Understanding Cross-site Leaks and Defenses

Tom Van Goethem
imec-DistriNet, KU Leuven
tom.vangoethem@kuleuven.be

Gertjan Franken
imec-DistriNet, KU Leuven
gertjan.franken@kuleuven.be

Iskander Sanchez-Rola
Norton Research Group
Iskander.Sanchez@nortonlifelock.com

David Dworken
Google
ddworken@google.com

Wouter Joosen
imec-DistriNet, KU Leuven
wouter.joosen@kuleuven.be

ABSTRACT

A web visit typically consists of the user's browser rendering generated HTML, CSS and JS content that is tailored to the user. This dynamic generation of responses based on the currently authenticated user, whose authentication credentials are automatically included in all requests including cross-site requests, have lead 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 reveals potentially sensitive information about the user. As this class of attacks has been the subject of a lot of recent research, and it affects many different components of the web platform, the causes and underlying techniques are not always very well understood. In this paper we define an abstraction of the XS-Leaks attacks, and introduce a model that we use to reason about the cause of different leaks and how the various defense mechanisms aim to defend against them. As the attacks are multifaceted and complex, multiple defenses need to be enabled to adequately thwart XS-Leak attacks. To facilitate deployment of these defenses, we introduce Leakbuster, a dynamic web application that provides web developers with suggestions to improve their website's security.

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 [50, 56], determine what their interests are [30], or infer who they were messaging with [31]. Through similar attacks, search functionality has been shown to leak information about undisclosed vulnerabilities [17, 57] or credit card details [16].

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-Leaks 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. To date, most vulnerabilities have been detected by manually evaluating specific browser mechanisms, such as the Application Cache [27] or the Quota API [58]. Because of the very varied nature of XS-Leaks, also a variety of defenses are needed to thwart them, and in fact, there is no single defense that is sufficient to prevent all XS-Leaks. This makes it very difficult for web developers to protect their users.

In this paper we aim to improve the understanding of XS-Leaks, and more specifically study the similarities between the different attacks, and the cause of their existence. To this end, we introduce a model of XS-Leaks and show how in every component that is involved with handling a response, a state-change may occur that depends on the response. For example, when the rendering of a document triggers certain resources to be cached, this alters the state of the HTTP cache. In the second stage of an XS-Leak attack, the state-change that occurred in a particular component is retrieved. In the prior example, this can be done by performing a timing attack to infer whether the resource was loaded from the cache. Based on this model, we also create a taxonomy that can be used to classify XS-Leaks and provide a clear overview of how the different attacks relate to each other.

Furthermore, we analyze the currently supported defenses against XS-Leaks and map these back to the model that we created, to better understand their intent. We identify three different strategies for defenses, namely trying to prevent state-changes from occurring in certain components, accepting that state-changes will occur and isolating these such that they can not be observed cross-site, and preventing the responses from being based on the state of the user by blocking illicit requests or removing authentication details. Based on this analysis, we determine which category of attacks are blocked by certain defenses and recommend the minimally required set of defenses that are needed to thwart all attacks. Finally, we introduce Leakbuster, which is an interactive web application that facilitates deploying XS-Leak defenses, and is based on our insights of a real-world case study where defenses were deployed on wide range of popular services.

In summary, we make the following contributions:

- We introduce a model for XS-Leaks showing how different components that are involved with handling responses may be abused to introduce state-changes based on the response.
- Based on this model we create a taxonomy for XS-Leaks and use it to classify known attacks that were discovered through an extensive literature review.
- We analyze the general strategies that are employed by defense mechanisms and find that a combination of isolation defenses and defenses that prevent illicit authenticated requests are needed to thwart all attacks.
- 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. As a result, the site defined in the URL was mainly there to easily access the web pages, and was not related to any security properties, as cross-site attacks had no practical impact. 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 unauthorized 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 [36] which 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. 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 TAXONOMY FOR XS-LEAKS

XS-Leaks have been known, although not always explicitly under that name, for well over a decade. Prior work has mainly focused on describing newly discovered techniques or discuss the consequences of known XS-Leaks. In this paper, we take a step back and focus on the, sometimes non-obvious, commonalities that exist between different XS-Leak attacks. To this end, we first propose a model that explains the source of certain leaks and use this model as the basis for creating a taxonomy of XS-Leaks.

