the websites were visited by victim. The outcome of the attack is a set $\hat{S} \subseteq W$. In the case of a successful attack, $\hat{S}$ should be as similar as possible to S.

The primary performance indicator used to evaluate the attack is its *accuracy*. In the evaluation presented in this paper, the test set is always balanced – it is constructed such that half of the websites are visited and half not visited, i.e. $|W \setminus S| = |S|$. In this setting, the best strategy for a naive classifier would yield a base rate accuracy of 50%, and the accuracy can be directly defined as the proportion of URLs correctly predicted as belonging to either S (true positive) or $W \setminus S$ (true negative):

$$\text{Accuracy} = \frac{|S \cap \hat{S}| + |(W \setminus S) \cap (W \setminus \hat{S})|}{|W|}$$

In cases where the test set is unbalanced, additional metrics such as precision, recall, and F1 score can be used as a replacement for direct accuracy measurements.

An additional important metric which we evaluate is the *attack rate*, measured in URLs per second. Since the attacker must declare the set W of sites of interest ahead of time, a higher attack rate immediately translates into a more effective attack, since more websites can be considered in a fixed time budget.

A final indicator we consider is the attack's *detectability*. An attack which creates significant visual artifacts on the victim's screen, or that noticeably slows down the victim's browser, will be less effective than a more stealthy attack.

4 New Browsing History Sniffing Attacks

4.1 The Attack Page

As stated in Section 3, the attacker begins the attack with a list of URLs W. The attacker's objective is to learn which of the URLs in that list have been visited by the victim (*i.e.*, are in the victim's browsing history). The attacker induces the user to visit an attack page, which contains one or more `<a>` tags. Next, the attack page tests each URL W_i in W one by one, by first setting the `href` attribute of the `<a>` tag to point to W_i, and then trying to detect whether visited link styling was applied to the `<a>` tag. Since direct measurements are disallowed by modern browsers, the attacker carries out this task by using JavaScript to measure the side channel.

Listing 2 outlines key components of the attack page that are common among the different side channel-based attacks described in this paper. Lines 5 and 6 define how the `<a>` tag is styled for unvisited and visited URLs. Line 10 defines

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2025.3538541

the <a> tag, including its class name accessible through the styles, its href attribute, and its text.

Listing 2: The history sniffing attack page.

```
1  <html>
2    <head>
3      <title>History Sniffing</title>
4      <style>
5        .attack { ... }
6        .attack:visited { ... }
7      </style>
8    </head>
9    <body>
10     <a class="attack" href="URL"> SOME
           TEXT </a>
11     <script id="worker">
12     - Measure the side channel
13     </script>
14     <script id="main">
15     - Setup callback for repaint that
16       1) updates the URL of the <a> tag
17       2) collects the measurement
18     - Predict the URL state based on the
           collected measurements
19     </script>
20   </body>
21 </html>
```

The code referenced in Line 12 is responsible for the side-channel measurements. A worker thread is used so that the side channel measurements can run in parallel and do not block the main thread. We implemented multiple variants of this code in our work, each of which evaluated a different side channel: CPU cache, GPU contention, and CPU port contention. In all cases, the worker thread uses a SharedArrayBuffer [22] in order to share the recorded measurements with the main thread. In addition to these measurement methods, we also reproduced the state-of-the-art rendering performance side channel, which does not require an extra worker thread.

The main script consists of two major parts: Lines 15-17 define the callback function for requestAnimationFrame, which updates the href attribute of the <a> tag and collects the measurements recorded in the worker thread; Line 18 predicts the URL's visited state based on the collected measurements.

4.2 Attack Methodology

Using the requestAnimationFrame API method. As mentioned in Section 2, the window.requestAnimationFrame method serves as the medium between the web browser and the webpage. It allows web developers to define what animations they wish to perform before the next repaint. Developers can define their code in a callback function and pass this callback as an input argument to requestAnimationFrame. The browser will then invoke this callback function right before the next frame repaint.

The requestAnimationFrame method can also be used for history sniffing attacks, as demonstrated in Listing 3. The JavaScript code defines the onAnimate callback

Listing 3: Using the window.requestAnimationFrame method for history sniffing attacks.

```
1  <script>
2    function onAnimate() {
3      // collect the measurement
4      // modify the <a> tag
5      requestAnimationFrame(onAnimate);
6    }
7    requestAnimationFrame(onAnimate);
8  </script>
```

function and calls the requestAnimationFrame API to register this callback (Line 7). As part of rendering the next frame, the browser calls onAnimate right before repainting that next frame. In our side channel-based history sniffing attack, the onAnimate callback performs three main operations: First, it collects the side channel measurement; Next, it modifies the <a> tag based on a specified pattern, as described below; Finally, it registers the function onAnimate as a callback for requestAnimationFrame (Line 5), so that the function is called again by the browser when it renders the next frame.

As explained in Sec. 2.3, prior work used the number of calls to requestAnimationFrame as a side channel to learn the visited status of a URL [4], [5]. We devise two new attack methods that significantly improve the attack's rate, compared to the existing state of the art, from 2 URLs/second to nearly to 60 URLs/second. The novelty of our approach consists of reducing the number of requestAnimationFrame API calls to a minimum of just one or two calls per tested URL, in conjunction with leveraging various types of side channels. The first method (Method A) makes two requestAnimationFrame API calls per tested URL, achieving an attack rate of 30 URLs/second. It is capable of maintaining a high attack accuracy of close to 100%,

The second method (Method B) makes only one requestAnimationFrame API call per tested URL, yielding an attack rate of up to 60 URLs/second. This increase in attack rate results in a reduction in accuracy, which can be mitigated by periodically resetting the <a> tag at the cost of a slightly reduced rate, as described below. The optimal working point we report this method depends on the particular type of side channel used in the attack: when using the CPU cache contention side channel, Method B yields an attack rate of 52.5 URLs/second and an attack accuracy of 86%, while using the GPU contention side channel yields an attack rate of 59.07 URLs/second with an accuracy of 99.6%. This suggests that Method B is always preferable for robust side channels.

Method A: two calls to the requestAnimationFrame API. Each time the attack page updates the href attribute of the <a> tag to a new URL, the :visited status of the <a> tag either remains the same or changes. If it remains the same, the browser does not recompute the <a> tag's style. If the :visited status changes, the browser has to recompute the style. This computation is observable through side channels during the interval starting from when the

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2025.3538541

`href` attribute is set to the new URL to the time the `requestAnimationFrame` API is called again. However, the mere fact that the `:visited` status has changed or not is not enough to determine if the new URL is visited by the user. It also depends on the status of the previous URL, *i.e.* the value of the `href` attribute before updating to the new URL. In order to overcome this dependency, the `href` is initialized before each measurement to a URL the attacker knows is not visited by the user. Since the previous URL is always unvisited, a change in the status means the new URL can be labeled as visited, whereas no change in the status means the new URL can be labeled as unvisited.

