XSinator.com: From a Formal Model to the Automatic Evaluation of Cross-Site Leaks in Web Browsers

Lukas Knittel
Ruhr University Bochum
lukas.knittel@rub.de

Christian Mainka
Ruhr University Bochum
christian.mainka@rub.de

Marcus Niemietz
Niederrhein University
of Applied Sciences
marcus.niemietz@hs-niederrhein.de

Dominik Trevor Noß
Ruhr University Bochum
dominik.noss@rub.de

Jörg Schwenk
Ruhr University Bochum
joerg.schwenk@rub.de

ABSTRACT

Cross-Site Leaks (XS-Leaks) describe a client-side bug that allows an attacker to collect side-channel information from a cross-origin HTTP resource. They are a significant threat to Internet privacy since simply visiting a web page may reveal if the victim is a drug addict or leak a sexual orientation. Numerous different attack vectors, as well as mitigation strategies, have been proposed, but a clear and systematic understanding of XS-Leak' root causes is still missing.

Recently, Sudhodanan et al. gave a first overview of XS-Leak at NDSS 2020. We build on their work by presenting the first formal model for XS-Leaks. Our comprehensive analysis of known XS-Leaks reveals that all of them fit into this new model. With the help of this formal approach, we (1) systematically searched for new XS-Leak attack classes, (2) implemented XSinator.com, a tool to automatically evaluate if a given web browser is vulnerable to XS-Leaks, and (3) systematically evaluated mitigations for XS-Leaks. We found 14 new attack classes, evaluated the resilience of 56 different browser/OS combinations against a total of 34 XS-Leaks, and propose a completely novel methodology to mitigate XS-Leaks.

CCS CONCEPTS

• **Information systems** → **Browsers**; Web applications; • **Security and privacy** → **Formal security models**.

KEYWORDS

XS-Leaks; Browser; Web Security

ACM Reference Format:
Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. 2021. XSinator.com: From a Formal Model to the Automatic Evaluation of Cross-Site Leaks in Web Browsers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3460120.3484739

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3484739

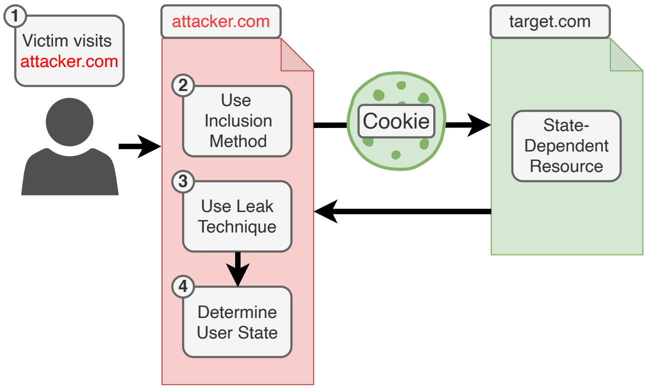

Figure 1: XS-Leak attack flow example. The victim (1) visits the attacker-controlled website, which (2) uses an inclusion method to request a state-dependent resource from a target website. The attacker then uses (3) a leak technique to (4) determine the victim's user state.

1 INTRODUCTION

Web Applications and User States. In a *web application*, a web browser interacts with several web servers through HTTP or WebSocket connections. The client-side logic of the web application is written in HTML, CSS, and JavaScript code, and is executed inside a tab of the browser, or inside an inline frame in another application. The *execution context* of a web application is defined through the concept of *web origins* [5]. Web applications may call and embed other web applications to enhance functionality. For example, a hotel reservation site may embed Google Maps and public transportation sites as an easy method to allow its customers to determine how to reach the hotel. In such situations, *cross-origin* HTTP requests between different web origins are necessary to retrieve data to embed and display in the web application.

When interacting with a website, a user has a well-defined *state* – this state typically contains the information whether the user is logged in or not. Besides the login status, the user state may contain account permissions, such as admin privileges, premium membership, or restricted accounts. The number of different user states is potentially unlimited. For example, in a webmail application, a user may or may not have received an email with the subject "top secret".

Privacy Risks of Cross-Origin Requests. Consider the following situation: the attacker has lured a victim on a malicious web application that executes hidden cross-origin HTTP requests to different drug counseling sites. If the attacker could learn whether the victim is logged in at one of these drug counseling sites, the attacker would gain highly privacy-critical information about the victim.

Cross-Site Leaks on the User States. To distinguish between two user states, the attacker's JavaScript code must be able to identify differences in its own execution environment resulting from different responses to cross-origin HTTP requests. These different responses must correspond to different user states at the target web application. If this differentiation is possible, we call this vulnerability an *XS-Leak*. The attacker can then craft a malicious website, which triggers the XS-Leak once the victim opens it (Figure 1). In the following two real-world examples, we try to exemplify the scope of the problem.

XS-Leak on Gitlab. Gitlab is a popular web application for collaborative software development hosted by many companies. Gitlab provides a profile URL https://git.company.com/profile: if the user is not logged in, this URL redirects the user to https://git.company.com/users/sign_in; if the user is logged in, the current user's profile information is shown. However, since the attacker embeds Gitlab cross-origin into the attacker's own web page, the attacker cannot directly read the URL. In Listing 1, we use the `window.length` property, which is readable cross-origin, to determine the user state; the profile page does not contain any iframes, but the login page includes three frames. If this property has the value 3, the user is not logged in. If it has the value the 0, the user is logged in. By scanning different company websites hosting Gitlab, the attacker may collect information on a programmer's affiliation.

Listing 1: XS-Leak on Gitlab.

```
let url = 'https://git.company.com/profile'
let ref = window.open(url, '_blank')
// wait until pop-up is loaded
let counted_frames = ref.window.length;
if (counted_frames === 0) {
    // User is logged in
} else if (counted_frames === 3) {
    // User is NOT logged in
}
```

XS-Leak on Google Mail. Google Mail is one of the most popular webmail applications. In 2019, Terjanq [53] reported an XS-Leak which could determine whether an email with a certain subject (e.g., "drug counseling") or content was present in the user's inbox cross-origin. The XS-Leak abused the common cache that web applications share. By using the advanced search option, which can be called cross-origin, Google Mail marks search results (if any exist) with a dedicated image. To perform an XS-Leak attack, the attacker first empties the web cache, then calls Google Mail advanced search, and finally checks if the dedicated image is available in the cache. If true, the search was successful, and the attacker learned that an email containing the used search term exists in the victim's inbox.

Formal Modelling and Testing. Sudhodanan et al. [51] gave the first classification of existing XS-Leaks. They surveyed related work, both academic and non-academic, added further attack classes, and showed that XS-Leaks are a novel paradigm in attacks on privacy. We build on their work to propose the first formal model for XS-Leak attacks (Section 2). This formalization allows us to extend their work in three aspects:

(1) By distinguishing between inclusion methods i and leak techniques t, we provide a clear methodology on how to search for novel, yet undiscovered, XS-Leaks. While we did not extend the set of inclusion methods I, we later show that mitigations may work for certain inclusion methods only. On the other hand, we could substantially extend the set T of leak techniques, by grouping known elements of T and systematically searching for new vectors which may fit in these groups.

(2) To systematically evaluate the three-dimensional matrix consisting of pairs (i, t, w), where $w \in W$ is a web browser from the set of tested web browsers, we build a tool called XSinator. This tool uses hand-crafted state-dependent resources to determine if there exists at least one pair i, t for which the state of this resource can be leaked in w. If at least one such pair exists, we label w as vulnerable against this attack vector. With this tool, we were able to detect major differences between browser implementations (Table 2). These findings are used as guides to propose new mitigation methods.

(3) By separating inclusion methods i and leak techniques t, we could evaluate existing and propose novel mitigations. Current mitigations mainly focus on the inclusion methods. 4 out of 7 mitigations discussed in [51] are related to HTTP headers, which only may be effective against certain inclusion methods. For example, `SameSite=Lax` cookies are only effective if the target resource is included as an iframe, but not if it is called with `window.open`. We introduce a novel class of mitigations, which in Sudhodanan et al. [51] is only present as a short analysis of the Tor browser. This class of mitigations targets the leak techniques T and can be sketched as follows: if our evaluation (Table 2) shows that some web browsers are immune to certain XS-Leaks, this indicates that the corresponding leak techniques can be fixed by changing the browser implementation. So in our discussion of mitigations, we clearly distinguish between mitigations targeting certain inclusion methods only, and mitigations targeting leak techniques.

Empirical Study with Reproducible Results. XSinator is an easy to deploy web application. With a single click, all XS-Leaks test cases can be automatically executed for the active browser. We used XSinator to evaluate 37 different web browsers on desktop operating systems, 9 on Android, and 10 on iOS. Surprisingly, we identified very different XS-Leaks in all major browser families such as Chromium, Firefox (FF), and Safari (SA). Browsers based on the Chromium engine (i.e., Google Chrome (GC), Microsoft Edge (MSE), Opera (OP)) behave identically. For example, XS-Leaks identified in GC are also applicable to all other browsers based on the same engine. Moreover, we could detect differences in FF compared to the Tor browser.

Contributions. We make the following key contributions:

- We are the first to present a formal model for XS-Leaks. We show that this model can be used to gain a systematic in-depth understanding of XS-Leaks' root causes (Section 2).
- We implemented XSinator, an easy-to-use, open-source website that is publicly available at XSinator.com. With a single click, XSinator can automatically scan for XS-Leaks vulnerabilities in every browser, including mobile and desktop (Section 3).
- We analyze known vulnerabilities and show that they fit into our formal model (Section 4).
- We significantly extend the set of known XS-Leak attack vectors by contributing 14 new XS-Leaks with the help of our formal model (Section 5).
- We evaluated 56 browser/OS combinations against a total set of 34 XS-Leaks XSinator fully automatically. We identified XS-Leaks in 37 desktop, 9 Android and 10 iOS web browsers. (Section 6).
- We use XSinator and the formal distinction between leak techniques and inclusion methods to propose a novel class of mitigation that disables leak techniques within web browser implementations (Section 7).

Artifact Availability. In the interest of open science, our tool and its source code, including all 34 XS-Leaks implementations, are available at XSinator.com.

Responsible Disclosure. We reported new leak techniques to Google and Mozilla and the disclosure process is still ongoing. We incorporated their feedback into our countermeasure discussion in Section 7.

2 FORMAL XS-LEAK DESCRIPTION

This section gives a formal description of XS-Leak attacks as a basis for further analysis. This formal description helps us to (1) classify existing XS-Leaks thoroughly, (2) systematically identify new candidates for XS-Leaks, and (3) classify and evaluate countermeasures.

2.1 Formal Description

Same Origin. We first formalize the well-known fact that requests to the same URL may yield different results, depending on which state s the web application is currently in.

Definition 2.1 (State-dependent resource). A *state-dependent resource sdr* is a 2-tuple $(url, (s, d))$, where $(s, d) \in \{(s_0, d_0), (s_1, d_1)\}$, and

- url is a URL resource on the target web application.
- $S = \{s_0, s_1\}$ is a set of two different *states* of the target web application.
- $D = \{d_0, d_1\}$ is a set that represents the *difference* of the web application's behavior that depends on s_0 and s_1.

