# Targeted Deanonymization via the Cache Side Channel: Attacks and Defenses

Mojtaba Zaheri
*New Jersey Institute of Technology*

Yossi Oren
*New Jersey Institute of Technology*

Reza Curtmola
*New Jersey Institute of Technology*

## Abstract

Targeted deanonymization attacks let a malicious website discover whether a visitor to the website bears a certain public identifier, such as an email address, a Twitter handle, an Instagram account name, or a TikTok user ID. These attacks were previously considered to rely on several assumptions, limiting their practical impact. In this work we challenge these assumptions, and show that the attack surface for targeted 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 cookies. 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, as we show experimentally.

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 individuals who are unable to install our browser extension.

## 1 Introduction

On the Internet, everybody knows it's 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* [75, 77] are an important class of attacks which threaten user anonymity. These attacks assume an attacker who is interested in tracking a certain target user. This target is only known to the attacker through a public identifier, such as an email address or a Twitter handle. The attacks also assume that there is a website under the complete or partial control of an attacker, and that the target user occasionally visits this website. The attacker is interested in learning whether the specific target is browsing the website.

*Leaky resources* are one method which has been leveraged for this purpose [75, 102]: An attacker uses a resource-sharing service such as YouTube, Google Drive, or Dropbox to privately share a resource (*e.g.*, an image or a video) with the target. Next, the attacker embeds this shared resource into her own website. Finally, the attacker attempts to determine whether a visitor to her website can access this resource – successful access to the resource 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*) [77] 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 [47]. As discussed by Staicu *et al.* [75], 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 of leaky resource attacks is that they rely on the browser's support for third-party cookies. This is due to the fact that 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.

A final limitation is that leaky resource attacks only work if the attacker can freely share content with the target, and are particularly effective if the target is not notified when content is shared with them. Some websites, however, do not allow content to be directly shared between strangers, especially without notifying them.

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 *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.* [35], 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 resolutions. As shown by Shusterman *et al.* [68, 70], cache occupancy attacks are highly effective for privacy attacks, in particular in the setting of 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 such as Tor.

By combining the side-channel technique with the blocking technique of Watanabe *et al.* [94, 95], 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 [80, 81]. As part of our responsible disclosure process, we will be actively reaching out to organizations which advocate for users with high risk of being targeted by deanonymization attacks, and providing 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 can potentially overcome most of the limitations of existing targeted deanonymization attacks, while remaining within the same threat model (Section 2).
- We experimentally demonstrate our attacks on a diverse set of targets, including both desktop and mobile systems with multiple CPU microarchitectures, multiple browsers, and multiple highly popular websites, including Google/YouTube, Twitter, LinkedIn, Facebook, Instagram, Reddit and TikTok. In particular, we demonstrate a practical end-to-end attack on the highly secured Tor Browser, which can scale to deanonymize thousands of users without their knowledge (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 mitigation countermeasures, such as adding random cache noise, are not effective in preventing 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 6).

## 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 certain journalist who has a well-known Twitter handle. The state adversary has also compelled a local website to include malicious code that can potentially 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.

As a third example, consider an advertising provider interested in tracking users across websites in order to build a more comprehensive profile of their personal habits. While this is traditionally achieved through the use of third-party cookies, some browsers, most notably Apple's Safari, completely block third-party cookies, making this form of tracking impossible. Instead of cookies, the advertiser must rely on browser fingerprinting. To do this, the advertiser collects a large variety of properties from the victim's machine using JavaScript, and then uses this collected data as a substitute for a cookie. Unfortunately for the advertiser, it is known that these properties evolve over time [89] and, as a result, the advertiser loses track of the user after a couple of weeks. To get around this problem, the advertiser may take advantage of targeted deanonymization. Users occasionally feed in their email addresses to shopping-related websites, lured by the promise of 10% off their first purchase. Consider such a user, first feeding his email to one site which engages in browser fingerprinting instead of cookies, and then, after a few weeks, going to another website which also attempts to fingerprint him. Because of the fingerprint's evolution, the fingerprinting algorithm returns several candidates for the user based on the Javascript properties, but is not certain which one of these is the right one. The website performs targeted deanonymization, based on the email addresses of the candidates, and finally pinpoints the targeted user.

## 2 Background

### 2.1 Leaky Resource Attacks

The leaky resource attack [47,75,77,102] is a targeted privacy attack, through which an individual browsing an attacker-controlled webpage can be uniquely identified. This is in contrast with other known deanonymization techniques, such as third-party tracking [20] (e.g., tracking pixels or tracking IPs) or social media fingerprinting, which do not provide this level of accuracy.

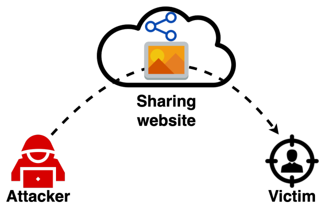
As originally proposed, the attack leveraged a media resource (e.g., an image, video, or audio file) hosted by a resource-sharing service that meets the following conditions: (1) The service relies on web cookies for user authentication, (2) the service allows a user to privately share hosted resources with other users of the service, and (3) the shared resource can be accessed directly via a URL. Such a URL is referred to as a *state-dependent URL* (SD-URL) [75], because the response to a user request for the URL is different depending on the user's state with respect to the sharing service: If the user is the target victim (*i.e.*, the user with which the resource was privately shared with), then the resource will be retrieved, because the request contains the authentication cookie; otherwise, the resource will not be retrieved. In our work, we show how deanonymization attacks can be applied to additional settings.

Many services meet these conditions, including: generic storage sites such as Google Drive [26], One Drive [57], Dropbox [19]; media sharing sites such as YouTube [101], Google Photos [32], Amazon Photos [3], Flickr [25]; code-hosting repositories such as GitHub [28], GitLab [29], Bitbucket [5], Assembla [4]; social media sites such as Facebook [22], Twitter [86], or Instagram [39]. We note that it is quite common for users to remain logged into such services for extended periods of time. For example, logging into GMail stores a Google login cookie which authenticates the user to services such as Google Drive and YouTube.

