# SoK: Exploring Current and Future Research Directions on XS-Leaks through an Extended Formal Model

Tom Van Goethem[*], Gertjan Franken[*]
Iskander Sanchez-Rola[†], David Dworken[§], Wouter Joosen[*]
[*]imec-DistriNet, KU Leuven
[†]Norton Research Group
[§]Google

## ABSTRACT

A web visit typically consists of the browser rendering a dynamically generated response that is specifically tailored to the user. This generation of responses based on the currently authenticated user, whose authentication credentials are automatically included via cookies in all (including cross-site) requests, have led to a multitude of issues. Through cross-site leaks (XS-Leaks), an adversary can try to circumvent the same-origin policy and extract information about responses, which in turn can reveal potentially sensitive information about the user. As research on this class of vulnerabilities only recently gained traction, and the attacks affect many different components of the web platform, the intrinsic characteristics and underlying causes remain largely unexplored.

In this paper we present an abstraction of XS-Leaks attacks and introduce an extended formal model that we use to reason about the cause of different leaks and which strategies the various defense mechanisms employ to defend against them. Furthermore, we provide a classification method for current attacks, and, guided by our model, propose a methodology to comprehensively detect new XS-Leak issues, or indicate their absence. Furthermore, we analyze the current defenses and identify gaps that still require further research to provide extensive solutions for sites that rely on cross-site interactions. Finally, we explore how XS-Leak defenses are currently deployed and which challenges website owners are still facing. As a first step towards facilitating the deployment of XS-Leak defenses, we introduce Leakbuster, a dynamic web interface that provides web developers with suggestions based on the insights provided throughout this paper.

## CCS CONCEPTS

• **Security and privacy → Browser security**; *Network security*; *Formal methods and theory of security*.

## KEYWORDS

xs-leaks; taxonomy; formal model; same-origin policy

**ACM Reference Format:**
Tom Van Goethem[*], Gertjan Franken[*], Iskander Sanchez-Rola[†], David Dworken[§], Wouter Joosen[*], [*]imec-DistriNet, KU Leuven, [†]Norton Research Group, and [§]Google. 2022. SoK: Exploring Current and Future Research Di-rections on XS-Leaks through an Extended Formal Model. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30–June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3488932.3517416

*ASIA CCS '22, May 30–June 3, 2022, Nagasaki, Japan*
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9140-5/22/05.
https://doi.org/10.1145/3488932.3517416

## 1 INTRODUCTION

For many, the web plays an important part of their daily life, ranging from sharing personal information with friends on social networks, or looking up health-related details. It is well known that people share a lot of sensitive information with trusted websites, and that if this data would be disclosed by adversarial parties, this could have significant consequences. Depending on the attack, there are a myriad of ways that the information could be abused. For instance, information leaked from social networks could be leveraged to identify a user [47, 53], determine what their interests are [25], or infer who they were messaging with [26]. Through similar attacks, the search functionality of web applications has been shown to leak information about undisclosed vulnerabilities [11, 54] or credit card details [10].

This class of vulnerabilities is typically referred to as cross-site leaks, or XS-Leaks, and has received a lot of interest by the security community in recent years. The XS-Leak techniques exploit a large variety of browser mechanisms to leak sensitive information about opaque cross-site responses that are based on the state that the unwitting visitor has with the targeted website. In essence, every mechanism that deals with handling responses may be susceptible to being abused to leak information about these responses. As the causes of XS-Leaks are very diverse, a wide variety of defenses are needed to thwart them. This makes it very difficult for web developers to protect their users.

In this paper we aim to improve the understanding of XS-Leaks by studying the root cause of the leaks based on their intrinsic features, and highlight opportunities for future research to capture the entire threat surface that XS-Leaks pose and determine which protections are needed to practically defend against them. To this end, we introduce an extended model of XS-Leaks and show how the state that a victim has with a web application can be transfered to the state of a component that is involved in handling the request and associated response. By later retrieving this state from the component in a the second stage of an XS-Leak attack, the adversary will be able to infer private information that the victim shared with the targeted website. For example, when rendering a page of the targeted website, a specific resource may only be added to the cache when the victim is in application state $s_0$, and not in state $s_1$ (the state is based on a secret property unknown to the adversary – this could be logged-in or logged out, being connected with someone one a social network site, receiving at least one result for a search query, ...). The attacker can then reveal whether the user is in state $s_0$ or $s_1$ by detecting whether or not the resource was cached by using a timing attack to extract this "detectable state-difference" from the HTTP cache.

Based on this formal model, we introduce a classification structure that can be used to uniquely identify different issues based on their intrinsic characteristics, i.e. the affected component, inclusion method and leak technique. We discuss how our new insights in the underlying causes of XS-Leaks can be used to identify new vulnerabilities and propose a methodology to this effect. Furthermore, we explore and analyze the different defenses that can be used to thwart a variety of XS-Leaks and find that these can be divided in three main strategies: preventing state-changes in components, isolation, and ensuring that responses are stateless. By categorizing these, we aim to improve the general understanding of these defenses and spark discussion about potentially novel protections, as some defenses may be too coarse-grained to be adopted. Finally, we analyze the adoption rate of XS-Leak defenses, introduce Leakbuster[1], a dynamic web interface that facilitates defense deployment, and report on a case-study where defenses were deployed at large scale.

In summary, we make the following contributions:

- We introduce a formal model for XS-Leaks, highlighting the underlying intricacies that cause the vulnerabilities, and capturing the different stages that occur during an attack.
- Based on this model, we propose a classification method that can be used to uniquely characterize XS-Leaks, and introduce a methodology that can be used to detect new vulnerabilities.
- As part of our analysis of the little-explored web server component, we identify two novel XS-Leak attacks.
- We analyze the general strategies that are employed by defenses and find that a combination of isolation and mechanisms that prevent illicit requests are necessary.
- We share the insights of a real-world case study where defenses were deployed at large scale, and use these to create Leakbuster, a dynamic web interface that can be used by web developers to facilitate the deployment of defenses.

## 2 BACKGROUND: SAME-ORIGIN POLICY

When the web was originally envisioned, its main goal was to facilitate the sharing of public static information. It was not until later, after cookies were introduced to the web platform, and users could authenticate with websites, and share private information with them, that security became more important. However, as cookies were not designed with security in mind, and thus are attached to all requests of the domain they were set on, this gave rise to a new class of vulnerabilities. For example, in a CSRF attack, the attacker tricks their victim to send an authenticated request that performs an unconsented action on the target website, e.g. change the victim's password.

The automatic inclusion of cookies in requests did not only enable state-changing attacks, but is also at the base of attacks that aim to uncover information that a user shared with a particular website. As this clearly has significant security and privacy consequences, the same-origin policy was devised [31], This policy is a set of security principles that ensure that one origin cannot leak any information about resources from another origin, unless permission is explicitly granted via CORS headers. As a result, the concept of an origin (scheme, host, port) and site (scheme, eTLD+1) now is a security boundary, and information should be confined within this boundary. This includes the content of the response body, the header values as well as metadata, e.g. size of the response. Due to historical and practical reasons, some metadata is intended to be known, such as the dimensions of an image. However, it has been shown that other, potentially sensitive, information can be leaked across site boundaries through various side-channel attacks. These are referred to as XS-Leaks and are the main focus of this paper.

## 3 A DETAILED MODEL FOR XS-LEAKS

In this section we provide the description for our model on XS-Leaks, which will be referred to throughout the remainder of the paper. This model was created by thoroughly analzying the underlying commonalities of the previously-known XS-Leak vulnerabilities, which were obtained by a systematic literature review. Our model extends that of Knittel et al. [18] and focuses on the intricacies that form the foundation of XS-Leak vulnerabilities. These novel constructs allow us to comprehensively classify current attacks, propose a methodology to detect new attacks, and formally evaluate different defense strategies.

### 3.1 Running example

To better understand the practical aspects of XS-Leak attacks and to map these to our formal model, we first introduce a running example of a web application that is vulnerable to various XS-Leak attacks. This application provides a fairly straightforward search functionality, and the web page showing the search results is implemented as a Jinja template, as shown in Listing 1. The underlying application authenticates the user based on the cookies that are attached to the request, and performs a textual search on the user's private information based on a string provided in a GET parameter. For each result, the description is shown along with an icon that is loaded from a CDN. We assume that the application is secured against "typical" web security vulnerabilities, such as SQL injection and cross-site scripting. Interestingly, despite this fairly trivial functionality, there are multiple XS-Leak techniques that can be used to leak the user's private information. In fact, this example is based on real-world vulnerabilities that were discovered in a series of Google products [51].

Listing 2 shows the JavaScript code that an adversary could run on their site to determine whether any results were shown for a specific (attacker-supplied) keyword. The attack code first loads the resource in an iframe (although it could also open it in a separate window using `window.open()` or `window.opener`). It then waits for the document to load, and subsequently uses a timing attack to determine whether the icon was loaded from cache. As the icon is added to the cache only when there is at least one result, the cache entry would be indicative that at least one result was returned for the query. In order to check multiple queries, the attacker would

[1] https://distrinet.github.io/leakbuster/

```
<!DOCTYPE html>
<html lang="en">
<body>
<h2>Search results </h2>
<div class="results">
{% for result in results %}
<img src="//cdn.com/result-icon.png">
{{ result.description }}
{% endfor %}
</div>
</body>
</html>
```

**Listing 1: Example template of a search application.**

```
const icon_url = 'https://cdn.com/result-icon.png'
iframe.src = 'https://service.com/?q=password'
iframe.onload = () => {
  const start = performance.now();
  await fetch(icon_url);
  const duration = performance.now() - start;
  if (duration < 5) // loaded resource from cache
    console.log('Query had results');
  else
    console.log("No results for query parameter");
}
```

**Listing 2: XS-Leak attack against example application.**

need to reset the state of the cache, and thus use one of the known techniques [59] to remove the icon from the cache. Note that as a result of the recent network isolation defenses, this specific attack is prevented, but it still allows us to clarify important points about XS-Leaks during the different sections of the paper.