Differences. Please note that the definition of D is, by intention, somewhat vague. Two different states s_0, s_1 on the same $url \in URL$ can result in different behavior by the web application. The difference $d \in D$ manifests itself either directly in the initial response (e.g., HTTP status code, or different HTML elements) or indirectly when the web application behaves differently (e.g., a navigation triggered by JavaScript code). Our notion of d does not only cover differences in HTTP requests and responses, but also side-effects on the APIs caused by these responses.

States. A web application may have (potentially infinitely many) different states $s \in S$ for a user v. To successively detect the desired state, the attacker may use a divide-and-conquer approach, for example, by detecting the subject on an email in a webmail application letter by letter. For this reason, in our model, we concentrate on distinguishing between two different states s_0, s_1. States are typically stored in the web browser, which sends them along with the HTTP Request (e.g., HTTP cookies, cache content).

Cross-Origin. In the web attacker model, an adversary can only access a state-dependent resource sdr from a different web origin. He thus must use an *inclusion method* to include sdr into his web page, and he may use different *leak techniques* to observe the victim's web browser from his malicious JavaScript code.

Definition 2.2 (Cross-Site Leak). A Cross-Site Leak is a function $xsl()$ that outputs a *bit* b', that is $b' = xsl(sdr, i, t)$ where

- $sdr \in SDR$ is a state-dependent resource.
- $i \in I$ is an *inclusion method* to request a cross-origin resource.
- $t \in T$ is a *leak technique* which can be used to observe state-dependent resources cross-origin.

The difference d in a state-dependent resource sdr is called *detectable* if there exists an inclusion method i and a leak technique t such that $xsl((url, (s_b, d_b)), i, t) = b$ for all requests.

Inclusion methods $i \in I$ trigger cross-origin requests to the target web application's state-dependent resource sdr in the victim's browser w_v. For example, the attacker can include the url of the sdr (i.e., $sdr.url$) in the `src` attribute of a specific HTML element, and may use the target when opening new browser tabs or windows, or can use the Fetch API. Note that since i issues a cross-origin request, t cannot directly access the server response due to the Same-Origin Policy (SOP) that forbids cross-origin access.

Leak techniques $t \in T$ provide information that malicious JavaScript can observe *cross-origin* when it runs in the victim's web browser w_v. This information may be rooted in the JavaScript execution context (e.g., event handlers, readable DOM attributes), in the global state of the web browser (e.g., global resource limits), or on Web APIs (e.g., the Performance API). It can vary in different browsers. For example, a piece of information may be accessible cross-origin in Firefox but inaccessible in Chrome.

Example: Cross-Site Leak on Gitlab. Let us illustrate these definitions with the XS-Leak on Gitlab as depicted in Listing 1. In that case, we have a state-dependent resource sdr with

- url = https://git.company.com/profile
- $s \in$ {logged-in, logged-out}
- d is the number of frames included in the page: $d_0 = 0$ for state logged-in, and $d_1 = 3$ for state logged-out.

Note that the reconnaissance phase, in which such state-dependent resources are found, is conducted on the web application itself (same origin) and thus is not limited by the cross-origin restrictions. Whether a state-dependent resource is exploitable in an XS-Leak depends on the attacker finding a suitable i and t that works in a victims browser. In this example, the attacker implements the XS-Leak $b' = xsl(sdr, i, t)$ as follows:

- `window.open(sdr, '_blank')` is the inclusion method i.
- The DOM property `window.length` is the leak technique t, which reads the number of frames d by using the pop-up window reference from i.
- The function $xsl()$ is given in Listing 1.

2.2 Attacker Model

Different attacker models can be derived from this formal description by giving the attacker control over the different components of an XS-Leak $d = xsl(sdr, i, t)$.

Real-World Attacks. In real-world attacks, the attacker only controls the inclusion method i and the leak technique t through the malicious HTML page that they created. The attacker has no control of $sdr.s$, which is stored in the victim's browser, or $sdr.d$, which depends on the target web application. To determine a state-dependent resource sdr, a reconnaissance phase is needed in which the attacker collects information on different states s that the web application may have, and on the resulting differences d in the HTTP responses.

Evaluation. Our evaluation uses a stronger attacker model, which also gives the adversary full control over the state-dependent resource sdr. That is, the attacker can choose the url, choose a state s and a difference d. The only element the attacker does *not* control is the web browser and its cross-origin isolation techniques. In an ideal world, the browser should reduce the set of detectable difference to the empty set. Thus, we consider the strongest possible attacker, and we want to determine which XS-Leak may *potentially* exist in any web application. We describe this approach's goal as follows: if we can reduce the set of detectable difference in this strong attacker model by strengthening cross-origin isolation techniques within the web browser, then we *automatically* reduce the attack surface in any weaker, more realistic attacker model. Our security experiment can be described as follows:

Definition 2.3 (XS-Leak Security Experiment). The XS-Leak Security Experiment operates as follows:

- **Setup.** In our security experiment, the attacker sets up a web application's state-dependent resources $sdr_0 = (url, (s_0, d_0))$, $sdr_1 = (url, (s_1, d_1))$, where the web application at url differs exactly in d based on s. The attacker then creates an XS-Leak xsl with the inclusion method i and the leak technique t, and deploys the resulting code in the web browser w_v under investigation.
- **Execution.** An unbiased random bit $b \in \{0, 1\}$ is chosen by the environment, and the resource sdr_b, which uses state s_b, is selected. The attacker does *not* learn b. The attacker may now issue a request to sdr_b, using inclusion method i. Once the response has been returned, the attacker may try to determine the state s by learning the detectable difference d through the use of leak technique t in w_v. From d, the attacker derives state s and, therefore, bit b'.
- **Winning condition.** The attacker wins the security experiment if $b' = b$.

2.3 Detectable Differences

A *detectable difference* is a difference D that can be observed cross-origin through at least one pair (i, t) to infer the actual state – we do not observe the difference directly, but rather through a side effect that this difference causes. We categorized them into five groups:

Status Code. An attacker can distinguish different HTTP response status codes cross-origin (e.g., server errors, client errors, or authentication errors).

API Usage. This detectable difference allows an attacker to detect Web APIs' usage across pages, allowing an attacker to infer whether a cross-origin page is using a specific JavaScript Web API.

Redirects. It is possible to detect if a web application has navigated the user to a different page. This is not limited to HTTP redirects but also includes redirects triggered by JavaScript or HTML.

Page Content. These detectable differences appear in the HTTP response body itself or in sub-resources included by the page. For example, this could be the number of included frames (cf. XS-Leak on Gitlab) or size differences of images.

HTTP Header. An attacker can detect the presence of a specific HTTP response header and may be able to gather its value. This includes headers such as `X-Frame-Options`, `Content-Disposition`, and `Cross-Origin-Resource-Policy`.

It is questionable that these groups are complete since new browser features or yet unknown XS-Leaks might unveil new detectable difference. However, they serve as a guideline for finding new XS-Leaks (see Table 1).

2.4 Real-world Inclusion Methods

In all XS-Leak attacks, the attacker's web page uses cross-origin inclusion methods to force the victim's browser in requesting the state-dependent resource. In theory, the SOP should prevent cross-origin information leakage, but this separation of web origins is not perfect and exceptions must be made; cf. Schwenk et al. [46] for a partial analysis. In the following, we discuss four different groups of inclusion methods:

HTML Elements. HTML offers a variety of elements that enable cross-origin resource inclusion. Elements like stylesheets, images, or scripts, force the victim's browser to request a specified non-HTML resource. A list that enumerates possible HTML elements for this purpose is available online [21].

Frames. Elements such as iframe, object, and embed may embed further HTML resources directly into the attacker page. If the page does not use framing protection, JavaScript code can access the framed resource's window object via the `contentWindow` property.

Pop-ups. The `window.open` method loads a resource in a new browser tab or window. The method returns a window handle that JavaScript code can use to access methods and properties, which comply with the SOP. These so-called pop-ups are often used in single sign-on. Modern browsers only allow pop-ups if they are triggered by certain user interactions [43]. For XS-Leak attacks, this method is especially helpful because it bypasses framing and cookie restrictions for a target resource. Newer browser versions recently added means to isolate window handles, as described in Section 7.

JavaScript Requests. JavaScript allows sending requests to target resources directly. There are two different ways for this purpose: *XMLHttpRequests* and its successor Fetch API [40]. In contrast to previous inclusion methods, an attacker has fine-grained control over the issued request, for example, whether an HTTP redirect must be automatically followed.

2.5 Real-world Leak Techniques

An attacker can observe various types of information from cross-origin resources. As it is not possible to directly access the response of a cross-origin request, an XS-Leak attacker relies on side effects that are caused by the included resource. We found that the techniques used to detect these side effects can ultimately be ascribed to a set of core problems. By analyzing the existing and new XS-Leaks we identified six groups:

Event Handler. Event handler can be seen as the *classical* leak technique for XS-Leaks. They are a well-known source of various pieces of information. For example, the trigger of `onload` indicates a successful resource loading in contrast to the `onerror` event.

Error Messages. Beyond event handlers, error messages can occur as JavaScript exceptions and special error pages. Error messages can be thrown in different steps, for example, directly by the leak technique. The leak technique can either use additional information directly contained in the error message, or distinguish between the appearance and absence of an error message.

Global Limits. Every computer has its physical limits, so does a browser. For example, the amount of available memory limits a browser's running tabs. The same holds for other browser limits that are enforced for the entire browser. If an attacker can determine when the limit is reached this can be used as a leak technique.

Global State. Browsers have global states that all pages can interact with. If this interaction is detectable from an attacker's website, it can be used as a leak technique. For example, the *History* interface allows manipulation of the pages visited in a tab or frame. This creates a *global state* because the number of entries allows an attacker to draw conclusions about cross-origin pages.

Performance API. The Performance API is used to access the performance information of the current page. Their entries include detailed network timing data for the document and every resource loaded by the page. This allows an attacker to draw conclusions about requested resources. For example, we identified cases where browsers will *not* create performance entries for some requests.

Readable Attributes. HTML has several attributes that are readable cross-origin. This read access can be used as a leak technique. For example, JavaScript code can read the number of frames included in a webpage cross-origin with the `window.frame.length` property.

3 XSINATOR: AUTOMATIC BROWSER EVALUATION

One of this paper's main contributions is to evaluate the impact of XS-Leak attacks on different web browsers $w \in W$. We systematically extend the work of Sudhodanan et al. [51] by including a broad set of relevant browsers, both desktop and mobile, and extending the set of XS-Leak attacks significantly.

