Targeted Deanonymization via the Cache Side Channel: Attacks and Defenses

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

Abstract

Targeted deanonymization attacks let a malicious website discover whether a website visitor bears a certain public identifier, such as an email address or a Twitter handle. These attacks were previously considered to rely on several assumptions, limiting their practical impact. In this work, we challenge these assumptions and show the attack surface for deanonymization attacks is drastically larger than previously considered. We achieve this by using the cache side channel for our attack, instead of relying on cross-site leaks. This makes our attack oblivious to recently proposed software-based isolation mechanisms, including cross-origin resource policies (CORP), cross-origin opener policies (COOP) and SameSite cookie attribute. We evaluate our attacks on multiple hardware microarchitectures, multiple operating systems and multiple browser versions, including the highly-secure Tor Browser, and demonstrate practical targeted deanonymization attacks on major sites, including Google, Twitter, LinkedIn, TikTok, Facebook, Instagram and Reddit. Our attack runs in less than 3 seconds in most cases, and can be scaled to target an exponentially large amount of users.

To stop these attacks, we present a full-featured defense deployed as a browser extension. To minimize the risk to vulnerable individuals, our defense is already available on the Chrome and Firefox app stores. We have also responsibly disclosed our findings to multiple tech vendors, as well as to the Electronic Frontier Foundation. Finally, we provide guidance to websites and browser vendors, as well as to users who cannot install the extension.

1 Introduction

On the Internet, everybody knows it is better to stay anonymous. For some types of users, however, anonymity is far more than a mere luxury, and losing it can have critical consequences. Individuals who organize and participate in political protest, who work as journalists reporting on inconvenient topics, network with fellow members of their minority group, or even purchase embarrassing or potentially incriminating personal items, may risk their life and liberty if their identity becomes known to malicious actors. *Targeted deanonymization attacks* [1, 2] are an important class of attacks which threaten user anonymity. These attacks assume an attacker who has complete or partial control over some website, and is interested in learning whether a specific target is browsing the website. The attacker knows this target only through a public identifier, such as an email address or a Twitter handle.

Leaky resources have been leveraged for this purpose [1, 3]: An attacker uses a resource-sharing service such as YouTube, Google Drive, or Dropbox to privately share a resource with the target. Next, the attacker embeds this shared resource into the attack website. Finally, the attacker checks if visitors to the website can access this embedded resource – successful access indicates that the current visitor is the intended target. Although the Same-Origin Policy (SoP) should normally prevent the attacker from learning this information, a family of mechanisms known as *cross-site leaks* (*XS-leaks*) [2] were found effective at bypassing the SoP and enabling this attack.

Whereas targeted deanonymization attacks based on leaky resource attacks were shown to be both practical and widespread, they make several limiting assumptions which cause them to be far less effective in practice. First, and most significantly, they assume the existence of a cross-site leak that allows the attacker to discover whether the embedded resource was loaded successfully. This is done by attaching error handlers, or alternative media handlers, to the embedded resource, and by checking whether they are triggered, or by otherwise exploiting behaviors which bypass the SoP, such as status code leaks, page content leaks, header leaks and other similar approaches [4]. As discussed by Staicu *et al.* [1], this behavior can be blocked through proper browser design, as well as by proper coding practices on the side of sharing websites. Second, leaky resource attacks commonly assume that the sharing website allows its resources to be embedded inside the attacker's website. However, many websites do not allow their content to be embedded in third-party websites, by using the `X-Frame-Options` header or the more modern and refined `Cross-Origin-Resource-Policy` header. A third limitation is that leaky resource attacks rely on the browser's support for third-party cookies, since the attacker's website must embed a resource from the sharing website. While the commonly-used Chrome browser exposes third-party cookies, several modern browsers, including Safari and Tor, disable third-party cookies for embedded resources. To get over these limitations, the attacker is forced to load the sharing website in a pop-up window, severely limiting the range of available cross-site exploitation methods.

In this work, we overcome these limitations by replacing cross-site leaks with browser-based side-channel attacks. Side-channel attacks are attacks that analyze the physical implementation artifacts of a system in order to gain an insight into its secret internal state. Of particular interest to our setting are *microarchitectural cache attacks*, which allow a *spy process* to observe the memory access patterns of a *victim process* over time, and use these access patterns to discover secrets about the victim. As shown by Gülmezoglu *et al.* [5], the combination of cache attacks and a deep learning-based machine learning pipeline lets an attacker effectively discover which video a user is viewing, what application he or she is running, and even which website he or she is currently browsing to. The *cache occupancy attack* is a variant of the standard cache attack, designed to work in settings with limited hardware access and limited timer resolution. As shown by Shusterman *et al*. [6,7], cache occupancy attacks are highly effective for privacy attacks, in particular for website fingerprinting, and can be mounted from within the browser through the use of untrusted JavaScript code or CSS directives, making them practical even in severely restricted settings.

By combining the side-channel technique with the blocking technique of Watanabe *et al*. [8, 9], we show that the attack surface of targeted deanonymization attacks is much larger than initially thought. In particular, we uncover a set of practical and scalable attacks that can deanonymize users in several important settings for which prior attack methods are not effective. This includes websites which use secure embedding methods or prevent embedding altogether, websites which do not allow private sharing of content between users, and browsers which block third-party cookies. Our attacks run in practical time (less than 3 seconds in most cases), and can be scaled to target an exponentially large amount of users.

More importantly, we provide a comprehensive countermeasure against all of the attacks we discovered. This countermeasure is already available on the Chrome and Firefox extension stores, and can be downloaded and installed immediately by concerned users [10, 11]. As part of our responsible disclosure process, we have reached out to the Electronic Frontier Foundation (EFF) and to multiple browser vendors and service operators, and provided guidance on how to install and use this countermeasure.

Our paper makes the following contributions:

- We introduce the concept of cache-based targeted deanonymization attacks, and show how they overcome the limitations of existing targeted deanonymization attacks, while remaining within the same threat model (Section 3).
- We experimentally demonstrate practical end-to-end attacks on a diverse set of targets, including desktop and mobile systems with multiple CPU microarchitectures, multiple browsers, and multiple highly popular websites. In particular, we present an attack on the Tor Browser which can scale to thousands of GMail users. (Section 4).
- We investigate the root cause of the attack, and show that it is caused both by a client-side and a server-side side channel working in concert. We show that generic countermeasures, such as adding random cache noise, are not effective against the attack. We design `Leakuidator+`, an open-source browser extension which successfully blocks the attack (Section 5).
- Finally, we discuss the ethical and practical implications of our findings, describe our responsible disclosure process, and provide guidance to users who may not be able to install the browser extension (Section 7).

1.1 Attacker Model

We assume the existence of one or more *victims*, which are of interest to some adversary. The adversary has some public information about the victims, for example, their Twitter handle or their email address. We also assume that the adversary has partial control over a website that the victims browse, and can inject JavaScript code into this website. The objective of the adversary is to discover whether the user currently browsing the attacker-controlled website is one of the victims. We note that the adversary does not need to control the resource-sharing service that is leveraged to execute the attack, only to be registered as a user of the service.

Motivating examples. Consider a state-sponsored adversary who has purchased, at great expense, a zero-day exploit, which it wishes to install on the computer of a journalist with a well-known Twitter handle. The adversary has also compelled a local website to include code that can install this exploit. If this exploit were to be installed on many devices, however, this would increase the risk of the exploit being detected by white-hat security researchers. Therefore, the state adversary wishes to first verify, using the well-known Twitter handle, that the user currently connecting to the website is the target journalist, and only then to deploy its exploit. As another example, consider the case where a law enforcement agency has covertly taken control of an underground extremist forum. The agency wishes to identify the users of this forum, but these users use pseudonyms to connect to the forum. The agency, however, has also gathered a list of Facebook accounts who are suspected to be users of this forum. The law enforcement agency would like to cross-reference the pseudonyms with this list of potential suspects.

2 Background

2.1 Leaky Resource Attacks

Leaky resource attacks [1–4, 8] are targeted privacy attacks, which can uniquely identify an individual browsing an attacker-controlled webpage. These attacks leverage a media resource (e.g., an image, video, or audio file) hosted by a resource-sharing service. They assume that (1) the service relies on cookies for user authentication, (2) users of this service can either privately share resources with other users, or block other users of the service, and (3) shared resources can be referenced via a canonical URL. This URL is called a *state-dependent URL* (SD-URL), since the site's response to a request for this URL depends on the user's identity.

The attack consists of two phases. In the setup phase, the attacker uploads a resource to the service, and then binds it to the victim's identity. There are two approaches to perform this binding. In the *sharing-based approach* [1], the attacker privately shares the resource with the target (e.g., by using the victim's email address or user ID with the service). In the *blocking-based approach* [8], the attacker makes the resource public, and then blocks the target from viewing any resources owned by the attacker. Next, the attacker embeds an SD-URL for this resource into an attacker-controlled webpage.

In the execution phase (Fig. 1), the attacker causes the target to visit this page (steps 1 and 2). As the target's browser renders the page, it makes a cross-site request for the embedded resource to the sharing service (steps 3 and 4), passing the user's authentication cookies. The response of the sharing website to this request depends on the target's identity. With the sharing-based approach, the response to this cross-site request contains the shared resource if the user is the target, and an error otherwise (step 5). With the blocking-based approach, the opposite happens – the response contains an error for the blocked target, and the shared resource for other users.

In the final step of the leaky resource attack, the attacker needs to discover whether the shared resource was loaded. The Same-Origin policy prevents the attacker from directly reading out the cross-origin response. The attacker can, however, bypass this policy using a *cross-site leak (XS-leak)* [2] to learn information about the response (step 6). Prior work [1] showed that different events were triggered when loading an SD-URL, allowing for a simple XS-leak. For example, when loading an image, the JavaScript *onload* callback is triggered if the image was loaded successfully, and the *onerror* callback is triggered otherwise. More subtle XS-leaks, uncovered through systematic analysis of websites and browser APIs [2, 4], include cross-origin communication between Window objects, the Performance API, and others. There are also script-less XS-leaks, that do not rely on JavaScript and instead used HTML tags that permit to load fallback content in case the primary content fails to load [1, 3].

Many services were shown to enable leaky resource attacks, including generic storage sites, media sharing sites, code-hosting repositories and social media sites. We note that it is quite common for users to remain logged into such services for extended periods of time.

Mitigations. The main weakness of leaky resource attacks lies in the final step, in which the attacker uses an XS-leak to discover whether the shared resource was loaded. Guided by recent academic research in the field, browser vendors are limiting the ability of websites to access and exploit XS-leaks, and sharing websites are redesigning their websites to reduce the XS-leak attack surface [4, 12]. As a result, it has become increasingly harder for an attacker to query the `<iframe>` belonging to the sharing website and discover whether the resource was loaded. The attack is even more challenging in browsers which implement cross-origin resource policies (CORP) [13] or SameSite cookies [14] which completely block third-party cookies. In these settings, authentication cookies are only sent to the sharing website if the site is loaded in a top-level window of its own – any sharing website content embedded in an `<iframe>` will be rendered without authentication, making classical leaky resource attacks impossible. In our work, we show how deanonymization can be performed even in the presence of all of these countermeasures.