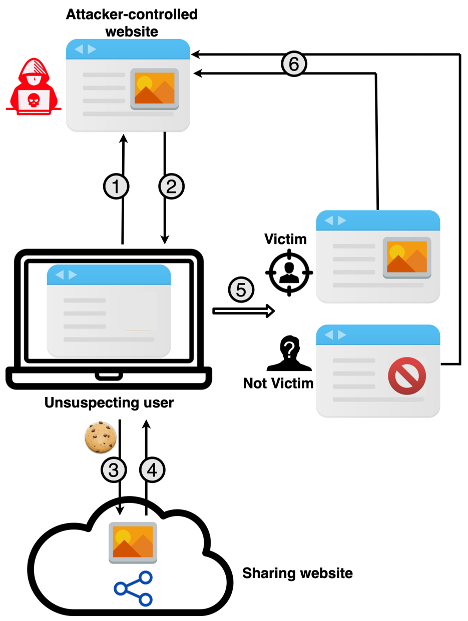
Many of these services do not need permission from target users to share a resource with them, and do not inform the target user when a resource is being shared with them. Some services, most notably Google Drive, even allow the sharer to explicitly choose not to notify the user with whom the resource is shared.

The attack proceeds as described in Fig. 1. In the attack setup phase (Fig. 1a), the attacker uploads a resource to the sharing service and then privately shares the resource with the target victim (e.g., by using the victim's email address or user ID with the service). To execute the attack (Fig. 1b), the attacker embeds the resource on an attacker-controlled webpage via an SD-URL. When a user visits the attack page (steps 1 and 2), the user's browser tries to render the page and so it makes a cross-site request to retrieve the embedded resource from the sharing service (steps 3 and 4).

If the user is logged into the service, the request for this resource (step 3) will be sent together with authentication cookies. If the user is the target victim, the response to this cross-site request contains the shared resource. Otherwise, it contains an error (step 5). The Same-Origin policy would normally prevent the attacker from reading the contents of the cross-origin response. However, the attacker can bypass this policy using a *cross-site leak (XS-leak)* [77] to learn information about the response (step 6): If the resource is



(a) Attack setup.



(b) Attack execution.

Figure 1: The leaky resource attack: Setup and execution.

successfully retrieved, then the attacker is certain that the target victim is visiting the attack page.

**Known XS-leaks.** Prior work [75] 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. 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. This fallback-based mechanism can be used to simulate an *if-then-else* control flow instruction in pure HTML. This functionality can be achieved using the `object`, `video`, or `audio` HTML tags [75, 102].

More subtle XS-leaks were uncovered through systematic analysis of websites and browser APIs [47,77]. These include cross-origin communication between Window objects, the browser's Performance API, and the number of WebSocket connections established by a webpage.

## 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 assumptions of the secure system [1]. Cache side-channel attacks are one type of micro-architectural attack. This attack exploits 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, or LLC, is shared between all cores of the processor. Caches are usually divided into smaller elements, called cache sets, with each cache set responsible for storing data from a certain part of physical memory. 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 higher-level security mechanisms such as sandboxing, virtual memory, privilege rings, hypervisors etc.,

There are several methods for performing cache attacks. This work uses the **Prime+Probe** technique, originally invented by Tromer *et al*. [60] and later adapted for use in the LLC by Liu *et al*. [53]. 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 cache set. The sets are chosen in a way that guarantees that the victim also uses them for its own purposes. In the second step, the attacker accesses the eviction set, bringing the cache set into a known state (prime step). In the third step, the attacker waits for the victim to use the cache. Since the attacker and the victim share sets, 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. This allows the attacker to detect whether the victim accessed a certain part of memory at a certain time, allowing it to learn about the victim's internal state.

Prime+Probe attacks usually require a time API with nanosecond-level accuracy. This resolution is typically not available through JavaScript in browsers. The **Cache Occupancy** attack is a variation on Prime+Probe introduced by Shusterman *et al*. [70]. In contrast to the Prime+Probe attack, where the attacker creates eviction sets which each cover a single cache set, in the cache occupancy attack the attacker allocates a single large buffer covering the entire LLC. Then, in the prime step, the attacker accesses this buffer, bringing the entire LLC into a known state. Any subsequent memory access by the victim will necessarily result in some of the attacker's memory being evicted from the cache, resulting in a longer runtime for the probe step, which can again be measured by the attacker. 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 attack is also platform-agnostic, running without modifications on different hardware architectures, since it does not make any assumptions on the way physical memory addresses are mapped into cache sets [68]. 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. 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 to be effective even when using the 10 Hertz 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, in several ways. First, we increase the attack's target population. We do this both by applying the attack to highly-popular services, 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. Second, we increase the attack's stealthiness. We do this by applying a technique, originally developed by Watanabe *et al.* [95], which is based on blocking the target user, as opposed to the traditional approach of privately sharing a resource with the target. This blocking method is particularly suited to our attack technique, since the countermeasures suggested by Watanabe *et al.* are hard to apply to an attacker observing the cache side channel. Third, we 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 cross-site leaks in order to determine whether the leaky resource is successfully loaded or not. This has the advantage of covering the novel scenarios introduced in this work, for which known cross-site 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.

In the remainder of this section, we first describe our general attack methodology, followed by the individual techniques that enable it.

### 3.1 General Attack Methodology

The attack has two phases. In the *training phase*, the attacker trains a machine learning classifier to detect the cache signature associated with successfully loading a leaky resource shared with the victim(s). The training phase can be potentially repeated under a variety of attack setups (*i.e.*, the combination of sharing service, browser, and device hardware), as it occurs without the involvement of the target victim(s) and the attacker is free to dedicate to this step whatever amount of time and resources it deems appropriate.

The second phase of the attack is the *online phase*. In this phase, the victim visits the attacker-controlled page. The attack webpage loads the leaky resource using one of the novel techniques described in the remainder of this Section [1]. As the leaky resource is being loaded and rendered, the attack page repeatedly measures the cache activity of the victim's computer. Finally, the attacker passes the collected cache measurements through the trained classifier, allowing it to identify the victim.

Because side-channel attacks take advantage of hardware-level properties of the victim's computer, they overcome architectural boundaries such as site isolation, process isolation and even VM isolation. In our particular case, as long as content from the attacker's website is rendered on the same computer as content from the sharing website, the former can spy on the latter using side-channel attacks, regardless of software-imposed boundaries.

The data analysis step views the data as a collection of samples. To collect one *sample*, the attack page needs to perform measurements for a certain amount of time, to which we refer as the *attack duration* and denote as $t_a$. This time is needed for the classifier to be able to differentiate between target and non-target users. The attack duration depends on the attack setup, *i.e.*. the combination of sharing service, the browser used by victim, and the cache measurement method. For the majority of attack setups, the attack duration is less than 3 seconds.