3.1 Running example

As a running example of XS-Leaks, we introduce a very straightforward search application, of which the Jinja template is shown in Listing 1. The underlying application authenticates the user based on the cookie that is 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

```
<!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.

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 [54].

Listing 2 shows the JavaScript code of what an adversary would run on their malicious site to determine whether any results were shown for a specific (attacker-supplied) keyword. The attack 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. This would indicate that loading the target page caused this, which can only occur when there was at least one result for the query. In order to check multiple queries, the attacker would need to reset the state of the cache, and thus use any of the known techniques [65] to remove the icon from the cache. Note that as a result of the recent network isolation defenses, this specific attack is prevented.

The example application is also be vulnerable to other XS-Leak attacks. For instance, detecting the size of the response gives leaks whether there were any, and possibly how many, results [4, 56, 58]. 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 [11]. 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 [16, 57].

3.2 Definition and threat model

For the attacks 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[1]. These cross-site resources are dynamically generated based on the identity of the victim, which is typically inferred from the included cookie, the requested endpoint, and (attacker-provided) query parameters. For simplicity, we consider that the returned response will be one of two distinctive options, depending on the state of the user, where one is considered the baseline, and the other has at least one differentiating aspect. Furthermore, the methods used by the adversary to distinguish the two possible responses can either exploit intentional cross-site leaks, e.g. the dimensions of images can be directly observed, or through a side-channel leak, e.g. the time required to fetch a resource.

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 regularly 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, 23, 46]). 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 A model for XS-Leaks

In order to have a better understanding of XS-Leaks, we first introduce a general model, where we abstracted specific aspects of certain attacks to focus on the commonalities of XS-Leaks. Our proposed model, which is depicted in Figure 1, is based on an analysis of all XS-Leak techniques that are known to date, and may be useful in directing future research on detecting novel attacks and reasoning about defenses. In this analysis, we determined the component that is responsible for the leak, and evaluated the reason why the leak came into existence, and how the sensitive data was extracted. The model reflects the threat model, i.e., we consider that the attack takes place when the user visits an attacker-controlled website (red browser tab in the model). This website can include a target resource that returns a different response based on the state of the user (depicted as the R/R' documents), either directly, in an iframe, or in a separate window (the inclusion is indicated by the black arrows).

We make the observation that whenever a resource is included, this introduces a chain of events that affects different components. Several of the actions that are taken during the fetching and rendering process introduce a side-effect that can be ephemeral or persistent. As such, every component that is involved in the resource inclusion process could potentially introduce or alter a certain state, which can be in the form of firing an event, consuming a shared

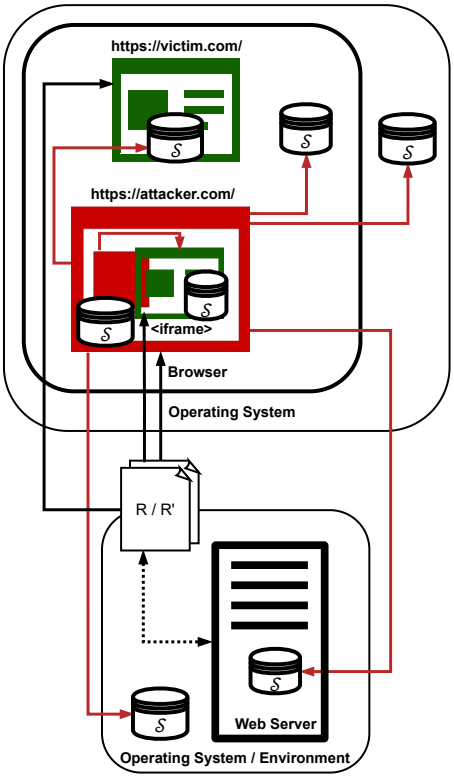


Figure 1: Model of XS-Leaks, indicating how the differentiating resource (R or R') can be included (black arrow), potentially leaving behind a state-change (S) in one of the components, that may later be retrieved from the attacking page (red arrow).

resource or changing the value of a global parameter. The state-changes that can occur in the various components are indicated with a database icon marked S. As the topology of the components is hierarchical, i.e., the client operating system embeds the browser, which in turn embeds the different tabs, whenever the resource traverses a component boundary (i.e. when a black line originating from the resource enters a component), a state-change may be introduces in this component.

When the state-change is related to the differentiating characteristic of the resource, an attack is successful if this state-change can be observed by the adversary (indicated by the red lines in the model). As such, we find that all XS-Leak attacks consist of two stages:

(1) the **state-introduction** stage that is forced by the attacker by including a specific resource in a particular way, and causes a state-change in one or more components
(2) the **state-retrieval** stage where the adversary aims to detect the state-change that occurred in the first stage, either directly, e.g. by reading the value of a certain property, or through side-channel information

Not only the occurrence or absence of a state-change may leak information, but also the time at which it occurs. For instance, in the example from Listing 2, the `Promise` returned by `fetch()` will always resolve, but the time it takes reveals whether the resource was served from the cache or accessed over the network.

[1]This definition is in line with that of prior work (XS-Leaks Wiki [49] and COSI attacks [52]), but makes it explicitly clear that the attacker aims to infer the state that the user has *with the targeted website*.

3.4 Taxonomy

The taxonomy that we propose is based on the two stages of state-changes that occur in a XS-Leak attack, and aims to capture characteristic aspects of the different attacks. As such, we consider the following classification properties:

Component retaining state-change When a request is made, a number of actions are taken by the different components that are involved with it, from the initialization of the request to the parsing and rendering. A state-change can occur in any of the involved components. This property indicates the component in which the state-change that discloses sensitive information about the resource occurs and is potentially kept. In our running example of the cache-probing attack, the state-change occurs in the *browser*, as this is where the HTTP cache is kept. Other components that are present on the client side are the *attacker page* and the *victim iframe*, in case a resource was included by the attacker in the iframe. An attacker can also open a resource in another window, and thus the state-change could occur in this *victim window*. Finally, state-changes may also occur in the underlying *operating system*. We also consider components at the server-side in which state-changes may occur; for instance in the *web server*. For simplicity, here we consider any system involved with the handling and processing of requests, e.g. nginx server, PHP code, database server, etc. The *operating system or (virtual) environment* in which the web server is run can also be the origin of state-changes.

For simplicity, we abstract the different sub-components within the browser or operating system. For example, although the HTTP cache and connection pool are two different sub-components, we join them in the same general component (browser).

Resource inclusion method In order to trigger the state-changes in the different components, the adversary can include the target resource in various ways, for example *directly, using a specific API*, for instance using an `<img>` to determine whether the target resource is a valid image. In certain cases it does not matter which method was used to include a resource and *any API* from the attacker's page can be used. The two remaining inclusion methods are *iframe*, where the resource will be loaded in a frame embedded in the attacker's page, and an *other window*, in a separate tab. This can be a newly opened tab, using the `window.open()` method, or the tab that opened the attacker's page, which can be accessed through `window.opener`.