The example application is also vulnerable to other XS-Leak attacks. For instance, detecting the size of the response leaks whether there were any, and possibly how many, results [4, 53, 55]. Furthermore, a new connection might need to be established to retrieve the icon image from the CDN, which also only happens if there is at least one result. This can be detected by leveraging the global limit on the connection pool [6]. In case the execution time of processing the request depends on how many results are generated, this may also create a timing side-channel that can be exploited [10, 54].

## 3.2 Definition and threat model

For the XS-Leaks discussed in this paper, we consider a threat model where the victim lands on a web page that is (partially) controlled by an adversary. This could either be a malicious web page containing arbitrary code, a compromised web page, or a web page containing a malicious advertisement. Unless stated otherwise, we consider that no restrictions, e.g. through the sandbox attribute of an iframe, are imposed on the attacker's web page.

We define XS-Leaks attacks as follows: an attack where the adversary leverages various browser operations and observes their direct or indirect effects in order to infer information about cross-site resources that reflect the state that the user has with a targeted website[2]. These cross-site resources are typically dynamically generated based on the identity of the victim, which is inferred from the included cookie, the requested endpoint, and (attacker-provided) query parameters. Although attacks that aim to determine which websites were previously visited by a user are closely related to XS-Leaks attacks, these are out of the scope, and are known as history sniffing attacks or history leaks. These attacks on user privacy typically aim to infer the changes that are persisted in the browser by interactions that were made by the user. For example, the browser keeps track (locally) of which web pages a user visited, and applies a different style based on whether the URL was visited (which could be abused to leak previously visited pages [1, 19, 43]). In contrast, XS-Leak attacks require any change that is persisted in the browser to be the result by an operation of the attacker, e.g., an attacker-triggered request made to the targeted server.

## 3.3 Formal model

In this section we first present the base model, as was introduced by Knittel et al. [18], and then extend this model to more extensively describe the underlying characteristics of XS-Leaks. Our model was constructed by thoroughly analyzing existing XS-Leak issues, and for each determining the actual cause and required operations. To obtain a list of existing XS-Leak attacks, we performed a systematic literature review in which we considered all papers that were published in the past five years at a well-known security conference (S&P, CCS, NDSS, USENIX Security, ACSAC, AsiaCCS), or an attack- or web-focused workshop (WOOT, W2SP, SecWeb). We included every paper that presented an attack where the threat model matches the one required for XS-Leaks, and where the attack could be used to leak information from cross-site resources. We further extended this set of attacks with those reported in blog posts, online articles, or in disclosed bug reports to major browser vendors. In total we consider 38 distinct XS-Leak attacks (including two that were discovered as a result of our research).

**Base model:**

- The URL resource of the web application that is being targeted by the attacker is defined as $url$
- This web application can be in two different states, depending on the user's authentication: $S = \{s_0, s_1\}$.
- Depending on the state $S$, the application will behave differently, e.g., by including an additional image; the differences in this behavior is defined as $D = \{d_0, d_1\}$.
- A cross-site leak is then defined as the function $xsl()$ that outputs a bit $b'$; more precisely: $b' = xsl(sdr, i, t)$, where $i \in I$ represents the inclusion method that is used to trigger the request to the state-dependent resource (e.g. including the resource in an iframe), and $t \in T$ is the leak technique that is used to observe the difference cross-site (e.g. counting the number of frames in the rendered web page).
- Finally, the state-dependent resource that is being requested is defined as $sdr \in SDR$, where $sdr_0 = (url, (s_0, d_0))$.

Applying this to our running example, we get the following:

- $url$ = https://example.com/search?q=query
- $s \in$ {no-search-results, at-least-one-result}
- $d \in$ {no-icon-on-page, icon-on-page}
- The inclusion method $i$ is embedding the URL as an iframe
- The leak technique $t$ consists of performing a timing attack that aims to determine whether the inclusion of the web page lead to the icon being cached

**Extended model:** We extend the model of Knittel et al. by introducing the concept of **components**. These components are the abstract representation of all the different aspects that are interacted

[2] This definition is in line with that of prior work (XS-Leaks Wiki [46] and COSI attacks [49]), but makes it explicitly clear that the attacker aims to infer the state that the user has *with the targeted website.*

with when a resource is requested, parsed or rendered. Examples of such components include the HTTP cache, the network connection pool (at the level of the browser), the different DOM properties of the attacker's page and that of the rendered resource, or various aspects of the operating system on the client side such as CPU caches. We also consider components located on the server, which may depend on the infrastructure and environment of the targeted server, but will typically include the network layer at the OS-level and the various aspects related to the execution of the web application's implementation. Based on the layer at which the components are located, we can group them together in seven *component groups*, five of which are located on the client side: attacker tab, victim frame, victim tab, browser, and client OS. The remaining two component groups are located on the server-side: environment (e.g. OS or serverless computing environment) and web server.

Each component consists of a 3-tuple, where the first element, $c$, represents the specific characteristics of the component. The latter two elements of the tuple, $\sigma_{s_0}$ and $\sigma_{s_1}$, represent the **state that the component is in**, one for each state of $S$, i.e. the two different states that the application can be in based on the user's authentication. Initially, these two states are equal, but they may differ after the state-dependent resources have been included. For example, the state of the HTTP cache is initially empty, but after rendering the resource in state $s_0$, a resource will be added to the cache, whereas this would not happen in application state $s_1$, thus resulting in different component states. We formally denote the set of components as follows: $C = \{C_0, C_1, \ldots, C_n\}$, where each individual component is defined as follows: $\{c, \sigma_{s_0}, \sigma_{s_1}\} \in C_i$. We reference the state of the $i^{\text{th}}$ component, $C_i$, for the first application state, $s_0$, as $C_i.\sigma_{s_0}$.

Next, we introduce the state-transfer and state-retrieval functions. The **state-transfer function**, $st$, will execute a specific inclusion technique $i$ on the targeted state-dependent resource, and thus might bring the components in a different state. As such, the aim of this function is to *transfer* the web application state into the component state. We formalize this function as follows: $st\colon (sdr, C, i) \longrightarrow C'$, where $C'$ represents the same set of components but after inclusion of the $sdr$. This inclusion may cause the initial state of a component to be altered, i.e. when $C_i.\sigma_{s_0} \neq C'_i.\sigma_{s_0}$ or $C_i.\sigma_{s_1} \neq C'_i.\sigma_{s_1}$. For each component that was involved in the state-transfer, the state may have changed differently depending on the web application state (and thus depending which version of the resource was included). We can describe the **differential component state** in our model as $\Sigma = C'_i.\sigma_{s_0} \oplus C'_i.\sigma_{s_1}$ (where $\oplus$ is the XOR operator). This differential component state reflects the difference in the final state of the component depending on whether the resource in application state $s_0$ or in $s_1$ was included. For simplicity, throughout the remainder of the text, we will refer to this differential component state as $C'_i.\Sigma$.

The **state-retrieval function**, $sr$, aims to subsequently extract the differentiating state-changes that were introduced by the state-transfer function. More precisely, given the set of components in their initial state and possibly-altered state, $C$ and $C'$, this function will return the set of state-changes that could be detected with the leak technique $t$. Formally, we represent this function as follows: $sr\colon (t, C_n, C'_n) \longrightarrow L$, where $L$ represents the set of detectable (using leak technique $t$) state-differences (between $C'_i.\sigma_{s_0}$ and $C'_i.\sigma_{s_1}$). In our running example, the state-difference is that in one application state the resource will be cached, but not in the other application state. We can detect this state-difference by using a timing attack based on the response time.

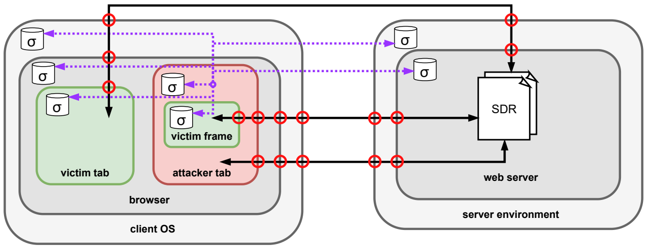


**Figure 1: Visual representation of our XS-Leaks model.**

The set of **detectable state-differences**, $L$, consists of two parts, where the first one, $\Delta$, represents valuative differences, e.g. a difference in the number of frames. The latter part, $\Theta$, represents differences in the time domain, where the resulting value of the leak technique is exactly the same in both states, but where the timing differs. Another example of this case, is where the processing time of a certain request takes longer depending on the state of the user. While the resulting response may be exactly the same, the time it takes for the response to be received by the victim's browser will differ. We formalize the set of differences in both domains as follows: $L = \{\Delta, \Theta\} = \{(\delta_0, \delta_1, \ldots, \delta_n), (\theta_0, \theta_1, \ldots, \theta_m)\}$.
In the case of our running example, we now get the following:

- We first transfer the application state to one of the components by embedding the resource as an iframe: $st(sdr, C, i)$.
- In application state $s_0$, the state of one component, the HTTP cache, will be altered, while this does not occur in the case of $s_1$: $C_i.\sigma_{s_0} \neq C'_i.\sigma_{s_0} \land C_i.\sigma_{s_1} = C'_i.\sigma_{s_1}$. As a result, the differential component state is non-empty: $C'_i.\Sigma \neq \emptyset$.
- Using a timing attack as leak technique, we can now retrieve the state from the component: $sr(t, C_i, C'_i)$.
- This results in a single detectable state-difference in the time domain: $L = \{\{\}, \{\theta_0\}\}$, where $\theta_0$ represents the timing difference when requesting a cached vs. non-cached resource.

A simplified visual representation of our model can be found in Figure 1, showing the seven component groups that each have their own state. Furthermore, when a state-dependent resource is included (black arrow), it can trigger state-changes in all the components that it passes through (as indicated by the red circles). Extracting the detectable state-differences is then performed by launching a leak technique in the state-retrieval function (indicated by the purple dotted line).