The techniques used by the attacker-controlled webpage to measure the cache activity while loading the leaky resource have in common the procedure described in Fig. 2. Specifically, `startCacheAttack()` allocates a buffer with the same size as the cache, primes the cache, and starts probing the cache. For the attack duration $t_a$, while the page is loading and rendering the leaky resource, `waitForPageToLoad(`$t_a$`)` probes the cache repeatedly and records the access time measurements. `uploadTraceData()` then uploads the collected trace to the attacker. We use the *prime* and *probe* operations from the PP0 repository [59].

[1] We note that the attacks can also work when the leaky resource is loaded using previously-known techniques which were described in Sec. 2.1.

```
startCacheAttack();
<load the leaky resource>
waitForPageToLoad(t_a);
finishCacheAttack(); uploadTraceData();
```

Figure 2: Common procedure for using the cache side channel in targeted deanonymization attacks.

## 3.2 Blocking-Based Attacks

**Challenge:** The first step in any conventional targeted deanonymization attack is to have the attacker share a resource with the victim. Several sharing websites, in particular Google Drive, allow resources to be shared with complete strangers, and furthermore allow the sharing party to prevent the sharing recipient from being notified about the newly-shared resource, making the attacks stealthy and effective. Not all websites, however, support this usability pattern. In some websites, content can only be shared between users who are already connected. In others, the sharing recipient is always notified when a resource is shared with them, causing the attacker to announce their intentions and reducing the stealthiness of the attack.

**Our Approach:** To make the attack more effective and increase its stealthiness, we leverage another cross-user mechanism which is quite the opposite of sharing – blocking. Many websites, including Twitter, LinkedIn, and Instagram, allow users to block other users whom they consider unworthy of viewing their content. Blocking is unidirectional – the blocking user does not have to be connected to the blocked user, and the blocked user's consent is not required. Furthermore, users are not notified when they are blocked, probably to avoid mental discomfort, and there is no way for a user to be notified when another user blocks them, or to display a list of all users who have blocked them.

In some instances, this blocking-based approach is made even easier because blocking a user in one product results in blocking the user across several other products. For example, when blocking a user in any of the following Google products, that user's account is blocked in all listed products: Drive, Chat, Photos, Maps, YouTube, Hangouts. Watanabe *et al.* [94, 95] already reported on this attack method in 2018, identifying that these properties make the blocking approach especially useful for targeted deanonymization. The main difference between our work and that of Watanabe *et al.* is the choice of side channel: Watanabe *et al.* used a high-level timing side channel to measure the time it takes to load the first landing page of the sharing website. In contrast, our work uses a cache side channel to observe the browser's real-time rendering activity. As a result, countermeasures which stymie the timing-based side channel, such as self-reloading landing pages, or "page shells", which load all their content using JavaScript [49], have no effect on our attack. Our attack also has a faster runtime than Watanabe *et al.* under practical network conditions. Most significantly, we can apply our attack even in cases where the attacker has no programmatic reference to the page loaded by the sharing website.

**Methodology:** In a blocking-based deanonymization attack, the attacker first creates a content item and shares it publicly. Next, the attacker selectively blocks the target user. Finally, the attacker causes the victim to load the public content item, and measures the cache side-channel generated by the victim's computer as this resource is loaded. If the side-channel trace indicates that the content was not loaded, this identifies the target.

We note that this blocking-based attack method works equally well with previously known XS-leaks (described in Sec. 2.1). For instance, we have successfully tested the attack with JavaScript events (using an `<img>` tag) for resources hosted on Google Drive.

## 3.3 Embedded Players

**Challenge:** 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, unless the embedded player's authors intentionally enabled this functionality.

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 [13]. 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 different states (the `CSPViolation` patch in LinkedIn [77]); 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 [44, 76, 77]); and for frame-counting leaks, the same number of frames can be returned (the `OP_FrameCount` patch in LinkedIn [77]).

**Our Approach:** To overcome the lack of XS-leaks when resources are loaded via embedded players (*e.g.*, using the `<iframe>` HTML tag), we measure the cache activity of the victim's computer while the resource-sharing website loads and renders the shared content.

**Methodology:** As described by the code snippet in Fig. 3, the attack web page uses JavaScript to insert an `<iframe>` HTML tag, load the leaky resource in the `<iframe>`, take cache measurements, and finally remove the `<iframe>` tag.

## 3.4 Pop-Unders and Tab-Unders

**Challenge:** We identified some scenarios that, up until the findings in this work, were considered safe from the at-

```
<script>
startCacheAttack();
let iframe = document.createElement("iframe");
iframe.src = "SD-URL";
document.getElementById("res").appendChild(iframe);
waitForPageToLoad(t_a);
iframe.remove();
finishCacheAttack(); uploadTraceData();
</script>
<div id="res"></div>
```

Figure 3: Embedding method: `iframe` HTML tag.

```
1  <script>
2  function go() {
3  startCacheAttack();
4  let popUnder = window.open("SD-URL", ...);
5  let ghost = window.open("about:blank");
6  ghost.focus(); ghost.close();
7  waitForPageToLoad(t_a);
8  popUnder.close();
9  finishCacheAttack(); uploadTraceData();}
10 </script>
```

Figure 4: Embedding method: pop-under.

tack's reach. 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 [47]. 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) [10].

**Our Approach:** To enable full-featured attacks even when embedding is not possible, we introduce attack variants that 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, our attack does not require programmatic access to the newly-created window and is executed stealthily.

To launch the attack, the attacker's page first lures the victim to click somewhere on the page. This can be achieved easily if the click gives some utility to the unsuspecting user. For example, a photo gallery page tempts the user to click the "Next" button for the next photo in the gallery. 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 pop-under or tab-under window, which loads in the background. As a result, the user still sees the original attack page, raising no suspicion of attack.

There are two variants to our method: In the pop-under variant, we load the sharing website inside a pop-up window, 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.

**Methodology:**
*Pop-under attack variant:* The code snippet in in Fig. 4 describes our methodology. The `go()` function, executed upon user click, initiates the cache activity measurement (line 3), and then opens a new window (line 4), referred to as the "pop-under window", which loads the leaky resource. Although the pop-under window gets focus for a brief amount of time, the attack proceeds to immediately return the focus to the attack page using a method described below. As a result, the user does not notice the pop-under window. The attack page, which is in focus again, takes cache measurements while the leaky resource is being loaded in the pop-under window (line 7). Once the measurements are collected, the pop-under window is closed (line 8), so the target user never notices such a window had been opened or existed.