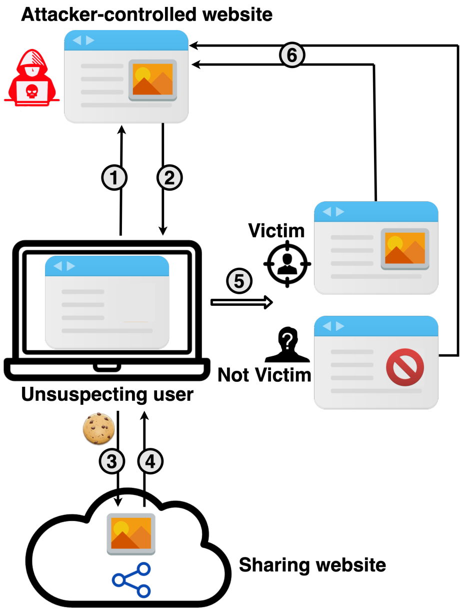


Figure 1: The leaky resource attack (sharing-based approach).

2.2 Cache-Based Side Channel Attacks

Modern computer systems prevent malicious code from accessing data belonging to other applications, users, or operating system services, by incorporating multiple trust boundary mechanisms. Micro-architectural side-channel attacks, defined by Aciiçmez as attacks which "exploit deeper processor ingredients below the trust architecture boundary", can get around these boundaries and thus compromise the confidentiality of the system [15]. Cache side-channel attacks are one type of micro-architectural attack. They exploit the high-speed cache memory, which is found in modern processors and used to interface between the fast CPU and the slower DRAM memory. This cache is typically divided into multiple levels: The fastest L1 cache is assigned to individual CPU cores, and the slowest, but largest, last-level cache (LLC) is shared between all cores. Cache attacks make use of the fact that all processes compete for the limited space available in these CPU caches. An attacker can exploit this contention to make inferences about the internal state of other processes, regardless of any software-based isolation mechanisms.

There are several methods for performing cache attacks. This work uses the **Prime+Probe** technique, originally invented by Tromer *et al.* [16] and later adapted for use in the LLC by Liu *et al.* [17]. The Prime+Probe attack has four steps. First, the attacker creates one or multiple eviction sets. Each eviction set is a list of memory addresses mapped by the CPU into the same region of the cache, a region also used by the victim for its own purposes. In the second step, the attacker accesses the eviction set, bringing the cache into a known state (prime step). Next, the attacker waits for the victim to use the cache. Since the attacker and the victim share the same region of the cache, this evicts some attacker data from the cache. In the fourth and final step, the attacker accesses the eviction set again, and measures the access time (probe step). A low access time means the eviction set is still in cache, while a high access time means it was evicted and replaced by the victim's data. Thus, the attacker detects whether the victim accessed a certain region of memory at a certain time, teaching it about the victim's internal state.

Prime+Probe attacks require a timer API with nanosecond-level accuracy, a resolution not typically available through JavaScript. The **Cache Occupancy** attack is a variation on Prime+Probe designed for this setting [6]. In contrast to the Prime+Probe attack, which monitors limited regions of the cache, in the cache occupancy attack the attacker allocates a large buffer covering the entire LLC. The attacker accesses this buffer in the prime step, bringing the entire LLC into a known state. Subsequent memory accesses by the victim will necessarily evict some of the attacker's memory from the cache, resulting in a longer runtime for the probe step. The use of a larger buffer allows the attack to be carried out with coarser-grained timers, such as the ones found within web browsers. The disadvantage of this attack is a reduced temporal and spatial accuracy, which makes it less appropriate for precise cryptanalytic attacks. **Sweep counting** is a modified version of the cache occupancy attack designed for even coarser-grained timers [7]. Instead of measuring the time it takes to go over the eviction buffer once, it counts the number of times the entire buffer can be accessed in a specified time interval. This attack was shown effective even when using the coarse timer found in the highly-secure Tor Browser.

3 Attack Techniques

In this section, we introduce several novel techniques to execute targeted deanonymization attacks. Our techniques significantly increase the potential impact of these attacks, when compared to previous work. We do so both by increasing the attack's target population by applying it to highly-popular services which have no currently-exploitable XS-leaks, including GMail, Twitter and Facebook, and by successfully executing it on browsers that have a strict policy of not allowing cookies to be attached to cross-site requests, including Safari and Tor. We also demonstrate the attack's scalability, by identifying concrete techniques to scale the attack from one target user to a group of target users.

Our overarching approach is to use CPU cache-based side channels, instead of XS-leaks, in order to determine whether the leaky resource is successfully loaded. This has the advantage of covering the novel scenarios introduced in this work, for which known XS-leaks are not effective. At the same time, we show that our approach is equally as effective in previously known attack scenarios, thus offering a unified framework for targeted deanonymization.

3.1 General Attack Methodology

Our attack has two phases, a training phase and an online phase[1]. In the *training phase*, the attacker trains a machine learning classifier to detect the cache signature associated with successfully loading a leaky resource. The training phase can be potentially repeated under a variety of combinations of sharing service, browser, and device hardware.

Next, in the *online phase*, the victim visits the attacker-controlled page, which loads the leaky resource. While the leaky resource is loaded and rendered, the attack page measures cache activity on the victim's computer. Finally, the attacker passes the collected cache measurements through the trained classifier, allowing it to identify the victim. The key advantage of our attack is that it needs no programmatic access to the leaky resource, and does not assume the existence of any XS-leak. This is because side-channel attacks take advantage of hardware-level properties of the victim's computer, and therefore disregard any software-imposed boundaries such as site, process and even VM isolation. In our particular case, side-channel attacks make deanonymization possible as long as content from the attacker's website is rendered on the same computer as content from the sharing website.

For the classifier to be able to differentiate between target and non-target users, the attack page needs to measure the cache for a certain *attack duration*, denoted as t_a, which depends on the attack setup, *i.e.*, the combination of sharing service, browser used by victim, and cache measurement method. For most attack setups, t_a is less than 3 seconds.

The techniques we present share a common structure: First, the attacker allocates a buffer as large as the cache. Next, the attacker causes the victim's browser to load the leaky resource. Then, for the attack duration t_a, the attacker repeatedly probes the buffer while the victim's browser loads and renders the leaky resource. Finally, the script uploads the collected side-channel trace to the attacker's server. Our cache occupancy code is based on the PP0 repository [18].

3.2 Embedded Content

Several highly popular services such as YouTube, LinkedIn, and TikTok present an ideal opportunity to maximize the attack's impact given their large user base. However, these services prevent direct sharing of resource URLs, instead requiring users to share embedded players, either as `<iframe>` objects or as included scripts. The embedded player will then attempt to load the shared resource, but would not indicate any success or error conditions to the parent frame.

In general, cross-site embedding through an `<iframe>` object minimizes the possibility of XS-leaks. This is because cross-origin access to `<iframe>` elements is very limited [19]. Moreover, a sharing website can directly address any known XS-leaks: For navigation leaks, the website can change the behavior so the `<iframe>` has the same navigation events in

[1] For an online-only variant of our attack, please see Appendix C.

```
1 startCacheAttack();
2 i = document.createElement("iframe");
3 i.src = "SD-URL";
4 document.body.appendChild(i);
5 waitForPageToLoad(t_a);
6 i.remove();
7 uploadTraceData();
```

Figure 2: Embedding method: `<iframe>` tag.

different states (the `CSPViolation` patch in LinkedIn [2]); for event-based leaks, the website can ensure that the same event is returned in different states (the `EF_StatusError` patch in Imgur and HotCRP [2, 20, 21]); and for frame-counting leaks, the same number of frames can be returned (the `OP_FrameCount` patch in LinkedIn [2]).

Our Approach: Instead of using XS-leaks, we measure the cache activity of the victim's computer while it loads and renders content from the resource-sharing website. As described by the code snippet in Fig. 2, the attack web page initiates the cache activity measurement (line 1), uses JavaScript to insert an `<iframe>` tag and load the leaky resource inside it (lines 2-4), takes cache measurements for the duration t_a (line 5), and finally removes the `<iframe>` (line 6) and uploads the traces to the server (line 7).

3.3 Pop-Unders and Tab-Unders

Up until the findings in this work, some scenarios were considered safe from the reach of deanonymization attacks. First, web browsers such as Safari, Tor, and Brave, have a strict policy to disable cookies by default when making cross-site requests. As such, users of these browsers may believe they are shielded from targeted deanonymization attacks via leaky resources. Second, popular services such as Twitter and Facebook explicitly prevent their content from being embedded inside other websites, either by using the `X-Frame-Options` or the `Content-Security-Policy frame-ancestors` headers to prevent cross-site embedding of their resources, or by using the `SameSite` cookie policy, which causes embedded content to be loaded without identifying cookies. Knittel *et al.* identified some cross-site leaks which can be applied even when the sharing website is loaded through a pop-up window, thereby bypassing framing and cookie restrictions [4]. This selection of leaks is, however, very limited, and still requires programmatic access to the new pop-up window, an ability which can be blocked in modern browsers by the cross-origin opener policy (COOP) [22].

Our Approach: We surreptitiously load the shared resource in a new browser window (*pop-under* variant) or in a browser tab (*tab-under* variant). In contrast to prior work on pop-up attacks [23, 24], we need no programmatic access to the newly-created window. Using a CPU cache side channel, the attacker indirectly learns private information cross-window or cross-tab, without necessarily needing a handle to the related tab or window. Also, unlike other work in which the pop-up window or the new tab remain in the foreground, our attack includes specific steps to put the new window/tab in the background, making the attack less noticeable by the user.

```
1 function go() {
2   startCacheAttack();
3   pu = window.open("SD-URL", ... );
4   ghost = window.open("about:blank");
5   ghost.focus(); ghost.close();
6   waitForPageToLoad(t_a);
7   pu.close();
8   uploadTraceData(); }
```

Figure 3: Embedding method: pop-under.

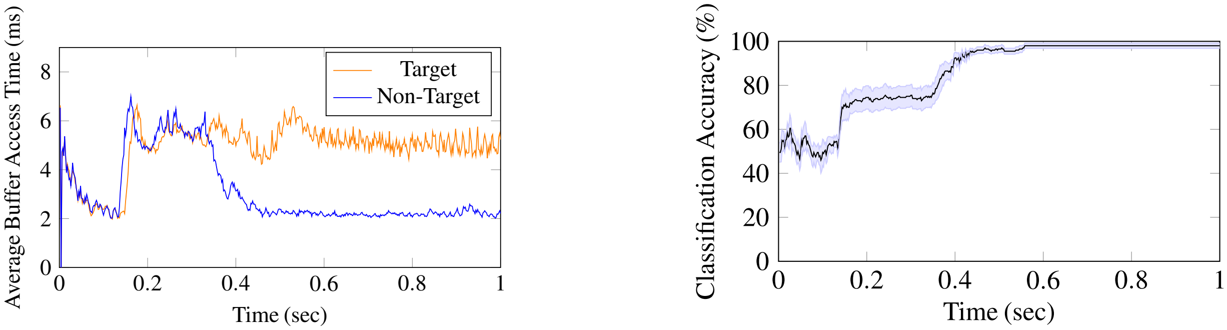
```
function go() {
  ownurl = document.URL + "?run=1";
  window.open(ownurl, ... );
  window.location.href = "SD-URL"; }
if(URLparams["run"] == 1) {
  startCacheAttack();
  waitForPageToLoad(t_a);
  uploadTraceData(); }
```

Figure 4: Embedding method: tab-under.