## 3.4 Conditions for XS-Leaks

Using the extended formal model, we can now express the conditions that need to be met for an XS-Leak issue to exist. A first prerequisite is that the state of a component needs to be changed as the result of an inclusion method, $i$, in at least one of the two application states, $s_0$ and $s_1$, expressed formally: $(\exists C_i, i)(C_i.\sigma_{s_0} \neq C'_i.\sigma_{s_0} \lor C_i.\sigma_{s_1} \neq C'_i.\sigma_{s_1})$, where $C'_i = st(sdr, C_i, i)$. Furthermore, the state-changes that occur in the component need to be different between the two web application states (which result in two different responses). Concretely, this means that the differential component state of at least one component needs to be non-empty: $(\exists C_i, i)(C'_i.\sigma_{s_0} \oplus C'_i.\sigma_{s_1} \neq \emptyset)$, or simplified: $(\exists C_i, i)(C'_i.\Sigma \neq \emptyset)$.

The other prerequisites for a successful XS-Leaks attack is that it needs to be possible to extract this differential state from the component. This means that there needs to exist a leak technique, $t$, for which the state-retrieval function returns a non-empty set of detectable state-differences. Formally, we can express this as follows: $(\exists C_i, i, t)\ [sr(t, C_i, C'_i) \neq \emptyset]$. These detectable state-differences can be either valuative differences or differences in the time domain: $L \neq \emptyset \Leftrightarrow \Delta \neq \emptyset \vee \Theta \neq \emptyset$.

Using this extended representation of the XS-Leaks model, we can better reason about the cause of the different attacks that are known to date (Section 4), determine various methods to detect new attacks (Section 5), analyze the different strategies used for defenses (Section 6), and whether these are sufficient to provide a complete protection (Section 7).

## 4 XS-LEAK ATTACKS: CURRENT STATE

Although the term XS-Leaks is relatively new, the issues have been known for more than two decades, e.g. in 2000, Felten and Schneider described how the time to request a resource leaked information about its cache status. Prior work on categorizing XS-Leaks has mainly focused on enumerating the different known techniques and grouping them by the technique that is used [46], or based on the differences in the resource that can be detected [18, 49]. In this section we introduce a new classification method that is based on our model and aims to capture the intrinsic properties of XS-Leaks; namely the component to which the web application state is transferred, the inclusion method that is used for this state-transfer, and the technique that is used to finally extract this state.

### 4.1 Classification attributes

Based on our model of XS-Leaks we now propose a classification that can be used to uniquely distinguish different XS-Leak attacks. This classification allows us to capture which types of XS-Leaks have already been found, and can thus be used as a means to guide future research (which we explore in more detail in Section 5). We determine several attributes that can be appointed to an XS-Leak, most of which originate directly from the model.

**Component group** As described in our model (Section 3), this attribute captures the group of components in which the relevant state-change occurs that eventually leads to the XS-Leak.

**Inclusion method** This property reflects the inclusion method that is used to trigger the state-transfer. For reasons of clarity, we divided these in groups: *iframe* (where the attacker embeds the state-dependent resource in an iframe), *other window* (where the resource is rendered in a different tab, using `window.open()` or `window.opener`), and *direct* (using a DOM API, e.g. `<img>` or `fetch()`, to include the resource – here we also indicate whether a specific API or any API is used).

**State difference** For this attribute we also summarized the possible values in three different groups: *event fired* (when an event is fired that can be observed by the attacker), *change of a property* (when the value of a certain property within the component changes, e.g. a new entry being added to the cache), and *consumption of a limited resource* (when there is a resource in the component that can only be consumed a limited amount of times, e.g. the Cache API has a global quota).

**Leak technique** The leak techniques extract the altered states from the components. These can be as straightforward as observing the time that an event occurs, or reading out the value of a certain property. Leak techniques may also be more complex, and may require probing. For instance, it is not possible to directly determine the cache status of a resource, but instead a timing attack needs to be used.

**Information in timing** For certain XS-Leaks the state-change that occurs does not leak any sensitive information about the resource. Instead, the time that the state-change occurs is what leaks the information. With this attribute we indicate whether the XS-Leaks only has detectable differences in the time domain; formally: $\Delta = \emptyset \wedge \Theta \neq \emptyset$.

**Idempotency** When after inclusion the state of certain components changes permanently and irreversibly, the XS-Leak is considered to be non-idempotent. For example, once a resource has been cached, it will remain cached until it becomes invalid. Consequently, the attack can only be executed once (unless there exists a technique to revert the state-changes). In other words, an XS-Leak is idempotent if there exists a method that can revert the state-changes made to the affected component: $(\exists\rho)[\rho(C'_i.\sigma) = C_n.\sigma]$.

**Differentiating aspect** Finally, we capture the various aspect(s) of the state-dependent resource that are detectable. For brevity, we categorize these in four groups: headers, content, metadata (such as size of the response), and generation process.

### 4.2 Applying the classification

We now apply our classification to all the different XS-Leaks that we found in the systematic literature review. We group together attacks based on the core mechanism that is used to leak the information. For instance, Bortz and Boneh showed that the time to download a response relates to the size of that response [4]. Gelernter and Herzberg introduce an amplification attack where the difference in response size grows extensively [10, §3.1], making the attack significantly more accurate, but still uses the original mechanism to leak the information. Similarly, in another attack introduced by the researchers, the execution time on the server is amplified [10, §4], which is similar to what Van Goethem et al. measure in their timeless timing attacks [54].

The classification of known XS-Leak attacks is shown in Table 1, and a brief summary of each attack is provided in the Appendix, Section A. Based on these results, we can make several observations. We find that there is a lot of diversity in the attacks: each component group is responsible for at least one attack. Furthermore, the majority of the leaks are caused by state-changes in the attacker page. We believe the reason for this is twofold. First, most mechanisms that can be used to include remote cross-site resources originate from the attacker page, and any resource-dependent parsing or processing is likely to introduce a leak. Secondly, the state-changes that occur in the attacker page are mostly directly observable and

**Table 1: Classification of known XS-Leak attacks**

| Attack | Component group<br>$\{C_i, \ldots, C_j\}$ | Inclusion method<br>($I$) | State difference<br>($L = \{\Delta, \Theta\}$) | Leak technique<br>($T$) | Information in timing? | Idempotent? | Differentiating aspect<br>(D) | Reference(s) |
|---|---|---|---|---|---|---|---|---|
| (in)valid script/image/... | attacker page | direct: specfic API | event fired | observation of an event | ○ | ● | content / headers | [62], [47] |
| client redirect (load event) | attacker page | iframe | event fired | observation of an event | ○ | ● | content | [65] |
| server redirect (max redirect count) | attacker page | direct: any API | consumption of limited resource | observation of an event | ○ | ● | headers | [12], [33] |
| server redirect (CSP violation) | attacker page | direct: any API | event fired | observation of an event | ○ | ● | headers | [65] |
| server redirect/status (AppCache) | attacker page | direct: any API | event fired | observation of an event | ○ | ● | headers | [23], [12] |
| server redirect (Fetch manual) | attacker page | direct: specfic API | event fired | observation of an event | ○ | ● | headers | [12] |
| element id focus | attacker page | iframe | event fired | observation of an event | ○ | ● | content | [64] |
| overly broad postMessage | attacker page | iframe / other window | event fired | observation of an event | ○ | ● | content | [66] |
| detect CORB'ed JSON responses | attacker page | direct: specfic API | event fired | observation of an event | ○ | ● | headers / content | [2] |
| detect CORP header | attacker page | direct: any API | event fired | observation of an event | ○ | ● | headers | [60] |
| detect COOP header | attacker page | other window | change of a property | reading a property | ○ | ● | headers | [61] |
| detect XFO (<object>) | attacker page | direct: specfic API | event fired | observation of an event | ○ | ● | headers | [49] |
| detect XFO (Resource Timing) | attacker page | direct: any API | change of a property | reading a property | ○ | ● | headers | [68] |
| response size estimate (parsing) | attacker page | direct: specfic API | event fired | observation of an event | ● | ● | metadata | [53] |
| response size estimate (Cache API) | attacker page | direct: specfic API | event fired | observation of an event | ● | ● | metadata | [53] |
| response size (Quota API) | attacker page | direct: specfic API | change of a property | reading a property | ○ | ● | metadata | [55, §3.4.1], [15] |
| response size (global quota eviction) | attacker page | direct: specfic API | consumption of limited resource | probing a property | ○ | ● | metadata | [55, §3.4.2] |
| cross-site pixel stealing | attacker page | iframe | event fired | observation of an event | ● | ● | content | [1], [19] |
| Performance API entries | attacker page | direct: any API | change of a property | reading a property | ○ | ● | headers / content | [18] |
| CORS error leaks redirect URL | attacker page | direct: specfic API | change of a property | reading a property | ○ | ● | headers | [18] |
| SRI leaks response size | attacker page | iframe / other window | consumption of limited resource | probing a property | ○ | ● | content | [18] |
| appearance of download bar | browser | other window | change of a property | reading a property | ○ | ○ | headers | [65] |
| cache probing | browser | iframe / other window | change of a property | probing a property | ○ | ○[†] | content | [59], [7] |
| Loophole (event loop timing) | browser | iframe / other window | consumption of limited resource | probing a property | ● | ● | content | [56] |
| Safari ITP leaks | browser | other window | change of a property | probing a property | ○ | ○ | content | [13] |
| detecting connections (pool limit) | browser | iframe / other window | consumption of limited resource | probing a property | ○ | ● | content | [6] |
| WebSocket global limit | browser | iframe / other window | consumption of limited resource | probing a property | ○ | ● | content | [18] |
| Payment API global limit | browser | iframe / other window | consumption of limited resource | probing a property | ○ | ● | content | [18] |
| CPU cache attacks | client-side OS | iframe / other window | change of a property | probing a property | ● | ● | content | [32], [42], [41] |
| response timing (size) | server environment | direct: any API | event fired | observation of an event | ● | ● | metadata | [4], [10, §3.1] |
| frame counting | victim iframe / victim window | iframe / other window | change of a property | reading a property | ○ | ● | content | [63] |
| no navigation due to download | victim iframe / victim window | iframe / other window | change of a property | reading a property | ○ | ● | headers | [65] |
| window.name leak | victim iframe / victim window | iframe / other window | change of a property | reading a property | ○ | ● | content | [58] |
| client redirect (History API) | victim window | other window | change of a property | reading a property | ○ | ● | content | [65] |
| response timing (execution time) | web server | direct: any API | event fired | observation of an event | ● | ● | generation process | [10, §4], [54], [36] |
| XSSI | web server | direct: specfic API | change of a property | reading a property | ○ | ● | content | [24] |
| request count: rate limit | web server | iframe / other window | consumption of limited resource | probing a property | ○ | ● | content | Section 5.1.1 |
| request count: max requests per connection | web server | direct: specfic API | event fired | reading a property | ○ | ● | content | Section 5.1.1 |

[†] Although the default technique is non-idempotent, there exists at least one technique that enables idempotent execution of this XS-Leak.

thus easier to detect, and in several cases even described in the specification. On the other hand, it is not directly clear that creating a new connection will consume a limited resource (the connection pool), which only has an observable side-effect after actively probing for the state-change.