State-changed aspect Also in the context of the state-introduction stage, we consider the particular aspect that changed during the state-change. We consider three different aspects: *change of a property* (e.g. the height of a window that is decreased because a download bar is shown to the user), *an event that is fired* (e.g. an `error` event on an `<img>` element), and the *consumption of a limited resource* (e.g. a connection that is used from the limited connection pool).

State observation method As part of the state-retrieval stage, we consider three different methods through which a previously introduced state-change can be observed: *observation of an event* (e.g. using an event listener), *reading a property* (e.g. the height for the window), and *probing a property*. For instance, the cache-probing attack requires active probing of the property, i.e. using a timing attack to detect whether the resource was loaded from the cache (e.g. as in Listing 2).

Affected DOM API If applicable, we record which DOM API or browser mechanism was responsible for causing the state-change. For example the History API is responsible for keeping track of the navigations that occurred in a window.

Information in timing For some XS-Leaks certain state-changes will always occur; for instance a response for the request will always be generated, but the aspect that leaks information about the state of the user with the targeted website, is the time that the state-change occurs. We indicate XS-Leaks attacks that can only be successful when the timing of the state-change can be dynamically observed by the attacker. We note that in real-world attacks, the timing and order of state-changes may leak additional information, e.g. by constantly monitoring the number of iframes embedded in the target document [31], but here we only record whether timing information is strictly necessary to launch an attack.

Intended cross-site information Certain exceptions to the same-origin policy exists, and some information is intended to be accessed cross-site. For example, the dimensions of an image are provided to the website that includes it.

Idempotency In certain cases the state-change that occurs may be permanent and can not be reverted to the previous state, thus preventing an attacker from making multiple observations. Such cases are thus non-idempotent. An example is the download bar that appears when the target resource is downloaded: as the user needs to actively close the download bar, the attack can only be used to detect the download navigation of a single resource.

Differentiating aspect Depending on the state of the user, a response will be generated that differs in a particular aspect. Typically, each XS-Leak technique can be used to infer one particular aspect. This could be a response *header* (e.g. `Content-Disposition`), the *contents* of the response body (e.g. the number of iframe elements it contains), *metadata* of the response (e.g. the size of the response body), or *generation characteristics* (e.g. the time required to process a request).

4 CLASSIFYING XS-LEAKS

In this section we apply our proposed taxonomy to known XS-Leaks attacks to better understand the underlying causes, which in turn is useful to understand how the defenses aim to tackle these (discussed in Section 5), and explore opportunities for future research. We aggregate known XS-Leaks by performing a systematic literature review: we consider all papers that were published in the last five years in a well-known security conference (Security & Privacy, CCS, NDSS, USENIX Security, ACSAC, AsiaCCS), or an attack- or web-focused workshop (WOOT, W2SP, and 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-origin 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 32 distinct XS-Leak techniques.

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 [16, §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 [16, §4], which is similar to what Van Goethem et al. measure in their timeless timing attacks [57].

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. It can be seen that there is a lot of diversity in the attacks: state-changes that occur in all components are responsible for at least one attack. The majority of the leaks are caused by state-changes in the attacker page. We believe the reason for this is twofold: 1) 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, and 2) the state-changes that occur in the attacker page are mostly directly observable and thus easier to detect, and in many 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. As such, we consider it an interesting direction for future research to systematically detect all different subcomponents in which state-changes may occur.

We found six attacks where the cross-site information is deliberately made available across sites. These can be roughly divided into three different categories. First, in some cases the correct functioning may require the information to be accessible across sites, or its functionality would make it virtually impossible to hide this leak. An example of this category is the `load` or `error` event on images or scripts, where the web developer needs to know whether an image was correctly loaded. A second category are the intended cross-site information transfers that exist out of historical reasons. For instance, many years ago, when web developers used an extensive number of frames to create their web pages, having access to the number of frames in a page was essential. However, as web pages start using them, it becomes more difficult to deprecate such features in order to remove them from the web platform [63]. The third category consists of mechanisms where the targeted web page intentionally discloses information to their (cross-site) embedder. In our literature review, we only found evidence of the `postMessage()` API, but also mechanisms that can relax the same-origin policy could be included, for instance CORS and Resource Timing API, when their policy is set too broad.

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 and the response that is generated, 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 DEFENSES BASED ON XS-LEAKS MODEL

In recent years, there has been an explosion of different defenses that aim to thwart XS-Leaks. In this section we analyze the different strategies that were taken by current defenses. We consider four classes of defenses, namely 1) those that prevent state-changes from occurring in the browser, 2) defenses that aim to isolate sites from each other, 3) defenses that aim to prevent the differentiation of the response and thus prevent user state from being embedded in the response, and finally 4) defenses that target specific issues.

5.1 Preventing state-changing effects

Whenever a resource is embedded, state-changes could be introduced in every component that this resource passes through, ranging from the attacker page to the client's operating system. By preventing the resource to be fully loaded or stopping its rendering, certain state-changes may not occur, and thus the components can be safeguarded from XS-Leaks. In the running example introduced in Section 3.1 the target resource is included as an iframe, and the parsing and rendering of this resource will trigger the icon image to be loaded and eventually added to the HTTP cache. By preventing the resource from being loaded in an iframe, this particular XS-Leak will thus be mitigated (note: an attacker can still trigger the rendering of the resource by opening it in another window). There are two main methods that can be used to enforce a **framing protection**, namely by setting the `X-Frame-Options` header or by using `frame-ancestors` directive of CSP. As soon as the browser encounters any of these two policies, it will stop loading the resource and prevent if from rendering.This thwarts attacks that require the target resource to be included in an iframe, and based on our taxonomy, we can make the following expression that captures all attacks that are mitigated by this defense:

Defense: framing protection
XS-Leaks thwarted: *inclusion_method = {iframe}*

By default, a website's resources can be included by any other (cross-site) web page, in any context (i.e., as a script, image, video, ...). There are two defenses that can be used to force the browser to block loading these resources. One can be set by the server through the `Cross-Origin-Resource-Policy` (**CORP**) header [35]. When the header's value is set to `same-site` or `same-origin`, the resource cannot be read by a cross-site web page, and the browser will even block loading the resource.A related mechanism is Cross-Origin Read Blocking (**CORB**), which is enabled by default in Chromium-based browsers [8]. Implementation in other browsers is expected to follow as a specification is being created for a similar mechanism [59]. Originally designed as a defense against Spectre attacks, CORB will block "sensitive" cross-site responses from being passed to the renderer process (thus making it inaccessible to Spectre attacks). Responses are considered sensitive based on their content type (i.e., JSON, HTML, and XML).Because blocking only occurs after the redirection chain, leaks that aim to infer this information are still feasible. Furthermore, CORB and CORP only affect resources that are included directly on the attacker's page, and not via an iframe or other window.

Table 1: Classification of known XS-Leak attacks

Attack	Component retaining state	Inclusion method	State-changed aspect	State observation method	Affected DOM API	Information in timing?	Intended XS information?	Idempotent?	Differentiating aspect	Reference(s)
(in)valid script/image/...	attacker page	direct: specfic API	event fired	observation of an event	img/script/...	○	●	●	content / headers	[68], [50]
client redirect (load event)	attacker page	iframe	event fired	observation of an event	iframe.onload	○	●	●	content	[71]
server redirect (max redirect count)	attacker page	direct: any API	consumption of limited resource	observation of an event	Fetch	○	○	●	headers	[18], [38]
server redirect (CSP violation)	attacker page	direct: any API	event fired	observation of an event	CSP	○	○	●	headers	[71]
server redirect/status (AppCache)	attacker page	direct: any API	event fired	observation of an event	AppCache	○	○	●	headers	[27], [18]
server redirect (Fetch manual)	attacker page	direct: specfic API	event fired	observation of an event	Fetch	○	○	●	headers	[18]
element id focus	attacker page	iframe	event fired	observation of an event	window.onblur	○	○	●	content	[70]
overly broad postMessage	attacker page	iframe / other window	event fired	observation of an event	postMessage	○	●	●	content	[72]
detect CORB'ed JSON responses	attacker page	direct: specfic API	event fired	observation of an event	CORB	○	○	●	headers / content	[2]
detect CORP header	attacker page	direct: any API	event fired	observation of an event	CORP	○	○	●	headers	[66]
detect COOP header	attacker page	other window	change of a property	reading a property	COOP	○	○	●	headers	[67]
detect XFO (<object>)	attacker page	direct: specfic API	event fired	observation of an event	XFO	○	○	●	headers	[52]
detect XFO (Resource Timing)	attacker page	direct: any API	change of a property	reading a property	Resource Timing API	○	○	●	headers	[74]
response size estimate (parsing)	attacker page	direct: specfic API	event fired	observation of an event	audio/video	●	○	●	metadata	[56]
response size estimate (Cache API)	attacker page	direct: specfic API	event fired	observation of an event	Cache API	●	○	●	metadata	[56]
response size (Quota API)	attacker page	direct: specfic API	change of a property	reading a property	Quota API	○	○	●	metadata	[58, §3.4.1], [21]
response size (global quota eviction)	attacker page	direct: specfic API	consumption of limited resource	probing a property	Storage API	○	○	●	metadata	[58, §3.4.2]
cross-site pixel stealing	attacker page	iframe	event fired	observation of an event	–	●	○	●	content	[1], [23]
response status (AppCache)	attacker page	direct: any API	event fired	observation of an event	AppCache	○	○	●	headers	[27]
appearance of download bar	browser	other window	change of a property	reading a property	window.height	○	○	○	headers	[71]
cache probing	browser	iframe / other window	change of a property	probing a property	HTTP cache	○	○	○†	content	[65], [12]
Loophole (event loop timing)	browser	iframe / other window	consumption of limited resource	probing a property	–	●	○	●	content	[60]
Safari ITP leaks	browser	other window	change of a property	probing a property	Intelligent Tracking Prevention	○	○	○	content	[19]
detecting connections (pool limit)	browser	iframe / other window	consumption of limited resource	probing a property	–	○	○	●	content	[11]
CPU cache attacks	client-side OS	iframe / other window	change of a property	probing a property	–	●	○	●	content	[37], [45], [44]
response timing (size)	server-side OS	direct: any API	event fired	observation of an event	–	●	○	●	metadata	[4], [16, §3.1]
frame counting	victim iframe / victim window	iframe / other window	change of a property	reading a property	frames.length	○	●	●	content	[69]
no navigation due to download	victim iframe / victim window	iframe / other window	change of a property	reading a property	–	○	○	●	headers	[71]
window.name leak	victim iframe / victim window	iframe / other window	change of a property	reading a property	window.name	○	●	●	content	[64]
client redirect (History API)	victim window	other window	change of a property	reading a property	History API	○	●	●	content	[71]
response timing (execution time)	web server	direct: any API	event fired	observation of an event	–	●	○	●	generation process	[16, §4], [57], [40]
XSSI	web server	direct: specfic API	change of a property	reading a property	–	○	○	●	content	[28]

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

Defense: CORB/CORP
XS-Leaks thwarted:
differentiating_aspect ∩ *{content, metadata}* ≠ ∅ &&
component = {attacker_page} && inclusion_method = {direct}

5.2 Isolation defenses

Another strategy that can be used to defend against XS-Leaks, is to allow the state-changes to occur, but either ensure that this happens in an isolated environment, or prevent access to the state, also by means of isolation. For example, as the result of the recently implemented **network isolation defenses**, Chrome and Firefox partition the HTTP cache as well as several other network-level properties based on the "site" of the top-level document, and, depending on the implementation, also based on the site of the embedding document (which could be an iframe). Safari has adopted partitioning of the HTTP Cache in 2013 [61], but did not isolate other network properties. If an attacker would use a separate window to perform the XS-Leak attack against our example application from Section 3.1 (e.g. because XFO prevent the attack via an iframe), the state-change would still occur, i.e., the icon image would be cached, but could not be directly observed by the attacker. More precisely, because the attacker does not share the same HTTP cache as the victim, changes made in the victim's cache can not be detected.

Defense: network isolation
XS-Leaks thwarted:
component = {browser} && inclusion_method ≠ {direct} &&
affected_API = {HTTP_cache[2]*}*

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 [34, 67]. At the time of this writing, the header is supported by all major browsers except for Safari [6]. In essence, when the policy's header value is set to `same-origin`, it ensures cross-origin pages do not have a reference to it. For example, this prevents the attack that counts the number of frames in a page from accessing the `window.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. An interesting consequence of this defense is that because the attacker loses a reference to a newly opened window, this window cannot be closed, and thus attacks that require opening the target resource in a new window will have to open many new windows, and will thus not be stealthy.

Defense: COOP
XS-Leaks thwarted: *inclusion_method = {other_window}* ||
component = {victim_window}