3.1 Implementation

Based on our formal description in Section 2, we evaluate all inclusion methods and leak techniques for a large set W of web browsers. For that, we built a web application named XSinator that consists of three main components:

(1) A *testing site* that acts as an XS-Leak attacker page. It implements known and novel XS-Leaks and evaluates them by running all of them with a single click.
(2) A *vulnerable web application*, which simulates the behavior of a *state-dependent resource* (*sdr*) for each XS-Leak. This web application has two states s_0 and s_1, which are triggered via a parameter in the HTTP requests. The states trigger different behavior in the *sdr*, for instance, in state s_0, the *sdr* has difference d_0 and d_1 in s_1.
(3) A *database* containing all previous test results. Security researchers can use this database to compare these results with the actual results of a new browser and track the XS-Leaks' exploitability over time.

The JavaScript code aims to distinguish states s_0 and s_1 based on the retrieved side-channel information. In our formal description, we consider an XS-Leak *exploitable* (denoted by ● in Table 2, Table 3, and Table 4), if there exists an inclusion method i and a leak technique t in web browser w such that $xsl((url, s_b, d_b), i, t) = b$. Otherwise, the XS-Leak is *not exploitable* (denoted by ○).

The test results from XSinator allow us to draw the following conclusion: If a certain XS-Leak is only exploitable in some of the tested web browsers, then the underlying leak technique could be fixed and will most probably not break existing web applications. This allows us to propose realistic countermeasures to known and novel XS-Leaks in Section 7.

3.2 Evaluation Challenges

The implementation of XSinator was far from being straightforward and revealed surprising insights.

Privileged Events. Some XS-Leaks require specific conditions or user interaction. For example, the Frame Count Leak, Web Payment API Leak, and the WebSocket Leak require a reference to an opened cross-origin window handle. JavaScript code can only create this handle with privileged events. For instance, a browser is only allowed to open a pop-up window if the `window.open()` function is triggered by human user interaction. XSinator uses the initial click on the "Run All Tests" button to initialize the environment as necessary.

Compatibility between Browsers. Since known XS-Leaks are often specific to one particular browser, we adapted them to be compatible with as many browsers as possible. If it was not possible to cover all browsers, XSinator implements variants of the same leak technique.

Mobile Browsers. The user interfaces of all mobile browsers restrict the number of visible windows/tabs to one. In all tested browsers except *FF Focus*, multiple windows/tabs can be opened in the background and are executed in parallel. Therefore, the test suite can be flawlessly executed. In FF Focus XSinator will not execute the test cases requiring a secondary window correctly due to the missing functionality. This has a low impact since an attacker would succumb to the same conditions.

Different Error Types. Many XS-Leaks distinguish between a *successful* leak technique's code execution and code execution triggering an error. For this reason, XSinator must distinguish if a triggered error can be interpreted as a *XS-Leak test result* or a *runtime error*. For example, when comparing SA to Chromium-based browsers, the *Web Payment API* is implemented differently. It, therefore, needs custom code adaptation. In contrast, FF does not implement the Web Payment API at all. Therefore it throws different errors, which XSinator must correctly interpret to give accurate results.

3.3 Limitations

XSinator comes with a few limitations and constraints.

No Automatic Detection of New Variants. XSinator is *not* designed to find new attack variants and automatically building new XS-Leaks remains an open problem. Although, as we implemented leaks that were thought of to be specific to one browser, we often found that they apply to others browser families as well by changing the leak technique or inclusion method.

From Browser to Website Evaluation. XSinator is not meant to be an automatic penetration testing tool. We use XSinator to systematically evaluate browser implementations against all known XS-Leaks. Although it cannot automatically detect weaknesses in real-world websites, developers can run the implemented XS-Leaks against a specified endpoint.

Excluded XS-Leaks. We had to exclude XS-Leaks that rely on service workers as they would interfere with other leaks in XSinator. Furthermore, we chose to exclude XS-Leaks that rely on misconfiguration and bugs in a specific web application. For example, Cross-Origin Resource Sharing (CORS) misconfigurations, postMessage leakage [19] or Cross-Site Scripting. Additionally, we excluded time-based XS-Leaks since they often suffer from being slow, noisy and inaccurate.

4 OVERVIEW OF XS-LEAK ATTACKS

We conducted a comprehensive analysis of known XS-Leaks. In Table 1, we present our results and show that all of them fit in our formal model. Each known XS-Leak can be described using $xsl(sdr, i, t)$ with $i \in I$ and $t \in T$. In total, there are 5 detectable differences and 34 XS-Leaks, including a contribution of 14 novel XS-Leaks (⊕) discussed in Section 5. We use 5 classes of detectable difference to structure this section, since they provide the basic information we want to observe through different inclusion methods and leak techniques based sidechannels.

Full details for the new attacks are provided in Section 5. Details for other attacks are given in Appendix A.

5 NEW XS-LEAK ATTACKS

Our formal model reveals that every XS-Leak consists of three main ingredients. To systematically identify novel XS-Leaks, we henceforth investigated them. First, there are inclusion methods. The sets of inclusion methods are well-known and mostly static. Novel XS-Leaks can be especially identified once browser vendors implement new features that are leak techniques. Second, there are leak techniques. Novel XS-Leaks are typically found by developing new *leak techniques*. In this section, we identified new XS-Leaks based on this ingredient. Third, there are detectable differences. We systematically created tests to extend the set of *detectable differences* when combined with known leak techniques, which led us to new XS-Leaks.

5.1 Leak Technique: Global Limits

WebSocket API. With this new technique, it is possible to identify if, and how many, WebSocket connections a target page uses. It allows an attacker to detect application states and leak information tied to the number of WebSocket connections.

Details: The WebSocket API allows the use of streaming connections between clients and servers using proprietary (e.g., binary) protocols. The client initiates the WebSocket Handshake using HTTPS. Upon success, the established TLS connection is used for tunneling the desired protocol. The specification recommends a limitation of WebSocket connections per client [15]. If one origin uses the maximum amount of WebSocket connection objects, regardless of their connections state, the creation of new objects will result in JavaScript exceptions. To execute this attack, the attacker website opens the target website in a pop-up or iframe and then attempts to create the maximum number of WebSockets connections possible. The number of thrown exceptions is the number of WebSocket connections used by the target website window.

Example Attack: Slack is a proprietary business communication platform that offers teams to work together in one *workspace.* Users can join one or more workspaces and communicate with other members in real-time. For this purpose, Slack uses WebSockets. We found that by detecting this WebSocket connection, it is possible to leak if a user is a member of a specific workspace.

Payment API. This XS-Leak enables an attacker to detect when a cross-origin page initiates a payment request.

Details: The Payment Request API enables a website to use the web browser to conduct payments. The user enters their payment credentials (e.g., credit card information) into the web browser.

	XS-Leak	Related Work	Leak Technique $t \in T$	Inclusion Method $i \in I$
	Detectable Difference: Status Code			
⊕	Perf. API Error	(Section 5.2)	A request that results in errors will not create a resource timing entry.	HTML Elements, Frames
⊕	Style Reload Error	(Section 5.2)	Due to a browser bug, requests that result in errors are loaded twice.	HTML Elements
⊕	Request Merging Error	(Section 5.2)	Requests that result in an error can not be merged.	HTML Elements
	Event Handler Error	Staicu and Pradel [50], Sudhodanan et al. [51]	Event handlers attached to HTML tags trigger on specific status codes.	HTML Elements, Frames
	MediaError	Acar and Danny Y. Huang [1]	In FF, it is possible to accurately leak a cross-origin request's status code.	HTML Elements (Video, Audio)
	Detectable Difference: Redirects			
⊕	CORS Error	(Section 5.3)	In SA CORS error messages leak the full URL of redirects.	Fetch API
⊕	Redirect Start	(Section 5.2)	Resource timing entry leaks the start time of a redirect.	Frames
⊕	Duration Redirect	(Section 5.2)	The duration of timing entries is negative when a redirect occurs.	Fetch API
	Fetch Redirect	Janc et al. [30]	GC and SA allow to check the response's type (*opaque-redirect*) after the redirect is finished.	Fetch API
	URL Max Length	Masas [38, 39]	Gather the length of a URL that triggers an error on a specific server.	Fetch API, HTML Elements
	Max Redirect	Herrera [23]	Abuse the redirect limit to detect redirects.	Fetch API, Frames
	History Length	Olejnik et al. [44], Smith et al. [47], terjanq [54], Wondracek et al. [75]	JavaScript code manipulates the browser history and can be accessed by the `length` property.	Pop-ups
	CSP Violation	Homakov [27], West [63]	The attacker sets up a CSP on `attacker.com` that only allows requests to `target.com`. If `attacker.com` issues a request to `target.com` that redirects to another cross-origin domain, the CSP blocks access and creates a violation report. Target location of the redirect may leak.	Fetch API, Frames
	CSP Detection	Homakov [27], West [63]	Similar to the above leak technique, but the location does not leak.	Fetch API, Frames
	Detectable Difference: API Usage			
⊕	WebSocket	(Section 5.1)	Exhausting the WebSocket connection limit leaks the number of WebSocket connections of a cross-origin page.	Frames, Pop-ups
⊕	Payment API	(Section 5.1)	Detect Payment Request because only one can be active at a time.	Frames, Pop-ups
	Service Worker	Karami et al. [32]	Detect if a service worker is registered for a specific origin.	Frames
	Detectable Difference: Page Content			
⊕	Perf. API Empty Page	(Section 5.2)	Empty responses do not create resource timing entries.	Frames
⊕	Perf. API XSS-Auditor	(Section 5.2)	Detect presence of specific elements in a webpage with the XSS-Auditor in SA.	Frames
	Cache	Vela [59]	Clear the file from the cache. Opens target page checks if the file is present in the cache.	Frames, Pop-ups
	Frame Count	Grossman [18], Masas [38]	Read number of frames (`window.length`).	Frames, Pop-ups
	Media Dimensions	Masas [38]	Read size of embedded media.	HTML Elements (Video, Audio)
	Media Duration	Masas [38]	Read duration of embedded media.	HTML Elements (Video, Audio)
	Id Attribute	Heyes [25]	Leak sensitive data from the `id` or `name` attribute.	Frames
	CSS Property	Evans [13]	Detect website styling depending on the status of the user.	HTML Elements
	Detectable Difference: Header			
⊕	SRI Error	(Section 5.3)	Subresource Integrity error messages leak the size of a response in SA.	Fetch API
⊕	Perf. API Download	(Section 5.2)	Downloads do not create resource timing entries in the Performance API.	Frames
⊕	Perf. API CORP	(Section 5.2)	Resource protected with CORP do not create resource timing entries.	Frames
⊕	COOP	(Section 5.4)	`COOP` protected pages can not be accessed.	Pop-ups
	Perf. API XFO	terjanq [55]	Resource with `X-Frame-Options` header does not create resource timing entry.	Frames
	CSP Directive	Yoneuchi [76]	CSP header directives can be probed with the CSP iframe attribute.	Frames
	CORP	Wiki [72]	Resource protected with CORP throws error when fetched.	Fetch API
	CORB	Wiki [71]	Detect presets of `Content-Type` and `Content-Type-Options` headers, because CORB is only enforced for specific content types together with the `nosniff` option.	HTML Elements
	ContentDocument XFO	Sudhodanan et al. [51]	In GC, when a page is not allowed to be embedded on a cross-origin page because of `X-Frame-Options`, an error page is shown.	Frames
	Download Detection	Masas [38]	Attacker can detect downloads by using iframes. If the iframe is still accessible, the file was downloaded.	Frames

Table 1: Overview of XS-Leaks attacks sorted by their detectable difference integrated into our formal model. We contribute novel attack techniques indicated by ⊕.