Another interesting observation is that the mechanisms aimed at defending against XS-Leaks can also be the source of a leak themselves. For instance, if the CORP header is only enabled based on the state of the user, an attacker could leverage this to leak information. Hence, it is important for web developers to consistently apply the defenses, and ensure features are not enabled conditionally.

## 5 XS-LEAK ATTACKS: FUTURE RESEARCH

To date, most XS-Leaks that have been discovered and reported are isolated issues that result from evaluating a particular API. Consequently, this does not provide any information about the possible number of XS-Leaks that currently remain undetected. We believe that systematic analyses are paramount for future research to discover previously unknown XS-Leaks, or to prove the absence of XS-Leaks in certain components. To guide this systematic analysis, the concept of components, whose state can change as the result of an inclusion, is instrumental. We believe that two aspects are needed to further map out the attack landscape on XS-Leaks, namely carefully listing all the possible components involved in the inclusion process, and for each component detecting whether the application state is transferred into the component state. Next, we explore these two aspects in more detail.

### 5.1 Enumerating components

In our classification we considered seven different component groups, both on the side of the client as well as on side of the server. However, each of these component groups can have numerous sub-components, which each may have their own state. As such, by mapping out all the different components in the smallest fragment possible, researchers can determine which aspects could *possibly* lead to XS-Leaks. More concretely, any interaction that is made with a state-dependent resource, regardless of whether this is persistent (e.g. adding it to the cache) or ephemeral (e.g. verifying the content type), could possibly trigger a change in the state of the component. Any such state-change could then result in an XS-Leak, provided there exists at least one leak technique that can infer this state-change. It is important to note however that not every state-change will lead to an XS-Leak, as it might not be feasible to retrieve the state from certain components.

Systematically enumerating all components is an arduous task as it involves operations that occur at various levels, each of which can be extremely complex or not well documented. As such, we believe that an iterative process guided by information that is already available (e.g. web specifications) would be most practical. More precisely, researchers could for each component group determine all the different operations that occur on the resource, and then for each operation determine which components are involved, in an increasingly fine-grained manner. As the list of components can be considered cumulative, any research that further refines the components would contribute to mapping out the threat surface.

*5.1.1 Novel attacks on determining triggered requests.* As part of our research, we partially analyzed the various sub-components of the little-explored server environment component group, resulting in two novel attacks that can leak the number of requests that were initiated by the user, e.g. as the result of rendering a state-dependent resource. Both attacks leverage a limit that is imposed on requests ($\Delta = \{consumption\_of\_limited\_resource\}$), which can be extracted by a probing leak technique. The first attack abuses the fact that web servers limit the number of requests that can be made over a single connection. For some of the most popular web servers, this limit is set by default to 100 (Apache, via the `MaxKeepAliveRequests` directive, and older nginx versions) or 1,000 (newer nginx versions, via the `keepalive_requests` directive). Once this limit is reached, the web server will close the connection. In an attack, the adversary could first trigger the rendering of the state-dependent resource (e.g. by loading it in an iframe), which would initiate either $n$ or $n-1$ requests, depending on the state of the user. Subsequently, the attacker sends $100-n$ requests and then determines whether the next request would require opening a new connection (e.g. based on a timing side-channel). From this, the attacker can infer whether $n$ or $n-1$ requests were made and thus reveal the user's state.

The second attack follows the same principle, but instead leverages the limit imposed by rate-limiting systems that aim to protect the website. Once the predefined maximum number of requests within the allotted time window has been reached, the web server will most likely generate a different response. This could include closing the connection, or returning an error message. This could in turn be probed by the attacker. The main difference between the two attacks is that the second one is URL-specific, whereas the first one only reveals information about the total number of requests sent to a specific origin. Furthermore, browsers with a keyed connection pool can defend against the first attack (under the condition that the embedding frame is part of the key), whereas the rate-limit is most likely to be IP-based. Consequently, a keyed connection pool would not deter this attack.

### 5.2 Detecting component state-changes

An important prerequisite for the existence of an XS-Leak, is that after inclusion of the state-dependent resource, the state is transferred to the component. As such, only components that experience a state-change can be the source of XS-Leaks. In order to detect this, we propose the following three-pronged approach, which is agnostic of the component, or component group, that is being evaluated. First, a web server should be set up that serves resources that each have a different differentiating aspect ($d \in D$), compared to a baseline resource. This is needed because a component may only trigger a state-change when the differentiating aspect is present.

Secondly, a state-monitoring system can be set up that tracks the state of the investigated components. How this can be achieved largely depends on the component group that is analyzed. For instance, for components that relate to the DOM, all the different property values, and APIs could be enumerated, capturing a snapshot of the state of the DOM. On the other hand, the state of other component groups such as the browser or client OS might be less accessible, and would thus require additional effort to evaluate.

For example, this might require monitoring the value of certain variables in the browser or the kernel, or determining whether certain resources such as the disk are accessed. The monitoring of variables would cover the cases whether the state difference is the consumption of a limited resource, or the change of a property. However, a state-change may also initiate side-effects, such as firing an event, and to capture these, the execution path should also be monitored as any (consistent) difference in execution could result in an observable side-effect.

Finally, using the exhaustive list of inclusion methods ($i \in I$), each resource, pairwise with the baseline without the differentiating aspect, can be included. By comparing the state of the analyzed component before and after the inclusion, and monitoring the execution trace, it now becomes possible to evaluate whether the differentiating aspect triggers a different state-change in the component. Through this method, every possible state-change will be captured, and, as a result, it is possible to determine whether the component could potentially be the source of an XS-Leak. Once a differentiating state-change has been detected, it is still required to determine whether there is any leak technique that can reveal this state-change, which can be highly specific to the associated component and the type of the state-change. Hence, we believe that this last step cannot be automated.

## 6 XS-LEAKS DEFENSES: CURRENT STATE

In recent years, several new defenses against XS-Leak attacks have been introduced to the web platform. In general, all defenses follow one of the three different strategies: preventing state-changes, isolating components, or ensuring that requests and responses are stateless. In this section we provide a brief overview of each strategy and the defenses that follow them.

### 6.1 Preventing state-changes

As we have shown through our model, XS-Leaks can exist when the state of a resource is transfered to the state of a component, in particular when the components is brought in a different state depending on the response that is returned. By preventing any state-changes from occurring in certain components, the XS-Leaks that they introduce will be mitigated. If we consider $\mathcal{D}$ as the model where the defense has been applied, and $\mathcal{U}$ as the unprotected model, we can formally describe the strategy as follows:

$$\mathcal{U} : (\forall x \in [i, j], j > i)(C'_x.\sigma_{s_0} \cup C'_x.\sigma_{s_1} \neq \emptyset \land C'_x.\Sigma \neq \emptyset)$$

$$\mathcal{D} : (\forall x \in [i, j])(C'_x.\sigma_{s_0} = C_x.\sigma_{s_0} \land C'_x.\sigma_{s_1} = C_x.\sigma_{s_1} \land C'_x.\Sigma = \emptyset)$$

More precisely, without the defense applied, there is at least one component, $C_x$, that after inclusion ($C'_x = st(sdr, C_x, i)$) has an altered component state in at least one of the two application states, and the resulting component state is different depending on the application state, i.e. the differential component state is non-empty. When the defense is in place, there are no state-changes that occur in the component, and therefore the differential component state will also be empty.

In practice there are two defenses that follow this strategy: framing protection and response blocking. Concretely, **framing protection** can be enabled by setting the `X-Frame-Options` header or by using the `frame-ancestors` directive of CSP. When the browser encounters any of these two policies that indicate the resource should not be embedded in an iframe, it will stop loading the resource and prevent if from rendering. Consequently, any state-change that would occur by rendering the resource, will now be prevented. In our running example, the icon image will no longer be added to the HTTP cache when a framing protection has been set on the page, and thus no state-change occurs. In general, this defense will block all XS-Leaks that rely on iframes as the inclusion method.

**Defense:** framing protection
**XS-Leaks thwarted:**

$$i = \{iframe\} \land (C_{i..j} = \{victim\_frame\} \lor d = \{\text{content}\})$$

In order to prevent cross-site responses from being accessible in the same renderer process, some browsers will block the loading of "sensitive" resources through the Cross-Origin Read Blocking (CORB) mechanism. This defense, originally intended to thwart Spectre attacks, will stop loading a response when it detects that its content type is potentially sensitive (i.e. JSON, HTML and XML). Alternatively, a website could also directly indicate that a certain resource is not intended to be included in a cross-site context by setting the `Cross-Origin-Resource-Policy` (CORP) header [30]. By blocking the further loading of such a response, no state-changes will occur after the check. Because this check occurs after the headers have been parsed, it is still possible that the parsing of the headers can cause certain state-changes. Furthermore, the check occurs in the browser component group, and only when it is included directly from an attack tab. As a result, state-changes may have still occurred in the component groups that were involved in the inclusion before this check was performed (client OS, browser, server environment and web server).