To launch the attack, the attacker's page first lures the victim to click on the page. The click event allows the attack page to open another window or tab. Instead, however, of launching a pop-up window on top of the existing page, the attacker opens a page which loads in the background. As a result, the user still sees the original attack page.There are two variants to our method: In the pop-under variant, we load the sharing website inside a pop-up window, and then abuse the victim browser's window ordering logic to force the attacker's webpage back into focus. In the tab-under variant, we load a copy of the attacker's webpage in a new tab, and then replace the attacker's old tab with the sharing website using the standard navigation API. The main difference between the two methods is in the programmatic access to the window containing the sharing website content – in the pop-up variant, the attacker has a reference to the sharing website window (as long as COOP does not prevent this), while in the tab-under variant, the attacker and sharing website are completely isolated from a programmatic standpoint.

Fig. 3 describes the *pop-under attack variant*. The `go()` function, executed on user click, starts cache activity measurement (line 2), and then opens a new pop-under window to load the leaky resource (line 3). Because the pop-under window loads in the background, the user does not notice the attack. The attack page, which is in focus, takes cache measurements while the leaky resource is loaded in the pop-under window (line 6). Once the measurements are collected, the pop-under window is closed (line 7). Because the content is loaded in a new window, and not in an `<iframe>`, requests to the sharing website are not subjected to cross-site embedding restrictions employed by these services, and browser third-party cookie-disabling policies do not trigger.

Causing a pop-under window to load behind the active page requires abusing the victim browser's window-ordering logic.



(a) Cache Side-Channel Traces for Targeted and Non-Targeted Users.

(b) Classifier Accuracy vs. Time.

Figure 5: A Proof-of-Concept Attack.

Advertisers are actively looking for these pop-under tricks, and browser vendors are constantly patching them [25, 26]. For the purpose of this paper, we identified a pop-under technique for Safari: Immediately after opening the pop-under window, the attack page opens a second window (ghost), brings focus to it, and then closes it (lines 4-5). In Safari 15.2, closing the ghost window returns focus to the attack page, placing the pop-under window in the background. This happens very quickly and, as a result, the victim does not notice anything unusual happening in the attack page.

Fig. 4 describes the *tab-under attack variant*, which can be used if a pop-under exploit cannot currently be found for the victim's browser. As described in the figure, the `go()` function, which runs upon user click, opens a second instance of the attack page, with an added URL parameter (lines 2-3). The focus is now on this second instance, which looks identical to the first instance, so this action is barely noticeable by the user. The second instance of the attack page now starts collecting cache measurements (lines 5-8). Meanwhile, after opening the new tab, the first instance of the attack page, which is now in the background, navigates to the SD-URL of the shared resource (line 4). Since the first tab is not in focus, the victim does not notice the leaky resource being loaded in this tab. In contrast to the pop-under variant, this variant does not abuse any window ordering APIs, and as such is supported by all of the browsers we evaluated. As a downside, this method does not grant the attacker programmatic access to the tab-under window, making it impossible to close the window after the attack concludes, or to cause it to navigate to another address. Using the tab-under variant, we executed the leaky resource attack successfully in all the browsers we tested, including Safari, Tor, and Chrome.

3.4 Playlists

Conventional deanonymization methods can be scaled to multiple users by loading multiple resources in a row [1, 8]. The tab-under attack variant can only load a single URL, since it lacks programmatic access to the new window. As a result, it is not clear how tab-under attacks can be scaled to target multiple users.

Our Approach: Despite this constraint, we were able to efficiently scale the attack in this setting to a group of target users, when the target users have accounts on a particularly important service – Google/Youtube. This feat is made possible by a unique property of YouTube, related to the way it processes playlists. In YouTube, a user can create a playlist containing multiple videos and share this playlist with viewers through a public URL. If there are private videos in this playlist, and the user currently viewing the playlist is not authorized to view some of them, the YouTube player simply skips the unauthorized videos and plays the rest. Different users, therefore, will each view a different sequence of movies when they view a shared playlist. To exploit this, the attacker shares YouTube videos with target users according to a certain sharing pattern, described in Sec. 4.4, creates a public YouTube playlist that includes these shared videos, and mounts a tab-under attack pointing to the URL of this playlist. The cache trace resulting from playing back this playlist lets the attacker deanonymize an amount of users exponential in the playlist's length.

4 Attacks

Proof of concept. Figure 5 illustrates the concept of our attack. In the experiment illustrated in the figure, the attacker causes the victim to load a resource from a sharing website, in this particular case YouTube, while capturing the side-channel trace using a cache occupancy attack. Sub-figure 5 (a) shows the side-channel trace as a function of time, measured as the average time required to access the attacker's eviction buffer. The two traces show averages made over 100 measurements each of target and non-target states, captured on a machine running Chrome for Windows. As the figure shows, the two traces start identically, but quickly diverge around the 200 ms point. The cache occupancy of the non-target state rises earlier and then returns to an idle state at around 500 ms, while for the target state the occupancy rises slightly later and then remains high. As possible interpretation of these two traces, we hypothesize that the server was slightly faster to respond in the case of a non-target, as previously identified by Watanabe *et al.* [9], but the non-target content returned by the server did not include video content. For the target, on the other hand, content took slightly longer to serve, but it included video content, which generated constant pressure on the cache. This difference in cache occupancy can be quickly captured through a machine learning classifier.

Sub-figure 5 (b) shows the accuracy of a logistic regression classifier, which is provided with increasingly large subsets of the side-channel trace. For each point t in the graph, the classifier is given the side-channel data for the time range $\{0...t\}$, and then its accuracy is measured using 10-fold cross validation. The bold line represents the mean accuracy over the folds, while the light area surrounding it indicates the standard deviation. We see that the accuracy of the classifier starts close to a random guess, rises significantly starting at the 200ms mark, and approaches perfect accuracy after 600ms. As this proof-of-concept experiment shows, an attacker observing the side-channel trace can quickly and effectively tell apart target and non-target states through the cache side channel, without relying on any cross-site leaks. In the following section, we systematically investigate this attack on a variety of websites, browsers, and target hardware microarchitectures.

4.1 Experimental setup

We examined three browsers, each with different default privacy policies and distinct browser engines. The Chrome browser (based on the Blink engine) allows third-party cookies, whereas Safari (based on the Webkit engine) and Tor (based on the Gecko engine) browsers do not allow third-party cookies. We conducted experiments using five system configurations, suggestively named using the combination of OS and browser: Win-Chrome, Win-Tor, Mac-Intel-Safari, Mac-M1-Chrome, Android-Chrome. A detailed specification of these configurations is provided in Table 4 of Appendix A.

Selected Services. We selected the following popular sharing websites to demonstrate the impact of our attacks: Google (including all Google properties such as YouTube, GDrive, Google Photos, GMail, etc.), Twitter, Facebook, Instagram, LinkedIn, Reddit, and TikTok. Together, their user base covers a vast majority of Internet users. The choice of services, browsers, and devices can be further expanded. We made these choices to cover a set of affected users as diverse, as large and as inclusive as possible in a limited amount of time. For YouTube and Reddit, we used the private sharing-based approach, whereas for Twitter, LinkedIn, TikTok, Facebook, and Instagram, we used the blocking-based approach. We alternated between the `<iframe>` embedding method, the *tab-under* method and the *pop-under* method, depending on whether the browser allows cookies with cross-site requests, and on whether the service allows cross-site embedding of its authenticated resources. Additional embedding details for each sharing service are provided in Appendix B.

The attack page. We prepared two attacker accounts on each sharing website, and uploaded a resource to each of the accounts. For the private sharing-based approach, the resource in the first attacker account (Resource A) was privately shared with the victim, and the resource in the second attacker account (Resource B) is not shared with the victim. For the blocking-based approach, both Resource A and Resource B were publicly shared, but the first attacker account blocks the victim, while the second account does not.We then prepared two attack pages: Page A embeds Resource A, and Page B embeds Resource B. Loading Page A simulates a target user, whereas loading Page B simulates a non-target user. We interleaved loading Page A and Page B, to make sure the classifier is trained on the difference between states, and not on the global state of the system. To automate the experiments, we used Selenium (for Windows-based systems), AppleScript (for MacOS-based systems), and Samsung Remote Test Lab (for mobiles). The attack pages were hosted on a Windows Server 2019 running on Amazon AWS EC2.

Data analysis methodology. We use supervised machine learning to analyze the cache measurement data. To build our data sets, we collect cache occupancy samples while a target and a non-target user load the attack page. For single-target attacks, we chose logistic regression after a pilot experiment with multiple classifiers. For multi-target attacks, we chose a long short-term memory (LSTM) neural network model which was shown to be effective for multiclass classification when used for website fingerprinting attacks using cache side channels [7]. The parameters used for these classifiers are provided in Appendix A.

For each attack setting, a subset of the samples is used to train a classifier, which is then used to predict whether a user loading the attack page is the targeted victim. We determined experimentally that a dataset of 200 samples (100 target samples and 100 non-target samples) is sufficient to yield high attack accuracy. To prevent over-fitting the classifier to the data set, we apply 10-fold cross validation. The per-fold accuracies are then combined to produce a single estimate for the mean and standard deviation of the attack accuracy. Analysis was performed using Scikit-Learn v1.0.1 [27] and TensorFlow v2.7.0 [28] with Python v2.7.12 in Google Colaboratory [29].

4.2 Experimental Results

Table 1 shows the attack accuracy for the various system configurations we considered. The table also includes results with the `Leakuidator+` defense enabled, which will be introduced in Sec. 5. Overall, the attacks have over 90% accuracy for a majority of the 28 attack setups considered, indicating that cache-based deanonymization attacks are effective across a variety of services, browsers, and microarchitectures.

Several factors affected the value of t_a. The primary factor was the precision of the browser's time measurement API. Due to the high precision of the time API on Chrome and Safari, t_a was generally under 3 seconds for the Win-Chrome and the Mac-Intel-Safari system. The Tor Browser has a lower-precision time API, and the Tor network has higher latency. As a result, the Win-Tor system had a higher t_a of 5 to

System	Win-Chrome			Win-Tor			Mac-Intel-Safari		Mac-M1-Chrome		
Service	t_a	Base	+Defense	t_a	Base	+Defense	t_a	Base	t_a	Base	+Defense
Google	1	98±2.5	51±8.6	5	92±8.1	47±6.8	1	100±0	1	100±0	45±7.4
Twitter	2	97.5±3.4	46.5±9.5	5	94.5±4.2	47.5±9.3	2	100±0	1	98.5±3.2	49±9.2
LinkedIn	2	100±0	55±7.8	5	84.5±6.1	44.5±10.1	1	86.5±6.3	1	98.5±2.3	53.5±14.5
TikTok	3	84.5±5.7	51.5±5.5	5	93±6	51±12.8	3	91.5±5	2	98±3.3	55.5±8.5
Facebook	2	100±0	41±10.4	5	96.5±5	44±10.4	5	84±7	1	97.5±3.4	44±7
Instagram	1	88.5±7.1	51±8.3	10	76.5±8.4	54±8.3	2	92.5±3.4	1	95.5±4.7	45±10.5
Reddit	3	89.5±8.5	45±11	8	70.5±12.5	48±9.3	3	88±5.6	3	81±7	51±11.6

Table 1: Summary of experimental results. Attack accuracy (%) is shown both before and after applying `Leakuidator+`. t_a is the attack duration in seconds. Average and standard deviation are obtained using 10-fold cross-validation as described in Sec. 4.1.