Returning focus to the attack page after loading a pop-up window requires abusing the victim browser's window-ordering logic. Advertisers are actively looking for these pop-under tricks, and browser vendors are constantly patching them [18,46]. For the purpose of this paper, we have identified a pop-under technique which can currently exploit the Safari browser. In this method, immediately after opening the pop-under window, the attack page opens a second window (ghost), brings the focus to it, and then closes it (lines 5-6). The consequence of this action in Safari is that after closing the ghost window, the focus returns to the attack page (from where the ghost window was opened and closed). As a result, the pop-under window is not visible to the victim. These steps happen fast, so that the victim does not even notice two new windows are being opened, and does not suspect anything unusual happened in the attack page.

Using the pop-under variant, we executed the leaky resource attack successfully in Safari by loading a YouTube video, thus bypassing Safari's cookie-disabling policy. We also were able to leverage social media posts from Facebook

```
 1  <script>
 2  function go() {
 3  let ownURL = top.document.location.
        toString() + "?run=1";
 4  window.open(ownURL, ... );
 5  window.location.href = "SD-URL";
 6  }
 7  if(URLparams["run"] == 1) {
 8  startCacheAttack();
 9  waitForPageToLoad(t_a);
10  finishCacheAttack(); uploadTraceData();}
11  </script>
```

Figure 5: Embedding method: tab-under.

and Twitter to execute the attack in Safari, because the posts are opened in a new window and this request is considered same origin (thus bypassing the protections employed by these services against cross-site embedding of posts).

*Tab-under attack variant:* If a pop-under trick cannot currently be found for the victim's browser, the attacker cannot return the focus to the initial attack page after opening a pop-up window. In this situation, we introduce a different approach to keep the attack stealthy. As described in Fig. 5, the first instance of the attack page executes the `go()` function upon user click. This function opens in a new tab a second instance of the attack page with a `run=1` URL parameter (lines 3-4). The focus is now on this second instance, which looks identical to the first instance, so the target user barely notices that the tab in focus is new. The second instance of the attack page is instructed to take cache activity measurements (lines 7-10). Meanwhile, after opening the new tab, the first tab (first instance of the attack page) navigates to the SD-URL of the leaky resource (line 5). This navigation can be initiated in the background using JavaScript and, since the first tab is not in focus anymore, the victim does not notice a 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 Brave. We also executed the attack using resources hosted on sharing services that do not allow cross origin requests with cookies, such as Facebook and Twitter. Since the SD-URL is opened in a different tab, it is considered same origin, and the request is made with cookies.

## 3.5 Playlists

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

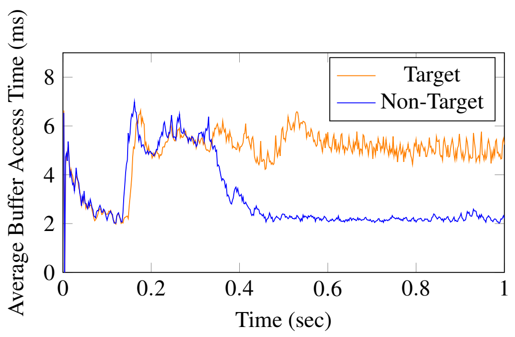
**Our Approach:** Despite this constraint, we were able to efficiently scale the attack in this setting to a group of target users, when the target users have accounts on a particularly important service – Google/Youtube. This feat is made possible by a unique property of YouTube, related to the way it processes playlists. In YouTube, a user can create a playlist containing multiple videos and share this playlist with viewers through a public URL – this will be the SD-URL to be used in the tab-under attack variant. 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, depending on the choice of the playlist's creator.

**Methodology:** The attacker shares YouTube videos with the target users according to a certain sharing pattern, and then creates a public YouTube playlist that includes the shared videos. The tab-under page points to the URL of this playlist. After collecting the cache traces resulting from playing back this playlist, the attacker can deanonymize an amount of users exponential in the playlist's length. In Sec. 4.5, we provide details about the sharing pattern.

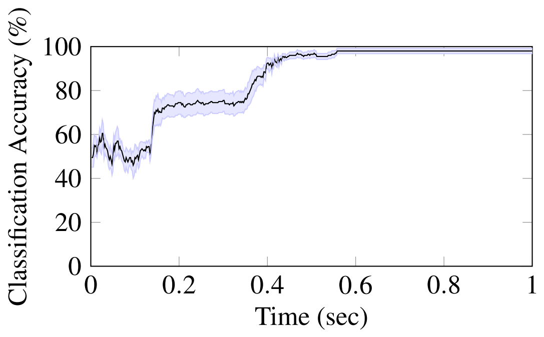
## 4 Attacks

### 4.1 A Proof-of-Concept Attack

Figure 6 illustrates the concept of our attack. In the experiment illustrated in the figure, the attacker's webpage 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. Subfigure 6 (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.* [95], but the non-target content returned by the server did not include video content. In the target state, on the other hand, the content took slightly longer to serve and included video content, which generated constant pressure on the cache. This difference in cache occupancy can be quickly



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



(b) Classifier Accuracy vs. Time

Figure 6: A Proof-of-Concept Attack

captured through a machine learning classifier, as the next Subfigure illustrates.

Subfigure 6 (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 $\{0 \ldots t\}$, 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 can see that the accuracy of the classifier starts close to a random guess, rises significantly starting at the 200ms mark, when the difference between the traces becomes apparent, 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.

## 4.2 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 victim user loads the attack page (labeled as "target state" samples), and while a non-victim user loads the attack page (labeled as "non-target state" samples).

For single target attacks, we use a logistic regression classifier, whereas for multi target attacks we use a long-short-term memory (LSTM) neural network model [2]. The parameters used for these classifiers are provided in Appendix A.

For each attack setting, a dataset of samples is used to train the 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 state samples and 100 non-target state samples) is sufficient to yield high attack accuracy. To fit the classifier onto the data set, we use 10-fold cross validation, described as follows. In a $k$-fold cross validation, the data set is partitioned into $k$ equally-sized subsets. Of the $k$ subsets, a single subset is retained for testing the model, and the remaining $k-1$ subsets are used as training data. The cross-validation procedure is repeated $k$ times, with each of the $k$ subsets used exactly once as the validation data. The $k$ results are then averaged to report a single estimation for the attack accuracy and for the standard deviation.