**Defense:** response blocking
**XS-Leaks thwarted:**

$$d \neq \{headers\} \land C_{i..j} = \{attacker\_page\} \land i = \{direct\}$$

### 6.2 Isolation defenses

A second strategy to defend against XS-Leaks is to allow the state-changes to occur in the components, but prevent the attacker's page from accessing them by means of isolation. Concretely, in the defended case there is no longer a leak technique that can be used to extract the state from the component:

$$\mathcal{U} : (\forall x \in [i, j], \exists t \in T)[C'_x.\Sigma \neq \emptyset \land sr(t, C_x, C'_x) \neq \emptyset]$$

$$\mathcal{D} : (\forall x \in [i, j], \forall t \in T)[C'_x.\Sigma \neq \emptyset \land sr(t, C_x, C'_x) = \emptyset]$$

To date, there are three defenses that follow this strategy: network isolation, cross-origin opener policy, and site isolation. The **network isolation defenses** partition certain aspects of the browser such as the HTTP cache and various network-level properties, e.g. the connection pool (a complete list can be found in the Chromium explainer document [27]), according to the top-level document from which the resource was requested. Chromium-based browsers additionally add the embedding frame as part of the isolation key. On Safari, only the HTTP Cache has been partitioned [57], while other network properties remain shared among different tabs. In an attack, when the resource is rendered in a new tab (victim window), the attack will not be able to determine whether this caused a resource to be added to the cache, because the attacker's HTTP cache is isolated from the HTTP cache used by the victim site.

**Defense:** network isolation
**XS-Leaks thwarted:**
$C_i \in$ *{HTTP_cache, connection_pool, …}* $\wedge\ i \neq$ *{direct}*

Another isolation defense is based on preventing attackers from retaining references to other windows. This defense can be enabled through the Cross-Origin Opener Policy (**COOP**), by setting the similarly named response header [29, 61]. At the time of this writing, the header is supported by all major browsers. In essence, when the policy's header value is set to `same-origin`, it ensures cross-origin pages do not have a reference to the other window. For example, this prevents the attack that counts the number of frames in a page from accessing the `win.frames.length` property. Because an attacker page can also include the target resource in an iframe, it is important that this defense is complemented with framing protection.

**Defense:** COOP
**XS-Leaks thwarted:** $C_{i..j} =$ *{victim_window}*

Finally, to counter attacks that leverage state-changes occurring at the microarchitectural level, or that are related to the process in which web pages are rendered and executed, a new isolation primitive, **site isolation**, was introduced [34]. In essence, site isolation ensures that documents of different sites are rendered in a separate process. This means that as long as no sensitive cross-site resources are loaded into the renderer, these are protected from attacks such as Spectre. As such, resources that are included directly may still cause μ-architectural state-changes.

**Defense:** site isolation
**XS-Leaks thwarted:** $C_{i..j} =$ *{μ-arch}* $\wedge\ i \neq$*{direct}*

### 6.3 Stateless responses

The third defense strategy that can be used to mitigate XS-Leaks is to ensure that all responses to illicit requests are exactly the same, regardless of the web application state. To accomplish this in all components, the generation of the response should not be based on the web application state. When the two web application states are identical, e.g. because the user's authentication is not included in the request, all components will remain in their original state.

$$\mathcal{U} : (\exists x \in [i, j])(s_0 \neq s_1 \wedge d_0 \neq d_1 \wedge C'_x.\Sigma \neq \emptyset)$$

$$\mathcal{D} : (\forall x \in [0, n])(s_0 = s_1 \wedge d_0 = d_1 \wedge C'_x.\sigma_{s_0} = C'_x.\sigma_{s_1} = C_x.\sigma_{s_0})$$

In practice, there exist two defenses that are based on ensuring the responses remain stateless, namely SameSite cookies and Fetch Metadata request headers. When a cookie's **SameSite** attribute is set to `Lax` (the default in Chromium-based browsers [5]), the cookie, and thus the user's authentication, will not be included in the request [28, 37]. As a result, the server will not be able to authenticate the user and will thus always return a stateless response. Because the SameSite cookie will still be included in navigational GET requests, attacks that rely on opening a new window are still possible.

**Defense:** SameSite cookie (Lax)
**XS-Leaks thwarted:** $i \in$ *{direct, iframe}*

To give web developers more insights on what caused the browser to send a request, the **Fetch Metadata** request headers provide information on the context in which a request was made. More specifically, the `Sec-Fetch-Site` header indicates whether the request was made in a cross-site, same-site or same-origin context, or whether the request was the result of a navigation request. Similarly, the `Sec-Fetch-Mode` indicates the "mode" in which the request was made: using CORS or not, as the result of a navigation, or for a WebSocket. By combining the different headers and evaluating their values before the request is processed, the server can determine the legitimacy of the request and return a static (stateless) error message when the request is considered illegitimate. A typical and recommended way of applying a policy based on these request headers is via the resource isolation policy (RIP) [67], which only allows same-site requests or GET requests that result from navigations or embedding.

**Defense:** Fetch Metadata (RIP)
**XS-Leaks thwarted:** $i =$ *{direct}*

## 7 XS-LEAKS DEFENSES: FUTURE RESEARCH

Although many different defenses exist, using one of the three different strategies, it remains to be seen whether these are sufficient. In this section we explore whether it is feasible to provide a complete protection against all possible XS-Leak attacks, including those that have not yet been discovered to date. Furthermore, we discuss directions for future work to explore whether the current set of defenses are sufficient to provide websites with the necessary protections.

### 7.1 Complete protection against XS-Leaks

As we have constructed an expression that captures the set of possible XS-Leaks each defense protects against, we can simply apply constraint solving to determine whether there exists any set of defenses for which no XS-Leak attack is still possible. Unfortunately, we find that there is no combination of defenses that provides a complete protection against all XS-Leaks, posing an interesting direction for future research. The main culprit for this are attacks where a different window is used as the inclusion method, as there is no defense that can completely protect against these. Although COOP prevents the attacker from reading out any state from the window in which the resource was rendered, any intermediary component group between the victim tab and web server may still exhibit state-changes as a result from rendering the resource (e.g. it is still possible to observe the response generation time). However, it is important to note that although COOP does not provide these complete protections in our model, it does significantly hinder attacks in practice. For each bit of information that the attacker would try to extract, a new window needs to be opened, requiring a user interaction.

The minimal set of currently available defenses that provides the best possible protection against XS-Leaks is the combination of SameSite cookies, COOP and site isolation. The SameSite attribute on cookies ensures that all cross-site requests, either included directly or via an iframe, are stateless. Additionally, COOP provides the best available protection for when resources are included in a different window. Finally, site isolation ensures that any microarchitectural state-changes that were caused by rendering the resource cannot be observed.

In case SameSite cannot be used, e.g. because the website relies on specific cross-site interactions, this defense could be exchanged for the combination of RIP (based on Fetch Metadata) and framing protection. However, the framing protection provides a strictly less extensive defense compared to SameSite cookies as the stateful response body will still traverse several components, possibly causing state-changes. Although no side-effects caused by rendering the resource can be observed, it might still be possible to leak information about the metadata, headers or generation process of the resource. Instead of using the framing protection set in response headers, websites could also use Fetch Metadata to block all requests that originate from iframes (using a Framing Isolation Policy, based on the `Sec-Fetch-Dest` header). Although this mechanism does not provide the same flexibility as the framing protections because it does not allow specific frame ancestors, blocking the request early on does prevent state-changes in all components. This example shows that cross-site interactions may complicate the implementation of XS-Leaks defenses. In the remainder of this section, we explore future research directions that could improve this situation.

### 7.2 Defenses and cross-site interactions

Many of the current defense mechanisms allow websites to either opt-in or opt-out of cross-site interactions. For example, the CORP header can only be set to `same-origin`, `same-site` or `cross-site`. This limits the granularity in which the defense can be applied, and it may be prohibitive to use this in conjunction with website functionality that relies on cross-site interactions. Examples of such cross-site interactions include social plugins, authentication flows, third-party video content, .... In many cases, these require the user's authentication to provide the intended user experience. It is for this reason that the SameSite-Lax-by-default policy in Chromium makes an explicit exception for cookies that were set within the last two minutes, as this would cause various authentication flows to break [35]. As the adoption of XS-Leaks defenses is slowly ramping up, we believe that it is an interesting direction for future work to evaluate whether the current set of defenses is sufficient to provide the maximum protection for websites that rely on cross-site interactions.

To further protect websites from XS-Leaks, web browser could enable certain policies by default (as the ones indicated above) or try to eradicate legacy features from the web platform (e.g. the `frames.length` property stems from the time when iframes were widely used to build websites). Based on the most recent data (October 2021) from the HTTP Archive[3] crawl over the home page of 5.5M sites, we find that there are only 3.07% of these sites to which at least one request was initiated in a cross-site context. This clearly indicates that most sites do not intend to be included by other sites at all, and thus would greatly benefit from policies that are enabled by default. As such, we believe that research in the direction of determining how XS-Leaks defenses can be enabled by default without negatively impacting intended functionality could significantly advance the state of security on the web.

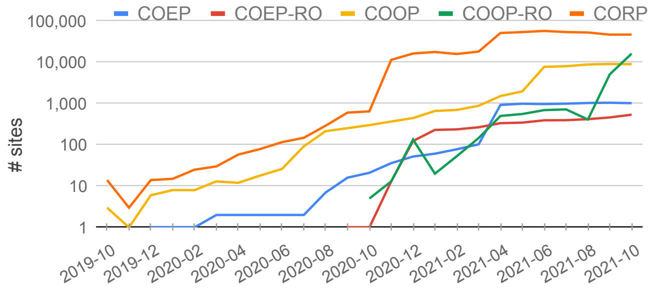


**Figure 2: Adoption of XS-Leak defenses over time (log y-axis).**

## 8 IMPROVING DEFENSE DEPLOYMENT

While several defense mechanisms exist that can adequately protect against virtually all practical attacks, an obvious prerequisite for these defenses to be effective, is that websites need to deploy them. In this section we explore to what extent websites already make use of them, introduce LEAKBUSTER, a web application to facilitate deployment, and discuss the difficulties that web developers are still facing based on a case-study where XS-Leaks defenses were deployed at large scale.

### 8.1 Current adoption rate

To analyze the evolution of the adoption of XS-Leaks defenses, we rely on the public datasets provided by HTTP Archive, containing detailed information on the visits of home pages of approximately 5.5M sites. In all same-origin responses, we look for the presence of the three main headers that control the XS-Leaks defenses in the browser. In Figure 2 we show the number of sites that adopt one of the policies, or a report-only (RO) version of it, over the past two years. This evolution shows that the adoption of security features is gradually increasing. Furthermore, we expect this growth to continue: e.g. in October 2021 we find more sites that enable the report-only mode of COOP (16,223), compared to those that implement the enforcing version (8,997). This is indicative that a growing number of websites intends to use this mechanism, but these are still in the early stages of deployment.