Figure 1 demonstrates how the attack works. The attacker's goal is to find out whether the user currently browsing the attacker's website has visited each of the following URL addresses: `google.com`, `youtube.com`, `pornhub.com`, `facebook.com` and `reddit.com`. The attack page contains an `<a>` tag with its `href` attribute initialized to `notvisited1368.foo` (*i.e.*, a randomly-generated URL which was, in all likelihood, not visited by the user). Next, the URL is changed to `google.com`. If the user has visited `google.com`, the status of the `<a>` tag changes from unvisited to visited. This triggers style recomputation, which the browser finishes at 16.66ms and becomes ready for the next call to the `requestAnimationFrame` API. At this point, the URL is reset to `notvisited9054.foo`, another randomly generated URL that was not visited by the user. At this time, the status changes from visited to unvisited. The browser recomputes the style again, and finishes the style recomputation at 33.33ms at the end of the second call to `requestAnimationFrame`. Thus, even though the status of the `google.com` URL is not directly visible to the attack

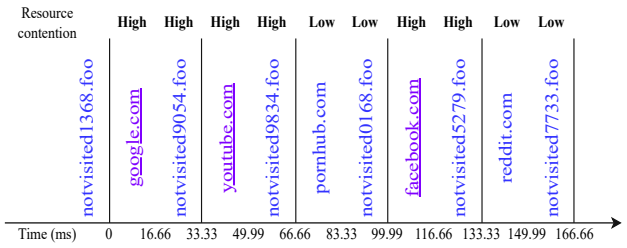

Fig. 1: Method A of history sniffing attacks: two calls to the `requestAnimationFrame` API. We use the standard color convention used by browsers to denote visited/unvisited URLs: blue for unvisited links and purple for visited links.

If there is a high resource contention during the interval between 0 and 33.33ms, it means that the URL status has changed from unvisited (`notvisited1368.foo`) to visited (`google.com`), and the attacker labels `google.com` as visited. In addition, at 33.33ms, the `<a>` tag is in a fresh, unvisited state, ready to test with next URL. Similarly, `youtube.com` and `facebook.com` are labeled as visited by observing high resource contentions during the intervals 33.33-66.66ms and 99.99-133.33ms.

The other case is when the new URL is unvisited. Consider `pornhub.com` as an example. At 66.66ms, the URL changes from `notvisited9834.foo` to `pornhub.com`. Since both of these URLs are unvisited, the `:visited` status of the `<a>` tag does not change, and the browser does not recompute the style. When `requestAnimationFrame` is called next at 83.33ms, the URL changes from `pornhub.com` to `notvisited0168.foo`. Again, there is no change in the state and so the browser does not recompute the style. This results in a low resource contention during the interval between 66.66 and 99.99ms, and as a result, `pornhub.com` is labeled as unvisited. Similarly, `reddit.com` is labeled as unvisited by observing low resource contention during the interval between 133.33 and 166.66ms.

Given the 60fps rendering rate in web browsers, the `requestAnimationFrame` API is called 60 times in one second. Since Method A requires two calls to the API to determine the visited status of each URL, the attack can test 30 URLs in one second, a significant improvement compared to 2 URLs in prior work.

Method B: one call to the `requestAnimationFrame` API. To further improve the attack rate, we devise a second attack method. Unlike Method A, which relies on a URL that the attacker knows was not visited by the user, Method B relies on the predicted status of the previous URL, and calls the `requestAnimationFrame` API only once for each tested URL. As such, the attack page can now test up to 60 URLs in one second, reaching the theoretical limit for attacks that rely on calling the `requestAnimationFrame` API. However, in Method B, the attack page only has 16.66ms to measure high or low resource contention, compared to 33.33ms in Method A. As a result, measurements are possibly less accurate. An incorrect labeling of a tested URL's status will propagate to subsequent tested URLs, resulting in the incorrect labeling of their visited status. To overcome this effect, the attack page resets the `<a>` tag to an unvisited URL periodically, correcting the occasional wrong labeling.

Figure 2 demonstrates this approach in more detail. The `href` value is initialized to `notvisited7329.foo`, a URL that was not visited by the user. Then, the attack page changes it to `google.com` and measures the resource contention during the interval between 0 and 16.66ms. High contention means that the URL status has changed. At this point, it is also known that the previous URL (`notvisited7329.foo`) has not been visited by the user. The combination of these two pieces of information leads the attacker to learn that the `<a>` tag's URL changed its status from unvisited to visited. As a result, `google.com` is labeled as visited. Then, at 16.66ms, the attack page sets the URL to `youtube.com`, and observes a high resource contention during the interval between 16.66 and 33.33ms, meaning that the status has changed. Knowing that the previous URL (`google.com`) was visited and the new URL (`youtube.com`) changed the status, the attacker learns that `youtube.com` was not visited.

One disadvantage of Method B is that it is more sensitive to errors than Method A. For example, consider a case where `youtube.com` is unvisited and `pornhub.com` is visited. As illustrated in Figure 2, at 33.33ms, the URL is changed from `youtube.com` (an unvisited URL) to

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2025.3538541

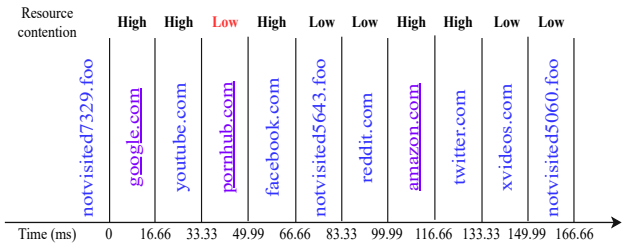

Fig. 2: Method B of history sniffing attacks: one call to the `requestAnimationFrame` API. We use the standard color convention used by browsers to denote visited/unvisited URLs: blue for unvisited links and purple for visited links.

`pornhub.com` (a visited URL), but the resource contention during 33.33-49.99ms was wrongly measured as low. As a result, the attack page incorrectly labels `pornhub.com` as unvisited. Even worse, this incorrect labeling will affect the subsequent URLs. For example, at 49.99ms, the URL is changed from `pornhub.com` (wrongly labeled as unvisited) to `facebook.com` and a high resource contention is observed, thus wrongly labeling `facebook.com` as visited. To stop this error from propagating further, the URL is reset at 66.66ms to `notvisited5643.foo`, which is known to be not visited. Next, the URL changes to `reddit.com` and a low contention is observed during 83.33-99.99ms interval, indicating that the state has not changed. Knowing that the previous URL (`notvisited5643.foo`) was not visited by the user, the attack page correctly labels `reddit.com` as unvisited. The example shown in Figure 2 resets the URL after testing every 4 URLs, but the exact URL reset number should be determined based on the level of accuracy and reliability of the side channel used for the attack.

Measurement Methodology. We now describe the methodology used to perform the measurements in the remainder of this attack section (*i.e.*, Sections 4.3, 4.4, 4.5). The experiments were conducted under Chrome `v109.0` on a Lenovo ThinkPad `P14s Gen 1` with `Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz`, running Microsoft Windows 11 Pro. We used `location.href` for visited URLs and `notvisited$X$.foo` for unvisited URLs, where `$X$` is a randomly generated number.

The attack requires a calibration step, in order to determine some parameter values that lead to high attack accuracy or to a good tradeoff between attack accuracy and attack rate on specific devices (recall that our threat model assumes that the attacker has some prior knowledge of the victim's hardware configuration). These parameters are the size of the text inside the <a> tag (for Methods A and B), and the URL reset value (for Method B). The calibration step can be executed offline, either manually or in an automated fashion.