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 [39]. 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. However, because resources *can* be included in a cross-site context by default, this defense mechanism should be considered a primitive, and needs to be combined with other defense, such as CORB/CORP, Fetch Metadata, or SameSite cookies. Nevertheless, as a side-effect of this defense, event loops are now executed per-site, and thus the Loophole attack that leverages the event loop of the renderer process has been mitigated.

Defense: site isolation
XS-Leaks thwarted: *attack = Loophole*
(only renderer event loop attack)

5.3 Stateless responses

A third defense strategy that can be applied to counter XS-Leaks is to prevent responses from containing any sensitive user-state. This type of defense is related to those aimed to thwart Cross-Site Request Forgery (CSRF) attacks as both need to validate the authenticity of requests and only allow those that are intended. In fact, defenses against CSRF can also be effective to counter certain XS-Leaks. For instance, requiring a unique, unguessable token (**CSRF token**) in the request ensures that only requests that were triggered by the web application itself are permitted. When a static response is returned for illicit requests that do not contain a valid token, the attacker cannot learn any user-specific information. However, as direct navigation requests to web pages should be allowed, and can by definition not contain a token, these pages can not be protected by a CSRF token (it is for this reason that CSRF tokens are typically only applied to POST requests). Moreover, if a web page would include subresources or make API calls that are protected by a CSRF token, these would still be loaded (and possibly cause state-changes) when the page is included as an iframe or window.

Defense: CSRF token
XS-Leaks thwarted: *inclusion_method = {direct} &&*
resource_type ≠ document

Another option to ensure that no action is taken on the victim's behalf (CSRF defense), or that no user-specific information is embedded in the response, is by preventing the cookie from being attached to the request. This can be achieved by using the **SameSite** attribute on cookies [32]. When the attribute is set to `Lax`, the cookie will not be included in cross-site requests, and thus the web server will not be able to authenticate the user, and thus can not include user-specific information in the response. This effectively defends against all attacks that require embedding the resource directly from the attacker page. However, when the target resource is included in an other window, the cookies will still be embedded in the request, and attacks such as measuring the execution time are still possible. At the time of this writing, the default value for the SameSite attribute is `Lax` in Chromium-based browsers [5], and Firefox intends to also make make this the default [29].

Defense: SameSite cookie
XS-Leaks thwarted: *component = attacker_page &&*
inclusion_method ∩ *{direct, iframe}* ≠ ∅

To give web developers more insights on what caused the browser to send a request, the **Fetch Metadata** request headers provide

[2] Any other mechanisms that are partitioned should also be included in this set.

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 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. When implemented correctly, this can effectively thwart many XS-Leak attacks.

Defense: Fetch Metadata
XS-Leaks thwarted: $inclusion\_method \cap \{direct, iframe\} \neq \emptyset$

5.4 Targeted defenses

Because the browser is run on physical hardware that has specific constraints, i.e., resources such as hard disk or number of connections are limited, there are constraints that will need to be enforced, or that may cause unintended side-effects, e.g. when the entire disk space is consumed. In our classification, we found four attacks that exploit the consumption of a limited resource in the browser level: event loop timing, response size leak by exploiting the global limit, detecting connection based on the connection pool limits, and detecting whether a redirect occurred. The former has been mitigated as a side-effect of site-isolation defenses [39] where sites are placed in separate processes, and thus no longer share the event loop with other sites. The response size leak was fixed by introducing randomness: whenever a cross-site resource is added to the cache, a random padding was added to the quota. This effectively prevents the adversary from learning the response size, as they can not account for the additional noise originating from the padding. To date, the leak based on the global connection pool has not been mitigated; we believe it is interesting to further explore whether this can also be mitigated by introducing noise or adding an isolation layer. To the best of our knowledge the attack that infers whether a redirect occurred by leveraging the maximum number of requests that can occur before a network error is returned, has not been mitigated either.

5.5 Stopping all XS-Leaks

Because of the variability in the types of XS-Leak attacks, there is not a single defense mechanism that prevents all XS-Leaks. Instead, a combination of multiple defenses, both those that are enabled by default in the browser, and those that need to be opted-in to by the website are required. We found that in order to defend against all currently known XS-Leaks, the smallest set of defenses that need to be combined are the following three: **Fetch Metadata, COOP and site isolation**. Fetch Metadata can prevent the server from processing requests that are sent in an unexpected (and thus likely illicit) manner, for instance when a cross-site document resource would be included directly on an attacker page. The server should only allows top-level navigations and same-origin or same-site requests based on the Fetch Metadata headers; in this case the request will also be blocked when the resource is included as an iframe. By explicitly blocking the request before processing it, instead of its rendering at the client side with framing protection, no timing information can be leaked from the response-generation process. To protect users of browsers that do not (yet) support Fetch Metadata, it is also possible to ensure that the cookie used for authentication has the SameSite attribute set to `Lax`. This will prevent the cookie from being sent along cross-site requests to subresources on the attacker page.

Because the server can not distinguish intended navigational requests from those that are triggered by the attacker, and SameSite Lax cookies will still be included in top-level navigations, COOP is also necessary to prevent the adversary from including resources in another window. Note that despite of COOP, windows can still be opened, but these can not be closed or navigated away from, making an attack practically infeasible. At the time of this writing, Cross Origin Opener Policy is not supported in Safari, making it infeasible to defend against attacks that leverage navigations in another window. Finally, browsers should enable site isolation to prevent cross-site secrets from ending up in the same process as the attacker's. Site isolation is enabled by default in Chrome [10], being deployed in Firefox [15] (and available behind a flag), and not supported in Safari.

6 REAL-WORLD DEPLOYABILITY CHALLENGES

As a case study of deploying defenses for XS-Leaks at scale, we report on the approach and challenges that a large technology company faced when deploying two important XS-Leak protections across 500+ services serving over a billion users. More specifically, COOP and a defense based on Fetch Metadata were deployed. For the latter, a policy was selected that only allows same-site requests and top-level GET navigations; this policy is typically referred to as Resource Isolation Policy (RIP) [62, 73], and will be referred to as such in the remainder of this section. The information and insights were obtained by working and discussing with the team that deployed these defenses.