10 seconds. An additional factor is the way website behavior impacts exposing differences in user states. For example, YouTube initially loads a player and then, depending on user state, either plays the privately shared video automatically, or does not load the video at all. As a result, the initial part of the side-channel trace, in which the player is loaded, does not contribute to accuracy. Similarly, some websites, in particular Instagram, TikTok and Reddit, expose smaller but continuous differences between user states. For these websites, the classifier needs a longer t_a to reach peak accuracy.

The attack on the Mac-M1-Chrome system required additional fine-tuning. Although Chrome has sub-millisecond time API precision, using the Cache Occupancy measurement method was not enough to capture cache activity patterns, due to the high speed of the M1's cache. To overcome this limitation, we used the Sweep Counting method with a 10 millisecond measurement interval. This yielded a high attack accuracy for all sharing websites.

In general, we experimented with videos of various sizes and durations, all resulting in successful attacks; the smallest video was 3.84KB with 1s duration. Since the videos are streamed by an embedded video player from the sharing service, the attacker is less concerned about their size or duration, as long as they play long enough for the attack.

4.3 Attacking Mobile Phones

The attacks described so far are desktop-centric. Most significantly, they assume the victim is logging in to the sharing website through a web browser. Many users, however, access sharing websites through their mobile phones. In contrast to desktop users, mobile phone users do not tend to use the web browser installed on the mobile phone to access services such as Twitter, GMail and Instagram, relying instead on dedicated apps. As a result, the mobile browser does not typically have cookies for the targeted websites.

There is one case, however, in which the mobile browser is almost universally logged in: The Chrome browser, which is installed on Android phones, is tightly integrated with Google services. The browser encourages users to "Sign into Chrome", an action that effectively causes the browser to log into all Google services. Due to this feature, it is possible to deanonymize Android users based on their GMail email addresses, as we show next.

We carried out a limited evaluation of our attack on the Android-Chrome system, an ARM-based Samsung Galaxy S21 device described in more detail in Table 4. We first opened the Android Chrome browser and followed the prompts to log in to Google services. Next, we browsed to a web page containing an embedded Google Drive video shared only with the target user, and collected side-channel traces using the sweep counting method. We collected 100 traces each for the target and non-target state, and evaluated the performance of our classifier, using the methodology described in Section 4.1. This yielded an attack accuracy of 91%, indicating that *our deanonymization attack is effective in a mobile setting as well.*

One concerning aspect of our mobile phone attack is the issue of mobile browser extensions. Whereas the desktop version of Chrome allows its behavior to be modified by third-party browser extensions, the mobile version of Chrome has no extension support. Thus, it is not possible to install our defense on this target, as we discuss in more detail in Sec. 7.

4.4 Scaling to Multiple Targets

In some situations, an attacker may want to target a group of users instead of a single user. The goal of the attacker is to identify which specific user among a list of n target users is visiting a particular website. Staicu *et al.* [1] showed this can be done efficiently in the context of previously known XS-leaks, by using $log(n)$ leaky resources.

A question then arises: *"How can we scale the attack to target a group of users under the new attack scenarios introduced in this work?"* In this section, we provide concrete techniques to scale the attacks under these new scenarios.

Staicu *et al.* [1] proposed to scale the attack by privately sharing each of the $log(n)$ resources with a subset of the n users, such that the specific subset of resources loaded by the attacker webpage will reveal the identity of the user. This basic pattern also works with the blocking-based approach, except that the attacker uses $log(n)$ attacker accounts, each of which owns one of the $log(n)$ resources [8].

The attack: In the *training* phase, the attacker takes cache measurements and builds a cache profile for each of n states,

Service	System	t_a	Base	+Defense
LinkedIn x 8	Win-Chrome	6	99.12±0.8	15.63±3.55
Twitter x 8	Mac-Intel-Safari	6	97±1.9	N/A
YouTube x 8	Win-Tor	45	78.88±3.33	11.75±3.07

Table 2: Scalable attack results for 8 user states. Attack accuracy (%) is shown both before and after applying the `Leakuidator+` defense. t_a is the attack duration in seconds.

corresponding for the target users. These measurements are used to train a machine learning classifier. As a prerequisite to scale up the attack, the attacker must be able to load multiple shared resources during a single visit of the victim to the attacker's website. If the browser allows third-party cookies, and if the sharing service allows cross-site embedding, we load multiple `<iframe>` elements, each containing a different shared resource. This method will not work, however, for sharing websites which do not provide an embedding option, or for browsers such as Safari or Tor which restrict third-party cookies in cross-site requests. In these settings, like in the single-user attack, we use the pop-under and tab-under methods, as described in Sec. 3.3: For Safari, we use a pop-under technique which allows us to load different shared resources sequentially in the pop-under window by changing the pop-under window's `window.location` field. For the Tor Browser, we use the tab-under technique, loading the URL of a YouTube playlist containing multiple shared videos in a background tab. We note that this approach works only for Google/YouTube.

A sharing pattern for playlists: The basic sharing pattern is not effective for Tor, because if one video in the playlist does not load, the playlist goes to play the next video and we cannot determine if the previous video was loaded or not. Instead, we leverage the fact that the duration of a video and the time the player switches between videos in the list is a source of difference in cache profiles. The playlist contains $log(n)$ pairs of videos, where each pair has two videos of different duration: one short (s) and one long (ℓ). Thus, the playlist contains videos $V_1^s, V_1^\ell, V_2^s, V_2^\ell, \ldots, V_{log(n)}^s, V_{log(n)}^\ell$. For each user i (with $1 \leq i \leq n$), consider the binary representation of i as $b_1 b_2 \ldots b_{log(n)}$. We associate each of the $log(n)$ pairs of videos with a bit in this binary representation. For $1 \leq j \leq log(n)$, if b_j is 0, then the attacker shares privately with user i the video V_j^s, otherwise the attacker shares the video V_j^ℓ. As a result, the playlist plays $log(n)$ videos, and the specific combination of videos will be used to identify the user.

Scaling Evaluation: Since the attack requires only a logarithmic number of leaky resources relative to the number of targeted users, the attack can be scaled to track thousands of users while still requiring a reasonable amount of time. As a proof of concept, we evaluated the effectiveness of the scaled multi-user attack with 8 states (seven states for the targeted users, plus one state for non-target users). We considered three settings: 1) LinkedIn under Chrome for Windows is a setting where third-party cookies are supported by the browser; 2) Twitter under Safari for Mac is a setting where third-party cookies are not supported by the browser, but there exists a pop-under method allowing post-popup navigation. 3) Google/Youtube under Tor for Windows is the most extreme setting, where third-party cookies are not supported by the browser, and there is no pop-under method which allows post-popup navigation. The attack could tell the victims apart with high accuracies in all three settings, as indicated in Table 2.

5 Defenses

We now turn to the design of a countermeasure against the attacks we discovered. Our attack operates at the microarchitectural level, learning about the victim's state by observing the CPU cache. As such, it cannot be obstructed neither by software-based isolation mechanisms such as `SameSite` cookies, cross-origin read blocking or cross-origin opener policies, nor by server-side isolation mechanisms such as self-reloading landing pages [12]. Instead, we turn to techniques from the field of side-channel defenses.

As stated by Mangard *et al.* [30], there are two general defense approaches against side-channel attacks. The first is mitigation, or hiding, which tries to make attacks impractical by reducing the signal-to-noise ratio of the side-channel trace. The second is prevention, or masking, which tries to make attacks theoretically impossible by removing all dependencies between the side-channel trace and any secret-bearing computation. We first evaluate a mitigation-type defense, which is simpler to design and implement. Specifically, we ran external code that generated artificial cache noise while the cache trace was collected, as well as playing videos or loading websites in other tabs of the browser. We found that a noise-based defense is not effective against a well-prepared attacker. As a result, we focused on a more systematic approach based on side-channel leakage prevention.

5.1 A First Approach: Adding Artificial Noise

The first defense we evaluated was a simple noise-based hiding defense. Specifically, we ran external code that generated artificial cache noise while the cache trace was collected, and checked whether this added noise can prevent the detection of the cache signatures required by our attacks.

We considered two sources for cache noise: CPU stress tests and web browsing activity. For stress-test noise, we evaluated four CPU cache-focused `stress-ng` tests [31]: binary search (*bsearch*), heap-sort (*heap*), wide-spread memory reads and writes (*cache*), CPU-intensive operations (*cpu*) [2].

For web-browsing noise, we evaluated two web-browsing activities: a YouTube video player (*play*), and a Wikipedia webpage which was reloaded once per second (*wiki*). These

[2] Exact command line parameters were as follows:
bsearch: stress-ng –bsearch 0
heap: stress-ng –heapsort 0
cache: stress-ng –cache 0 –cache-level 3 –cache-ways 16
cpu: stress-ng –cpu 8

	bsearch	*heap*	*cache*	*cpu*	*play*	*wiki*
No-Noise	47.5%	50%	50%	49%	50%	99%
Known-Noise	87%	87.5%	84%	79%	99%	99%
Unknown-Noise	N/A	N/A	83%	84%	95%	98%

Table 3: Attack accuracy under various types of noise. Stress tests: binary search (*bsearch*), heapsort (*heap*), wide spread memory reads and writes (*cache*), and CPU intensive operations (*cpu*). Web browsing: YouTube player (*play*) and Wikipedia page (*wiki*).

activities were performed in a second browser tab loaded together with the attack page.

We considered three attack scenarios, which simulate different amounts of information the adversary has about the defenses employed by the victim. In the *No-Noise Scenario*, training was done in the absence of noise, while testing was done in the presence of noise. In the *Known-Noise Scenario*, the traces used both for training and for testing were gathered in the presence of the same type of noise. Finally, in the *Unknown-Noise*, training was done on data gathered under four types of noise (no noise, *bsearch, heapsort, play*), and testing was done on data gathered under the four other types of noise (*cache, cpu, play, wiki*). All three scenarios were evaluated with an attack page that embeds a YouTube video in an `<iframe>` in the Chrome browser under Windows, and that uses a regular cache occupancy method.

Table 3 shows the results of this noise-based defense. We observe that under the *No-Noise* scenario, the attack accuracy remains around 50% for all noise sources other than *wiki*, suggesting the noise-based approach may be effective against an unprepared adversary. Unfortunately, this is not the case when the adversary has prior awareness of this noise-based defense: Under the *Known-Noise* scenario, the attack accuracy varies between 79% and 87.5%, which is somewhat lower than the 98% accuracy observed in the absence of all defenses, but still significantly higher than the base rate of 50%. To make things worse, as the *Unknown-Noise* scenario shows, attacks are still possible even when the attacker does not know the type of noise the victim plans to use as a defense. We therefore conclude that a simple noise-based defense is not an effective countermeasure against our attacks.

5.2 Leakuidator+

We now describe the design and implementation of our main defense proposed in this work, `Leakuidator+`. The countermeasure is compatible with the desktop versions of Chrome, Firefox and Tor Browser, and is already available on the Chrome and Firefox extension stores [10, 11].