The online portion of the attack proceeds in three phases:

Phase 1: The attack page first establishes baseline values, which will be used to label the target URLs. For Method A, the page performs 10 measurements for URLs that are known to be in a visited state, followed by 10 measurements for URLs that are known to be in an unvisited state. The medians of these two sets of measurements are used as baseline values for the visited and unvisited states. For Method B, the attack page performs 10 measurements for scenarios that are known to change the visited status of the <a> tag's URL, followed by 10 measurements for scenarios that are known not to change the URL's visited status. The medians of these measurements are used as baseline values for the changed and not-changed states.

Phase 2: The attack page then tests 100 target URLs, out of which 50 are visited and 50 are unvisited, shuffled at random. For Method A, each tested URL is labeled as visited or unvisited, depending on whether the side-channel measurement is closer to the visited or to the unvisited baseline value, respectively. For Method B, for each tested URL, the attack page compares the measured value with the two baseline values to determine if the state has changed or not. Combining this information with the `:visited` status of the previous URL, the web page decides whether the tested URL is visited or not.

Phase 3: Finally, the attack page reports the accuracy of the attack on these 100 target URLs. All of the reported attack accuracies and attack rates represent the mean over 5 different experiments [1]. In a real-world attack, the attack page would also report the determined visited status of the target URLs.

4.3 CPU Cache-Based Side Channel Attacks

Prior work demonstrates the effectiveness of the CPU cache side channel in multiple privacy attacks including website fingerprinting and targeted deanonymization [16], [23]. Here, we use sweep counting, a side channel technique that uses a buffer as large as the size of the last level cache. This technique records the number of times it can access, *i.e.* sweep, the entire buffer in a specified time interval; if the cache is occupied with other system activity, it takes a longer time to sweep the buffer. Thus, a low number of buffer sweeps indicates a high contention in the cache, whereas a high number of buffer sweeps indicates a low contention.

To leverage this side channel in the context of history sniffing attacks, we employ a very large text inside an <a> tag. The code snippet in Listing 4 demonstrates how the attack page applies simple CSS styles such as `font-size` and `color` on the <a> tag.

Listing 4: CSS style for the attack page that uses a CPU cache contention-based side channel.

```
<style>
  .attack {
    font-size: 2px;
    color: white;
  }
  .attack: visited  {
    color: #feffff;
  }
</style>
<a id="a0" class="attack">
  [VERY LARGE TEXT]
</a>
```

1. We have opted not to report the standard deviation because the accuracy values that we obtained were quite similar.

URL Text Length	28K	29K	30K	31K	32K
Accuracy	91.2%	91%	96.4%	93.8%	96.8%
URL Text Length	33K	34K	35K	36K	37K
Accuracy	97.6%	94.4%	92.6%	91.6%	**98%**
URL Text Length	38K	39K	40K	41K	42K
Accuracy	**98.4%**	**98.6%**	95.8%	96.6%	92.4%

TABLE 1: Attack accuracy with Method A using the CPU cache side channel. The text length ranges from 28K to 42K characters. The attack rate is 30 URLs/second.

URL Reset Value	3	4	5	6
Accuracy	78%	87.6%	84%	90.6%
Rate (URL/s)	45	48	50	51.43
URL Reset Value	7	8	9	10
Accuracy	**86%**	81.8%	72.8%	79%
Rate (URL/s)	**52.5**	53.33	54	54.54

TABLE 2: Attack accuracy and attack rate with Method B using the CPU cache side channel. The text length is 38K and the URL reset value ranges from 3 to 10.

Different shades of white color are used for visited and unvisited states to hide the <a> tag from users and keep the attack stealthy. Whenever the visited status changes, the browser recomputes the CSS styles on the large text. This results in a high cache contention, which can be observed through sweep counting. When the visited status does not change, on the other hand, cache contention is low.

Method A. The attack page performs sweep counting measurements during 33.33ms intervals. As the level of cache contention relies on the text length, we experiment in the calibration step with texts of different lengths to find which one is more effective. We covered a large search space for the text length by first experimenting with text lengths that are powers of two, starting with 512 characters up to 128K characters. Among them, the experiment with 32K text length had the highest accuracy of 96.8%. We then experimented with text lengths surrounding 32K, in multiples of 1K characters, as shown in Table 1. Among them, 37K, 38K and 39K yield the highest accuracy, whereas accuracy decreases for other lengths in this range. As a result, we chose the text length in the middle of this range, 38K, which yields an attack accuracy of 98.4% and an attack rate of 30 URLs/second. To determine the root cause of this accuracy, we performed two ablation experiments. In the first, we reduced the text size to 0 characters, and obtained an accuracy no better than a random guess. In the second experiment, we changed the cache sweeping code such that it did not make any cache accesses. In this latter case, accuracy was significantly degraded, yielding accuracies as low as 60%. This negative result, while perhaps intuitive, seems not to conform to the claims of Cook et al. from [24].

Method B. The attack page takes sweep counting measurements during 16.66ms intervals. As a result of our prior experiments with Method A, we chose 38K as the text length. We then experiment with various values for the URL reset, which is specific to Method B. After experimenting with powers of two between 1 and 64, the top accuracy of 87.6% was obtained for a URL reset value of 4. A higher URL reset value yields a higher attack rate, but results in a lower attack accuracy. We then experimented with values surrounding 4, from 3 to 10, as shown in Table 2. Most of these had an accuracy above 80%. As a tradeoff between attack accuracy and attack rate, we chose to reset after every 7 URL tests, yielding an accuracy of 86% and a rate of 52.5 URLs/s.

4.4 GPU-Based Side Channel Attacks

Inspired by the sweep counting technique that measures the CPU cache contention side channel, we now introduce a new technique based on GPU contention. As described in Sec. 2.5, prior work used WebGL for device fingerprinting by taking advantage of the fact that GPU execution units may have different performance due to the hardware manufacturing process [20]. Here, we revisit this attack vector, and introduce a novel side channel based on GPU contention and WebGL that can be used to learn about other activities in the system. We rely on the fact that a GPU busy with style computations would take a longer time to respond to WebGL queries. Thus, the attack page tasks the GPU with a simple operation, and measures the time it takes to complete this task. A slow or fast completion of this simple task indicates whether the GPU is engaged with heavy operations from other processes at the same time.

We use computation-heavy CSS styles on the <a> tag to impose a heavy workload on the GPU when a style recomputation occurs. These CSS styles are similar to those used by prior work [4], [5]. More specifically, we use `transform:rotateY()` as a mechanism to cause future animations and filters to be performed on the GPU [25], as described in Listing 5.

A worker thread defined in the attack page uses WebGL code to measure GPU contention, as described in Listing 6. First, in the `prepare()` function, the webpage uses the `OffscreenCanvas` interface [26] to initialize an empty one-pixel canvas that can be rendered off screen. Then, in the `go()` function, a call to the `convertToBlob` method of this interface causes all the instructions currently in the WebGL pipeline to execute and ultimately return a binary object representing the image contained in the canvas, as in [20]. The attack page calls `convertToBlob` repeatedly, and counts the number of times this call can be completed during a specified time interval. A low count means that the GPU experiences high contention due to other operations in the system such as CSS style computations. In contrast, a high count means the GPU experiences low contention. This approach differs from previous work in that it measures the GPU's ongoing contention, rather than its hardware capabilities. As a result, this new technique can be used with history sniffing attacks, similar to the cache sweep counting technique used in Sec. 4.3. Interestingly, our method works even if the attack page dispatches no OpenGL drawing operations to the GPU, and does nothing but repeatedly call `convertToBlob`.