When dealing with hundreds of different services, it is not practical for the security team to directly collaborate with each service on every change. Instead, a predefined process for large scale changes was followed; this process is used company-wide whenever broad changes that affect hundreds of different packages are affected. When deploying security features, the general methodology followed in the process is as follows:

(1) Prototype with 2-3 high priority products to carefully roll out enforcement, working closely with the product teams.
   → This ensures that the security policy is possible to deploy in practice and indicates potential deployment challenges.
(2) Change a core framework so that all newly created services already enforce a given security policy.
   → This prevents backsliding so that progress can be made over time and newly created services remain unaffected.
(3) Deploy a report-only version of the policy in order to gather quantifiable data and insights about compatibility.
   → This makes it possible to understand how users interact with products in the real world. Oftentimes a product may not have been designed to be used in one way (for example,

in a popup) but in the real world users often depend on unexpected workflows that are important to not break.

(4) Monitor reported violations in order to find endpoints that need special exemptions.
(5) Once there are no remaining violations for a given service, enable enforcement.
    → Enforcement is rolled out gradually in stages. Over a 1 week period the enforcement is first gradually ramped up for company employees, after which a similar deployment is done for all users.

The deployment of RIP and COOP was very similar, with a few exceptions, as the former is enforced at the server side (based on request headers) and the latter at the client side (based on response headers). A key difference is in the reporting: the initial COOP header did not have a report-only header (required for step 3). Hence, the security team worked with browser vendors to add support for the header. For RIP, the reporting is done at the server side, and thus it can easily be analyzed which requests would be blocked (and thus may require a relaxation of the policy).

COOP enforcement was first deployed such that newly created services using a core framework would have it enabled. Over the next couple months, about two dozen newly created services were launched with COOP enabled by default, providing confidence that COOP can safely be rolled out at a larger scale. Subsequently, a report-only policy was gradually rolled out across 500+ services. Interestingly, a bug in Chrome's implementation of COOP caused the browser to crash in specific cases [7]. Because of the gradual roll-out, this was detected early on, and the impact remained limited. Once this issue was mitigated by Chrome, reporting was ramped up to 100% of the traffic.

The reported violations are automatically collected, processed, deduplicated, and denoised before they are manually analyzed. A report is characterized as "noise" if it is non-actionable for a product team (whether due to browser bugs, irregular user behavior, or other software interfering with how a website is supposed to work). In contrast to CSP reports, which can be noisy [25], COOP reports were found to contain minimal noise, and the noise is generally easy to identify. Before switching to an enforcing COOP policy, all reports were first resolved, either by understanding the source of the noise or by fixing the root cause of the report. The main source of noise are:

(1) Third-party sites opening one of the services in a popup and monitoring the `closed` attribute to determine if the user has closed the popup. In these cases it was discussed with the product team to determine whether this behavior should be supported.
(2) A variety of browser extensions trigger COOP violations (often by injecting code that interacts with `window.opener`). In a 1 week period, these represented ~300 reports across 50+ services. These reports were filtered out by checking the `sourceFile` of the report to determine if it was caused by a browser extension.
(3) A commonly used library checked `window.opener.length`. Filtering out reports from this library filtered out ~2% of all violation reports.

After the noisy reports were filtered out, hundreds of endpoints were identified that needed to be opened in a popup, and required a relaxation of the policy, prompting hundreds of changes that were automatically created, tested, shared for review, and merged.

For COOP, ~2% of all endpoints (spread across 14% of services) needed an exempted policy of `unsafe-none`. Since all of these services use client-side navigation, `same-origin-allow-popups` needs to be set for any endpoints on a service if a service ever needs to open and interact with popups. This represented another 14% of services. For RIP, ~3% of all endpoints (representing 8% of services) needed an exemption.

Unfortunately, to prevent introducing new XS-Leaks attacks, the COOP reporting API has certain known gaps where it will not trigger a report even though a violation occurred. This means that a lack of reports does not necessarily mean that it is safe to enable enforcement. A list of known ways in which this can happen was created and each product team was asked whether they thought they fell into any of the incompatible patterns. We elaborate on the different reporting gaps is listed in the Appendix, Section B.

Lessons learned. To deploy XS-Leaks defenses at large scale, we believe that a data-oriented approach driven by user-generated reports has several key advantages. First, not every service needs to be manually analyzed for the sometimes subtle or unclear causes that would violate a policy. Second, due to a large variety in users' systems, and in the way that services may be used by third-party services, manual testing in an isolated environment would result in many violations to be missed. Finally, by observing violations over an extended period of time, accurate predictions can be made on the potential side-effects of enforcing new defenses, and an educated judgment can be made whether these are outweighed by the security benefits. When deploying features that are relatively new, and possibly not extensively tested, a gradual roll-out process can ensure that issues are uncovered early on, and few users are affected. In this case study, XS-Leaks protections were successfully deployed on 500+ services, providing additional security to over a billion users.

7 FACILITATING DEFENSE DEPLOYMENT

Defenses against XS-Leaks still have a low adoption rate. Based on the latest dataset from HTTP Archive (May 2021), CORP is adopted by 0.84% of websites, and the adoption of COOP is even lower, at 0.03%. We believe there are several factors that play a role in the limited deployment of these defenses. First, the defenses have only been introduced relatively recently (first supported by a major browser: CORP in March 2019, COOP in May 2020). Second, because XS-Leak attacks have only recently gained in popularity, a large fraction of website developers may not be aware of these attacks and thus do not try to protect against them. Finally, because of the diversity of XS-Leaks, it may not be clear to developers which defenses are needed to protect against all attacks, and what the potential consequences of relaxing a policy for a particular endpoint can be (and how those can be further protected).

7.1 Leakbuster

To facilitate the adoption of XS-Leak defenses and assess their impact, we created Leakbuster, a dynamic web interface that is based on our model and analysis of the defenses presented in

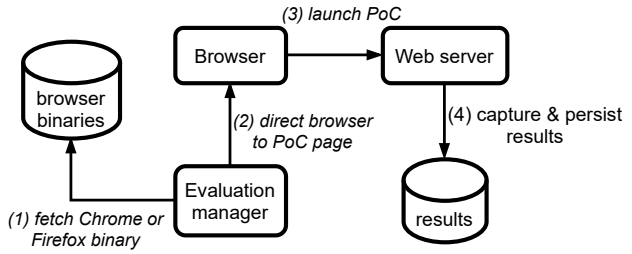


Figure 2: The framework used to track XS-Leaks prevalence across browser versions.

Section 5. Leakbuster allows web developers to indicate which XS-Leak defenses they have currently enabled, and will automatically determine and display which XS-Leak attacks are defended against and which might still be exploitable. Furthermore, it will provide suggestions on how to further secure their website against XS-Leaks, and provide an overview of points of attention when the website needs to exempt certain endpoint from security policies.