Afterwards, the website can query the API to request payment. The browser then shows a UI pop-up to the user, and the user can confirm the purchase with one single click on a button. Similar to the WebSocket API, the standard specification recommends a global limit of one singular UI element [28]. If the target website is using the Payment Request API, any further attempts to show this UI will be blocked, and cause a JavaScript exception. The attacker can exploit this by periodically attempting to show the Payment API UI. If one attempt causes an exception, the target website is currently using it. The attacker can hide these periodical attempts by immediately closing the UI after creation. Instead of opening the UI and then closing it, the browser never shows the UI, and the user takes no notice of the attack.

Example Attack: A specific product provided by the website *shop.org* is advertised on a website, for example, *blog.com*, using an affiliate link. The operator of *shop.org* can use our attack to identify if a customer bought the product after clicking on provided affiliate link.

5.2 Leak Technique: Performance API

We developed new XS-Leak attacks based on the *Performance API*, which allows an attacker to leak various characteristics of the target page.

Details: The Performance API is used to access the performance information of the current page. This includes detailed network timing data for the document and every resource the page loads. Terjanq [55] showed how to detect the `X-Frame-Options` header in GC. We used his work as a base to create novel attacks that allow to differentiate between status codes, to detect empty pages, to detect if the XSS-Auditor is executed, and we improved terjanq's work to detect `X-Frame-Options` in non-Chromium-based browsers. The Performance API specifies that all fetched resources *must* create a performance entry [29]. We identified cases where browsers will *not* create an entry for a specific request. That means an attacker can differentiate requests by checking if a performance entry is created. The following 8 new XS-Leak attacks are based on this observation:

- **Error Leak**: It is possible to differentiate between HTTP response status codes because requests that lead to an error do not create a performance entry. This has a similar impact to XS-Leak described in Section A.1. We also identified two cases where browser bugs in GC lead to resources being loaded twice when they fail to load. This will result in multiple entries in the Performance API and can thus be detected.
- **Empty Page Leak**: An attacker can detect if a request resulted in an empty HTTP response body because empty pages do not create a performance entry in some browsers.
- **XSS-Auditor Leak**: In SA, it is possible to detect if the XSS-Auditor was triggered and thus leak sensitive information. The XSS-Auditor is a built-in feature of SA and GC (now removed [10]) designed to mitigate Cross-Site Scripting (XSS) attacks. It aims to protect against reflected XSS by checking query parameters. In 2013, Braun and Heiderich [7] showed that the XSS-Auditor can be used to block benign scripts with false positives. Based on their technique, researchers exfiltrate information and detect specific content on a cross-origin page. These XS-Leaks were first described in a bug report by Terada [52] and later in a blog post by Heyes [24]. However, the discovered techniques applied only to the XSS-Auditor in GC and do not work in SA. We found that blocked pages will not create Performance API entries. That means an attacker can still leak sensitive information with the XSS-Auditor in SA.
- **X-Frame Leak**: If a page is not allowed to be rendered in an iframe it does not create a performance entry. As a result, an attacker can detect the response header `X-Frame-Options`.
- **Download Detection**: Similar, to the XS-Leak described in Section A.5, a resource that is downloaded because of the `Content-Disposition` header, also does not create a performance entry. This technique works in all major browsers.
- **Redirect Start Leak:** We found one XS-Leak instance that abuses the behavior of some browsers which log too much information for cross-origin requests. The standard defines a subset of attributes that should be set to zero for cross-origin resources. However, in SA it is possible to detect if the user is redirected by the target page, by querying the Performance API and checking for the `redirectStart` timing data.
- **Duration Redirect Leak:** In GC, the `duration` for requests that result in a redirect is negative and can thus be distinguished from requests that do not result in a redirect.
- **CORP Leak:** In some cases, the `nextHopProtocol` entry can be used as a leak technique. In GC, when the CORP header is set, the `nextHopProtocol` will be empty. Note that SA will not create a performance entry at all for CORP-enabled resources.

5.3 Leak Technique: Error Messages

CORS Error. This technique allows an attacker to leak the target of a redirect that is initiated by a cross-origin site.

Details: CORS is used to explicitly allow access between cross-origin sites that would otherwise be forbidden by the SOP. `Access-Control` headers let servers describe which origins are permitted to access the response and whether credentials should be included with the request. CORS allows publicly accessible web resources to be read and used from any website. In Webkit-based browsers, it is possible to access CORS error messages when a CORS request fails. An attacker can send a CORS-enabled request to a target website which redirects based on the user state. When the browser denies the request, the full URL of the redirect target is leaked in the error message. With this attack, it is possible to detect redirects, leak redirect locations, and sensitive query parameters.

SRI Error. An attacker can leak the size of cross-origin responses due to verbose error messages.

Details: The `integrity` attribute defines a cryptographic hash by which the browser can verify that a fetched resource has not been manipulated. This security mechanism is called Subresource Integrity (SRI) [3]. It is used for integrity verification of resources served from content delivery networks (CDNs). To prevent data leaks, cross-origin resources must be CORS-enabled. Otherwise, the response is not eligible for integrity validation. Similar to the CORS error XS-Leak, it is possible to catch the error message after a fetch request with an integrity attribute fails. An attacker can forcefully trigger this error on any request by specifying a bogus hash value. In SA, this error message leaks the content length of the requested resource. An attacker can use this leak to detect differences in response size, which enables powerful XS-Leak attacks.

5.4 Leak Technique Readable Attributes

COOP. An attacker can leak if the Cross-Origin Opener Policy (COOP) header is available within an cross-origin HTTP response.

Details: Web applications can deploy COOP response header to prevent other websites from gaining arbitrary window references to the application. However, this header can easily be detected by trying to read the `contentWindow` reference. If a site only deploys COOP in one state, this property is undefined, otherwise it is defined.

6 EVALUATION RESULTS

Table 2 shows evaluation results which are automatically generated by using XSinator. We used browsers that are available on BrowserStack in our evaluation. This ensures two aspects. First, the results

	Chrome			Edge			Firefox			Opera			Safari		Tor Browser		
XS-Leak	89.0	89.0	86.0	46.3.4	90.0	46.3.7	81.1.4	87.0	33.0	60.1	75.0.3	3.0.2	14.0	14.0	10.0.15	10.0.16	10.0.16 (safer)
OS	🤖	🖥	🍎	🤖	🖥	🍎	🤖	🖥	🍎	🤖	🖥	🍎	🖥	🍎	🤖	🖥	🖥
Detectable Difference: Status Code																	
Performance API Error	○	○	●	●	○	●	○	○	●	○	○	●	●	●	○	○	○
Style Reload Error	●	●	●	●	●	●	○	○	●	●	●	●	●	●	○	○	○
Request Merging Error	●	●	●	●	●	●	○	○	●	●	●	●	●	●	○	○	○
Event Handler Error	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
MediaError	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Detectable Difference: Redirects																	
CORS Error Leak	○	○	●	○	○	●	○	○	●	○	○	●	●	●	○	○	○
Redirect Start	○	○	●	○	○	●	○	○	●	○	○	●	●	●	○	○	○
Duration Redirect	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○
Fetch Redirect	●	●	○	○	●	○	○	○	○	●	●	○	○	○	○	○	○
URL Max Length	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Max Redirect	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
History Length	●	●	●	●	●	●	○	○	●	●	●	●	●	●	○	○	○
CSP Violation	○	○	●	●	○	●	○	○	●	○	○	●	●	●	○	○	○
CSP Redirect	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	○
Detectable Difference: API Usage																	
WebSocket	●	●	○	●	●	○	●	●	○	●	●	○	○	○	●	●	●
Payment API	○	●	○	○	●	○	○	○	○	○	●	○	○	○	○	○	○
Detectable Difference: Page Content																	
Performance API Empty Page	○	○	●	○	○	●	○	○	●	○	○	●	●	●	○	○	○
Performance XSS Auditor	○	○	●	○	○	●	○	○	●	○	○	●	●	●	○	○	○
Cache	○	○	●	●	○	●	●	○	●	●	○	●	●	●	○	○	○
Frame Count	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Media Dimensions	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Media Duration	●	●	○	●	●	○	●	●	○	●	●	○	○	○	●	●	○
Id Attribute	●	●	○	●	●	○	○	○	○	●	●	○	○	○	○	○	○
CSS Property	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Detectable Difference: Header																	
SRI Error	○	○	●	○	○	●	○	○	●	○	○	●	●	●	○	○	○
Performance API Download	●	●	○	●	●	○	●	●	○	●	●	●	●	●	○	○	○
Performance API CORP	●	●	●	●	●	●	○	○	●	●	●	●	●	●	○	○	○
COOP Leak	●	●	○	○	●	○	○	○	○	○	●	○	○	○	○	○	○
Performance API XFO	○	○	●	●	○	●	○	○	●	●	○	●	●	●	○	○	○
CSP Directive	●	●	○	●	●	○	○	○	○	●	●	○	○	○	○	○	○
CORP	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
CORB	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
ContentDocument XFO	●	●	○	●	●	○	○	○	○	●	●	○	○	○	○	○	○
Download Detection	●	●	○	●	●	○	●	●	○	●	●	●	●	●	●	●	●
∑ Attackable (max. 34)	22	23	22	24	23	22	14	13	22	24	23	24	24	24	11	11	10

Table 2: Evaluation results overview categorized by its detectable differences. Successful attacks are depicted with full circuits (●), safe browser are indicated with empty circuits (○). The results for Android 🤖 in comparison with Desktop 🖥 Browsers are almost identical, while iOS 🍎 browsers behave differently. Only a few XS-Leaks are susceptible to all browsers. More detailed evaluation tables found in the Appendix, Tables 3 and 4. The used inclusion methods and leak techniques are listed in Table 1.

are generated in a fully automatic fashion; including different operating systems combined with Selenium for browser automation. Second, the results can be reproduced by other researchers. We additionally evaluated Tor using XSinator because of its privacy goals that are directly targeted with XS-Leaks. More browser results, including a real-time browser evaluation, are available on XSinator's website and in the Appendix (Tables 3 and 4).

First, we identified clear differences between each browser family, although GC and SA still have certain similarities. On mobile devices, it is evident how restricted the iOS browser ecosystem is, while browsers on Android often behave just like their desktop counterparts. Second, we analyze how vulnerabilities propagate between different browser versions over time. For this, we compared the results from XSinator for all popular desktop browsers over the last year, which gave interesting insights into the adoption of new features and the effectiveness of security patches.