We performed the data analysis using Scikit-Learn v1.0.1 [71] and TensorFlow v2.7.0 [85] with Python v2.7.12 in Google Colaboratory [65].

## 4.3 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. The Tor browser connects to the Tor network in order to anonymize web user activity. We conducted experiments using five system configurations, described in Table 1.

**Services and embedding methods.** We selected the following popular sharing websites to demonstrate the impact of our attacks: Google (which includes all Google properties including Google Drive, Google Photos, GMail, YouTube and so on), Twitter, Facebook, Instagram, LinkedIn, Reddit, and TikTok. Together, their user base covers a vast majority of Internet users.

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. The choice of resource embedding method depends on whether

[2]We selected these classifiers as they perform slightly better all around, after experimenting with the following additional classifiers: gaussian naive bayes, multinomial naive bayes, K-nearest neighbors, support-vector machines, and random forest.

| 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 1: 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.

the browser allows cookies with cross-site requests, and on whether the service allows cross-site embedding of its authenticated resources. 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. More 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. By default, these resources are private to their accounts. For the private sharing-based approach, the resource in the first attacker account (Resource A) is 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 are publicly shared. The first attacker account blocks the victim, and the second attacker 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 victim user, whereas loading Page B simulates a non-victim user. To make sure the classifier is trained on the difference between states, and not on the global state of the system, we load these two pages in an interleaved fashion, *i.e.* repeatedly load Page A followed by Page B.

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. In Reddit, which 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.

**Other details.** To automate the experiments, we used the following frameworks: 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.

## 4.4 Experimental Results

Table 2 shows the attack accuracy for the various system configurations we considered. For easy side-by-side comparison, 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, which indicates that our cache-based targeted deanonymization attacks are effective across a variety of sharing services, web browsers, and microarchitectures.

We noticed a correlation between the browser’s time measurement API precision and the attack duration. Due to Chrome’s sub-millisecond time API precision, the attack reaches high accuracy in under 3 seconds for the Win-Chrome system. Similarly, Safari’s millisecond time API precision leads to high attack accuracy in under 5 seconds for the Mac-Intel-Safari system. Since the Tor browser has lower time API precision, we used the Sweep Counting cache measurement method for the Win-Tor system. The attack duration is larger in this system for two reasons: 1) The Sweep Counting method sacrifices spacial and temporal details and requires longer time to collect a sufficient number of samples, and 2) the longer latency of the Tor network causes delays when loading the leaky resources. For Tor, we observed high attack accuracy within the first 5 seconds for five services, ranging from 84.5% in LinkedIn to 96.5% in Facebook. The classifier yielded lower attack accuracy for Instagram and Reddit, even when using up to 10 seconds for attack duration.

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 the differences in cache patterns, due to the high speed of the M1 CPU. To achieve a balance between the time API precision and the cache measurement method, we used the Sweep Counting method with a 10 millisecond measurement interval. This yielded high attack accuracy, ranging from 81% in Reddit to 100% in YouTube.

We observed additional factors that influence the attack duration $t_a$. One important factor is the way website behavior

| System | Win-Chrome | | | Win-Tor | | | Mac-Intel-Safari | | Mac-M1-Chrome | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $t_a$ | Accuracy (%) | | $t_a$ | Accuracy (%) | | $t_a$ | Accuracy (%) | $t_a$ | Accuracy (%) | |
| Service | (s) | w/o def. | w/ def. | (s) | w/o def. | w/ def. | (s) | w/o def. | (s) | w/o def. | w/ def. |
| 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 2: Summary of experimental results. For each attack configuration, the attack accuracy is shown both without the `Leakuidator+` defense and with the defense. $t_a$ represents the attack duration in seconds. For each data point, the average and the standard deviation are provided, obtained using the 10-fold cross-validation methodology described in Sec. 4.2.

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. Browsers usually allow videos to be played automatically if the video is muted. If websites also provide such feature, it can be used by an attacker to amplify differences of cache patterns and lead to high attack accuracy in a shorter amount of time. Another factor is that some types of leaky resources, such as Instagram and TikTok posts and private subreddits, expose smaller but continuous differences in user states. Hence, the classifier needs a longer attack duration to gain confidence in attack accuracy. For example, in Reddit with Tor, the attack accuracy improves up until the eighth second due to these continued differences.

## 4.5 Scaling to Multiple Users

In some situations, an attacker may want to target a group of users instead of a single user. In this setting the goal of the attacker is to identify which specific user among a list of target $n$ users is visiting a particular website. Staicu *et al.* [75] 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.

**A basic sharing pattern:** Staicu *et al.* [75] proposed to scale the attack by privately sharing each of the $log(n)$ resources with a subset of the $n$ users as follows: Given resources $R_1, \ldots, R_{log(n)}$, for each user $i$ (with $1 \le i \le n$), consider the binary representation of $i$ as $b_1 b_2 \ldots b_{log(n)}$. The attacker shares privately with user $i$ the resources $R_j$ for all $j$ such that bit $b_j$ is 1. When the attacker-controlled page attempts to load the $log(n)$ resources, the specific resources that end up being loaded will reveal the identity of the user. This 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. User $i$ is blocked by the attacker from accessing resource $R_j$ for all $j$ such that bit $b_j$ is 1.

**The attack:** In the *training* phase, the attacker takes cache measurements and builds a cache profile for each of $n$ states, where a state corresponds to one of the $n$ target users. These measurements are used to train a machine learning classifier, which is then used to identify a specific target during the *attack* phase.

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 cross-site requests with cookies (such as Chrome and Firefox), and if the sharing service allows cross-site embedding, we programmatically create and delete multiple `<iframe>` elements, each containing a different shared resource. Using the basic sharing pattern described earlier, this approach works for Google/YouTube, LinkedIn, and TikTok.

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 follow an approach based on the pop-under and tab-under methods described in Sec. 3.4.

For the Safari browser, we use a pop-under technique which allows the attacker to retain a handle to the new window. This programmatic access allows us to replace the contents of the pop-under window multiple times. Different shared resources are loaded sequentially in the pop-under window by setting `window.location` to the SD-URL of the shared resource and then taking cache measurements in the attacker page. Using the basic sharing pattern described earlier, this approach works for all the seven sharing services we considered.

For the Tor browser, we use the tab-under technique, which does not provide the attacker with programmatic access to the sharing website. Inside this background tab we load the URL of a YouTube playlist containing multiple shared videos.