`Leakuidator+` is based on `Leakuidator` [3], a previously-proposed client-side defense designed to protect against XS-leak-based targeted deanonymization. We first describe the original `Leakuidator` defense, highlight the changes made to this countermeasure to make it effective against the novel attacks described in this paper, and finally show an experimental validation of the effectiveness of the proposed defense.

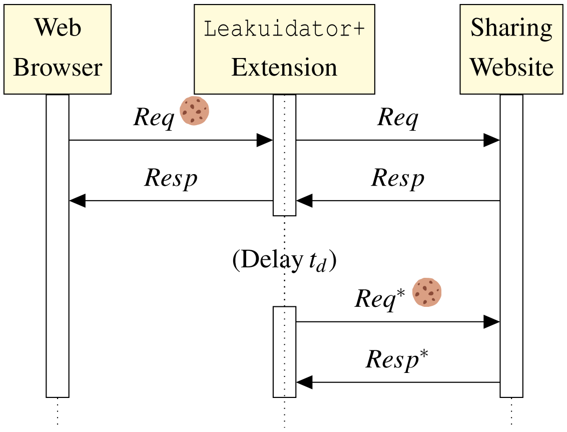


Figure 6: Interaction Diagram for `Leakuidator+`.

Figure 6 shows the exchange of messages between the browser, the extension, and ultimately the sharing website. As the figure shows, the process starts when the browser sends a web request *Req*, together with cookies, to the sharing website. When the extension intercepts this request, it classifies the request as potentially risky if the request contains cookies and is cross-site. In that case, the extension strips the authentication cookies from the request, and only then passes it on to the sharing website. Since the sharing website does not have access to the authentication cookies, its response *Resp* trivially contains no identifying information about the user. When the extension receives the response *Resp*, it passes it directly to the browser for rendering.

This mechanism can be used to block all third-party cookies. Such behavior, however, is not appropriate in many cases – cookies are important for many existing web functionalities, including analytics and tracking. To remain compatible with these use-cases, the extension generates a fresh request, labeled *Req**, containing the cookies stripped from *Req*. The extension then sends *Req** to the sharing website in the form of a HEAD request. The additional delay t_d between the transmission of *Req* and the transmission of *Req** is unique for `Leakuidator+`, and the reason for its inclusion will be described below. The personalized response that the sharing website sends back, labeled *Resp**, is never forwarded to the browser for rendering – it is only analyzed by the browser extension. As long as the browser's extension API prevents the webpage from accessing the fields of *Resp**, the user is again protected from XS-leak-based deanonymization. The extension finally compares *Resp* and *Resp**. If there are any observable differences between the two, the extension indicates this to the user through the browser toolbar.

We made a series of changes to `Leakuidator` so that it protects against the new attacks proposed in this paper.

Protecting against pop-unders and tab-unders. The original Leakuidator was only configured to offer protection in cases of cross-origin web requests. This covers the existing class of XS-leak-based attacks, but specifically excludes any first-party requests from protection. Hence, no protection is provided against the pop-under and tab-under embedding methods used by our attacks. Since wholesale blocking of all first-party cookies would immediately break the functionality of many web pages, we selected a more refined approach to decide when to activate our protection. Specifically, Leakuidator+ keeps track of which browser tabs and windows were created by which webpage, creating groups of related tabs and windows. This is done by monitoring the *webNavigation* API to detect when a new tab or window is opened, and recording the relations between parent and child. When a request is made in either a parent or a child window/tab, the extension detects if the request's domain is different from the top-level domain in the related window/tab, and applies the defense. Leakuidator+ excludes from the defense any tabs that are manually created by the user (*e.g.*. by clicking on the "+" or "New Tab" button).

Removing residual side-channel leaks. As Leakuidator was shown to be effective at preventing leaky resource attacks based on known XS-leaks, we expected it to be immediately effective against the side channel-based leaks investigated in this work. We were instead surprised to find that it is ineffective. For example, when we launched an attack using an image hosted on Google Drive in the Chrome browser, we could visually observe differences in the CPU cache side-channel measurements between target and non-target users, even when Leakuidator was enabled. These differences were also exploitable by our machine learning classifier. As a result, our attack remained highly effective despite the presence of Leakuidator, achieving a 86% attack accuracy instead of the expected base-rate of 50%. This finding is counter-intuitive – when Leakuidator is installed, the server does not respond with the image content, and the browser does not render any leaky resources.

We performed a detailed analysis to understand this finding, and uncovered two subtle reasons that cause observable differences in the side-channel measurements. First, there is a *server side channel* related to *Resp**, (*i.e.*, the response to *Req**, the second request initiated by Leakuidator). In particular, we noticed that the server takes a different time to respond, if a user is allowed or not to access the shared resource. This timing side channel was originally used by Watanabe *et al.* to launch deanonymization attacks on several popular services using XS-leaks [8, 9]. In our case, even though *Resp** is not forwarded to the browser, and therefore not available to an attacker using XS-leaks, the mere fact that it is processed by the browser's extension framework is enough to cause an exploitable side-channel difference.

Second, we discovered a *client side channel* in the extension itself. The extension performs various operations on *Req* and *Resp*, including recording header names and values, using them to prepare *Req**, and finally comparing the fields of *Resp** and *Resp*. Thus, the extension itself amplified the differences between the target and non-target user states, resulting in observable differences.

To mitigate the *server side channel*, Leakuidator+ adds a small random delay t_d before sending the second request, *Req**. This delay also randomizes the arrival time of the server response *Resp**, making it impractical for an attacker to perform an attack based on this signal. Since the largest value we observed for the timing side channel was on the order of 100 ms, we chose t_d uniformly between 0 and 1 second. We note that since *Resp** is not sent to the browser for rendering, the only user-noticeable side-effect of this added delay is a slightly delayed notification in the browser toolbar.

To mitigate the *client side channel*, Leakuidator+ minimizes the operations performed while analyzing the request and response headers, limiting itself to only inspect and record the headers that are strictly necessary. Instead of recording and using the headers from *Req*, Leakuidator+ relies on the browser to prepare *Req** headers. Also, Leakuidator+ only records the *Resp* headers used for comparison with *Resp**, instead of recording all *Resp* headers.

Preserving legitimate functionality. It is natural to analyze the impact of our defense on legitimate website functionality. We note that Leakuidator+ is built on top of Leakuidator, which was designed to preserve functionality such as user tracking and analytics. In legitimate scenarios such as third-party authentication requests, Leakuidator+ notifies the user about the request, requiring them to interact with the extension and complete the process after marking the request as safe.

Extending support to additional browsers. The original Leakuidator extension was only usable on Google Chrome and other Chromium-based browsers such as Microsoft Edge, Brave, Opera and Yandex. Since additional browsers now support the same WebExtension API offered by Chrome, we ported the extension to the Firefox and Tor browsers as well.

Evaluation. We performed a comprehensive set of experiments to validate Leakuidator+'s effectiveness. As summarized in Table 1, when Leakuidator+ is enabled, the attack accuracy becomes equivalent to that of a random guess. Our countermeasure was able to prevent the attacks we discovered on multiple websites, multiple browsers, and multiple hardware microarchitectures, all while remaining compatible with existing uses for cookies in navigation, tracking and analytics.

We also evaluated Leakuidator+'s effectiveness against attacks targeting a group of users. Table 2 shows the attack's accuracy drops near the baseline level of 12.5%, which is equivalent to a random guess for the considered 8-state setup.

Leakuidator [3] was shown to incur a small performance overhead, thus minimally impacting user experience. Leakuidator+ does not make changes that would significantly affect those overheads, including load time, number of requests, and data transferred.

Security Analysis. Leakuidator+ provides by design protection against the main known XS-leak types, such as those described in Sec. 2.1. We have also shown experimentally that the defense renders cache-based attacks impractical.

Recently, Knittel *et al.* [4] introduced a formal model for XS-leaks, building on work of Sudhodanan *et al.* [2]. The authors systematically search for XS-leaks and find 14 new attack types grouped in four categories. In the remainder of this section, we describe how `Leakuidator+` protects against these. Although our analysis does not necessarily guarantee protection against new unknown XS-leaks, we view it as compelling evidence that `Leakuidator+` is an effective defense mechanism against targeted deanonymization attacks.

Leak Technique: Global Limits exploits browser limits. The limit on number of WebSocket connections allows an attacker to differentiate user states by detecting a webpage's number of WebSocket connections [32]. `Leakuidator+` removes cookies from the initial GET request, resulting in same number of connections in different user states. The response to the HEAD request initiated by `Leakuidator+` is not rendered by the browser, thus no connections are established.

The limit on the number of UI elements for the Payment API allows an advertiser to learn whether a user attempted to purchase an advertised item after clicking on an affiliate link, and is not a deanonymization attack.

Leak Technique: Performance API allows an attacker to differentiate user states by inspecting the browser's Performance entries. It was previously used to detect the X-Frame-Options header in Google Chrome [33]. When `Leakuidator+` is enabled, *Resp* has same effect on these entries in both target and non-target states. To eliminate any potential leak that could arise from Performance entries related to $Resp^*$, `Leakuidator+` removes these entries when $Resp^*$ arrives.

Leak Technique: Error Messages allow an attacker to learn the target of a redirect. In Webkit-based browsers, primarily Safari, if a CORS-enabled request fails, it is possible to access CORS error messages, including the full URL of the redirect target; in addition, the Subresource Integrity error message can leak the response size. These XS-leaks could possibly be used for targeted deanonymization if the errors rely on authentication cookies sent along with cross-site requests. However, Safari blocks third-party cookies by default.

Leak Technique: Readable Attributes. Web apps can use the Cross-Origin Opener Policy (COOP) to prevent other websites from gaining arbitrary window references to the application, *e.g.*, through pop-up windows. Reading the value of the *contentWindow* attribute may allow an attacker to learn if COOP is enabled and thus potentially differentiate between user states. `Leakuidator+` protects against this by applying the defense to groups of related tabs/windows that have different top-level domains.

6 Related Work

XS-leaks usually exploit cross-site information in a binary form: questions with YES or NO answers, where the response is visible to the attacker. xsleaks.dev is a community-driven website dedicated to collecting knowledge about APIs that can be used for such cross-site leaks. These include window references, frame counting, error events, navigation, response cache probing, ID attribute, postMessage broadcasts, CORB and CORP leaks, and timing attacks [1, 2, 23, 24, 34–55].

There has been academic effort to give a structure to XS-leaks by classification. Recently, Knittel *et al.* [4] introduced a formal model for XS-leaks, building on work of Sudhodanan *et al.* [2], and systematically searched for new XS-leak attack classes. On the defense side, they argue that if at least one browser is immune to a certain leak technique, this technique can be fixed in other browsers as well, by changing their implementation. Our client-side defense, in contrast, does not depend on browser vendors and website owners, and can be used immediately. We observe that many of the defenses proposed to mitigate XS-leaks were not designed to protect against side channels. As our work shows, attacks based on side channels bypass these software-imposed boundaries.

Targeted deanonymization is an example of privacy leakage through XS-leaks [1, 3]. In response to XS-leaks, a number of defense mechanisms were proposed, including response cache protections, subresource protections, fetch metadata, cross-origin opener and resource policies, framing protections, SameSite cookies, isolation policies, cross-origin read blocking, and the partitioned HTTP cache [13, 14, 22, 56–67].