The most common XS-Leaks defense mechanism in use is CORP, with 46,805 sites setting the header in October 2021. It should be noted that the vast majority of these set the CORP header to `cross-site`, which does not provide any additional protection. However, if sites want to use COEP (and thus be able to make use of certain sensitive browser features), all resources that it includes need to set a CORP header. As websites include many third-party resources, many of which they do not have under their control, this will likely complicate the adoption of COEP. To facilitate the deployment of COEP, browser vendors are now experimenting with a `credentialless` version of COEP, allowing resources that were requested without credentials to be included without a CORP header [16].

[3] https://httparchive.org/

## 8.2 Leakbuster

Although the adoption of XS-Leak defenses is steadily growing, sites that deploy them only represent a minute fraction of the web: 0.84% for CORP, 0.16% for COOP, and 0.02% for COEP. A possible reason for this low adoption rate is that XS-Leaks are still not widely known or understood. Furthermore, given the extensive number of defenses, and various intricacies that need to be considered to deploy a complete solution, it may be hard for web developers to decide which defenses need to be deployed, and whether there are any alternatives. To tackle these issues, and facilitate the deployment of defenses, we introduce Leakbuster as a dynamic web interface[4]. The tool is based on our model, the classification of attacks, and the expressions that capture which attacks are thwarted by certain defenses. Upon entering the values for the relevant security headers and policies, web developers will be presented with a list of XS-Leak attacks they are still susceptible to along with a list of prioritized suggestions. By following the recommendations of Leakbuster, website owners will be able to protect their users to the best extent possible.

## 8.3 Challenges of adoption

As we discussed in Section 7.2, website are still facing several challenges when trying to adopt XS-Leak defenses, especially when the website wants to support cross-site interactions. In this section we elaborate on these challenges based on a case-study of a large technology company that deployed COOP and RIP based on Fetch Metadata across 500+ services, serving over a billion users. The information and insights presented here were obtained by working and discussing with the team that deployed these defenses.

When a web application is published, other sites may (legitimately) interact with it in ways that are not clear to the web developers; e.g. other sites might want to open the application in a pop-up and close this pop-up after some time. If the web application would set a COOP policy, this behavior would break as the embedding web site would not be able to close the pop-up. As a result, just setting an enforcing policy could interfere with intended behavior and deteriorate users' browsing experience. To overcome this challenge, it is generally advised to gradually roll out a defense and first enable a report-only version of the defense; these are available for both COOP and COEP, and as Fetch Metadata is enforced on the server side, it is possible to generate reports when a violation occurs. Unfortunately, the reporting of COOP violations is not complete, as this could possibly leak additional information in specific cases. We document these reporting gaps in Appendix B. We believe that further research on this topic is needed to ensure that the violations captured in report-only mode match those that would occur in the enforcing mode. This is necessary for web developers to reliably determine which violations could still occur when switching to an enforcing policy.

In our case study, violations of COOP and RIP were captured for several weeks in a report-only mode. Surprisingly, in contrast to CSP violations, which are known to be very noisy [21], the unintended violations were fairly limited, and were mainly due to browser extensions or JavaScript libraries that would interact with `window.opener`. After filtering these undesirable interactions, ~2% of all endpoints (spread across 14% of services) were deemed to need an exempted policy of `unsafe-none`, as these relied on cross-site interactions. For another 14% of services, the COOP header on all endpoints needed to be relaxed to `same-origin-allow-popups` in order to allow these single-page applications to interact with windows opened as popups, e.g. for authentication. For RIP, ~3% of all endpoints (representing 8% of services) needed an exemption. Given that only a minority of services required an exemption of the policy, this is encouraging for future work that aims to enable policies by default. Nevertheless, the fraction of exempted endpoints remains non-negligible, and additional research is needed to determine how these endpoints can be further protected, possibly with a more fine-grained policy.

# 9 RELATED WORK

Most closely related to our work, is the concurrent work by Knittel et al. [18], who also introduce a formal model. Via a preprint provided to us by the authors, we detected many similarities in both models, and chose to adopt the same formulation for reasons of consistency. By introducing the concept of stateful "components", we are able to better capture the underlying characteristics that cause XS-Leaks, and use that to propose a methodology to identify new vulnerabilities, or ensure the absence thereof. Knittel et al. also perform an evaluation of different browsers and identify several novel XS-Leak attacks as a result of their analysis. Other works that provide an extensive overview of XS-Leaks, is the research by Sudhodanan et al. [49] and the XS-Leaks wiki [46]. These works mainly focus on the type of information that can be inferred, and introduce classes that are mainly descriptive of how an attack is performed. In our work, we introduce an extended model that abstracts away from the specific technicalities of the different attacks, allowing us to capture the intrinsic characteristics. Furthermore, we leverage this model to analyze XS-Leak defenses, categorizing these according to three main strategies.

For an overview of known XS-Leaks attacks, we refer to the classification in Table 1, with an accompanying summary of each attack in Appendix A.

In the context of violations to the same-origin policy, Schwenk et al. evaluated the implementation of the same-origin policy in different browsers and detect varying browser behavior because of the lack of a formal specification [40]. Other violations of the same-origin policy have been explored by Somé, who found that the permissions of browser extensions could be leveraged to leak data across site-boundaries [45]. Schuster et al. present an attack where contention on the network layer is abused to infer which videos are being played by the user based on the bursts on the network [39]. We believe that this technique could potentially also be used as an XS-Leak, to infer information about web pages. Finally, Franken et al. explored how same-origin policy violations could be abused in browser engines that are used to display e-books [8].

A related research topic that often leverages similar techniques as XS-Leak attacks, is history sniffing. In contrast to XS-Leaks, history leaks aim to infer the state that is retained in the browser by a prior visit of the user, and thus the state-introduction occurs inadvertently (and is not initiated by the attacker). The state-retrieval stage can be very similar to XS-Leak attacks. For instance, an adversary can

[4] https://distrinet.github.io/leakbuster/.

exploit a timing leak in CSS filters to extract whether a link is displayed in the `:visited` style [1, 43]. Similarly, in older browser versions, the attacker could simply read out the computed style to infer whether a certain URL was previously visited [3, 43]. More recently, Karami et al. [14] and Lee et al. [22] showed that the service worker cache could also be exploited to infer whether a user previously visited a specific website.

## 10 CONCLUSION

XS-Leaks are complex and can occur in many different components across the web ecosystem. In this paper we abstract away from the specific details of the techniques and introduce an extended formal model to create a better understanding of the intrinsic characteristics of the attacks. We show that a successful XS-Leak attack is performed in several steps: first, the adversary aims to transfer the web application state to a susceptible component, and subsequently retrieve the detectable state-differences. By considering these three aspects (stateful components, state-transfer, and state-retrieval) separately, it becomes clear that the strategies used to defend against these attacks can be linked directly to disabling one of the steps in the attack process. Based on the logic expressions that capture which XS-Leak issues are mitigated by a certain defense, we find that with the current set of defenses, not a single combination exists that can theoretically thwart all attacks. As some of the defenses add additional constraints (from the attacker's perspective), we believe that with strict policies, a complete solution against practical attacks is available. Finally, throughout this paper, we discuss several possible directions for future research that aim to improve the general state of security in the context of XS-Leaks, both in uncovering the complete threat surface, exploring defenses that are suitable for all intended web functionalities, and finding ways how these can be easily deployed at large scale.

## ACKNOWLEDGMENT

This research is partially funded by the Research Fund KU Leuven, and by the Flemish Research Programme Cybersecurity. This project was partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101019206 (TESTABLE).

## REFERENCES

[1] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 623–639.

[2] Lukasz Anforowicz. 2019. CORB vs side channels. https://docs.google.com/document/d/1kdqstoT1uH5JafGmRXrtKE4yVfjUVmXitjcvJ4tbBvM/edit.

[3] David Baron. 2002. :visited support allows queries into global history. https://bugzilla.mozilla.org/show_bug.cgi?id=147777.

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

[5] Chrome Platform Status. 2021. Feature: Cookies default to SameSite=Lax. https://www.chromestatus.com/feature/5088147346030592.

[6] Chromium bugs. 2018. Issue 843157: Security: leak cross-window request timing by exhausting connection pool. https://bugs.chromium.org/p/chromium/issues/detail?id=843157.

[7] Edward W Felten and Michael A Schneider. 2000. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*. 25–32.

[8] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. [n.d.]. Reading Between the Lines: An Extensive Evaluation of the Security and Privacy Implications of EPUB Reading Systems. In *2021 IEEE Symposium on Security and Privacy*. IEEE, 247–264.

[9] Brent Fulgham. 2018. Protecting Against HSTS Abuse. https://webkit.org/blog/8146/protecting-against-hsts-abuse/.

[10] Nethanel Gelernter and Amir Herzberg. 2015. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1394–1405.

[11] Luan Herrera. 2018. XS-Searching Google's bug tracker to find out vulnerable source code. https://medium.com/@luanherrera/xs-searching-googles-bug-tracker-to-find-out-vulnerable-source-code-50d8135b7549.

[12] Luan Herrera. 2021. XS-Leaks in redirect flows. https://docs.google.com/presentation/d/1rlnxXUYHY9CHgCMckZsCGH4VopLo4DYMvAcOltma0og/.

[13] Artur Janc, Krzysztof Kotowicz, Lukas Weichselbaum, and Roberto Clapis. 2020. Information Leaks via Safari's Intelligent Tracking Prevention. *arXiv preprint arXiv:2001.07421* (2020).

[14] Soroush Karami, Panagiotis Ilia, and Jason Polakis. 2021. Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage. In *Network and Distributed System Security Symposium (NDSS)*.

[15] Hyungsub Kim, Sangho Lee, and Jong Kim. 2016. Inferring browser activity and status through remote monitoring of storage usage. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 410–421.

[16] Eiji Kitamura. 2021. Load cross-origin resources without CORP headers using COEP: credentialless. https://developer.chrome.com/blog/coep-credentialless-origin-trial/.