We evaluate the GPU attack using two methods, similar to the CPU cache attack. **Method A.** The attack page

Listing 5: CSS style for the attack page that uses a GPU contention-based side channel.

```
<style>
  .attack {
    transform:
      perspective(100px) rotateY(37deg);
    filter:
      contrast(200%);
      drop-shadow(16px 16px 10px #fefefe
          );
      saturate(200%);
      text-shadow: 16px 16px 10px #
          fefefe;
      outline-width: 24px;
      font-size: 2px;
      text-align: center;
      display: inline-block;
      color: white;
      background-color: white;
      outline-color: white;
    }
  .attack: visited {
    color: #feffff;
    background-color: #feffff;
    outline-color: #feffff;
  }
</style>
<a id="a0" class="attack">
  [VERY LARGE TEXT]
</a>
```

Listing 6: Measuring the GPU contention side channel.

```
<script>
  async function prepare() {
    offscreenCan = new OffscreenCanvas
        (1,1);
    offscreenCan.getContext("webgl2",
        ... );
  }

  async function go(interval) {
    count = 0;
    start = performance.now();
    while(performance.now() - start <
        interval) {
      blob = await offscreenCan.
          convertToBlob();
      count++;
    }
    return count;
  }
</script>
```

performs GPU contention measurements during 33.33ms intervals. It works similar to Method A of Section 4.3, except for using the CSS style shown in Listing 5 and a different side channel (GPU contention instead of CPU cache contention). In the calibration step, we first experimented with text lengths which are powers of two, starting with 512 characters up to 16K characters. Among them, the experiment with 4K text length had the highest accuracy of 99.4%. We then experimented with other text lengths surrounding 4K, in multiples of 1K characters, from 2K to 9K, as shown in Table 3. Among them, the 5K and 6K text lengths yielded the highest accuracy of 99.8% with an attack rate of 30 URLs/second.

URL Text Length	2K	3K	4K	**5K**
Accuracy	61%	52.2%	99.4%	**99.8%**
URL Text Length	6K	7K	8K	9K
Accuracy	99.8%	98.2%	85.8%	75%

TABLE 3: Attack accuracy with Method A using the GPU contention side channel. The text length ranges from 2K to 9K characters. The attack rate is 30 URLs/second.

URL Reset Value	2	4	8
Accuracy	80.2%	92%	95.6%
Rate (URL/s)	40	48	53.33
URL Reset Value	16	**32**	**64**
Accuracy	98.4%	**99.6%**	**99.6%**
Rate (URL/s)	56.47	**58.18**	**59.07**

TABLE 4: Attack accuracy and attack rate with Method B using the GPU cache side channel. The text length is 5K and the URL reset values are powers of two from 2 to 64.

Method B. The attack page takes measurements during 16.66ms intervals. It works similar to Method B of Sec. 4.3, except for the CSS styles and the side channel. We chose 5K as the text length, based on the experiments with Method A. We then experiment with various URL reset values, namely powers of two between 1 and 64, as shown in Table 4. Most of these had an accuracy above 90%, suggesting that the GPU contention side channel is highly robust and accurate. Still, to compensate for occasional inaccurate measurements while considering the tradeoff for attack rate, it seems useful to reset the URL to an unvisited URL after every 32 or 64 tests. This yields an attack accuracy of 99.6% and an attack rate of 58-59 URLs/second.

4.5 Rendering Performance-Based Side Channel Attacks

Prior work has already experimented with the rendering performance side channel, as described in Sec. 2.3. We take a different approach to this attack, compared to prior work, by measuring the time it takes to complete a small, constant number of `requestAnimationFrame` API calls. This approach still impacts the browser's rendering performance, limiting the stealthiness of the attack, but it leads to an improved attack rate compared to prior work [4], [5].

Method A. In Listing 3, the attack page uses the JavaScript `performance.now()` API to record the start time (Line 3) and sets the `href` attribute of the <a> tag to the new URL (Line 4). The next call to `onAnimate`, sets the `href` to an unvisited URL (Line 4). In the next call to `onAnimate`, the attack page records the end time (Line 3). The difference between the start and end times is used as a side channel to measure the rendering performance and predict the visited status.

We used the CSS styles described in Sec. 4.3, but with a much larger text length to overwhelm the browser's 60

URL Text Length	128K	**256K**	512K	1M
Accuracy	75.8%	**86%**	95.4%	99%
Rate (URL/s)	26.54	**15.61**	5.88	3.13

TABLE 5: Attack accuracy and attack rate with Method A using the rendering performance side channel. The text length ranges from 128K to 1M characters.

URL Reset Value	**2**	4	8	16
Accuracy	**77.8%**	70%	67.6%	51.4%
Rate (URL/s)	**22.04**	30.11	30.22	34.19

TABLE 6: Attack accuracy and attack rate with Method B using the rendering performance side channel. The text length is 256K and the URL reset values are powers of two from 2 to 16.

fps rendering rate and induce a timing side channel. For the calibration step, we experimented with text lengths that are powers of two, between 512 and 1M characters. We noticed that for lengths between 512 and 64K, the Chrome browser preserved the 60fps frame rate and the attack accuracy remained low at 50%. Starting from 128K, the frame rate decreased and as a result, the attack accuracy improved increasingly with larger texts. Table 5 shows the attack accuracy and rate for the 128K, 256K, 512K, and 1M text sizes. As a trade-off between attack accuracy and attack rate, we chose 256K as text length, which yields an attack accuracy of 86% and attack rate of 15.61 URLs/second.

Method B. In Listing 3, the attack page records the start time (Line 3) and sets `href` attribute of the `<a>` tag to the new URL (Line 4). At the next call to `onAnimate`, it records the end time (Line 3). The difference between the start and end times is used to predict the changed or not-changed state. We chose 256K as the text length based on the experiments with Method A and focused on varying the URL reset value. Experimenting with different reset values that are powers of two between 2 and 16, we observed that the attack accuracy decreases with higher reset numbers, as shown in Table 6. Due to the low accuracy of the rendering performance side channel and shorter interval of just one call to `requestAnimationFrame` in Method B, the accuracy decreases dramatically with increasing the URL reset value. As a result, we chose to reset the URL after testing every 2 URLs, which yields an attack accuracy of 77.8% and an attack rate of 22.04 URLs/second.

4.6 CPU Port Contention-Based Side Channel Attacks

Previous studies have confirmed the efficacy of CPU port contention as a covert channel [27]. In modern Intel CPUs, each physical core has two logical cores, with the execution engine responsible for scheduling micro operations between them. Different ports are used to fetch instructions based on their type; for example, arithmetic micro operations are distributed across ports 0, 1, 5, or 6. Rokicki *et al.* investigated port contention on ports 1 and 5 and identified several Web Assembly instructions that can exploit this side channel.

The authors utilized this side channel as a covert channel attack to transfer data between processes, achieving a bit rate of 200bps in cross-browser communication. However, this approach has a limitation - the Web Assembly code in the first process can only extract information from the second process if it runs on the same physical core as the second process, which can be one of several physical cores in a CPU. To address this, the authors suggested detecting the system's core count using the `navigator.hardwareConcurrency` API and running multiple worker threads.