Cache attacks were proposed simultaneously by Percival and by Osvik *et al.* [16, 68], and first demonstrated on the last-level cache by Liu *et al.* [17]. Oren *et al.* presented a JavaScript implementation of the last-level cache attack [69], and Shusterman *et al.* presented the cache occupancy and sweep counting variants, which can be run in more restricted browser environments [6, 70]. Several works have explored the use of micro-architectural side-channel attacks for attacks on privacy. Jana *et al.* introduced the memory footprint side channel, and showed how a malicious Android app can infer fine-grained web-related information about a user, including personal interests and login status [71]. Gülmezoglu *et al.* showed how cache attacks can learn about running applications in a cloud scenario [72]. Gülmezoglu *et al.* also showed how a native Android app can use the cache to discover running applications, website activity and even which videos the victim was streaming [5]. To the best of our knowledge, we are the first to introduce targeted deanonymization on the web using the CPU cache side channel.

Many works have tried using different side-channel attack methods to infer browsing activity, including power consumption, GPU leaks, data statistics, performance counters, and event loops [46, 73–80]. In contrast with the deanonymization scenario, in which the attacker actively induces the victim to load a resource, most of these works assume that the attacker passively observes the victim. It is interesting to consider how these additional methods could be applied to targeted deanonymization, but we believe our defense should be effective regardless of the method used by the attacker.

7 Ethics, Disclosure and Guidance

The deanonymization attacks described in this paper are both practical and dangerous, and can impact the privacy of journalists, activists, and other vulnerable populations. While we provide a browser extension that serves as a countermeasure against these attacks, and experimentally verify its effectiveness, there are several scenarios in which this countermeasure is impossible to deploy. Most significantly, the current implementation of the WebExtension API on Apple's Safari browser is not compatible with our extension, and the official mobile version of Chrome provided by Google does not support extensions at all. Users of these browsers will thus be unable to defend themselves against the attacks described in this paper, until content sharing sites make non-trivial changes to the way they allow content to be shared.

It is our ethical responsibility to minimize the risk to these users. We have opened bug reports with browser vendors (Chromium [81], Firefox [82], Edge, Safari, WebKit, the Tor Project), and are sharing a draft of this paper with affected services including Google, Twitter, Meta, Microsoft, TikTok, Reddit, and Apple. We also consider journalists and activists part of the disclosure process, and have reached out to the organizations who advocate for them through the EFF. Until the responsible disclosure process concludes, we plan to embargo the results. Our countermeasure is already available in the Chrome and Firefox extension stores, and can be immediately installed even before the attacks are publicized. When the disclosure process is over, we plan to publicize an easy-to-understand description of the attack and how to mitigate it, and to work with relevant stakeholders to make sure potential victims know how to protect themselves. Below we provide additional advice on ways to limit the attack's effectiveness.

The attack works on websites even when using VPN, since it targets the browser's rendering process and not the network stack. The attack will not work on websites opened in incognito mode, unless the user explicitly logs in to the website from the incognito session.

Guidance to Website Owners. As discussed in section 5.2, there are two main causes for differences in the observed side-channel leakages between targeted and non-targeted users – a server-side timing difference and a client-side rendering difference. These differences can be mitigated through careful design by website owners. As a positive example, we note that Apple's iCloud service applies most of these design principles, and, as a result, we were not able to attack it using our technique. Web servers typically have an authorization module, which checks if a user is allowed to access a resource, followed by a content delivery module, which actually makes the resource available to the client. If an authorized user loads the resource, both authorization and content delivery modules need to run. For non-authorized users, on the other hand, content delivery is not invoked at all, resulting in a faster response time which can be observed. While we measured this faster response time using a cache side channel, any other side channel that can monitor traffic can also detect this difference, for example the congestion-based method used by Schuster *et al.* [83]. To mitigate the timing side channel, web servers should thus be designed to **return their responses in constant time, regardless of the authorization status of the user**. To mitigate client-side rendering side channels, web servers should **make their error pages as similar as possible to their content pages**. This will make it more difficult for a side-channel attacker to distinguish between the two. As an example, if an authorized user was going to be shown a video, the error page for the non-targeted user should also be made to show a video. In general, website owners should minimize any kind of attacker-observable difference in responses they send between the two states. In addition, websites should **require user interaction before rendering content:** The scalable attack we showed on Tor, as well as several of the video-based attacks, relied on the fact that browsers automatically play shared videos, even if they are loaded in a background page. The added cache activity resulting from this video playback makes it very easy for the classifier to tell apart users. To prevent this, website owners can make sure that videos shared with only a subset of users require some sort of user interaction before they are played. In general, if there are any operations which cause an unavoidable difference in cache activity (for example playing a video or decompressing a file), we recommend that the website first asks the user to confirm this activity. Websites should also consider **replacing blocking with "shadow-banning"**. Blocking public content from a particular user is arguably an exercise in inconvenience – all the blocked user needs to do to access this content is simply open a private browsing window. The shadow-banning technique applies a different approach to blocking. A shadow-banned user is apparently able to interact with the website, including viewing content, creating posts and posting comments. All of the user's comments and posts, however, are invisible to other users. In this approach, the public posts of users are always accessible to other users, including those whom they ban. As a result, it is not possible to use selective shadow-banning to apply targeted deanonymization. On the other hand, since other users are not exposed to the shadow-banned user's content, the website operators can achieve their goal of controlling the discourse on the website. Finally, websites should **notify users upon sharing or blocking.** Google Drive and other sharing websites allow content to be shared without notifying the recipient. Similarly, many sites do not provide any way for a user to know when, or by whom, they are blocked. This behavior increases the risk of the attacks we described, since the target user has no way of knowing he or she is targeted. To reduce this risk, website owners should always notify users when they have content shared with them, or when they are blocked by another user. To minimize cognitive and emotional discomfort, websites can consider how to selectively suppress some of these notifications without sacrificing security.

Guidance to Browser Vendors. The browser serves as host both for the attacker and for the victim. A browser which can isolate the cache activity of the victim from the spying eyes of the attacker, or which can prevent the attacker's code from performing cache occupancy measurements, would be the ideal countermeasure. This is, unfortunately, a task which may be impossible to carry out without redesigning the browser, the operating system or even the CPU [84]. Even before this protected browser becomes available, several engineering fixes to current browsers can raise the bar for the attackers. First, browsers should **consider pop-unders as a security threat**. Pop-under windows and tabs are truly annoying. Advertisers are always looking for new methods for launching these pop-unders, and browser vendors are constantly tweaking the browser's window management logic to prevent them [25, 26]. Going forward from the results in this work, we argue that browser vendors should no longer consider pop-unders as a mere annoyance, but instead consider them as security risks. This includes both actively blocking this browsing pattern, and applying cross-site protections to content loaded into pop-ups. Browser vendors should also **allow browser extensions to modify request headers**. The defense we presented works by carefully processing the header fields of the requests sent out by the browser – inspecting fields in the request headers, comparing two responses to search for privacy leaks, and ultimately changing or removing fields – removing cookie headers from requests and set-cookie headers from responses. All of these stateful processing steps are made possible by an extension API named `webRequestBlocking`, which allows extensions to intercept, block, or modify requests in-flight. Unfortunately, Google has announced that this API is being phased out [85]. Firefox did not currently announce plans to remove support for `webRequestBlocking`, and the Safari extension API does not support it at all. The API designed to replace `webRequestBlocking`, named `declarativeNetRequest`, may be appropriate for list-based ad blockers, but is not usable for our browser extension. Our work shows the importance of allowing browser extensions to statefully intercept and modify web requests. We urge browser vendors to keep the `webRequestBlocking` option available for extensions, and urge vendors who do not currently support it to make this feature a priority.

Guidance to Standards Bodies. The WWW specification already includes a set of standards designed to isolate web content from malicious third parties. These include resource policies, opener policies, cookie isolation, and similar defenses. Common to all of these defenses is the assumption that two pages programmatically isolated from each other are not able to interact. The attacks presented in this work show that this assumption must be reconsidered. In particular, the cross-origin read blocking (CORB) feature was already shown to be less effective in the presence of side channels [52]. We suggest that the CORB feature be extended to pop-under and tab-under contexts, similar to the way in which we extended `Leakuidator`: Any web page *opened* by another web page should also be subjected to CORB restrictions, even if it is opened in a separate window.

Guidance to Users. Users who are at increased risk of being targeted online, such as journalists, activists, and religious leaders, are already instructed to be more careful online than other users, for instance when opening attachments, responding to friend requests, clicking on unknown links, and so on. We provide here some guidance specific to the cache-based targeted deanonymization attack, and will be cooperating with advocacy groups to bring this guidance to the knowledge of relevant users as part of the disclosure process. The best suggestion we can provide is to **install our browser extension,** `Leakuidator+`, which is already available on both the Google Play Store and the Firefox Add-ons website [10, 11]. As Sec. 5 describes in detail, the extension protects against all of the attacks we described in the paper, with a minimal impact on functionality and compatibility. It should be noted that the current Android version of Chrome does not support extensions. Firefox for Android, as well as several third-party Android browsers based on the open-source Chromium code (notably Kiwi and Yandex), do support extensions, but testing the compatibility of our extension with these browsers remains a task for future work. Users should also **avoid unnecessary logins**. Websites such as GMail, Twitter, Facebook and Instagram make it useful and convenient to be constantly logged in. This behavior pattern is especially enforced by Google, through their control over the browser, the website, and in some cases the device itself. This behavior also unfortunately increases the risk of deanonymization attacks. To protect themselves, privacy-conscious users should only log in to websites when they plan to actively use them, and make sure to log out when they are done. It should be noted that the Tor Browser keeps cookies stored in memory as long as the browser is running – if a user opens GMail, and then closes the tab, the Google cookie remains present on the browser until the user manually logs out, deletes cookies or quits the browser. Users should also **consider using multiple devices:** The best way to prevent side-channel attacks is to isolate the source from the receiver. A reasonable and practical way to achieve this for sensitive users would be to invest in multiple cheap devices, each dedicated to a single online service. As a moderate alternative to above, users can use multiple sessions in their browser, for example by using the Multi-Account Containers add-on in Firefox, the Add Profile feature in Edge, or the Multiple People feature in Chrome. Tor Browser also has a (very prominent) "New Identity" button that closes and reopens the browser with a click of a button.

An Alternative Defense Approach: A new potential defense strategy against the popunder and tabunder attack variants emerged from our discussions with the affected services and the browser vendors. Instead of relying on users to install a browser extension, a similar functionality can be achieved by dividing the responsibility for detecting and reacting to potentially suspicious requests between browser vendors and sharing service operators: The browser provides additional information about the context in which a request is made (through the request headers), and the sharing service uses this information to decide how to respond to potentially suspicious requests. We initiated a proposal to extend the W3C standard for fetch metadata HTTP request headers [86].

8 Concluding Remarks

In this paper, we have introduced novel attack techniques for targeted deanonymization on the web, which can uniquely identify a target user when leaky resources are rendered in the user's browser. The attacks leverage CPU cache side channels to bypass software-imposed boundaries and are shown to be effective across multiple architectures. Our work reveals that the attack surface for targeted deanonymization attacks is drastically larger than previously considered. We experimentally show that several popular resource sharing services can be leveraged to conduct the attack. When considering together the collection of users of these services, we conclude that a large majority of Internet users are vulnerable.