6.1 Browser Comparison

Blink vs. Webkit. On the Desktop, GC 90, MSE 90, and OP 75, which are based on Chromium's Blink engine, behave equal in our test suite. Moreover, we detected that Blink-based browsers are vulnerable to the fetch redirect XS-Leak. This vulnerability was surprising since it was a known bug in SA that was fixed in February 2020 [73]. Blink- and Webkit-based browsers are vulnerable to multiple Performance API XS-Leaks. Although Google developers have been addressing theses XS-Leaks (cf. Table 3), it is still possible to detect new security headers like the CORP header. In contrast, Webkit-based browsers are still vulnerable to a variety of XS-Leaks based on the Performance API. XSinator shows the possibility to detect empty pages and therefore pages that the XSS-Auditor blocks in Webkit.

Desktop Tor vs. Firefox. Two of the main targets of Tor are to defend surveillance and resist fingerprinting. To verify its resilience to XS-Leaks, we used XSinator to evaluate the behavior of Tor in the *default* and *safer* mode. We compared Tor browser (based on FF78) with FF due to the same underlying browser engine called Gecko. Tor has more restrictive browser settings; some APIs are not activated because they are explicitly deactivated using Tor browser's configuration flags. Despite this hardening, our evaluation shows that a subset of FF XS-Leaks still works. Regarding XS-Leak attacks, Tor's *secure* mode is more restrictive than the *default* mode; for example, attackers cannot automatically fingerprint sound and image files due to an activated click-to-play functionality. However, eleven XS-Leaks like the WebSocket XS-Leak still work in Tor's *secure* mode. While FF is already quite resistant against XS-Leaks based on the Performance API leak technique, our tests show that the Performance API is completely disabled in Tor. This restriction drastically limits the attack surface, even for undiscovered XS-Leaks.

Desktop vs. Mobile. We evaluated desktop as well as mobile browsers (cf. Table 2). Most android browsers behave almost identical regarding XS-Leaks on both platforms when comparing the same browser engine versions. Given that most Android browsers are built on outdated Chromium-based browser engines, XS-Leaks are of prime importance. Our results show that an older browser engine is usually an indicator for older browser bugs. For example, the Performance API Error XS-Leak does not appear in Chrome engines since version 84. Additionally, some new functionalities like the Payment API or COOP are only available and exploitable in the newest browser engines.

Our evaluation results in Table 4 show that browsers installed in iOS behave similarly due to the same underlying browser engine. Note that the browser versions do not necessarily match cross operating systems. For example, GC 86 was the latest version on iOS while GC 89 was the latest version on Desktop and Android. Some iOS browsers do not support downloads and are thus not vulnerable to download detection XS-Leaks. The mobile and desktop versions of SA behave identically are thus vulnerable to the same XS-Leaks.

6.2 Patch History

Table 3 shows the results of popular desktop browsers over time. This timeline provides interesting insights into patch behavior and how vulnerabilities propagate between different versions. SA only patched the fetch redirect XS-Leak, while support for new features led to more XS-Leaks. FF fixed the media Error XS-Leak in version 80, and cache partitioning in version 85 mitigated the HTTP cache XS-Leak. Cache portioning was also introduced in Chromium-based browser in version 88. The Chromium developers are actively trying to combat XS-Leaks. For example, they addressed the CSP violation XS-Leak, and two XS-Leaks based on the Performance API. However, we believe that patching the Performance API Error XS-Leak in version 84 introduced the Duration Redirect, as it appeared in the same version. This assumption was later confirmed when the bug was closed in version 88 [56].

7 XS-LEAK DEFENSES

In our formal model, an XS-Leak is a function with three inputs: (1) a state-dependent resource *sdr*, (2) an inclusion method i, and (3) a leak technique t. In this section, we discuss how to mitigate the XS-Leak threat based on this definition.

The existence of state-dependent resources cannot be prevented since most web applications (except simple static sites) use them by nature. Mitigations based on a single reported state-dependent resource, for example, the status code fix of HotCRP reported in [51], may not fix the problem since typically many state-dependent resources exist in a single web application.

Several new extensions to the HTTP ecosystem have been proposed and implemented recently. For example, Same-Site Cookies set to the mode lax will prevent protected cookies from being sent, if the target web application is embedded in an iframe in a cross-site context. If a pop-up window is used as inclusion method, the Same-Site mode lax is not enough to protect cookies. Such mitigation approaches have the benefit of being standardized and will most probably work in all modern browsers. However, they may be limited to a subset of the inclusion methods.

The basic paradigm behind these mitigations is that only a single state s_0 of the web application can be reached when using a specific inclusion method i. We describe these mitigations in Section 7.1.

Our evaluation of XS-Leaks in Section 6 showed that for some XS-Leaks, only some browsers are vulnerable, and others are not (cf. Table 2). This observation gives rise to a new class of countermeasures – to investigate the differences in browser implementations and to identify the root cause for non-vulnerability. Browser implementations which are *not* vulnerable to the described XS-Leaks attacks have somehow managed to block the leak technique t used in the attack; regardless of the inclusion method used. We describe these mitigations in Section 7.2.

7.1 Inclusion Method Mitigations

Inclusion methods enable an attacker to trigger cross-origin requests on a specified state-dependent resource. To mitigate XS-Leaks on the inclusion method level, there are two possibilities. First, cross-origin requests can be allowed but they do not result in a detectable difference. Second, cross-origin requests can be denied under specific circumstances.

HTML elements. In the case of XS-Leaks requesting resources, a web application can apply different mitigations. It can use the CORP header to control if pages can embed a resource [66]. CORP can either be set to `same-origin` or `same-site` and blocks any cross-origin respectively cross-site requests to that resource.

On the client site, Chromium-based browsers use the CORB algorithm to decide whether cross-origin resource requests should be allowed or denied. CORB was primarily implemented to protect against side-channel attacks such as Spectre [35], but it mitigates XS-Leak inclusion methods in addition. For example, it protects HTML, XML, and JSON files by blocking requests such as `<img src="http://target.com/">`.

Frames. The main defense to prevent iframe elements from loading HTML resources is the usage of *X-Frame-Options*. This HTTP response header indicates whether a browser is allowed to embed a document. For example, `X-Frame-Options: DENY` mitigates any XS-Leak which relies on iframe, objects, or embed elements. Alternatively, the CSP directive *frame-ancestors* can achieve a similar result [61]. If the embedding is denied, the inclusion method cannot detect a difference in the responses.

Pop-ups. Inclusion methods that use pop-ups are more difficult to handle because a specific HTML element do not trigger them and HTTP response headers such as *X-Frame-Options* do not apply. For restricting the access to `window.opener`, the COOP HTTP response header defines three different values: `unsafe-none` (default), `same-origin-allow-popups`, and `same-origin`. These values can be used to isolate browsing tabs and pop-ups and thus, mitigates leak techniques based on pop-ups.

JavaScript Requests. Cross-origin JavaScript requests are often used in XS-Leak attacks, because an attacker has fine-grained control over the issued request. However, since these request are not CORS enabled they fall under the same restrictions as requests send by HTML elements, like scripts or images. Thus the impact of this leak technique can also be mitigated by CORP and CORB.

Generic Request Policies. In the following discussion, we shed light on browser features that help on a generic level to mitigate XS-Leaks on multiple inclusion methods.

Fetch Metadata. These request headers allow server owners to understand better how the user's browser caused a specific request. In Chrome, `Sec-Fetch-*` headers are automatically added to each request and provide metadata about the request's provenance [65]. For example, `Sec-Fetch-Dest: image` was triggered from an image element. Web applications can then choose to block requests based on that information.

Same-Site Cookies. The *Same-Site* cookie flag allows websites to declare whether a cookie should be restricted to same-site or first-party context. All major browsers support Same-Site cookies. In GC, cookies without the attribute are now `Lax` by default. For XS-Leaks, Same-Site cookies drastically limit the leak attack possibilities. On the other hand, leak techniques that rely on `window.open` still work with `SameSite=Lax`. Websites that use other authentication methods, such as client-side certificates and HTTP authentication, remain vulnerable.

Cross-Origin Identifier Unlinkability (COIU). COIU, also known as First-Party Isolation (FPI), is an optional security feature that users can enable in FF's expert settings (`about:config`) and was initially introduced in Tor Browser. In an abstract view, it is an extended same-site context. It binds multiple resources (e.g., Cookies, Cache, Client-side storages) to the first-party instead of sharing them among all visited websites. If enabled, COIU drastically decreases the applicability of XS-Leaks, since only methods using pop-ups are still possible to fit the policy's first-party requirement.

Tracking Protections. Apple implemented a privacy mechanism called Intelligent Tracking Prevention (ITP) in SA that aims to combat cross-site tracking by limiting the capabilities of cookies and other web APIs [4]. In newer versions of SA, ITP blocks all third-party cookies by default without any exceptions [74]. This blocking prevents all leaks that are not based on pop-ups. FF took a similar approach with Enhanced Tracking Prevention (ETP), but they only block specific third-party cookies belonging to tracking providers. In the context of XS-Leaks, ETP only mitigates leak techniques that target these tracking domains.

Browser Extensions. Security aware users can use browser extensions to prevent certain inclusion methods. Since numerous extensions allow controlling a browser's HTTP requests, we discuss this prevention by the example of uBlock Origin (UBO), one of the most prominent *wide spectrum blockers*.

UBO provides three [26] blocking modes, which we evaluated using XSinator in GC. UBO uses the *easy* mode in its default installation and only blocks advertisements based on URL patterns. When running UBO on XSinator, it does not block a single request and henceforth does not prevent any XS-Leak. In the *medium* mode, UBO blocks third-party frames and third-party scripts globally. This block prevents all XS-Leaks relying on inclusion methods that use HTML elements. In UBO's *hard* mode, it blocks all third-party resources. Every HTTP resource that does not belong to the first-party must be manually enabled. This blocking prevents XS-Leaks that rely on inclusion methods using JavaScript requests or other HTML tags (e.g., link), leaving only pop-up based XS-Leaks as still working. To prevent them, users can configure UBO to globally disable pop-ups.

7.2 Leak Technique Mitigations

XS-Leaks use various leak techniques for exploitation. We identified various XS-Leaks that only work in some browsers, while others are immune. In the following, we discuss mitigations that are based on the leak technique level.

Event Handler. The most prominent leak technique for XS-Leak is probably the *event handler* because event messages contained in the trigger are a rich source of information (cf. Sudhodanan et al. [51]). The most effective mitigation on this leak technique would be to deny them all, but this would break the majority of web applications on the Internet. We therefore propose to reduce the number of the necessary information that can be gathered within events. For example, the *CSP violation event* should not contain the redirection target URL in the `blockedURI` field. This behavior is implemented in FF and in newer versions of GC – only SA remains vulnerable.

Error Messages. To mitigate XS-Leaks based on the leak technique *error messages*, there are two major requirements. First, error messages must not contain detailed information, similarly to event handler messages. For example, the MediaError and CORS error XS-Leak abuses details provided in the error message, such as the HTTP response status code. Second, browsers must minimize error message occurrences. XS-Leaks such as SRI Error, ContentDocument XFO, or Fetch Redirect detect whether an error message is thrown or not. For example, GC could mimic the behavior of FF and SA, that is, they do not throw an error at all in these particular cases.