We believe that the CPU port contention side channel is suitable for history sniffing attacks, much like other hardware-based side channels, including CPU cache and GPU contention. To test this hypothesis, we used various Web Assembly codes [28], such as Count Trailing Zeros, Unsigned Remainder, and bitwise operations on 128-bit vectors. Our experiments showed that the `NOT`, `AND`, `XOR`, and `ANDNOT` operations on vectors produced positive results with attack accuracies up to 76.3% using Method A, with side channel measurements visibly different - the baseline value for visited was approximately half of the baseline value for unvisited. However, we were unable to reproduce these results consistently over time, suggesting the observed behavior may be unreliable.

In our efforts to identify the CPU port related to visited styling operations, we used Intel VTune Profiler [29] to profile the Chrome browser while running a rendering performance-based side channel attack. The profiling data revealed that port 6, responsible for branch and simple ALU, was the busiest during the attack. As a result, we ran the CPU port contention-based side channel attack again, this time using other operations to target port 6. We experimented with operations such as `ADC`, `LE`, `GT`, and even a simple loop in JavaScript, but were not able to improve the attack accuracy. We consider this to be future work and subject to more systematic analysis.

4.7 Firefox

Firefox deployed a fix in 2020 [8] in an effort to mitigate history sniffing attacks that rely on browser rendering performance. With this patch, the browser relies on always doing the same amount of work for visited and unvisited links [30]. After this fix, the browser always recomputes the style when swapping between two URLs, and the measured contention level is the same, both when the target URL is visited and when it is not visited. This countermeasure is effective against prior history sniffing attacks, and also against Methods A and B introduced in this work. As a result, in Firefox, we were not able to produce the same results we obtained with Chrome.

Despite this, we were able to detect a specific pattern in our experiments with Firefox v.110 with a technique similar to Method A. We used rendering performance as the side channel and a very large `<a>` tag text (512K). Based on the results, the first call (when setting the `href` to the new URL) takes a long time both for visited and unvisited URLs (around 40ms rather than the typical 16.66ms), but the second call (when setting the `href` to a randomly generated unvisited URL) takes even a longer time than the first call when the new URL in the first call is visited. Based on this observation, if the second call takes a longer time than the first call, we label the URL as visited, otherwise we label it

Setup	Accuracy	Rate (URL/s)
Method A - CPU cache	98.4%	30
Method A - GPU	99.8%	30
Method A - Rendering performance	86%	15.61
Method B - CPU cache	86%	52.5
Method B - GPU	99.6%	59.07
Method B - Rendering performance	77.8%	22.04

TABLE 7: Summary of the attack results.

as unvisited. In some of our experiments, this led to highly accurate attacks with accuracies above 90%. However, we were not able to reproduce these results consistently over a span of multiple days, suggesting that this specific pattern may be unreliable. We hypothesize that although a pattern may exist, it is dependent on the Firefox browser's heuristics and/or rendering performance optimization algorithms. To conclude, this particular strategy deployed by Firefox appears to provide protection against history sniffing attacks, but its implementation may not always be effective.

4.8 Discussion

Table 7 summarizes the main attack results of this work. For all three side channels, Method A, which utilizes two calls to `requestAnimationFrame`, achieves a lower attack rate but higher attack accuracy compared to Method B. Therefore, Method B is always preferred when using a robust side channel technique for history sniffing (*i.e.*, a side channel that yields a high attack accuracy).

Among the three side channels that were experimented in this paper, the GPU contention side channel resulted in the most robust attacks with higher speeds, followed closely by the CPU cache side channel. Compared to prior work that relies on the rendering performance-based side channel, our findings demonstrate that the use of micro-architectural side channels leads to highly accurate and higher-rate attacks. In addition, these new attacks do not affect the browser rendering performance, making them difficult to detect. By using a text color that matches the color of the background (*e.g.*, white text on white background), the attacks remain stealthy. The rendering performance-based side channel was found to be less effective, but the new attack methods introduced in the paper still obtain a higher attack rate than the existing state of the art.

5 Defenses

There is a long history of browsing history sniffing attacks, resembling a cat-and-mouse game: Typically, once an attack is disclosed against a browser, the browser vendor rushes to plug the hole. Many of these attacks exploit various design and/or implementation flaws related to the `:visited` CSS selector. This pattern suggests that vulnerabilities will continue being found. A better solution would be to address the root cause at a more fundamental level, especially if the goal is to provide defenses against strong attacks such as those that leverage side channels. There are two approaches for addressing this root cause: the first is to modify the way the browser exposes browsing history to users, effectively meaning that the adversary has nothing to measure; the second is to systematically mitigate side-channel attacks in web browsers, making it impossible for an attacker to measure any leakage, including the one generated by visited link styling. We explore both approaches in this section.

5.1 Modifying Browser Behavior

The most radical solution to prevent browsing history sniffing attacks would be to simply not expose the browsing history to webpages. This would mean, for example, disabling the functionality of CSS selectors such as `:visited` and `:link`. As a result, developers would not be able to style differently links that were visited from links that were not visited. In general, browser vendors have been reluctant to deploy such radical solutions by default, the argument being that many users prefer the benefits provided by `:visited`. On a positive note, Firefox provides a configuration flag `layout.css.visited_links_enabled` which, when disabled, deactivates the styling of visited links [2].

To reduce the attack surface against history sniffing, Firefox deployed in 2020 (starting with Firefox v. 77) the following strategy: it always recomputes the style of links regardless of changes to their `:visited` status [8]. This approach, which is enabled by default, has been effective in safeguarding against the side channel-based attacks described in this paper. However, we believe that the execution of this strategy is rather complex and difficult to properly implement, as observed in our experiments described in Sec. 4.7 which show that information can sometimes leak. We speculate there may be two potential reasons. It is possible that even though both visited and unvisited links are restyled, the system-wide effects of restyling visited links are different from restyling unvisited links (*i.e.*, the computation time to restyle visited and unvisited links is different). It may also be the case that the intended restyling of all links may not be happening properly for the particular case of our attack pages. Finally, we note that even Firefox engineers also hint that this strategy does not fully mitigate timing-based attacks [30]. Ultimately, the possibility of different final styles being rendered on visited and unvisited URLs leaves the door open for potential vulnerabilities in the future, similar to the one we explored in Sec. 4.7.

Proposed Defense: Double-Keying. One solution that works at a more fundamental level is to key the status of a link (visited or unvisited) both by the origin URL (*i.e.*, the domain in the URL bar) **and** the link's target URL. This approach offers an additional layer of isolation, when compared to current practice, which only uses the link's target URL. In this way, a link's visited status would be enabled on a webpage only if that link was visited in the past from that particular webpage. This approach does not leak additional information, because the origin website could know anyway if the user had already visited that target URL in the past, for example by monitoring mouse click events on the link element.

This solution has a long history of discussions by the W3C standardization forum [31], and similar ideas have also been discussed in the context of Firefox [30] (referred to as "first-party isolation" (FPI) for `:visited`), Chromium [32]

2. By default, this flag is enabled, but it can be disabled by setting it to *false*.

and WebKit [33], or proposed explicitly by the research community [4].

This solution is being revisited periodically in meetings of the W3C CSS Working Group, where its benefits and drawbacks are being discussed, together with considering paths to move closer to adoption. Recently, there seems to be some traction to experiment with it in Chrome. We summarize next these discussions.

Such a solution, if adopted by browser vendors, should be effective against the attacks we disclose in this paper, especially if the defense is implemented at the lowest level possible (*i.e.*, do the filtering at the storage engine responsible for executing history data queries, rather than at the renderer [31]).