To defend against this threat, we provide a comprehensive countermeasure against all of the attacks we discovered. `Leakuidator+` is a client-side defense that can be deployed right away as a browser extension, without depending on browser vendors and website owners. We also provide guidance to websites and browser vendors, as well as to individuals who are unable to install our browser extension.

Future Work. Targeted deanonymization via the cache side channel is a powerful attack mechanism. Whereas we showed multiple avenues that are readily available to attackers, it would be desirable to further improve the protection landscape. As future work, we plan to further explore and improve usability aspects of the proposed `Leakuidator+` defense. In addition, we believe it is crucial to work with browser vendors and standards bodies to explore comprehensive mechanisms that can start addressing the fundamental underlying causes of cache side channel-based targeted deanonymization attacks.

Artifact Availability. We provide a dataset of cache traces for single and multi-target attacks, together with a Google Colab document showing how to use classifiers on these datasets, as well as sample attack pages for the `<iframe>`, *pop-under* and *tab-under* embedding methods. In addition, we provide an online-only attack page (as described in Appendix C). The artifact repository can be accessed using: `git clone https://github.com/leakuidatorplusteam/artifacts.git`.

The complete source code for `Leakuidator+` is available through the Firefox and Chrome extension stores [10, 11].

Acknowledgments. We would like to thank the USENIX Security reviewers and Giancarlo Pellegrino for reviewing this paper. This research was supported by the US NSF (National Science Foundation) under Grants No. CNS 1801430, DGE 1565478, and DGE 2043104.

References

[1] Cristian-Alexandru Staicu and Michael Pradel. Leaky Images: Targeted Privacy Attacks in the Web. In *USENIX Security Symposium*, pages 923–939. USENIX Association, 2019.

[2] Avinash Sudhodanan, Soheil Khodayari, and Juan Caballero. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In *NDSS*, 2020.

[3] Mojtaba Zaheri and Reza Curtmola. Leakuidator: Leaky resource attacks and countermeasures. In *Proc. of the 7th EAI SecureComm*, 2021.

[4] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. XSinator.com: From a Formal Model to the Automatic Evaluation of Cross-Site Leaks in Web Browsers. In *ACM CCS*, pages 1771–1788, 2021.

[5] Berk Gülmezoglu, Andreas Zankl, M. Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. Undermining User Privacy on Mobile Devices Using AI. In *AsiaCCS*, pages 214–227. ACM, 2019.

[6] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *USENIX Security Symposium*, 2019.

[7] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *USENIX Security Symposium*, 2021.

[8] Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, Keito Sasaoka, Takeshi Yagi, and Tatsuya Mori. User Blocking Considered Harmful? An Attacker-Controllable Side Channel to Identify Social Accounts. In *EuroS&P*. IEEE, 2018.

[9] Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, Keito Sasaoka, Takeshi Yagi, and Tatsuya Mori. Follow Your Silhouette: Identifying the Social Account of Website Visitors through User-Blocking Side Channel. *IEICE Trans. Inf. Syst.*, 103-D(2):239–255, 2020.

[10] Leaquidator+ Team. Leakuidator+ for Firefox. https://addons.mozilla.org/en-US/firefox/, 2021.

[11] Leaquidator+ Team. Leakuidator+ for Chrome. https://chrome.google.com/webstore/, 2021.

[12] Kenneth Kufluk and Gregory Baker. Protecting user identity against Silhouette. https://blog.twitter.com/engineering/en_us/topics/insights/2018/twitter_silhouette.

[13] MDN Web Docs. Cross-Origin Resource Policy (CORP). https://developer.mozilla.org/en-US/docs/Web/HTTP/Cross-Origin_Resource_Policy_(CORP).

[14] The Chromium Projects. SameSite Updates. https://www.chromium.org/updates/same-site.

[15] Onur Aciiçmez. Yet another MicroArchitectural Attack: exploiting I-Cache. In *CSAW*, pages 11–18. ACM, 2007.

[16] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.

[17] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE S&P*, pages 605–622, 2015.

[18] Yossi Oren. PP0 GitHub Repository. https:

//github.com/Yossioren/pp0, 2021.

[19] MDN Web Docs. Same-origin policy: Cross-origin script API access. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#cross-origin_script_api_access.

[20] Soheil Khodayari. De-anonymization attack: Cross site information leakage. https://hackerone.com/reports/723175, 2019.

[21] Avinash Sudhodanan. HotCRP: Attempt to plug an information leak represented by http status. https://github.com/kohler/hotcrp/commit/406a966aad00a762460fbc62cfb04a7532fc9fbd, 2019.

[22] MDN Web Docs. Cross-Origin-Opener-Policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy.

[23] Ron Masas. Patched Facebook Vulnerability Could Have Exposed Private Information About You and Your Friends. https://www.imperva.com/blog/facebook-privacy-bug/.

[24] XS-leaks Wiki: Window References. https://xsleaks.dev/docs/attacks/window-references/, October 2020.

[25] Avi Drissman. WebUSB dialog allows popunders. https://bugs.chromium.org/p/chromium/issues/detail?id=838314.

[26] Masato Kinugawa. Popunder restriction bypass with Presentation API. https://bugs.chromium.org/p/chromium/issues/detail?id=768900.

[27] Scikit-learn: Machine Learning in Python. https://scikit-learn.org/, 2021.

[28] TensorFlow: An end-to-end open source machine learning platform. https://www.tensorflow.org, 2021.

[29] Google Research. Colaboratory. https://colab.research.google.com/, 2021.

[30] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[31] Ubuntu Wiki. Stress-NG. https://wiki.ubuntu.com/Kernel/Reference/stress-ng, November 2021.

[32] Leak cross-window request timing by exhausting connection pool. https://bugs.chromium.org/p/chromium/issues/detail?id=843157, May 2018.

[33] Terjanq. Twitter: Detect X-Frame-Options header in Chrome. https://twitter.com/terjanq/status/1111600071014080517, March 2019.

[34] Egor Homakov. Disclose domain of redirect destination taking advantage of CSP. https://bugs.chromium.org/p/chromium/issues/detail?id=313737.

[35] Egor Homakov. Using Content-Security-Policy for Evil. http://homakov.blogspot.com/2014/01/using-content-security-policy-for-evil.html.

[36] Terjanq. Protected tweets exposure through the url. https://hackerone.com/reports/491473.

[37] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *23rd IEEE Computer Security Foundations Symposium*, pages 200–214. IEEE, 2010.

[38] Eduardo Vela. HTTP Cache Cross-Site Leaks. https://sirdarckcat.blogspot.com/2019/03/http-cache-cross-site-leaks.html.

[39] Jens Müller. CORS misconfiguration. https://web-in-security.blogspot.com/2017/07/cors-misconfigurations-on-large-scale.html.

[40] Terjanq. Mass XS-Search using Cache Attack. https://terjanq.github.io/Bug-Bounty/Google/cache-attack-06jd2d2mz2r0/index.html#VIII-YouTube-watching-history.

[41] Gareth Heyes. Leaking IDs using focus. https://portswigger.net/research/xs-leak-leaking-ids-using-focus.

[42] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proc. of WWW*, 2007.

[43] Chris Evan. Cross-domain search timing. https://scarybeastsecurity.blogspot.com/2009/12/cross-domain-search-timing.html.

[44] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The Clock is Still Ticking: Timing Attacks in the Modern Web. In *Proc. of the 22nd ACM CCS*, pages 1382–1393. ACM, 2015.

[45] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless timing attacks: Exploiting concurrency to leak secrets over remote connections. In *Proc. of the 29th USENIX Security Symposium*, pages 1985–2002, 2020.

[46] Pepe Vila and Boris Köpf. Loophole: Timing Attacks on Shared Event Loops in Chrome. In *USENIX Security*, 2017.

[47] Eduardo Vela. Matryoshka - Web Application Timing Attacks (or.. Timing Attacks against JavaScript Applications in Browsers). https://sirdarckcat.blogspot.com/2014/05/matryoshka-web-application-timing.html.

[48] Eduardo Vela. Security: XS-Search + XSS Auditor = Not Cool. https://bugs.chromium.org/p/chromium/issues/detail?id=922829.

[49] Juan Manuel Fernández. CSS Injection Primitives. https://x-c3ll.github.io/posts/CSS-Injection-Primitives/.

[50] XS-leaks Wiki: postMessage Broadcasts. https://xsleaks.dev/docs/attacks/postmessage-broadcasts/, October 2020.

[51] cure53.de. HTTPLeaks. https://github.com/cure53/HTTPLeaks/.

[52] Łukasz Anforowicz. CORB vs side channels. https://docs.google.com/document/d/1kdqstoT1uH5JafGmRXrtKE4yVfjUVmXitjcvJ4tbBvM/edit?ts=5f2c8004.

[53] Sigurd Kolltveit. A timing attack with CSS selectors and Javascript. https://blog.sheddow.xyz/css-timing-attack/.

[54] Takashi Yoneuchi. A Rough Idea of Blind Regular Expression Injection Attack. https://diary.shift-js.info/blind-regular-expression-injection/.

[55] Soroush Karami, Panagiotis Ilia, and Jason Polakis. Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage. In *Proc. of NDSS '21*, 2021.

[56] MDN Web Docs. Sec-Fetch-Site. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-Fetch-Site.

[57] MDN Web Docs. Vary. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Vary.

[58] XS-leaks Wiki: Subresource Protections. https://xsleaks.dev/docs/defenses/design-protections/subresource-protections/, October 2020.

[59] W3C Working Draft. Fetch Metadata Request Headers. https://www.w3.org/TR/fetch-metadata/.

[60] Anne van Kesteren. Cross-Origin-Opener-Policy response header (also known as COOP). https://gist.github.com/annevk/6f2dd8c79c77123f39797f6bdac43f3e.

[61] MDN Web Docs. X-Frame-Options. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options.

[62] MDN Web Docs. CSP: frame-ancestors. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors.

[63] XS-leaks Wiki: Isolation Policies. https://xsleaks.dev/docs/defenses/isolation-policies/, December 2020.

[64] The Chromium Projects. Cross-Origin Read Blocking for Web Developers. https://www.chromium.org/Home/chromium-security/corb-for-developers.

[65] Vicki Pfau. Optionally partition cache to prevent using cache for tracking. https://bugs.webkit.org/show_bug.cgi?id=110269.

[66] Josh Karlin. Split Disk Cache Meta Bug. https://bugs.chromium.org/p/chromium/issues/detail?id=910708.

[67] Anne van Kesteren. Top-level site partitioning. https://bugzilla.mozilla.org/show_bug.cgi?id=1590107.

[68] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan 2005*, 2005.

[69] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*, pages 1406–1418, 2015.

[70] Anatoly Shusterman, Zohar Avraham, Eliezer Croitoru, Yarden Haskal, Lachlan Kang, Dvir Levi, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Website Fingerprinting Through the Cache Occupancy Channel and its Real World Practicality. *IEEE Trans. Dependable Secur. Comput.*, 18(5):2042–2060, 2021.

[71] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy*, pages 143–157. IEEE Computer Society, 2012.