Global Limits. Fixing leak techniques that abuse *global limits* are relatively complex because they rely on physical restrictions. For example, the maximum number of available TCP connections cannot be changed. The general recommendation thereby is to restrict global limits on a small per-site basis. The WebSocket API XS-Leak leverages the shared limit of concurrent WebSocket connections; FF has a global limit of 200 connections. Upon exhausting this limit, new connections result in JavaScript exceptions. A partial mitigation is to change this limit from a global to a per-site limit, possibly with a small randomized value. The global limit for the Payment API is *one*, that is, the attacker can silently attempt to activate the WebPayment UI at any time, which only succeeds if the UI is not being used concurrently by any other tab. We recommend to only access to the Payment API when a trustworthy event [43] was used. By this means, the global limit is set to *zero* unless the user provides consent like a left mouse click on a dialog window, which sets the global limit to *one*.

Global State. Any properties of a browser's *global state* should not be accessible. For example, FF is the only browser that updates the global state *history* when a redirect occurs, which results in reading `history.length`. Browsers should create a new history property when a redirection occurs instead of storing it globally. Other examples are shared resources, such as caches. Cache leaks abuse the shared cache used for all open websites in a browser. To completely mitigate cache leak techniques, the HTTP cache must be partitioned on a per-site base, as implemented by SA, GC and FF [34]. Note that in SA iframes are not effected by cache partitioning.

Performance API. We proved the Performance API to be an excellent leak technique. In many XS-Leaks, we could detect the difference whether a cross-origin request's response has or has not a performance entry. As a unification, we recommend ensuring that all request must create such an entry and only the correct subset of timing information is recorded for cross-origin requests.

8 RELATED WORK

Our work relates to research of four different categories that we elaborate in the following. An overview of known XS-Leaks is depicted in Table 1 in Section 4.

XS-Leak Attacks. Recently, Sudhodanan et al. [51] systematically summarized prior work on XS-Leaks and presented how they can be used to infer state information from a cross-origin web application. The authors implemented a crawler-based attack vector generation tool for websites called BASTA-COSI. We considered their XS-Leaks in our work. While the authors showcased successful attacks on various websites, we contribute a comprehensive test suite for browsers and an extensive evaluation thereof. They did not consider a formal model of XS-Leaks.

Lee et al. [36] created XS-Leaks which use crafted AppCache manifest to leak redirections in state-dependent resources. Similarly, Eriksson and Sabelfeld [12] explore XS-Leak to detect redirects with the new `navigate-to` CSP directive. In 2015 Lekies et al. [37] showed how the cross-origin inclusion of dynamically generated JavaScript poses the risk of leaking sensitive information like usernames and passwords. They focused on websites and did not evaluate the behavior of different browsers. In 2018 Acar et al. [2] presented how attacker sites can attack IoT devices on the victim's LAN via a Media Error XS-Leak. Gulyas et al. [20] showed how XS-Leaks on login pages can be used for browser fingerprint web browsers. In 2019 Staicu and Pradel [50] showed how accounts on file sharing platforms can be uniquely identified by granting sole access rights to an image to a sole user and including it in the attacker website.

In Bortz and Boneh [6] analyzed *cross-site timing* attacks, in which the time a site takes to respond to a cross-site request could be used as leak technique. In 2009 Chris Evans [9] was the first described the concept of cross-site search attacks based on timing attacks. Gelernter and Herzberg [16] improved their work with attacks based on statistical tests, algorithms, and some application-specific behaviors in 2015. Van Goethem et al. [57, 58] reiterated the feasibility of fast and precise timing side-channel attacks utilizing HTML5 features to leak size, type, and transmission speed of cross-origin web resources.

Browser Security. In 2017 two white papers by Heiderich et al. [22] and Vervier et al. [60] summarized the recent threats to browser users. In 2019 Mirheidari et al. [41] showed how Web Cache Deception (WCD) can lead to caches exposing sensitive data. In 2020 Janc and West [31] discuss their plans to remove unsafe features and behaviors from the web platform. Calzavara et al. [8] highlighted how inconsistent framing policies of websites in the wild can impede browser security. Narayan et al. [42] proposed changes to the Firefox source code to mitigate binary exploitation through web-served passive media, an orthogonal threat besides XS-Leak. Karami et al. [33] researched how browser extensions can be fingerprinted by enumeration of its Web Accessible Resources (WARs) and behavior. Roth et al. [45] revealed the challenges posed to website maintainers by the complex and ever-developing Content Security Policy (CSP).

Online Evaluation Test Suites Schwenk et al. [46] provided a test suite for stress-testing the SOP. The tabular presentation of historical data collected across different browsers allows to quickly identify outliers. We used their concept as a basis for XSinator. As a collection of test suites for testing browsers based on W3C/WHATWG specifications, Web Platform Tests [49] can be used.

Formal Models. In 2005 Gross et al. [17] formally model a web browser for the purpose of analyzing browser-based protocols. In 2014 Fett et al. [14] formalized an expressive model of the Web Infrastructure based a model for public key cryptography by Dolev and Yao [11]. In 2017 Schwenk et al. [46] published a formal model for the SOP and systematically analyzed and highlighted problematic differences between implementations across browser families. Our proposed model takes the SOP into consideration as an important factor in the context of XS-Leaks, as it is more concise and applicable than the general formalization of a whole web browser.

9 CONCLUDING REMARKS

In this paper, we proposed a formal model for XS-Leaks. This model fits all previously published results. By identifying the three ingredients of an XS-Leak, which are the *detectable difference* D, *inclusion method* i and *leak technique* t, we were able to gain novel in-depth insights and systematically produce new attacks. (1) To detect a difference D, different inclusion methods i and leak techniques t can be used. This yields a systematic way for detecting 14 new XS-Leaks. (2) Our evaluation of existing browser/OS combinations with XSinator showed that non-vulnerable browser implementations for each class of XS-Leak exist (except in 5 cases). (3) This finding allowed us to investigate a new class of XS-Leak mitigations that target the used leak techniques. Our results show that a small, dedicated formal model may help to develop a more thorough understanding of web attacks.

Future Work. While the set I of inclusion methods is relatively static, new web technologies may introduce novel detectable differences, and novel leak techniques. Our contributed formal model provides a clear methodology to check such novel technologies for possible XS-Leaks. A similar observation holds for the mitigations. The inclusion method-based countermeasures should be re-evaluated once new inclusion methods are defined, and browser implementations need to be constantly re-evaluated once their functionality is enhanced. Moreover, we think that the community could benefit from an evaluation whether fixes against XS-Leaks can be done without interfering with Web functionality. However, crawling sites would probably yield only incomplete data, since many sites hide their functionalities behind logins or on sub-sites. Therefore, it would be a difficult but commendable future work to look on web applications, identify their states, and solve session management problems.

ACKNOWLEDGEMENTS

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. Lukas Knittel was supported by the German Federal Ministry of Economics and Technology (BMWi) project "Industrie 4.0 Recht-Testbed" (13I40V002C). Dominik Noss was supported by the research project "MITSicherheit.NRW" funded by the European Regional Development Fund North Rhine-Westphalia (EFRE.NRW).

REFERENCES

[1] Gunes Acar and Frank Li of UC Berkeley Danny Y. Huang, Princeton University. 2018. MediaError message property leaks cross-origin response status. https://bugs.chromium.org/p/chromium/issues/detail?id=828265. (April 2018).

[2] Gunes Acar, Danny Yuxing Huang, Frank Li, Arvind Narayanan, and Nick Feamster. 2018. Web-Based Attacks to Discover and Control Local IoT Devices. In *Proceedings of the 2018 Workshop on IoT Security and Privacy (IoT S&P '18)*. Association for Computing Machinery, New York, NY, USA, 29–35. https://doi.org/10.1145/3229565.3229568

[3] Devdatta Akhawe, Frederik Braun, Francois Marier, and Joel Weinberger. 2016. *Subresource Integrity*. W3C Recommendation. W3C. https://www.w3.org/TR/2016/REC-SRI-20160623/.

[4] Apple. 2019. Safari Privacy Overview. https://www.apple.com/safari/docs/Safari_White_Paper_Nov_2019.pdf. (November 2019).

[5] A. Barth. 2011. *The Web Origin Concept*. RFC 6454. IETF. http://tools.ietf.org/rfc/rfc6454.txt

[6] Andrew Bortz and Dan Boneh. 2007. Exposing Private Information by Timing Web Applications. In *Proceedings of the 16th International Conference on World Wide Web (WWW '07)*. Association for Computing Machinery, New York, NY, USA, 621–628. https://doi.org/10.1145/1242572.1242656

[7] Frederik Braun and Mario Heiderich. 2013. X-frame-options: All about clickjacking. (2013). https://cure53.de/xfo-clickjacking.pdf

[8] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. 2020. A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 683–697. https://www.usenix.org/conference/usenixsecurity20/presentation/calzavara

[9] Chris Evans. 2009. Cross-domain search timing. (2009). https://scarybeastsecurity.blogspot.com/2009/12/cross-domain-search-timing.html

[10] Chrome Platform Status. 2020. XSS Auditor (removed). https://www.chromestatus.com/feature/5021976655560704. (June 2020).

[11] D. Dolev and A. Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory* 29, 2 (1983), 198–208. https://doi.org/10.1109/TIT.1983.1056650

[12] Benjamin Eriksson and Andrei Sabelfeld. 2020. Autonav: Evaluation and automatization of web navigation policies. In *Proceedings of The Web Conference 2020*. 1320–1331.

[13] Chris Evans. 2008. Cross-domain leaks of site logins. https://scarybeastsecurity.blogspot.com/2008/08/cross-domain-leaks-of-site-logins.html. (August 2008).

[14] Daniel Fett, Ralf Kuesters, and Guido Schmitz. 2014. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. (2014). arXiv:cs.CR/1403.1866

[15] I. Fette and A. Melnikov. 2011. *The WebSocket Protocol*. RFC 6455. IETF. http://tools.ietf.org/rfc/rfc6455.txt

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

[17] Thomas Gross, Birgit Pfitzmann, and Ahmad-Reza Sadeghi. 2005. Browser Model for Security Analysis of Browser-Based Protocols. *IACR Cryptology ePrint Archive* 2005, 127. https://doi.org/10.1007/11555827_28

[18] Jeremiah Grossman. 2012. I Know What Websites You Are Logged-In To (Login-Detection via CSRF). `http://blog.whitehatsec.com/i-know-what-websites-you-are-logged-in-to-login-detection-via-csrf/`. (October 2012).

[19] Chong Guan, Kun Sun, Zhan Wang, and WenTao Zhu. 2016. Privacy breach by exploiting postmessage in html5: Identification, evaluation, and countermeasure. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 629–640.

[20] Gabor Gyorgy Gulyas, Doliere Francis Some, Nataliia Bielova, and Claude Castelluccia. 2018. To extend or not to extend: on the uniqueness of browser extensions and web logins. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*. 14–27.

[21] Mario Heiderich. 2020. HTTPLeaks. https://github.com/cure53/HTTPLeaks. (June 2020).