As support for different XS-Leak defenses differ depending on the browser engine and version, and certain browsers have built-in security mechanisms, it is important to take this into account when providing guidelines on how to adequately protect all users. As new defenses are regularly implemented, and it may not always be evident which side-effects a certain change in the browser could bring along, we created a framework as part of Leakbuster to easily keep the information up-to-date. An overview of the framework that we use to automatically assess a browser's built-in protections against certain XS-Leaks, as well as validate their support for defense mechanisms, is shown in Figure 2. For all XS-Leaks attacks that we managed to reliably replicate, we created a proof-of-concept implementation, and then used an automation framework to test the PoCs on all available versions for Chromium (75 versions [9]) and Firefox (74 versions [33]). Whenever a new browser version is released, it can be tested using only a single instruction. When the PoCs are launched in the browsers, they access the resources from a dynamic web server that returns responses with the required headers and body. It will then be determined whether the XS-Leak could be successfully exploited, or was thwarted by a countermeasure.

The results of running the framework on historic browser versions is shown in the Appendix, in Table 2. Interestingly, we find that several XS-Leaks such as frame counting and detecting redirects by means of the History API have been present in the web platform (for both Chromium and Firefox) for a very long time; starting from the first browser version that we could use for automation, which was released in 2012. For most other leaks, we find that the presence of the leak often coincides with the introduction of a new feature or API (e.g. AppCache, Fetch). As new features are constantly being added to the web platform, these should be thoroughly analyzed for any state-changes that they may cause, or whether they could be exploited to infer the state introduced by another mechanism. The code of Leakbuster (both the web interface and automation framework) will be made publicly available upon publication of this paper.

8 RELATED WORK

Most related to our paper, is the research by Sudhodanan et al. [52] and the XS-Leaks wiki [49]. Both provide an extensive overview of XS-Leaks, and 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 focus on uncovering the cause of XS-Leaks by introducing a model and taxonomy, showing the different stages of XS-Leaks and that state-changes can occur in different components. Through this view, we also provide a better understanding of how current defenses work against these leaks 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 Section A.

In their research, 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 [43]. 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 [48]. 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 [42]. 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 [13].

Two related research topics that often leverage similar techniques that can leak previously visited websites and fingerprint the browser and system environment of the user. 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 exploit a timing leak in CSS filters to extract whether a link is displayed in the `:visited` style [1, 46]. Similarly, in older browser versions, the attacker could simply read out the computed style to infer whether a certain URL was previously visited [3, 46]. More recently, Karami et al. [20] and Lee et al. [26] showed that the service worker cache could also be exploited to infer whether a user previously visited a specific website.

In fingerprinting attacks, the adversary may also resort to using techniques similar to those in the XS-Leaks to infer the state of the browser or system (which is typically introduced by the interaction between the client and the browser). For example, it has been shown that it is possible to reveal which browser extensions are installed through timing attacks [41, 55], or by the changes that the extension make to web pages [24, 51]. Finally, prior work on user tracking has focused on introducing a known state in the browser that can later be retrieved to re-identify the user. These methods typically include leveraging a method that persists information based on the resource's response, such as caches [47], DNS [22], TLS session state [14, 53], etc.

9 CONCLUSION

XS-Leaks are complex and can occur in all different components of the web ecosystem. In this paper we abstracted away from the

specific details of the techniques and created a model to create a better view of what causes the attacks. We find that XS-Leaks are based on two subsequent phases. In the state-introduction phase the attacker includes a target resource in a specific way, in an attempt to introduce a response-dependent state-change in a particular component. In the following state-retrieval phase, this state-change is then detected, leaking information about the cross-site response. Based on this model, we also analyze the current defenses, and find that these either aim to prevent state-changes from occurring, prevent the state-changes to be observed, or ensure that the cross-site response is not based on the user's information. Finally, motivated by a real-world case-study on deploying XS-Leaks defenses at large scale, we introduce Leakbuster, a web application that aims to facilitate the deployment of defenses by giving suggestions based on already-deployed mechanisms, and indicating which attacks could still be launched when certain exceptions need to be made.

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] Chrome Platform Status. 2021. Feature: Cross-Origin-Opener-Policy. https://www.chromestatus.com/feature/5432089535053824.

[7] Chromium. 2021. [COOP access reporting] Fix crash, invalid cast. https://chromium-review.googlesource.com/c/chromium/src/+/2732471.

[8] Chromium. 2021. Cross-Origin Read Blocking (CORB). https://chromium.googlesource.com/chromium/src/+/refs/heads/main/services/network/cross_origin_read_blocking_explainer.md.

[9] Chromium. 2021. Index of chromium-browser-snapshots/. https://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html.

[10] Chromium. 2021. Site Isolation. https://www.chromium.org/Home/chromium-security/site-isolation.

[11] 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.

[12] 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.

[13] 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.

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

[15] Anny Gakhokidze and Neha Kochar. 2021. Introducing Site Isolation in Firefox. https://blog.mozilla.org/security/2021/05/18/introducing-site-isolation-in-firefox/.

[16] 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.

[17] 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.

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

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

[20] 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)*.

[21] 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.

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

[23] David Kohlbrenner and Hovav Shacham. 2017. On the effectiveness of mitigations against floating-point timing channels. In *26th USENIX Security Symposium (USENIX Security 17)*. 69–81.

[24] 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 (USENIX Security 21)*.

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

[26] 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.

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

[28] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. 2015. The unexpected dangers of dynamic JavaScript. In *24th USENIX Security Symposium (USENIX Security 15)*. 723–735.

[29] Andrea Marchesini. 2019. Prototype SameSite=Lax by default. https://bugzilla.mozilla.org/show_bug.cgi?id=1551798.

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

[31] 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/.

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