**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. We evaluated different approaches toward creating this playlist, and eventually converged on a method based on the duration of videos. Our experiments show that the duration of each video, and also 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. For simplicity, we refer to the duration of these videos as short ($s$) and long ($\ell$). 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 $1 \le i \le n$), 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 $1 \le j \le log(n)$, if $b_j$ is 0, then the attacker shares privately with user $i$ the video $V_j^s$, otherwise the attacker shares the video $V_j^\ell$. 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. The first experiment was performed on LinkedIn under Chrome for Windows, representing the setting where third-party cookies are supported by the browser. The second experiment was performed on Twitter under Safari for Mac, representing a setting where third-party cookies are not supported by the browser, but there exists a pop-under method which allows post-popup navigation. The third experiment was performed on Google/Youtube under Tor for Windows, representing 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. Table 3 shows that the attack allows the victims to be told apart with high accuracies in all three settings.

| *Service* | *System* | $t_a$ *(s)* | *Accuracy (%) w/o def.* | *Accuracy (%) w/ def.* |
|---|---|---|---|---|
| LinkedIn | Win-Chrome | 6 | $99.12 \pm 0.8$ | $15.63 \pm 3.55$ |
| Twitter | Mac-Intel-Safari | 6 | $97 \pm 1.9$ | N/A |
| YouTube | Win-Tor | 45 | $78.88 \pm 3.33$ | $11.75 \pm 3.07$ |

Table 3: Summary of multi target attack results for 8 user states. $t_a$ is the attack duration. For each attack configuration, the attack accuracy is shown both without the `Leakuidator+` defense and with the defense. For the Win-Chrome and Mac-Intel-Safari systems, we took 2s measurements for each of the three resources. For the Win-Tor system, we used videos of 5s and 10s and measured for an additional 15s to account for delays in the Tor network.

## 4.6 Attacking Mobile Phones

The attacks described so far in the paper 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 below.

We evaluated this attack on the Android-Chrome device, an ARM-based Samsung Galaxy S21 device described in more detail in Table 1. 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 4.2. Our attack on this mobile device yielded an accuracy of 91%, indicating that our deanonymization attack is effective in the mobile setting as well.

One concerning aspect of our mobile phone attack is the issue of mobile browser extensions. While 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. As a result, it is not possible to install our countermeasure on this target, as we discuss in more detail below.

## 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 request blocking (CORB) or cross-origin opener policies, nor by server-side isolation mechanisms such as self-reloading landing pages [49]. Instead, we attempt to prevent the attack using techniques from the field of side-channel defenses.

As stated by Mangard *et al*. [54], there are two general defense approaches against side-channel attacks. The first approach is mitigation, or hiding, which attempts to make attacks impractical by reducing the signal-to-noise ratio of the side-channel trace. The second approach is prevention, or masking, which attempts 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, and show that this defense is not effective against a well-prepared attacker. Next, we present 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 [97]: binary search (*bsearch*), heap-sort (*heap*), wide-spread memory reads and writes (*cache*), and CPU-intensive operations (*cpu*). Exact command line parameters are provided in Appendix D.

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 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 4 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. In the next subsection, we present a more systematic approach.

| | *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 4: 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*).

## 5.2 `Leakuidator+`

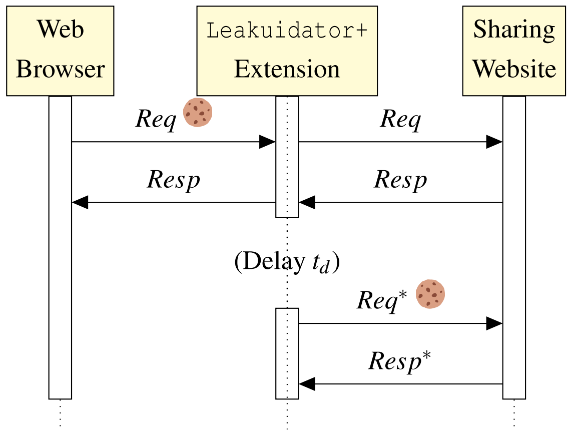


Figure 7: Interaction Diagram for `Leakuidator+`

We now describe the design and implementation of our main defense we propose 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 [80, 81].

`Leakuidator+` is based on `Leakuidator` [102], a previously-proposed client-side defense against targeted anonymization attacks. The source code for this extension was provided to us by the authors. This defense was originally designed to protect users from XS-leak-based deanonymization, even if the user is using otherwise vulnerable web browsers and websites. We first describe the original Leakuidator, 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 7 shows the exchange of messages between the browser, the extension, and ultimately the 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 first 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. This 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 provides a visual indication to the user through a notification on 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 *web-Navigation* API to detect when a new tab or window is opened, and recording the relations between parent and child and their top-level URLs. If a webpage from one domain opens a pop-up window from a different domain, Leakuidator+ detects this condition and applies the defense, even though all requests made by the pop-up window are considered by the browser to be first-party requests. Leakuidator+ uses *transition type* and *transition qualifiers* to exclude 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.** Since 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 targeting a user 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 why the attack remains effective even when Leakuidator is installed. We modified Leakuidator source code and attack pages to record time of events, impose delays on various events, generate artificial cache occupancy in different steps, and modify operations related to each step in Leakuidator algorithm. We also inspected the requests and responses using the Burp Suite and the browser's developer tools while experimenting with various sharing websites and embedding methods. This allowed us to correlate the measured side-channel differences with the sequence of events performed collectively by the sharing service, by the extension, and by the web browser's rendering process.

Our analysis 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, depending on whether 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 [94,95]. 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 a side-channel difference which the attacker could detect and exploit.

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*. We observed that the extension code itself amplifies small differences in cache occupancy that may exist 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 uniformly chose a value between 0 and 1 second for $t_d$. 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 *Req* headers used for comparison with *Req*$^*$, instead of recording all *Req* headers.

**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. As we describe in Sec. 6.3.2, the Apple Safari browser, as well as the Android version of Chrome, do not currently support our extension.

**Evaluation.** We performed a comprehensive set of experiments to validate `Leakuidator+`'s effectiveness. As summarized in Table 2, 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. As shown in Table 3, 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.

## 5.3 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.* [47] introduced a formal model for XS-leaks, building on work of Sudhodanan *et al.* [77]. 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 [96]. `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 [84]. 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 recording relations between tabs and windows, and by applying the protection to tabs and windows that are opened cross-site.