[22] Mario Heiderich, Alex Inführ, Fabian Fäßler, Nikolei Krein, Masato Kinugawa, Tsang-Chi Hong, Dario Weißler, and Paula Pustulka. 2017. Cure53's Browser Security White Paper. (2017). https://raw.githubusercontent.com/cure53/browser-sec-whitepaper/master/browser-security-whitepaper.pdf

[23] Luan Herrera. 2020. XS-Leaks in redirect flows. https://docs.google.com/presentation/d/1rlnxXUYHY9CHgCMckZsCGH4VopLo4DYMvAcOltma0og. (January 2020).

[24] Gareth Heyes. 2015. Abusing Chrome's XSS auditor to steal tokens. https://portswigger.net/research/abusing-chromes-xss-auditor-to-steal-tokens. (August 2015).

[25] Gareth Heyes. 2019. XS-Leak: Leaking IDs using focus. https://portswigger.net/research/xs-leak-leaking-ids-using-focus. (October 2019).

[26] Raymond Hill. 2020. Blocking mode uBlock Wiki. https://github.com/gorhill/uBlock/wiki/Blocking-mode. (October 2020).

[27] Egor Homakov. 2013. Disclose domain of redirect destination taking adventadge of CSP. https://bugs.chromium.org/p/chromium/issues/detail?id=313737. (October 2013).

[28] Ian Jacobs, Zach Koch, Domenic Denicola, Roy McElmurry, Rouslan Solomakhin, and Marcos Caceres. 2019. *Payment Request API*. Candidate Recommendation. W3C. https://www.w3.org/TR/2019/CR-payment-request-20191212/#show-method.

[29] Arvind Jain, Zhiheng Wang, Anderson Quach, Jatinder Mann, and Todd Reifsteck. 2017. *Resource Timing Level 1*. Candidate Recommendation. W3C. https://www.w3.org/TR/2017/CR-resource-timing-1-20170330/#resources-included.

[30] Artur Janc, Krzysztof Kotowicz, Lukas Weichselbaum, and Roberto Clapis. 2020. Information Leaks via Safari's Intelligent Tracking Prevention. https://arxiv.org/abs/2001.07421. (January 2020).

[31] Artur Janc and Mike West. 2020. Oh, the Places You'll Go! Finding Our Way Back from the Web Platform's Ill-conceived Jaunts. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 673–680.

[32] 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)*.

[33] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. 2020. Carnus: Exploring the Privacy Threats of Browser Extension Fingerprinting. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society.

[34] Eiji Kitamura. 2020. Gaining security and privacy by partitioning the cache. https://developers.google.com/web/updates/2020/10/http-cache-partitioning. (October 2020).

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[36] Sangho Lee, Hyungsub Kim, and Jong Kim. 2015. Identifying Cross-origin Resource Status Using Application Cache. In *22nd Network and Distributed System Security Symposium (NDSS 2015)*. https://www.microsoft.com/en-us/research/publication/identifying-cross-origin-resource-status-using-application-cache/

[37] 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. https://publications.cispa.saarland/987/ pub_id: 1055 Bibtex: lekies2015unexpected URL date: None.

[38] Ron Masas. 2019. Browser Side Channels. https://github.com/xsleaks/xsleaks/wiki/Browser-Side-Channels. (September 2019).

[39] Ron Masas. 2019. Server Side Redirect Detection. https://xsleaks.github.io/xsleaks/examples/redirect/. (September 2019).

[40] MDN web docs. 2020. Fetch API. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API. (June 2020).

[41] Seyed Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. 2019. Cached and Confused: Web Cache Deception in the Wild. (12 2019).

[42] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 699–716. https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[43] Marcus Niemietz and Jörg Schwenk. 2018. Out of the Dark: UI Redressing and Trustworthy Events. In *Cryptology and Network Security*, Srdjan Capkun and Sherman S. M. Chow (Eds.). Springer International Publishing, Cham, 229–249.

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

[45] S. Roth, Timothy Barron, S. Calzavara, Nick Nikiforakis, and Ben Stock. 2020. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies. In *NDSS*.

[46] 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.

[47] 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)*.

[48] Jungkee Song, Alex Russell, Marijn Kruisselbrink, and Jake Archibald. 2019. *Service Workers 1*. Candidate Recommendation. W3C. https://www.w3.org/TR/2019/CR-service-workers-1-20191119/.

[49] Web Platform Tests / Open Source. 2021. Web Platform Tests Github Page. (2021). https://github.com/web-platform-tests/wpt

[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] Avinash Sudhodanan, Soheil Khodayari, and Juan Caballero. 2020. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In *27nd Network and Distributed System Security Symposium (NDSS 20)*.

[52] Takeshi Terada. 2014. Security: XSS filter information leak. https://bugs.chromium.org/p/chromium/issues/detail?id=396544. (July 2014).

[53] Terjanq. 2019. Mass XS-Search using Cache Attack. https://medium.com/@terjanq/massive-xs-search-over-multiple-google-products-416e50dd2ec6. (November 2019).

[54] terjanq. 2019. Protected tweets exposure through the URL. https://hackerone.com/reports/491473. (April 2019).

[55] terjanq. 2019. Twitter: Detect X-Frame-Options header in Chrome. https://twitter.com/terjanq/status/1111600071014080517. (March 2019).

[56] terjanq. 2020. Issue 1157818: performance API reveals information about redirects (XS-Leak). https://crbug.com/1157818. (December 2020).

[57] 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*. ACM, 1382–1393.

[58] 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.

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

[60] Markus Vervier, Michele Orrù, Berend-Jan Wever, and Eric Sesterhenn. 2017. Cure53's Browser Security White Paper. (2017). https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf

[61] MDN web docs. 2019. CSP: frame-ancestors. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors. (November 2019).

[62] MDN web docs. 2020. Web APIs: History. https://developer.mozilla.org/en-US/docs/Web/API/History. (February 2020).

[63] Mike West. 2013. Cross-origin leakage with securitypolicyviolation events and paths in source expressions. https://lists.w3.org/Archives/Public/public-webappsec/2013May/0022.html. (May 2013).

[64] Mike West. 2018. *Content Security Policy Level 3*. W3C Working Draft. W3C. https://www.w3.org/TR/2018/WD-CSP3-20181015/.

[65] Mike West. 2020. Fetch Metadata Request Headers. https://w3c.github.io/webappsec-fetch-metadata/. (April 2020).

[66] Web Hypertext Application Technology Working Group (WHATWG). 2020. Fetch - Living Standard: Cross-Origin-Resource-Policy header. https://fetch.spec.whatwg.org/#cross-origin-resource-policy-header. (August 2020).

[67] Web Hypertext Application Technology Working Group (WHATWG). 2020. Fetch - Living Standard: HTTP-redirect fetch. https://fetch.spec.whatwg.org/#http-redirect-fetch. (August 2020).

[68] Web Hypertext Application Technology Working Group (WHATWG). 2020. Fetch - Living Standard: Requests. https://fetch.spec.whatwg.org/#requests. (August 2020).

[69] Web Hypertext Application Technology Working Group (WHATWG). 2020. HTML - Living Standard: contentDocument. https://html.spec.whatwg.org/multipage/iframe-embed-object.html#dom-iframe-contentdocument. (August 2020).

[70] Web Hypertext Application Technology Working Group (WHATWG). 2020. HTML - Living Standard: Navigating to a fragment. https://html.spec.whatwg.org/multipage/browsing-the-web.html#scroll-to-fragid. (August 2020).

[71] XS-Leaks Wiki. 2020. CORB Leaks. https://xsleaks.com/docs/attacks/browser-features/corb/. (October 2020).

[72] XS-Leaks Wiki. 2020. CORP Leaks. https://xsleaks.com/docs/attacks/browser-features/corp/. (October 2020).

[73] John Wilander. 2019. Preventing Tracking Prevention Tracking. https://webkit.org/blog/9661/preventing-tracking-prevention-tracking/. (December 2019).

[74] John Wilander. 2020. Full Third-Party Cookie Blocking and More. https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/. (March 2020).

[75] Gilbert Wondracek, Thorsten Holz, Engin Kirda, and Christopher Kruegel. 2010. A practical attack to de-anonymize social network users. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 223–238.

[76] Takashi Yoneuchi. 2019. XS-Leak with Resource Timing API and CSP Embedded Enforcement. https://crbug.com/1105875. (July 2019).

A OVERVIEW OF XS-LEAK ATTACKS

A.1 Detectable Difference: Status Code

An attacker is able to distinguish different HTTP response status codes to detect server errors, client errors, authentication errors, or server redirects cross-origin.

Leak Technique: Event Handler. Event handlers can be attached to an HTML tag which includes a cross-origin resource. Depending on the status code, content-type, and other response headers, different events handlers are triggered (e.g., `onload` or `onerror`) [50, 51]. By using event handler as a leak technique, an attacker is allowed to detect the presence of headers, HTTP errors, and media types.

Leak Technique: MediaError. In FF, it is possible to accurately leak a cross-origin request's status code by using an error message leak technique generated by a video or audio element as inclusion methods.

A.2 Detectable Difference: API Usage

An attacker is able to detect the usage of Web APIs cross-origin.

Service Worker. Service workers are event-driven script contexts that run at an origin [48]. They run in the background of a web page and can intercept, modify, and cache resources to create offline web application. Karami et al. [32] introduced leak techniques to detect if a service worker is registered for a specific origin. They used iframes as an inclusion method on resources that have previously been cached by a service worker.

A.3 Detectable Difference: Redirects

It is possible to detect if a web application has navigated the user to a different page. This is not limited to HTTP redirects but redirects triggered by JavaScript or HTML code can also be detected.

Fetch Redirect. The Fetch API provides an interface for fetching resources by using JavaScript code. The API allows various configuration options. One of these options is the *redirect mode* [68]. If it is set to `manual`, redirects are not automatically followed. Based on the discussion of Janc et al. [30], browsers like GC and SA allow the attacker to check the response's type as a leak technique after the redirect is finished.

URL Max Length. Web applications may not adequately handle long URLs. Usually, web servers reject a request when the URL exceeds a specific limit and return an error with status code `414 URI Too Long` or `413 Payload Too Large`. Modern browsers can typically handle longer URLs, although 2 megabytes are a common limit. If a web application redirects, it may try to preserve any query parameters that are attached to the original request (e.g., as implemented by `twitter.com`). Additionally, the URL length may increase in size when a redirect occurs. By considering a global limit as a leak technique, an attacker can consequently craft a request that exceeds the server's URL character limit to trigger an error once redirected. As an example, Masas [38, 39] showed how an attacker could gather the length of a URL that triggers an error on a specific server to be used within an XS-Leak.

Max Redirect Browsers try to prevent infinite redirect loops. The Fetch standard specifies that after *twenty* redirects a network error should be returned [67]. Herrera [23] showed that this limit can be abused to detect the number of redirects of a cross-origin resource. For example, an attacker can redirect nineteen times before redirecting to the target site that may potentially redirect. If it redirects, an error is triggered that can then be detected by the attacker page.