Unlike other defenses that focus on addressing specific flaws (e.g., adding resistance to browser fingerprinting in Firefox [34]), this solution is a more robust solution and is more provably correct than trying to patch individual flaws as they appear.

The main drawback of this solution is that it may affect usability, in that it would break certain use cases. If a user visits a webpage linked from one website (or even directly by providing the URL in the URL bar), and another website links to that webpage, the links on the second website would appear unvisited. This behavior would come apparent, for example, when a user goes to some aggregator site and wants to see what URLs are already visited (even if not from this aggregator site). We note that if a user clicked on the results of a Google search in the past, those links that were clicked would appear as visited in a future Google search, since they are being accessed from the Google search engine.

To reduce the negative usability impact, several situations could be exempted from this double-keying policy, because the visited status of a link could be exposed safely without having a privacy impact. One such situation is same-origin links, *i.e.*, links to URLs on the same domain. In this case, the visited status can be reported, because the server can track its own cross-links anyway. One other situation is to allow the user to mark certain origins as "safe" to expose the visited status of a link. For example, users can mark their favorite search engine as "safe", in which case if the user has already visited site A directly and then searches for it on Google, the link to A in the Google results would show as visited. This would be similar to how browsers expose an option for allowing third-party cookies in certain cases.

In our opinion, this solution provides a good trade off between the value that visited links provide to users and stronger privacy guarantees. We hope that our work, which we have reported to browser vendors, provides additional data points to support the case for adopting this solution.

5.2 Systematically Mitigating Side-Channel Attacks in Web Browsers

The history sniffing attack presented here is only one example of a side-channel attack which can be launched through a malicious webpage. A defense that could systematically prevent the browser from being used to launch *any* side-channel attack would be highly desirable. Unfortunately, this is a highly challenging task, as the entire programming model of the web is based on the ability of a remote, potentially untrusted server to execute code on the user's machine. To systematically prevent side-channel attacks, the browser would either need to be able to execute code in a way that is completely isolated from the rest of the system, or otherwise to prevent the code from being able to measure the side channel. Achieving total isolation is very difficult, considering the significant amount of shared resources between the browser and the rest of the system, including not only CPU and GPU components, but also the network stack, the file system, and other resources. In addition, such a defense must isolate individual web pages, and even the subcomponents of a single web page, from each other, forcing a very uncomfortable tradeoff between security and performance and ultimately introducing measurable contention within the browser itself [35]. Systematically preventing side channels from being observed is similarly challenging; side-channel attacks have been demonstrated in cases where the attacker is prevented from accessing memory, from taking timing measurements or even completely prevented from executing code [36]. A defense which will be able to overcome these challenges would be a significant step forward in the field of web security.

6 Discussion

6.1 Limitations

Our work shows how both cache-based and GPU-based contention channels can be used in a practical and effective side channel attack which infers a user's browsing history. We note that we limit the evaluation of the attacks we introduced to the Chrome web browser. Our attack claim does not include other major browsers. Although we found evidence which suggests that the Firefox browser leaks some information about a user's browsing history when confronted with our attacks, developing this leakage into an attack that can be executed in a reliable fashion remains an open challenge. We did not attempt the attacks against the Safari and Edge browsers, although we hypothesize that the Edge browser is also vulnerable due to its use of the Blink rendering engine also used by Chrome.

We also note that we did not evaluate our attack through a large-scale user study. In such a setting, the attacker will also have to consider how to effectively trick users into visiting the malicious website required to carry out the attack, and will also have to contend with the variability stemming from the diversity of hardware and software configurations that users have.

6.2 Related Work

We describe prior work related to the various techniques used by the attacks introduced in this paper.

6.2.1 Browser History Sniffing Attacks

There is a line of direct attacks that can leak the browser history by exploiting vulnerabilities in the browser. These types of attacks usually have a very high rate, of thousands of URLs/second. For example, as recently as 2002, it was possible to obtain the visited status of a link by simply

defining a background URL for the CSS :visited pseudo-class [37]. Or, an attacker could exploit an inadequate implementation of the CSS Paint API in the Chrome browser [4]. Fortunately, such vulnerabilities are usually promptly fixed by the browser vendors.

Mishra et al. [38] have shown an attack based on the browser cache, more precisely on HTTP response headers that are cached by a browser, which can reveal not only a user's browsing history, but also to build a timeline of a user's visits. This attack is prevented after modern browsers have adopted browser cache partitioning [39], [40].

There is another line of indirect attacks, such as those relying on side channels, which affect the browser on a more subtle level by exploiting contention to a computer's resources. Usually, the fix against this type of attacks is not always straightforward to design or deploy, as it may require more fundamental changes to the browser, which may affect performance or even break backwards compatibility. The work of Smith et al. [4] and Huang et al. [5] is closest to our work. They apply heavy CSS styles on `:visited` pseudo-class and count number of calls to `requestAnimationFrame` in a specified interval as rendering performance side channel, and learn indirectly if a site has been visited. In this paper, we improve the speed and stealthiness of these indirect attacks by introducing two new attack methods and using other types of side channels.

Sánchez-Rola et al. [6], [41] introduce a history-sniffing technique based on timing the execution of server-side request processing code. Basically, if a user has cookies with a website, it can reveal whether the user has visited that website and also the user's login status with the website. This type of attack is executed over a network and depends on the server-side configuration and on the network round-trip time between the user and the website. As such, it is less reliable and has a lower attack rate.

Karami et al. [42] showed that an attacker can misuse the prefetching and caching capabilities of the Service Workers technology to execute history sniffing attacks. Among the two attacks they introduce, the one based on the Performance API has been fixed by browser vendors.

An interactive history sniffing attack is a type of attack that typically involves giving the user a task to complete, such as a CAPTCHA, a game, or a puzzle, as described by Weinberg et al. [43]. The interactive task is typically constructed from hyperlinks to the sites the malicious page wants to probe, without the user realizing it. Although these attacks require user interaction, they are challenging to protect against, because web browsers allow different styles to be displayed to the user for visited and unvisited URLs. One potential solution to this type of attack is to implement the double keying defense, as outlined in Sec. 5.

Recent research has presented significant improvements to interactive history sniffing attacks. O'Neal et al. [44] and Zalewski [45] have demonstrated improved interactive attacks using CSS mix-blend-mode to perform `XOR` and `AND` operations in order to increase the attack rate. These techniques allow more efficient user-assisted history sniffing attacks.

Felten and Schneider [46] were the first to demonstrate the potential for browser history sniffing attacks through timing attacks on the web browser's cache, which was later refined by Zalewski [47]. When a resource is accessed for the first time, it is stored in the browser's cache for future use. If an attacker's page can retrieve a webpage or its embedded resources quickly, it means that the resource was likely retrieved from the browser cache, rather than from the server, which takes longer. This allows the attacker to infer that the user has previously visited that webpage. However, modern web browsers have implemented cache partitioning [39], [40] as a protective measure against these types of attacks.

Attack Vector	Rate (URLs / sec)	Dwell Req'd	Stealthy
Subnormal Floating Point [49]	16.4	No	No
CPU Frequency [50]	1-3	No	No
Paintlets [4]	3000	No	Yes
Thermal Side Channel [51]	0.005	No	No
Rendering Contention [52]	0.33	No	No
Auxiliary Links [5]	2	No	No
Prime + Probe [53]	267	Yes	Yes
This work	**59**	**No**	**Yes**

TABLE 8: Comparison of different browser history sniffing attacks.