[17] Amit Klein and Benny Pinkas. 2019. DNS Cache-Based User Tracking.. In *Network and Distributed System Security Symposium (NDSS)*.

[18] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. 2021. XSinator.com: From a Formal Model to the Automatic Evaluation of Cross-Site Leaks in Web Browsers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*.

[19] David Kohlbrenner and Hovav Shacham. 2017. On the effectiveness of mitigations against floating-point timing channels. In *26th USENIX Security Symposium*. 69–81.

[20] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. 2021. Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets. In *30th USENIX Security Symposium*.

[21] Stuart Larsen. 2020. Filtering the Crap, Content Security Policy (CSP) Reports. https://csper.io/blog/csp-report-filtering.

[22] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. 2018. Pride and prejudice in progressive web apps: Abusing native app-like features in web applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1731–1746.

[23] Sangho Lee, Hyungsub Kim, and Jong Kim. 2015. Identifying Cross-origin Resource Status Using Application Cache.. In *Network and Distributed System Security Symposium (NDSS)*.

[24] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. 2015. The unexpected dangers of dynamic JavaScript. In *24th USENIX Security Symposium*. 723–735.

[25] Ron Masas. 2018. Patched Facebook Vulnerability Could Have Exposed Private Information About You and Your Friends. https://www.imperva.com/blog/facebook-privacy-bug/.

[26] Ron Masas. 2019. A now-patched vulnerability in the web version of Facebook Messenger allowed any website to expose who you have been messaging with. https://www.imperva.com/blog/mapping-communication-between-facebook-accounts-using-a-browser-based-side-channel-attack/.

[27] Matt Menke. 2020. Storage Isolation Project. https://docs.google.com/document/d/1V8sFDCEYTXZmwKa_qWUfTVNAuBcPsu6FC0PhqMD6KKQ/.

[28] Rowan Merewood. 2019. SameSite cookies explained. https://web.dev/samesite-cookies-explained/.

[29] Mozilla Developer Network. 2021. Cross-Origin Opener Policy (COOP). https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy.

[30] Mozilla Developer Network. 2021. Cross-Origin Resource Policy (CORP). https://developer.mozilla.org/en-US/docs/Web/HTTP/Cross-Origin_Resource_Policy_(CORP).

[31] Mozilla Developer Network and Jesse Ruderman. 2020. Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[32] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. 2015. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1406–1418.

[33] Charlie Osborne. 2021. Playing Fetch: New XS-Leak exploits browser redirects to break user privacy. https://portswigger.net/daily-swig/playing-fetch-new-xs-leak-exploits-browser-redirects-to-break-user-privacy.

[34] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site isolation: process separation for web sites within the browser. In *28th USENIX Security Symposium*. 1661–1678.

[35] Renwa. 2020. Bypass SameSite Cookies Default to Lax and get CSRF. https://medium.com/@renwa/bypass-samesite-cookies-default-to-lax-and-get-csrf-343ba09b9f2b.

[36] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. 2019. Bakingtimer: privacy analysis of server-side request processing time. In *Proceedings of the 35th Annual Computer Security Applications Conference.* 478–488.

[37] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. 2020. Cookies from the Past: Timing Server-Side Request Processing Code for History Sniffing. *ACM Dgital Threats: Research and Practice Journal (DTRAP)* (2020).

[38] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. 2017. Extension breakdown: Security analysis of browsers extension resources control policies. In *26th USENIX Security Symposium.* 679–694.

[39] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. 2017. Beauty and the burst: Remote identification of encrypted video streams. In *26th USENIX Security Symposium.* 1357–1374.

[40] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. 2017. Same-origin policy: Evaluation in modern browsers. In *26th USENIX Security Symposium.* 713–727.

[41] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. 2021. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *30th USENIX Security Symposium.*

[42] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium.* 639–656.

[43] Michael Smith, Craig Disselkoen, Shravan Narayan, Fraser Brown, and Deian Stefan. 2018. Browser history re: visited. In *12th USENIX Workshop on Offensive Technologies (WOOT 18).*

[44] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. 2021. Persistent Tracking in Modern Browsers. (2021).

[45] Dolière Francis Somé. 2019. Empoweb: empowering web applications with browser extensions. In *2019 IEEE Symposium on Security and Privacy (SP).* IEEE, 227–245.

[46] Manuel Sousa, terjanq, Roberto Clapis, David Dworken, and NDevTK. 2020. XS-Leaks Wiki. https://xsleaks.dev/.

[47] Cristian-Alexandru Staicu and Michael Pradel. 2019. Leaky images: Targeted privacy attacks in the web. In *28th USENIX Security Symposium.* 923–939.

[48] Oleksii Starov and Nick Nikiforakis. 2017. Xhound: Quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy (SP).* IEEE, 941–956.

[49] Avinash Sudhodanan, Soheil Khodayari, and Juan Caballero. 2019. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. *arXiv preprint arXiv:1908.02204* (2019).

[50] Erik Sy, Christian Burkert, Hannes Federrath, and Mathias Fischer. 2018. Tracking users across the web via TLS session resumption. In *Proceedings of the 34th Annual Computer Security Applications Conference.* 289–299.

[51] terjanq. 2019. Mass XS-Search using Cache Attack. https://terjanq.github.io/Bug-Bounty/Google/cache-attack-06jd2d2mz2r0/index.html.

[52] Tom Van Goethem and Wouter Joosen. 2017. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In *11th USENIX Workshop on Offensive Technologies (WOOT 17).*

[53] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. 2015. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* 1382–1393.

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

[55] Tom Van Goethem, Mathy Vanhoef, Frank Piessens, and Wouter Joosen. 2016. Request and conquer: Exposing cross-origin resource size. In *25th USENIX Security Symposium.* 447–462.

[56] Pepe Vila and Boris Köpf. 2017. Loophole: Timing attacks on shared event loops in chrome. In *26th USENIX Security Symposium.* 849–864.

[57] WebKit. 2013. Optionally partition cache to prevent using cache for tracking. https://bugs.webkit.org/show_bug.cgi?id=110269.

[58] WHATWG. 2021. HTML Living Standard. 4.8.5 The iframe element. https://html.spec.whatwg.org/multipage/iframe-embed-object.html#attr-iframe-name.

[59] XS-Leaks Wiki. 2020. Cache Probing. https://xsleaks.dev/docs/attacks/cache-probing/.

[60] XS-Leaks Wiki. 2020. CORP Leaks. https://xsleaks.dev/docs/attacks/browser-features/corp/.

[61] XS-Leaks Wiki. 2020. Cross-Origin-Opener-Policy. https://xsleaks.dev/docs/defenses/opt-in/coop/.

[62] XS-Leaks Wiki. 2020. Error Events. https://xsleaks.dev/docs/attacks/error-events/.

[63] XS-Leaks Wiki. 2020. Frame Counting. https://xsleaks.dev/docs/attacks/frame-counting/.

[64] XS-Leaks Wiki. 2020. ID Attribute. https://xsleaks.dev/docs/attacks/id-attribute/.

[65] XS-Leaks Wiki. 2020. Navigations. https://xsleaks.dev/docs/attacks/navigations/.

[66] XS-Leaks Wiki. 2020. postMessage Broadcasts. https://xsleaks.dev/docs/attacks/postmessage-broadcasts/.

[67] XS-Leaks Wiki. 2020. Resource Isolation Policy. https://xsleaks.dev/docs/defenses/isolation-policies/resource-isolation/.

[68] XS-Leaks Wiki. 2020. X-Frame-Options and Status Type Detector. https://xsleaks.github.io/xsleaks/examples/x-frame/index.html.

# A SUMMARY OF XS-LEAKS

**(in)valid script/image/...** When including a resource as a script or an image, this will trigger an error event in case the content of the resource does not match the expected content. For instance, including an HTML resource in an `<img>` element will trigger an `error` event, whereas a valid image resource will result in a `load` event.

**client redirect (load event)** When a document resource is included as an iframe, the `load` event will be fired every time a document is loaded. Therefore, when a client-side redirect occurs after the document has loaded, the `load` event will be fired multiple times.

**server redirect (max redirect count)** According to the Fetch specification, when twenty server-side redirects occur, a network error will be returned. As such, to determine whether a specific resource causes a redirect, the attacker can first make a request to their own server and redirect 19 times, after which a redirect to the target resource occurs. If this resource redirects, a network error can be observed, otherwise the resource will be loaded. Note that this method can also be used to determine the exact number of redirects that occur.

**server redirect (CSP violation)** By defining a (restrictive) Content Security Policy on their page, the attacker can determine from which hosts resources are allowed to be loaded. In case a resource from a different host is loaded, this will result in a violation of the CSP, which can be observed by listening for a `securitypolicyviolation` event. As such, this allows an attacker do determine whether a resource redirects to a host that is not defined in the allowed sources according to the attacker's defined CSP policy.

**server redirect/status (AppCache)** Entries in the AppCache manifest that redirect or have a non-200 status code will cause an `error` event on the `applicationCache` object; in case no redirect occurs, the `cached` event will be fired. This allows an attacker to determine whether a certain resource will cause a redirect.

**server redirect (Fetch manual)** When the `redirect` option of in the `fetch()` call is set to `manual`, the returned `Promise` will resolve in case a redirect happens, otherwise the promise will be rejected. The attacker can determine whether a redirect occurred by interpreting the resolution of the promise.

**element id focus** When a document resource is loaded in an iframe where the URL fragment is set to the ID of a DOM element on the page, the browser will focus this iframe, causing the embedding (attacker) page to lose focus, which can be observed by listening for the `blur` event.

**overly broad postMessage** To allow for cross-site communication, the postMessage API can be used. For instance, an embedded page can send a message to the top-most page by using `top.postMessage(msg, origin)`. The second argument of this function defines the origin for which the message is intended. If this is set to the wildcard `*`, the message will be sent regardless of the origin of the attacker.

**detect CORB'ed JSON responses** When a valid JSON document is included as in `<script>` element, it will cause a `SyntaxError`, which can be observed by listening to the `error` event. However, if the response is blocked by CORB, the body will be emptied, and no syntax error will occur.