## 6 Discussion

### 6.1 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 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 the Appendix C.

### 6.2 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. Examples of APIs provided by browsers that can be used to infer such information are window references [98], frame counting [55], error events [75, 77], navigation [37, 38, 83], response cache probing [23, 56, 82, 90], ID attribute [36], postMessage broadcasts [73], CORB and CORP leaks [2], and timing attacks [7, 9, 21, 24, 30, 31, 42, 48, 91–93, 100]. Recently, there has been academic effort to give a structure to XS-leaks by classification [47, 77]. Targeted deanonymization is an example of privacy leakage through XS-leaks [75, 102].

In response to XS-leaks, a number of defense mechanisms were proposed, including response cache protections [14, 15], subresource protections [74], fetch metadata [17], cross-origin opener policy [10,87], cross-origin resource policy [11], framing protections [12, 16], SameSite cookies [64], isolation policies [40], cross-origin read blocking [63], and the partitioned HTTP cache [43, 62, 88]. xsleaks.dev is a community driven website encouraging developers and researchers to contribute to the collective knowledge for ongoing concern about cross-site leaks.

Recently, Knittel *et al.* [47] introduced a formal model for XS-leaks, building on work of Sudhodanan *et al.* [77]. This allowed them to systematically search for new XS-leak attack classes. On the defense side, they suggest an interesting approach: If a browser is immune to a leak technique, the corresponding leak technique can be fixed in other browsers as well by changing the implementation. As opposed to this approach, we propose a client-side defense that can be deployed right away, without depending on browser vendors and website owners.

As a general observation, many of the defenses proposed to mitigate XS-leaks were not designed to protect against side channels. Thus, as our work shows, attacks based on side channels may bypass software-imposed boundaries.

Cache attacks were proposed simultaneously by Percival and by Osvik *et al.* [60, 61], and first demonstrated on the last-level cache by Liu *et al.* [53]. Oren *et al.* presented the first JavaScript implementation of the last-level cache attack [58], and Shusterman *et al.* presented the cache occupancy and sweep counting variants of the attack, which can be run in more restricted browser environments such as Tor [69, 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 [41]. Gülmezoglu *et al.* showed how attacks using the cache can learn about running applications in a cloud scenario [33]. Gülmezoglu *et al.* also showed how a native Android app can use the cache side channel to undermine user privacy, discovering running applications, website activity and even detecting which videos the victim was streaming [35]. Multiple authors have explored the use of various side-channel attacks to infer browsing activity, including power consumption, GPU memory leaks, data-usage statistics, performance counters, and event loops [6, 8, 34, 45, 50, 52, 72, 93, 99]. In general, most of these works assume that the attacker passively observes the victim. This contrasts with the deanonymization scenario, in which the attacker actively induces the victim to load a certain website, and then learns about the victim's secrets from the way this website is rendered. While it is interesting to consider how these additional side channels could be applied to the task of targeted deanonymization, we believe our defense should be effective regardless of the side-channel method chosen by the attacker.

## 6.3 Guidance

The deanonymization attacks described in this paper are both practical and dangerous. We provide a browser extension which can prevent these attacks, and recommend that security-conscious users install this extension. We also provide advice in this subsection on other ways to limit the attack's effectiveness.

### 6.3.1 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 difference and a client-side difference. The server-side difference, already discussed by Watanabe *et al.* [94], is due to the server's different response times for targeted and non-targeted users. The client-side difference is due to the different way the browser processes the response for targeted and non-targeted users – for the targeted user, a resource-heavy media player is typically loaded, while for the non-targeted user, a relatively static error is displayed, resulting in an observable difference in the cache occupancy. Both of these differences can be mitigated through careful design by website owners, reducing the risk of the attack we present. 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.

**Mitigate server-side timing side channels:** A typical web server design includes an authorization module, which checks whether a user is authorized to access a certain response, followed by a content delivery module, which actually fetches the resource and makes it available to the client. When the targeted user attempts to load the resource, both the authorization and the content delivery modules need to run. For a non-targeted user, on the other hand, content delivery is not invoked at all, resulting in a faster response time which can be observed by the client. Although we measured this faster response time using a cache side channel, any other method of monitoring network traffic can also detect this difference, for example the congestion-based method used by Schuster *et al.* [67]. To mitigate this timing side channel, web servers can be designed to return their responses in constant time, regardless of the authorization status of the user. Since this step only affects non-targeted users, this mitigation should not have any effect on the usability of the website for users who are authorized to view the media.

**Mitigate client-side rendering side channels:** Web servers should attempt to make the structure of their error pages as similar as possible to that of their content pages. This will make it more difficult for a side-channel attacker to distinguish between the two. As an example, if the targeted user was supposed to be shown a video, the error page for the non-targeted user should be made to show a video as well. In general, website owners should strive to minimize any kind of attacker-observable difference in responses they send between the two states, whether it is the response time, size, duration, type, dimensions, counts, etc.

**Require user interaction before rendering cache-intensive 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 targeted and non-targeted users. To prevent this, website owners can make sure that videos and playlists which are 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. We observed this behavior in Apple's iCloud Drive service.

**Notify users upon sharing or blocking:** Google Drive and other sharing websites allow content to be shared without notifying the user, as a convenience feature. Twitter goes even further and does not provide any way for a user to know by whom he or she is blocked, presumably to avoid the mental discomfort associated with this knowledge. 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 the cognitive and emotional discomfort related to these notifications, website owners can consider how to selectively and intelligently suppress some of these notifications without sacrificing security.

**Replace blocking with "shadow-banning":** Blocking public content from a particular logged-in 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, originally invented by SlashDot and currently used by Reddit, applies a different approach to blocking than this traditional approach. 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 all other users. In this approach, the public posts of users are always accessible to other users, including those who are shadow-banned. 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 available on the sharing website.

### 6.3.2 Guidance to Browser Vendors

The browser serves as host both for the attacker and for the victim. The ideal countermeasure would therefore be a browser which can somehow isolate the cache activity of the victim from the spying eyes of the attacker, or which can somehow prevent the attacker's code from performing cache occupancy measurements. This is a tall task which may be impossible to carry out without redesigning the browser, the operating system or even the CPU [27]. Even before this protected browser becomes available, several engineering fixes to current browsers can raise the bar for the attackers.