Once a user's history is leaked, a user's history profile can be used to track the user across the web, because a user's history profile provides useful information regarding the user's uniqueness and re-identifiability [1], [48].

Table 8 quantitatively compares the extraction rate of several recent browser history sniffing attacks designed to work under a threat model similar to ours. We note that several other works did not report their attack rate [6], [44]. As the Table shows, our attack is the fastest of all previously reported history sniffing attacks, with the exception of the Paintlet attack of Smith *et al*. [4], which has been mitigated in Chrome version 67, and the Prime+Probe attack of O'Connell *et al*. [53]. In contrast to the attack of O'Connell *et al*., which requires the victim to *dwell* on the attack page for a minute or more while the eviction set data structure is constructed, our attack can run immediately as soon as the page is loaded. This makes the amortized running time of our attack faster in practice in most situations. We also note that both our attack and the attack of O'Connell *et al*. are unique in being *stealthy*, since they run in the background and do not interfere with the user's browsing experience. Other attacks work by artificially slowing down the browser's rendering engine, and measuring this slowdown through its impact on the `requestAnimationFrame` function. This slowdown noticeably affects the user's browsing experience, making the attack detectable.

6.2.2 Side Channel-Based Attacks against Browsers

Lim et al. [54] provide a systematic overview of the security landscape of modern web browsers, including their architectures, bug reports, and common attacks and defenses. This work names four sources of side channels that have been used to leak sensitive information in browsers: mi-

croarchitectural state, GPU, floating-point timing channels, and browser-specific side channels.

Next, we review prior work that leverages side channels to infer sensitive information in browsers.

Rendering-based side channels. Pixel stealing attacks [11], [49] exploit a rendering performance side channel to extract cross-origin content that is embedded in an `iframe` or an `object` HTML tag. Normally, this type of access is forbidden by the Same-origin Policy implemented in browsers. The main idea behind these attacks is that different pixel colors take varying amounts of time to render in a browser, and this difference can be magnified by expanding the target pixel. Particularly, the floating-point side channel was leveraged to amplify this difference [25], [49], [55]. These works also suggest using pixel stealing attacks for browser history sniffing [11], [49], where a custom style is applied to links on the sniffing page (e.g., using the black color for visited links and white for unvisited ones), and a single pixel of the link is read to determine its state.

Wu et al. [56] introduce a rendering contention side channel that stresses the browser rendering resource to execute several attacks, such as browser history sniffing, website fingerprinting, and keystroke logging. This side channel is caused by contention between the CPU, GPU and the screen buffer. Unlike other work that focuses on rendering a single frame, this work measures the time needed to render a sequence of frames. This, in turn, makes this attack not particularly effective for browser history sniffing attacks, as the attack rate is low.

CPU cache-based side channels. Zaheri et al. [23] leverage CPU cache side channels to execute targeted deanonymization attacks, in order to uniquely determine a user's identity on the web. Our goal in this paper is different, as we seek to infer the user's browsing history.

GPU-based side channels. Lee et al. [57] and Naghibijouybari et al. [19] utilized the power of native APIs such as OpenCL, OpenGL, and CUDA to assess the impact of a victim process on a GPU, as well as to extract sensitive user data, such as their visited websites. In contrast, our approach to the history sniffing attack is different, as we execute it from an untrusted webpage within the browser, without direct access to these native APIs.

Laor et al. [20] conducted experiments running WebGL code on an attack page to discover the physical characteristics of individual execution units of a GPU. They used this information to build a unique profile for each device for the purpose of device fingerprinting. Inspired by this idea, we have repurposed it to learn about dynamic content processed in the GPU with the goal of carrying out browsing history sniffing attacks.

7 Conclusion

In this work, we showed how attackers who make use of contention-based side channels can mount browsing history-sniffing attacks which are both faster and stealthier than state-of-the-art history-sniffing attacks based on rendering performance alone. We also discussed several potential directions for countermeasures.

Considering this attack from a wider context, it is clear that the root cause of this privacy leak is the constant trade-off made by browser vendors as they balance convenience, speed and privacy: The visited link styling feature was introduced to make browsers more convenient to use, while the demand to make web pages render faster caused browser vendors to introduce code optimizations which resulted in non-uniform execution paths for visited and unvisited link rendering. As a result, the users' privacy was compromised.

Even if browser vendors fix the particular set of leaks we exploited in this work, namely, visited link inference through cache contention and GPU contention, the root cause of this privacy leak remains, and novel side-channel attacks which may emerge in the future could still be used to exploit it. We therefore recommend that vendors adopt the double-keying approach for storing browsing history. This countermeasure is a classical example of approaches which stop privacy leaks at their source, making it inaccessible to any kind of present and future attack. Implementing it will push back against the constant drive to make browsers faster and more engaging, giving a priority to the user's right to privacy. We hope that our work provides compelling additional data points showing that piecemeal defenses are insufficient, and will help tip the scale towards defending user privacy at a more fundamental level.

References

[1] C. C. Lukasz Olejnik and A. Janc, "Why johnny can't browse in peace: On the uniqueness of web browsing history patterns," in *5th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2012)*, 2012.

[2] P. Stone, "Pixel perfect timing attacks with HTML5," in *Black Hat*, 2013, https://media.blackhat.com/us-13/US-13-Stone-Pixel-Perfect-Timing-Attacks-with-HTML5-WP.pdf.

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

[4] M. Smith, C. Disselkoen, S. Narayan, F. Brown, and D. Stefan, "Browser history re: visited," in *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*, 2018.

[5] A. Huang, C. Zhu, D. Wu, Y. Xie, and X. Luo, "An Adaptive Method for Cross-Platform Browser History Sniffing," in *Proc. of the Workshop on Measurements, Attacks and Defenses for the Web (MADWeb '20)*, 2020.

[6] I. Sánchez-Rola, D. Balzarotti, and I. Santos, "Cookies from the past: Timing server-side request processing code for history sniffing," *Digital Threats*, vol. 1, no. 4, dec 2020.

[7] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in javascript web applications," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 2010, p. 270–283.

[8] "Bugzilla: Turn on the visited link mitigations," https://bugzilla.mozilla.org/show_bug.cgi?id=1632765, 2020.

[9] "Chromium bugs: Issue 1446288: Security: Side-channel attack allows accessing the browsing history," https://bugs.chromium.org/p/chromium/issues/detail?id=1446288, 2023.

[10] "Side-channel attack allows accessing the browsing history (PoC for Chrome, preliminary results for Firefox)," https://bugzilla.mozilla.org/show_bug.cgi?id=1833918, 2023.

[11] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: Timing attacks using css filters," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. Association for Computing Machinery, 2013, p. 1055–1062.

[12] P. Vila, P. Ganty, M. Guarnieri, and B. Köpf, "Cachequery: learning replacement policies from hardware caches," in *PLDI*. ACM, 2020, pp. 519–532.

[13] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *IEEE S&P*, 2015, pp. 605–622.

[14] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *CT-RSA*, ser. LNCS, vol. 3860. Springer, 2006, pp. 1–20.