**detect CORP header** Similar to the CORB'ed responses, if a CORP header is present and it is not set to `cross-origin`, it will be blocked. Alternatively, to detect it when it is set to `cross-origin`, the attacker can set the COEP header on their page to `require-corp`, which will prevent the resource from loading if the CORP header is not present.

**detect COOP header** When a page sets the COOP header, it will prevent other pages from retaining a reference to it. Hence, to check whether the COOP header is set, the attacker could open the resource in a new window, and then verify whether the reference to this window is still available.

**detect XFO (<object>)** When a document resource is included in an `<object>` element, and it sets the `X-Frame-Options` header to `DENY`, no `load` event will be fired on the object element. Without the XFO header, the event will be fired.

**detect XFO (Resource Timing)** Typically, when a resource is loaded, a new `PerformanceResourceTiming` entry is created. However, in Chromium-based browsers, this does not happen when an XFO-enabled document resource is loaded in an iframe.

**response size estimate (parsing)** The time it takes to parse a resource as an audio or video element depends on the size of the resource. Hence, by measuring the time (repeatedly), an estimation of the response size can be made.

**response size estimate (Cache API)** The time it takes to add or remove a response to the cache (using the Cache API), depends on its size. By repeated measurements, an estimate of this size can be obtained.

**response size (Quota API)** To prevent abuse, the available quota that each website is provided with, is limited. The available quota can be retrieved from by calling the API. A website can observe the currently allotted quota, force the target resource to be cached (using the Cache API), and observe the quota again. The size of the resource will be the difference between the two values.

**response size (global quota eviction)** Next to a per-site quota, there also exists a global storage quota. When this limit is reached, the least-recently used site will be evicted. If an attacker can force one of their sites to be evicted (of which they know the size), they can retrieve the size of the response by adding the target resource to the cache and then fill up the remainder of the global quota byte by byte, until another eviction occurs.

**cross-site pixel stealing** When a page embeds a document resource in a frame, it can perform certain manipulation on what is visually displayed. For instance, SVG filters and CSS rules can be applied. In case the execution of applying these filters on untrusted data, i.e., the pixels of a cross-site page, is not performed in constant-time, the timing information can be abused to extract text and other visuals from the targeted page.

**Performance API entries** Depending on the response that is returned, an entry in the Performance API may or may not be made. For instance, in case of an error (status code 500), or an empty response, no entry will be created. In case of a redirect, the timing values may also differ. Furthermore, in case the target is included as an iframe and is blocked by the XSS auditor in Safari, no performance entry will be created.

**CORS error leaks redirect URL** In Safari browsers, the URL to which a page redirects can be read out when the CORS request results in an error.

**SRI leaks response size** In Safari the message embedded in the error event when a mismatch for SRI is detected leaks the size of the response.

**appearance of download bar** When a resource that is opened in a new window is served with the `Content-Disposition: attachment` header, it will be downloaded by the browser. In Chromium-based browsers, this will cause the download bar to appear, causing the height of the window to be reduced. This download bar will remain there until it is closed by the user.

**cache probing** When a resource is added to the HTTP cache, it will be loaded much faster in comparison to retrieving the resource over the network. When a specific document resource is loaded in an iframe or window, this may cause specific (other) resources to be added to the cache. The attacker can then use a timing attack to determine if any resource was cached. Cached resources will remain in the cache until they are invalidated, hence making this technique non-idempotent. Nevertheless, there exist various techniques that allow an adversary to remove specific entries from the cache.

**Loophole (event loop timing)** Browsers make use of event loops to handle the different events that happen. When an event loop is shared by different cross-site pages (in particular the attacker page and its target page), the attacker page can leak the time that the other page requires to handle events by continuously triggering events and observing the delay between them (a larger delay indicates that an event had to be handled for the other page).

**Safari ITP leaks** The Intelligent Tracking Prevention mechanism in Safari maintains a list of domains to which several cross-site requests are made. Whenever a cross-site request is made to a particular site, that domain is given a strike, and after sufficient strikes from a sufficient amount of top-level sites, the domain will be added to the ITP list. When cross-site requests to domains on the ITP list are made, the `Referer` header and any cookies will be stripped. To perform an XS-Leak attack, the attacker can trigger the loading of the target document resource, and afterwards use various side-channels to infer whether this caused any particular domain to be added to the ITP list. Because the ITP list is only cleared when the browsing history is cleared, the attack is non-idempotent. Note that this issue has been mitigated in Safari.

**detecting connections (pool limit)** The global number of concurrent connections is limited, and when it is reached, the least-recently used connection will be terminated. An attacker can thus determine how many new connections the rendering of the target document resource caused by first establishing the maximum number of concurrent connections to their server, and then detecting how many of those were closed by the user.

**WebSocket global limit** Browsers set a global limit on the total number of WebSocket connections (Chrome: 256, Firefox: 200), when this limit is reached, no new connections can be established. This can be used to leak whether the target page initiated a new WebSocket connection.

**Payment API global limit** Only a single UI pop-up for the Payment API can be shown at the time. By probing whether it is possible to open this UI, the attacker can determine whether the targeted page opened it.

**CPU cache attacks** When a document resource is rendered in the browser, this typically results in various executions on the CPU, which in turn results in various changes at the microarchitectural level. Prior work has (repeatedly) shown that the trace of changes made to the last-level cache can identify which websites are being visited, i.e., in a website fingerprinting attack. As this technique allows distinguishing two different execution traces of rendering a document resource, it could in theory also be leveraged to launch XS-Leak attacks.

**response timing (size)** On the downstream connection between the server and the client, the time it takes the server to send the entire response to the client will depend on the size of the response. By measuring this time, an adversary can distinguish small and large responses.

**frame counting** When the attacker has a reference to a window in which the target resource was loaded, they can retrieve the number of frames that are loaded in this document by accessing the `frames.length` property. It is also possible to determine the number of frames in the entire frame-tree, for instance by checking `frames[0].length` for the number of iframes embedded in the first frame.

**no navigation due to download** If a resource with the `Content-Disposition` header set to `attachment` is loaded in an iframe or window, this will cause the resource to be downloaded and no navigation will occur in the iframe or window. As a result, the document's origin remains `about:blank`. In this case, the attacker can still access SOP-protected attributes, such as `window.origin`, thereby allowing them to tell whether or not a download was triggered

**window.name leak** By setting the `window.name` property, a name is given to the current browsing context. When the document within this browsing context is navigated to a different page, this name is retained, and thus becomes available across origins.

**client redirect (History API)** The History API keeps track of which navigations occurred, which is accessible through the `History.length` property. An adversary can navigate a separate window to the target resource and wait for it to finish loading, and subsequently navigate that window to an attacker-controlled web page. By accessing the `history.length` property, the attacker will be able to infer how many client-side redirects or calls to the `history.pushState()` API were made by the target resource. Note that server-side redirects using the `Location` header are not counted.

**response timing (execution time)** The time it takes to generate a response might depend on the state of the user; for instance if the user is able to access a particular resource such as a private group on a social network site, the server might take additional steps to retrieve these, resulting in a timing difference. The computation time can be observed with a typical cross-site timing or a timeless timing attack. An attacker could try to inflate the measured timing difference to reduce the measurement noise introduced by jitter.

**XSSI** Some websites may dynamically generate JavaScript that contains potentially sensitive user information. An attacker can embed this JavaScript in their page, and then read out the sensitive data, either by accessing a global property, overwriting a prototype, or redefining global APIs.

**request count: rate limit** Some web servers impose a maximum number of requests that can made to certain endpoints within a predefined timeslot. Once this limit is reached, an error page will be returned. The attacker can detect after how many requests this error page is returned to infer how many requests were previously made as the result of rendering a target web page.

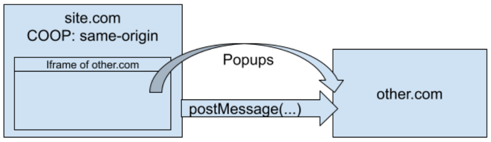
**request count: max requests per connection** Web servers limit the number of requests that can be sent over a single connection; after the maximum has been reached, the connection will be closed. An attacker could determine how many requests were needed to close the connection to infer how many requests were previously sent.

# B COOP REPORTING GAPS

The experience of deploying COOP at a large technology company discussed in Section 8 gave insight into 3 gaps in COOP reporting where no reports would be triggered, but enforcing COOP could lead to a breakage. These three scenarios are included below so as to assist other sites in deploying COOP.

## B.1 Iframe Window Interactions

If a page enables COOP, all iframes on that page also get COOP enforced. This means that if a page enables COOP and it embeds a page that needs to open popups and interact with them, it may break:

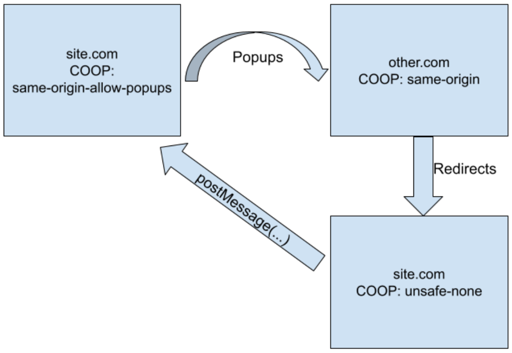


## B.2 Redirects

If:

(1) A page enables COOP enforcement
(2) That page redirects to a page without COOP enforcement
(3) Some other service `window.open`s the page with COOP enforcement
(4) That service then tries to use the window reference for cross-origin communication

Then things can break without triggering a COOP report:

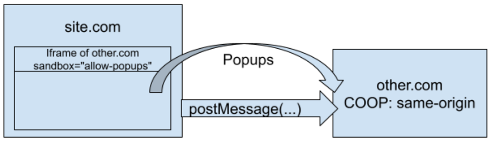


## B.3 Iframe Sandbox

If:

(1) A page contains an iframe with `sandbox="allow-popups"` but without `allow-popups-to-escape-sandbox`
(2) That iframe opens a popup to `example.com/endpoint`
(3) `example.com/endpoint` enforces COOP

Then the opened popup will show a network error page with the error `CoopSandboxedIFrameCannotNavigateToCoopPage`:


site.com
Iframe of other.com
sandbox="allow-popups"
Popups
postMessage(...)
other.com
COOP: same-origin