[72] Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. Cache-Based Application Detection in the Cloud Using Machine Learning. In *AsiaCCS*, pages 288–300. ACM, 2017.

[73] Shane S. Clark, Hossen A. Mustafa, Benjamin Ransford, Jacob Sorber, Kevin Fu, and Wenyuan Xu. Current Events: Identifying Webpages by Tapping the Electrical Outlet. In *ESORICS*, pages 700–717, 2013.

[74] Qing Yang, Paolo Gasti, Gang Zhou, Aydin Farajidavar, and Kiran S. Balagani. On Inferring Browsing Activity on Smartphones via USB Power Analysis Side-Channel. *IEEE Trans. Information Forensics and Security*, 12(5):1056–1066, 2017.

[75] Pavel Lifshits, Roni Forte, Yedid Hoshen, Matt Halpern, Manuel Philipose, Mohit Tiwari, and Mark Silberstein. Power to peep-all: Inference Attacks by Malicious Batteries on Mobile Devices. *PoPETs*, 2018(4):1–1, 2018.

[76] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *IEEE SP*, pages 19–33, 2014.

[77] Raphael Spreitzer, Simone Griesmayr, Thomas Korak, and Stefan Mangard. Exploiting Data-Usage Statistics for Website Fingerprinting Attacks on Android. In *WISEC*, pages 49–60, 2016.

[78] Berk Gülmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to Violate Web Privacy with Hardware Performance Events. In *ESORICS (2)*, pages 80–97, 2017.

[79] Jo M. Booth. Not So Incognito: Exploiting Resource-Based Side Channels in JavaScript Engines. Bachelor thesis, Harvard, April 2015.

[80] Hyungsub Kim, Sangho Lee, and Jong Kim. Inferring browser activity and status through remote monitoring of storage usage. In *ACSAC*, pages 410–421, 2016.

[81] Chromium bugs: Side-channel attack can deanonymize users (potential risk to journalists and activists). https://bugs.chromium.org/p/chromium/issues/detail?id=1285604, 2022.

[82] Bugzilla: Side-channel attack can deanonymize users (potential risk to journalists and activists). https://bugzilla.mozilla.org/show_bug.cgi?id=1749129, 2022.

[83] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Beauty and the Burst: Remote Identification of Encrypted Video Streams. In *USENIX Security Symposium*, pages 1357–1374, 2017.

[84] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *EuroSys*, 2019.

[85] David Li. The transition of Chrome extensions to Manifest V3. https://developer.chrome.com/blog/mv2-transition/, September 2021.

[86] Extending the fetch metadata headers: related tabs/windows #83. https://github.com/w3c/webappsec-fetch-metadata/issues/83, 2022.

[87] Stan Salvador and Philip Chan. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*, 11(5):561–580, 2007.

A Additional Experimental Setup Details

In this section, we provide additional details about our experimental setup. Table 4 provides details about the five system configurations we used in the experiments.

Machine Learning Classifier Parameters. The LSTM neural network model was used with the hyper-parameters described in Table 5. The logistic regression classifier was used with 1000 max iterations.

B Embedding Details For Various Services

In this section, we provide details about the method used to embed leaky resources for each sharing service. The embedding methods are based on specific SD-URLs we identified for these services.

In Chrome, we used the `<iframe>` embedding method for YouTube, LinkedIn and TikTok, and the *tab-under* method for Facebook, Instagram, Reddit and Twitter. In Safari, we used the *pop-under* method, whereas in Tor we used the *tab-under* method.

System	Device	OS	CPU	Browser	Measurement Method
Win-Chrome	Dell Latitude	Windows 10 Pro 20H2	Intel Core i7 7820HQ	Chrome 96.0	C, 8MB, 2ms
Win-Tor	Dell Latitude	Windows 10 Pro 20H2	Intel Core i7 7820HQ	Tor 11.0.1	S, 8MB, 100ms
Mac-Intel-Safari	MacBook Pro	macOS Catalina 10.15.7	Intel Core i7 3540M	Safari 15.0	C, 4MB, 2ms
Mac-M1-Chrome	Mac mini	macOS Big Sur 11.4	Apple M1 8-Core	Chrome 96.0	S, 4MB, 10ms
Android-Chrome	Samsung Galaxy S21 5G	Android 11, One UI 3.1	Qualcomm SM8350	Chrome Android 92.0	S, 4MB, 10ms

Table 4: System configurations used for the attacks. The "Measurement Method" column describes the setup used for cache measurements, in the format (Method, Buffer size, Interval). "Method" denotes the cache measurement method used: C for Cache Occupancy, S for Sweep Counting. "Interval" is related to the accuracy of the time measurement API, which is determined by the combination browser/device. For Cache Occupancy, "Interval" denotes the time between consecutive cache measurements. For Sweep Counting, "Interval" denotes the time needed to take one cache measurement.

Hyperparameter	Value
Optimizer	Adam
Learning rate	0.001
Batch size	128
Training Epoch	early stop by validation accuracy
Input units	vector size of the input
Convolution layers	1
Convolution activation	relu
Convolution kernels	256
Convolution kernel size	32
Pool size	4
LSTM activation	tanh
LSTM units	32
Dropout	0.7

Table 5: Hyper-parameters for neural network classifier.

For all the services tested, except for Reddit, the leaky resource was a video, because it causes cache activity over an extended period of time. In some cases, the video does not auto-play, but the video player loads a preview of the video that generates sufficient cache activity.

Additional details for the individual sharing websites is provided below.

YouTube. The SD-URL for the leaky resource points to a video player playing a video. The attack uses the private sharing-based approach. YouTube complies with cookies from both cross-site and same-site requests. As a result, we use the `<iframe>` embedding method for the Chrome browser: When the private resource is shared with the victim, the video is loaded in the embedded YouTube player; when the private resource is not shared with the victim, the video is not loaded in the embedded YouTube player. In the Safari and Tor browsers, cookies are disabled for cross-site requests, so an embedding method should be used that allows sending cookies as first party along with the requests. As a result, in the Safari browser we used the pop-under embedding method, whereas in the Tor browser we used the tab-under embedding method.

LinkedIn and TikTok. The SD-URL for the leaky resource points to a publicly shared post containing a video. The attack uses the blocking-based approach. These services comply with cookies from both cross-site and same-site requests. As a result, we use the `<iframe>` embedding method for the Chrome browser: If the account holder of the publicly shared post (the attacker) blocks the victim account, then the post does not load in the victim's Chrome browser; if the victim is not blocked, the post is loaded in the victim's Chrome browser. In the Safari and Tor browsers cookies are disabled for cross-site requests, hence we use the pop-under and tab-under embedding methods, respectively.

Twitter, Instagram and Facebook. The SD-URL for the leaky resource points to a publicly shared post containing a video. The attack uses the blocking-based approach. These services ignore cookies from cross-site requests. For example, consider a post embedded cross-site using an `<iframe>`: If the post is public, it is loaded in the browser regardless of user state; if the post is private, it is not loaded in the browser regardless of the user state. Therefore, an embedding approach is needed that attaches the cookies to the requests as first party cookies. In the Safari browser, we used the pop-under embedding method. In the Chrome and Tor browsers, we used the tab-under embedding method.

Reddit. The SD-URL for the leaky resource points to a private subreddit page. The attack uses the private sharing-based approach. The attacker creates a private subreddit and approves the victim to the private subreddit. We were not able to embed the subreddit page cross-site, so we used embedding methods that make first party requests: The pop-under method for the Safari browser and the tab-under method for the Chrome and Tor browsers. Since Reddit does not allow posting of videos in private subreddits, we modified the default layout of the private subreddit so that it loads multiple images when displayed.

C An Online-Only Attack

The attacks described in Section 4 implicitly assume that the attacker has some prior information about the victim's system configuration. This prior information lets the attacker

	Win-Chrome		Win-Tor		Mac-Intel-Safari		Mac-M1-Chrome	
	Accuracy (%)		*Accuracy (%)*		*Accuracy (%)*		*Accuracy (%)*	
Service	*w/ MSE*	*w/ FastDTW*	*w/ MSE*	*w/ FastDTW*	*w/ MSE*	*w/ FastDTW*	*w/ MSE*	*w/ FastDTW*
Google	97	98	65	80	100	98	76	87
Twitter	76	82	71	74	98	98	63	71
LinkedIn	100	100	52	54	56	68	82	74
TikTok	67	69	78	82	68	80	67	82
Facebook	100	100	65	66	65	65	93	96
Instagram	60	71	61	66	63	81	54	67
Reddit	67	69	53	63	60	86	61	64

Table 6: Attack accuracy for the Online-Only attack simulation. `MSE` is mean squared error and `FastDTW` [87] is an approximate dynamic time warping (DTW) algorithm that has a linear time and space complexity.

carry out an offline step, in which it trains a machine learning classifier on a system similar to the victim's. Although this assumption is reasonable under our threat model, it is still interesting to consider the case where the attacker does not have the ability to prepare for the attack.

We now describe a variant of our attack which can be carried out without a training step, at the cost of a longer online attack time. In this setting, the attacker prepares three shared resources, R_{victim}, R_{other} and R_{all}. R_{victim} is shared with the victim, R_{other} is shared with a single user who is not the victim (*i.e.*, another attacker account), and R_{all} is shared publicly with everyone.

The attack page loads the three shared resources one after the other while taking cache measurements. Next, the attacker uses a similarity metric, such as mean squared error (MSE) or dynamic time warping distance (DTW), to detect whether the trace collected for R_{victim} is more similar to the trace collected for R_{other}, or to the trace collected for R_{all}. That is, if $MSE(Trace(R_{victim}), Trace(R_{all})) < MSE(Trace(R_{victim}), Trace(R_{other}))$, then the attacker concludes that it is targeting the victim.

To experimentally validate this attack, we performed an experiment in Chrome for Windows targeting the Google/YouTube cookie, using three YouTube videos loaded into an `<iframe>` element. We collected 1 second side-channel measurements for each of three videos, resulting in a total attack time of 3 seconds. Then, we applied the MSE metric to identify the presence of the victim. We repeated the experiment 200 times, 100 for a victim user and 100 for a non-victim user. An implementation of this online-only attack can be found in the paper's artifact repository. Our results showed that all 100 predictions in the victim state were correct, and 98 out of 100 predictions in non-victim state were correct, resulting in an overall attack accuracy of 99%. We therefore conclude that our attacks are feasible in some settings even if the attacker cannot carry out a training step.

To see if this attack can be extended to other websites and browsers, we simulated the online-only attack using traces from our dataset. We did so by repeatedly selecting one pair of target and non-target traces as references, then measuring the distance between these reference traces and the subsequent pair of target and non-target traces from the same dataset. We discovered that while the simulated online-only attack was effective in many settings, including Google, LinkedIn and Facebook on Win-Chrome, Google and Twitter on Mac-Intel-Safari, and Facebook on Mac-M1-Chrome, it was far less effective than the classifier-based method in several settings, including TikTok, Instagram and Reddit on Win-Chrome, LinkedIn and Facebook on Mac-Intel-Safari, Twitter, Instagram and Reddit on Mac-M1-Chrome, and most of the services on Win-Tor. A table listing the full accuracy results for this simulated experiment can be found in Table 6. Note that an online-only attack beyond simulation is limited to the settings where it is possible to load multiple resources through the attack page.