[15] T. Rokicki, C. Maurice, and P. Laperdrix, "Sok: In search of lost time: A review of javascript timers in browsers," in *EuroS&P*. IEEE, 2021, pp. 472–486.

[16] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust Website Fingerprinting Through the Cache Occupancy Channel," in *USENIX Security Symposium*, 2019.

[17] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: cross-cores cache covert channel," in *DIMVA*, ser. Lecture Notes in Computer Science, vol. 9148. Springer, 2015, pp. 46–64.

[18] "WebGL," https://www.khronos.org/webgl/.

[19] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered insecure: Gpu side channel attacks are practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. Association for Computing Machinery, 2018, p. 2139–2153.

[20] T. Laor, N. Mehanna, A. Durey, V. Dyadyuk, P. Laperdrix, C. Maurice, Y. Oren, R. Rouvoy, W. Rudametkin, and Y. Yarom, "DRAWNAPART: A Device Identification Technique based on Remote GPU Fingerprinting," in *Proc. of NDSS '22*, 2022.

[21] A. Shusterman, Z. Avraham, E. Croitoru, Y. Haskal, L. Kang, D. Levi, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Website Fingerprinting Through the Cache Occupancy Channel and its Real World Practicality," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 5, pp. 2042–2060, 2021.

[22] "SharedArrayBuffer," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer, 2023.

[23] M. Zaheri, Y. Oren, and R. Curtmola, "Targeted Deanonymization via the Cache Side Channel: Attacks and Defenses," in *USENIX Security Symposium*. USENIX Association, 2022, pp. 1505–1523.

[24] J. Cook, J. Drean, J. Behrens, and M. Yan, "There's always a bigger fish: a clarifying analysis of a machine-learning-assisted side-channel attack," in *ISCA*. ACM, 2022, pp. 204–217.

[25] D. Kohlbrenner and H. Shacham, "On the effectiveness of mitigations against floating-point timing channels," in *USENIX Security Symposium*. USENIX Association, 2017, pp. 69–81.

[26] "MDN Web Docs: OffscreenCanvas," https://developer.mozilla.org/en-US/docs/Web/API/OffscreenCanvas, 2023.

[27] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren, "Port contention goes portable: Port contention side channels in web browsers," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 1182–1194.

[28] T. Rokicki, "Port Contention Goes Portable: The Code!" https://github.com/MIAOUS-group/web-port-contention, 2022.

[29] Intel, "Intel® VTune™ Profiler: Find and Fix Performance Bottlenecks Quickly and Realize All the Value of Your Hardware," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.udp6m4, 2023.

[30] "Bugzilla: Key :visited per origin (first-party-isolation for :visited)," https://bugzilla.mozilla.org/show_bug.cgi?id=1398414.

[31] "[selectors] Solve :visited once and for all #3012," https://github.com/w3c/csswg-drafts/issues/3012.

[32] "bugs chromium: Issue 713521: Eliminate :visited privacy issues once and for all," https://bugs.chromium.org/p/chromium/issues/detail?id=713521.

[33] "WebKit BugZilla: Bug 37443 - CSSStyleSelector should pass through origin information when determined if link visited," https://bugs.webkit.org/show_bug.cgi?id=37443.

[34] "Bugzilla: Simplify ResistFingerprinting callers in nsMediaFeatures," https://bugzilla.mozilla.org/show_bug.cgi?id=1434215, 2018.

[35] P. Snyder, S. Karami, A. Edelstein, B. Livshits, and H. Haddadi, "Pool-party: Exploiting browser resource pools for web tracking," in *USENIX Security Symposium*. USENIX Association, 2023, pp. 7091–7105.

[36] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses," in *USENIX Security Symposium*, 2021.

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

[38] V. Mishra, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "Déjà vu: Abusing Browser Cache Headers to Identify and Track Online Users," in *PETS 2021 - The 21th International Symposium on Privacy Enhancing Technologies*, Jul. 2021. [Online]. Available: https://hal.inria.fr/hal-03017222

[39] M. W. Docs, "State Partitioning," https://developer.mozilla.org/en-US/docs/Web/Privacy/State_Partitioning#network_partitioning, 2023.

[40] E. Kitamura, "Gaining security and privacy by partitioning the cache," https://developer.chrome.com/en/blog/http-cache-partitioning/, 2020.

[41] I. Sánchez-Rola, D. Balzarotti, and I. Santos, "Bakingtimer: Privacy analysis of server-side request processing time," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. Association for Computing Machinery, 2019, p. 478–488.

[42] S. Karami, P. Ilia, and J. Polakis, "Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage," in *Proc. of NDSS '21*, 2021.

[43] 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 *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 147–161.

[44] K. O'Neal and S. Yilek, "Interactive history sniffing with dynamically-generated qr codes and css difference blending," in *2022 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2022, pp. 335–341.

[45] M. Zalewski, "CSS mix-blend-mode is bad for your browsing history," https://lcamtuf.blogspot.com/2016/08/, 2016.

[46] E. W. Felten and M. A. Schneider, "Timing attacks on web privacy," in *Proceedings of the 7th ACM Conference on Computer and Communications Security*. ACM, 2000, pp. 25–32.

[47] M. Zalewski, "Rapid history extraction through nondestructive cache timing," https://lcamtuf.coredump.cx/cachetime/, 2011.

[48] S. Bird, I. Segall, and M. Lopatka, "Replication: Why we still can't browse in peace: On the uniqueness and reidentifiability of web browsing histories," in *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*. USENIX Association, Aug. 2020, pp. 489–503.

[49] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 623–639.

[50] Y. Wang, R. Paccagnella, A. Wandke, Z. Gang, G. Garrett-Grossman, C. W. Fletcher, D. Kohlbrenner, and H. Shacham, "DVFS frequently leaks secrets: Hertzbleed attacks beyond sike, cryptography, and cpu-only data," in *SP*. IEEE, 2023, pp. 2306–2320.

[51] H. Taneja, J. Kim, J. J. Xu, S. van Schaik, D. Genkin, and Y. Yarom, "Hot pixels: Frequency, power, and temperature attacks on gpus and arm socs," in *USENIX Security Symposium*. USENIX Association, 2023, pp. 6275–6292.

[52] S. Wu, J. Yu, M. Yang, and Y. Cao, "Rendering contention channel made practical in web browsers," in *USENIX Security Symposium*. USENIX Association, 2022, pp. 3183–3199.

[53] S. O'Connell, L. A. Sour, R. Magen, D. Genkin, Y. Oren, H. Shacham, and Y. Yarom, "Pixel thief: Exploiting SVG filter leakage in firefox and chrome," in *USENIX Security Symposium*. USENIX Association, 2024.

[54] J. Lim, Y. Jin, M. Alharthi, X. Zhang, J. Jung, R. Gupta, K. Li, D. Jang, and T. Kim, "SOK: on the analysis of web browser security," *CoRR*, vol. abs/2112.15561, 2021. [Online]. Available: https://arxiv.org/abs/2112.15561

[55] A. Rane, C. Lin, and M. Tiwari, "Secure, precise, and fast Floating-Point operations on x86 processors," in *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Aug. 2016, pp. 71–86.

[56] S. Wu, J. Yu, M. Yang, and Y. Cao, "Rendering contention channel made practical in web browsers," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3183–3199.

[57] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities," in *IEEE Symposium on Security and Privacy*, 2014, pp. 19–33.