**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 [18, 46]. 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.

**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 will be phased out starting January 2022, and ultimately removed during January 2023 [51]. The Edge browser, which is based on Chromium, follows the same schedule. 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.

### 6.3.3 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 shown in this work, as well as similar works on web-based side-channel attacks, show that this assumption might need to be reconsidered. In particular, the cross-origin request blocking (CORB) feature was already shown to be less effective in the presence of side channels [2]. 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.

### 6.3.4 Guidance to Users

There are already advocacy groups who publish guidance for users who are at increased risk of being targeted online, such as journalists, activists, religious leaders and so on. Such users are 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. We will also be cooperating with advocacy groups to bring this guidance to the knowledge of relevant users as part of the disclosure process, as described further in the following subsection.

**Install `Leakuidator+`:** The best suggestion we can provide is to install our browser extension, which is already available on both the Google Play Store and the Firefox Add-ons website [80, 81]. 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. Whereas mobile users can use

Firefox for Android or one of the several third-party Android browsers based on the open-source Chromium which do support extensions (*e.g.*, Kiwi Browser or Yandex Browser), testing the compatibility of our extension with these browsers remains a task for future work.

**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 targeted 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, delete cookies, or make use of Incognito or Private Browsing modes whenever possible. 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 window, the Google cookie will not disappear, but will rather remain present on the browser until the user manually logs out, deletes cookies or quits the Tor Browser.

**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 imprudent as it may sound, having a separate Twitter-phone, Facebook-phone, GMail-phone and browsing-phone can be the best way for a user to stay safe online. 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. This strategy can help reduce the attack surface of any kind of XS-leak, including the ones introduced in this paper based on cache side channels.

## 6.4 Ethics and Disclosure

The attacks presented in this paper can impact the privacy of journalists, activists, and other vulnerable populations. While we also 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 the content sharing sites make non-trivial changes to the way they allow content to be shared.

We plan to carry out a series of steps to minimize the risk to these users. First, we plan to proactively flag the paper as a potential privacy risk, and have already asked the conference's research ethics committee for guidance during the review process. We are sharing a draft of this paper with Google, Twitter, Meta, Microsoft, the Tor Foundation, TikTok, Reddit, Apple, and the W3C. We also consider journalists and activists part of the disclosure process, and plan to reach out to the organizations who advocate for them through the Surveillance Self-Defense team at 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.

## 6.5 Artifact Availability

We provide a dataset of cache traces collected during experiments for single and multi target attacks and a Google Colab document demonstrating how to use classifiers on the dataset. These artifacts are available together with sample attack pages for the `<iframe>`, *pop-under* and *tab-under* embedding methods [78]. In addition, we provide an Online-Only attack page (as described in Sec. 6.1) [79]. The complete source code for `Leakuidator+` is available through the Firefox and Chrome extension stores [80, 81].

## 7 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.

## References

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

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

[3] Amazon Photos. https://www.amazon.com/photos.

[4] Assembla. https://www.assembla.com/.

[5] Bitbucket. https://bitbucket.org.

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

[7] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proc. of the 16th international conference on World Wide Web*, pages 621–628, 2007.

[8] 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.

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

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

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

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

[13] 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.

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

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

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

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

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

[19] Dropbox. https://www.dropbox.com/.

[20] Steven Englehardt and Arvind Narayanan. Online Tracking: A 1-Million-Site Measurement and Analysis. In *Proc. of ACM CCS '16*, CCS '16, pages 1388–1401. ACM, 2016.

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

[22] Facebook. https://www.facebook.com.

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

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

[25] Flickr. https://www.flickr.com.

[26] Google Drive. https://drive.google.com/.

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

[28] GitHub. https://github.com.

[29] GitLab. https://gitlab.com.

[30] 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 SIGSAC Conference on Computer and Communications Security*, pages 1382–1393. ACM, 2015.

[31] 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.

[32] Google Photos. https://photos.google.com/.

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

[34] 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.

[35] 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.

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

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

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

[39] Instagram. https://www.instagram.com.

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

[41] 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.

[42] 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.

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

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

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

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

[47] 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 *2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1771–1788, 2021.

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

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

[50] 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.

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

[52] 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.

[53] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy*, pages 605–622. IEEE Computer Society, 2015.

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

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

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

[57] OneDrive. https://onedrive.live.com/.

[58] 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.

[59] Yossi Oren. PP0 GitHub Repository. https://github.com/Yossioren/pp0, 2021.

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

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

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

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

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

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

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

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

[68] 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*, pages 2863–2880. USENIX Association, 2021.

[69] 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.

[70] 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*, pages 639–656. USENIX Association, 2019.

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

[72] 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.

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

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

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

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

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

[78] Leakuidator+ Team. Artifact Repository for "Targeted Deanonymization via the Cache Side Channel: Attacks and Defenses". https://github.com/leakuidatorplusteam/artifacts, 2022.

[79] Leakuidator+ Team. Targeted Deanonymization via the Cache Side Channel: Online-Only Attack. https://leakuidatorplusteam.github.io/artifacts/wp_mse.html, 2022.

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

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

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

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

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

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

[86] Twitter. https://twitter.com.

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

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

[89] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. FP-STALKER: Tracking Browser Fingerprint Evolutions. In *IEEE Symposium on Security and Privacy*, pages 728–741. IEEE Computer Society, 2018.

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

[91] 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.

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

[93] Pepe Vila and Boris Köpf. Loophole: Timing Attacks on Shared Event Loops in Chrome. In *USENIX Security*, pages 849–864, 2017.

[94] 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*, pages 323–337. IEEE, 2018.

[95] 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.

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

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

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

[99] 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.

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

[101] YouTube. https://www.youtube.com.

[102] Mojtaba Zaheri and Reza Curtmola. Leakuidator: Leaky resource attacks and countermeasures. In *Proc. of the 7th EAI International Conference on Security and Privacy in Communication Networks (SecureComm '21)*, 2021.

## A Machine Learning Classifier Parameters

This section provides details about the parameters used by the machine learning classifiers. 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.

**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

| 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

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 where the UI is modified to load multiple images for the target state in order to affect cache contention. 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.

## C Summary Of Simulated Online-only Attacks

We provide details about the simulated online-only attacks 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.

## D Parameters For CPU Stress Tests

We provide the exact command line parameters used for the CPU stress tests described in Sec. 5.1:
*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

| | *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` [66] is an approximate dynamic time warping (DTW) algorithm that has a linear time and space complexity.