[33] Mozilla. 2021. Index of /pub/firefox/releases/. https://ftp.mozilla.org/pub/firefox/releases/.

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

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

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

[37] 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.

[38] 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.

[39] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site isolation: process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*. 1661–1678.

[40] 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.

[41] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. 2017. Extension breakdown: Security analysis of browsers extension resources control policies. In *26th USENIX Security Symposium (USENIX Security 17)*. 679–694.

[42] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. 2017. Beauty and the burst: Remote identification of encrypted video streams. In *26th USENIX Security Symposium (USENIX Security 17)*. 1357–1374.

[43] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. 2017. Same-origin policy: Evaluation in modern browsers. In *26th USENIX Security Symposium (USENIX Security 17)*. 713–727.

[44] 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 (USENIX Security 21)*.

[45] 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 (USENIX Security 19)*. 639–656.

[46] 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)*.

[47] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. 2021. Persistent Tracking in Modern Browsers. (2021).
[48] Dolière Francis Somé. 2019. Empoweb: empowering web applications with browser extensions. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 227–245.
[49] Manuel Sousa, terjanq, Roberto Clapis, David Dworken, and NDevTK. 2020. XS-Leaks Wiki. https://xsleaks.dev/.
[50] Cristian-Alexandru Staicu and Michael Pradel. 2019. Leaky images: Targeted privacy attacks in the web. In *28th USENIX Security Symposium (USENIX Security 19)*. 923–939.
[51] 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.
[52] 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).
[53] 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.
[54] terjanq. 2019. Mass XS-Search using Cache Attack. https://terjanq.github.io/Bug-Bounty/Google/cache-attack-06jd2d2mz2r0/index.html.
[55] 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)*.
[56] 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.
[57] 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 (USENIX Security 20)*. 1985–2002.
[58] Tom Van Goethem, Mathy Vanhoef, Frank Piessens, and Wouter Joosen. 2016. Request and conquer: Exposing cross-origin resource size. In *25th USENIX Security Symposium (USENIX Security 16)*. 447–462.
[59] Anne van Kesteren. 2021. Opaque Response Blocking (ORB, aka CORB++). https://github.com/annevk/orb.
[60] Pepe Vila and Boris Köpf. 2017. Loophole: Timing attacks on shared event loops in chrome. In *26th USENIX Security Symposium (USENIX Security 17)*. 849–864.
[61] WebKit. 2013. Optionally partition cache to prevent using cache for tracking. https://bugs.webkit.org/show_bug.cgi?id=110269.
[62] Lukas Weichselbaum. 2020. Protect your resources from web attacks with Fetch Metadata. https://web.dev/fetch-metadata/.
[63] Mike West. 2021. Web Deprecation Metrics. https://deprecate.it/.
[64] WHATWG. 2021. HTML Living Standard. 4.8.5 The iframe element. https://html.spec.whatwg.org/multipage/iframe-embed-object.html#attr-iframe-name.
[65] XS-Leaks Wiki. 2020. Cache Probing. https://xsleaks.dev/docs/attacks/cache-probing/.
[66] XS-Leaks Wiki. 2020. CORP Leaks. https://xsleaks.dev/docs/attacks/browser-features/corp/.
[67] XS-Leaks Wiki. 2020. Cross-Origin-Opener-Policy. https://xsleaks.dev/docs/defenses/opt-in/coop/.
[68] XS-Leaks Wiki. 2020. Error Events. https://xsleaks.dev/docs/attacks/error-events/.
[69] XS-Leaks Wiki. 2020. Frame Counting. https://xsleaks.dev/docs/attacks/frame-counting/.
[70] XS-Leaks Wiki. 2020. ID Attribute. https://xsleaks.dev/docs/attacks/id-attribute/.
[71] XS-Leaks Wiki. 2020. Navigations. https://xsleaks.dev/docs/attacks/navigations/.
[72] XS-Leaks Wiki. 2020. postMessage Broadcasts. https://xsleaks.dev/docs/attacks/postmessage-broadcasts/.
[73] XS-Leaks Wiki. 2020. Resource Isolation Policy. https://xsleaks.dev/docs/defenses/isolation-policies/resource-isolation/.
[74] 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, which can then be observed in the attacker page. 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 following API: `navigator.storage.estimate()`. 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.

response status (AppCache)

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 browsers 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.

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 microachitectural 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`.

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.

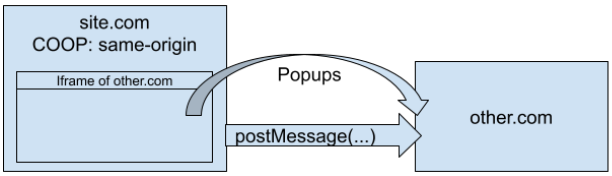
XSSI Some websites may dynamically generate JavaScript that contains potentially sensitive user information. An attacker can embed this JavaScript in their page, and then read out the sensitive data, either by accessing a global property, overwriting a prototype, or redefining global APIs.

B COOP REPORTING GAPS

The experience of deploying COOP at a large technology company discussed in Section 6 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. See this diagram:

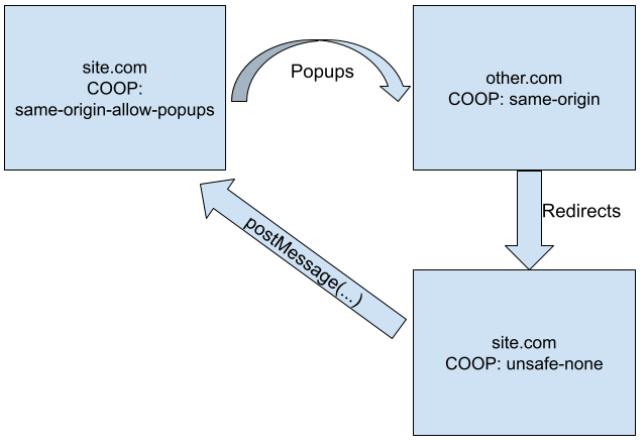


B.2 Redirects

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. See this diagram:



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`. See this diagram:

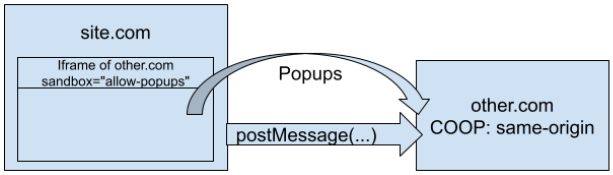


Table 2: Evaluation results

Attack	Chromium	Firefox
(in)valid script/image/..	21 - 91	16 - 89
frame counting	18 - 91	16 - 89
client redirect (History API)	18 - 91	16 - 89
client redirect (load event)	18 - 91	16 - 89
appearance of download bar	39 - 56, 59 - 91	-
no navigation due to download	30 - 91	16 - 89
server redirect (max redirect count)	43 - 91	39 - 89
server redirect (CSP violation)	76 - 91	-
server redirect/status (AppCache)	20 - 91	27 - 83
server redirect (Fetch manual)	75 - 78, 80 - 91	-
element id focus	48 - 91	-
overly broad postMessage	18 - 91	16 - 89
detect CORB'ed JSON responses	68 - 91	-
detect CORP header	73 - 91	74 - 89
detect COOP header	83 - 91	74 - 89
detect XFO (<object>)	72 - 91	81 - 89
response size (Quota API)	43 - 62	-
window.name leak	18 - 91	16 - 89

C LEAKBUSTER: SUPPLEMENTARY DATA

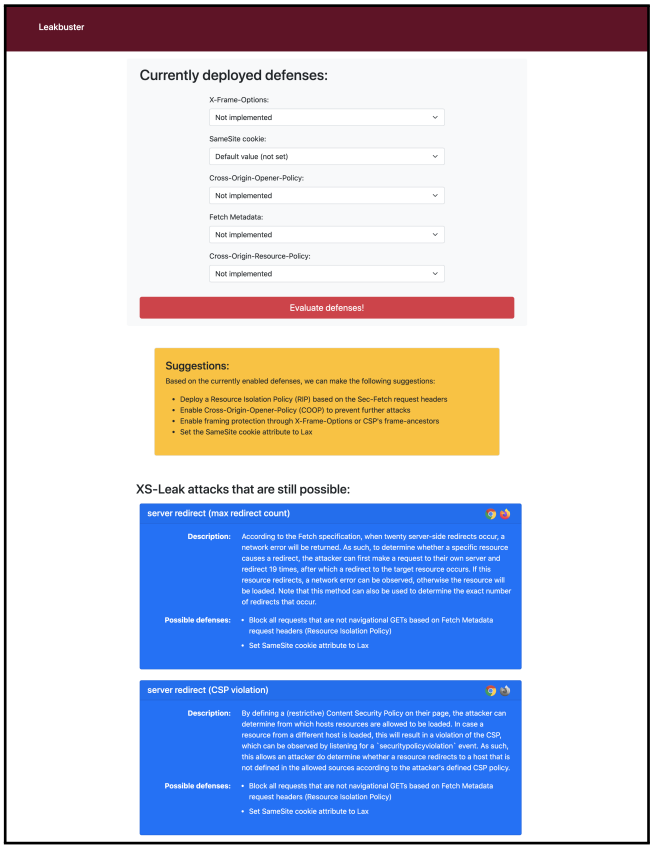


Figure 3: A screenshot of LEAKBUSTER; the user selects which defenses are currently deployed for a specific endpoint, and based on these the tool will provide several suggestions on how the security of the website can be enhanced, and provide an overview of the different XS-Leak attacks that are still possible.

A screenshot of LEAKBUSTER is displayed in Figure 3. On top a web developer can enter which defenses were already deployed for a specific endpoint. Based on this provided information, LEAKBUSTER will make an assessment of the website's ability to counter XS-Leaks with the current defenses. If the protections are insufficient, a list of suggestions is provided that can be used to improve protection; we plan to also include concrete guidelines of how to deploy the defenses and point to the relevant resources for more information, and to give a prioritization for deploying different defenses. Additionally we show the list of XS-Leaks that are still possible, and indicate which browsers are susceptible to them (including the version number).

The results of testing XS-Leaks attacks in historical browser versions (Chromium: versions 18 until 91, Firefox: 16 until 89), are shown in Table 2. A timeline for when support for various defenses that can be used to thwart XS-Leaks was first enabled in Chromium and Firefox is shown in Figure 4.

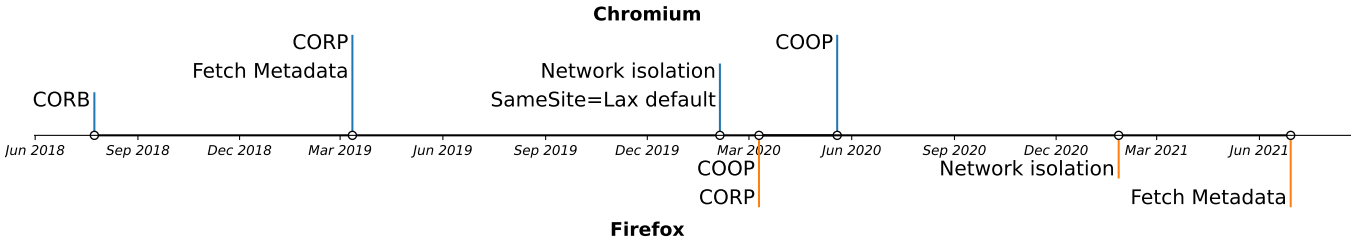


Figure 4: Timeline of the dates when XS-Leaks defenses were enabled and supported in Chromium and Firefox. Initial support for the SameSite cookie is not shown in the diagram; this is May 2016 for Chromium and May 2018 for Firefox.