History Length. The *History API* allows JavaScript code to manipulate the browser history, which saves the pages visited by a user [62]. An attacker can use the `length` property as an inclusion method: to detect JavaScript and HTML navigation. Multiple works have studied the browser history and show how to abuse it to determine whether a user has accessed a certain website [44, 47, 54, 75].

CSP Violation. An XS-Leak can use the CSP [64] to detect if a cross-origin site was redirected to a different origin. This leak can detect the redirect, but additionally, the domain of the redirect target leaks. These techniques were discussed by West [63] and Homakov [27]. The basic idea of this attack is to allow the target domain on the attacker site. Once a request is issued to the target domain, it redirects to a cross-origin domain. CSP blocks the access to it and creates a violation report used as a leak technique. Depending on the browser, this report may leak the target location of the redirect.

CSP Detection. Newer browsers do not leak the target location of the redirect in the violation report. However it is still possible to detect if a cross-origin redirect occurred, because the request is still blocked by the CSP, i.e., the violation report can be used as a leak technique.

A.4 Detectable Difference: Page Content

An attacker can detect content in HTML documents or resources. The detection includes HTML attributes, embedded resources, and CSS rules.

Cache. For avoiding unnecessary data transfers and server requests, browsers implement *HTTP caching* to increase performance when loading web pages. Vela [59] discovered that most browsers use one shared cache for all websites. Regardless of their origin, it is possible to deduct whether a target page has requested a specific file. To detect this request, the attacker page executes the following leak technique it clears the file from the cache, it opens the target page in a pop-up or iframe, and finally, it checks if the file is present in the cache again.

Frame Count. HTML iframe elements can be used to embed other documents within the current one. Developers use them to isolate third-party content or to include widgets and advertisements. Although JavaScript APIs only allow limited access to cross-origin window objects, the number of frames on a page can still be read. As shown by Grossman [18] and Masas [38], an attacker can use this length to detect state differences if the number of frames changes between them. A target web page may include a different number of frames depending on the user state. With readable attributes as a leak technique, properties like `length` help an attacker to leak the frame number if it is possible to obtain a window handle to the target. A window handle can be obtained by the following inclusion methods: embedding the target page in an iframe or by opening a pop-up using `window.open`.

Media Dimensions. Media elements, such as video and image, can sometimes differ in size. Web applications may dynamically generate media files depending on user information or add watermarks that change media size. An attacker can use standard DOM APIs to detect these differences [38]. For example, media resources can be embedded cross-origin with `<video>` or `<img>` HTML tags and properties such as the dimensions for image elements can be read.

Media Duration. Similarly, the duration of audio and video elements can be read cross-origin.

Id Attribute. Hyperlinks are often used to link to specific parts of a document, and browsers will automatically scroll to them when the identifier is specified in the fragment part of a URL [70]. This behavior can be used to detect if a specific identifier is present on a page. For certain HTML elements as inclusion methods, GC and SA will not only scroll to them but also focus them. For example, `<input id="leakme">` will gain focus when the fragment of the URL is set to `leakme`. In GC, this even works in cross-origin iframes. However, an attacker can not directly detect that the element got focused because of the SOP. To get around this, an `onblur` handler can be registered on the attacker page. Once the iframe receives focus, the attacker page will lose focus and a *blur* event triggers. Heyes [25] discovered this leak technique and it can be used to detect login pages or to leak sensitive data from the `id` or `name` attribute.

CSS Property. Web applications may change website styling depending on the status of the user. As described by Evans [13], these changes can be used to detect differences in Cascading Style Sheets (CSS) rules. Cross-origin CSS files can be embedded on the attacker page with the HTML `link` element, and the rules will be applied to the attacker page. If a page dynamically changes these rules, an attacker can detect these differences depending on the user state. For example, websites will often serve different CSS depending on whether the user is logged in. As a leak technique, the attacker can include the targeted CSS file to use the `window.getComputedStyle` method to read CSS properties of a specific HTML element. As a result, an attacker can read arbitrary CSS properties if the affected element and property name is known.

A.5 Detectable Difference: Header

In some cases, the presence of HTTP headers can be detected. This includes headers such as `X-Frame-Options`, `Content-Type`, and `Content-Disposition`.

Performance API XFO The Performance API is used to access performance information of the current page [29]. This includes detailed network timing data for the document and every resource the page loads. Terjanq [55] showed that when a resource has `X-Frame-Options` header set and is included with an object tag, it will not create an resource timing entry in the Performance API (cf. Section 5.2).

CSP Directive. A new feature in GC allows web pages to proposes a CSP by setting an attribute on an iframe element. The policy directives are transmitted along with the HTTP request. Normally, the embedded content must explicitly allow this with an HTTP header, otherwise an error page is displayed. However, if the iframe already ships a CSP and the new policy is not stricter, the page will display normally.

This allows an attacker to detect specific CSP directive of a cross-origin page, if it is possible to detect the error page. This leak was reported to GC by Takashi Yoneuchi [76]. Although, this bug is now marked as fixed, we found a new leak technique that can detect the error page, because the underlying problem was never fixed.

CORP. The CORP header is a relatively new web platform security feature that when set blocks no-cors cross-origin requests to the given resource (cf. Section 7). The presence of the header can be detected, because a resource protected with CORP will throw an error when fetched.

CORB. CORB is an algorithm in the browser that blocks *dubious* cross-origin resource requests before they reach the webserver (cf. Section 7). This feature can be used to detect the presence of `Content-Type` and `Content-Type-Options` headers, because CORB is only enforced for specific content-types together with the nosniff option. An attacker can use a combination of event handlers as a leak technique to detect CORB.

ContentDocument XFO. In Chrome, when a page is not allowed to be embedded on a cross-origin page, because the X-Frame-Options (XFO) header is set to `deny` or `same-origin`, an error page is shown instead. For objects, this error page can be detected by checking the `contentDocument` property [69]. Typically, this property returns `null` because access to a cross-origin embedded document is not allowed. However, due to Chrome's rendering of the error page, an empty document object is returned instead. This does not work for iframes or in other browsers. Developers may forget to set `X-Frame-Options` for all pages and especially error pages often miss this header. As a leak technique, an attacker may be able to differentiate between different user states by checking for it.

Download Detection. The `Content-Disposition` header indicates if the browser is either supposed to downloaded content or displayed it *inline*. Masas [38] has demonstrated with a leak technique that an attacker can detect downloads by using the inclusion method: iframe. If the iframe is still accessible, the file was downloaded; this is the case because in most browsers a download does not trigger a navigation and the iframe is still considered same-origin. This attack also works with pop-ups created with `window.open`.

Browser	Chrome and Edge							Firefox			Safari			
XS-Leak	80.0	81.0	83.0	84.0	85.0 - 87.0	88.0	89.0 - 90.0	79.0	80.0 - 84.0	85.0 - 88.0	11.1	12.1	13.1	14.0
Detectable Difference: Status Code														
Performance API Error	●	●	●	○	○	○	○	○	○	○	●	●	●	●
Style Reload Error	●	●	●	●	●	●	●	○	○	○	●	●	●	●
Request Merging Error	●	●	●	●	●	●	●	○	○	○	●	●	●	●
Event Handler Error	●	●	●	●	●	●	●	●	●	●	●	●	●	●
MediaError	○	○	○	○	○	○	○	●	○	○	○	○	○	○
Detectable Difference: Redirects														
CORS Error Leak	○	○	○	○	○	○	○	○	○	○	●	●	●	●
Redirect Start	○	○	○	○	○	○	○	○	○	○	●	●	●	●
Duration Redirect	○	○	○	●	●	○	○	○	○	○	○	○	○	○
Fetch Redirect	○	●	●	●	●	●	●	○	○	○	●	○	○	○
URL Max Length	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Max Redirect	●	●	●	●	●	●	●	●	●	●	○	○	○	●
History Length	●	●	●	●	●	●	●	○	○	○	●	●	●	●
CSP Violation	●	●	●	●	○	○	○	○	○	○	●	●	●	●
CSP Redirect	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Detectable Difference: API Usage														
WebSocket	●	●	●	●	●	●	●	●	●	●	○	○	○	○
Payment API	●	●	●	●	●	●	●	○	○	○	○	○	○	○
Detectable Difference: Page Content														
Performance API Empty Page	○	○	○	○	○	○	○	○	○	○	●	●	●	●
Performance XSS Auditor	○	○	○	○	○	○	○	○	○	○	●	●	●	●
Cache	●	●	●	●	●	○	○	●	●	○	●	●	●	●
Frame Count	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Media Dimensions	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Media Duration	●	●	●	●	●	●	●	●	●	●	○	○	○	○
Id Attribute	●	●	●	●	●	●	●	○	○	○	○	○	○	○
CSS Property	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Detectable Difference: Header														
SRI Error	○	○	○	○	○	○	○	○	○	○	○	○	●	●
Performance API Download	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Performance API CORP	●	●	●	●	●	●	●	○	○	○	○	●	●	●
COOP Leak	○	○	●	●	●	●	●	○	○	○	○	○	○	○
Performance API XFO	●	●	●	●	●	●	○	○	○	○	●	●	●	●
CSP Directive	●	●	●	●	●	●	●	○	○	○	○	○	○	○
CORP	●	●	●	●	●	●	●	●	●	●	○	●	●	●
CORB	●	●	●	●	●	●	●	●	●	●	●	●	●	●
ContentDocument XFO	●	●	●	●	●	●	●	○	○	○	○	○	○	○
Download Detection	●	●	●	●	●	●	●	●	●	●	○	●	●	●
∑ Attackable (max. 34)	25	26	27	27	26	24	23	15	14	13	20	22	23	24

Table 3: Evaluation results for popular desktop browsers show how vulnerabilities propagate between different versions. Browser versions that did not show any differences have been merged. Chrome and Edge behave identically.

iOS Version	iOS 14							iOS 13	iOS 12	iOS 11
XS-Leak	Safari 14.0	Opera 3.0.2	Chrome 87.0	Chrome 86.0	Firefox 33.0	Edge 46.3.7	Firefox Focus 8.1.7	Safari 13.0	Safari 12.1	Safari 11.0
Detectable Difference: Status Code										
Performance API Error	●	●	●	●	●	●	●	●	●	●
Style Reload Error	●	●	●	●	●	●	●	●	●	●
Request Merging Error	●	●	●	●	●	●	●	●	●	●
Event Handler Error	●	●	●	●	●	●	●	●	●	●
MediaError	○	○	○	○	○	○	○	○	○	○
Detectable Difference: Redirects										
CORS Error Leak	●	●	●	●	●	●	●	●	●	○
Redirect Start	●	●	●	●	●	●	●	●	●	●
Duration Redirect	○	○	○	○	○	○	○	○	○	○
Fetch Redirect	○	○	○	○	○	○	○	○	○	●
URL Max Length	●	●	●	●	●	●	●	●	●	●
Max Redirect	●	●	●	●	●	●	●	○	○	○
History Length	●	●	●	●	●	●	○	●	●	●
CSP Violation	●	●	●	●	●	●	●	●	●	●
CSP Redirect	●	●	●	●	●	●	●	●	●	●
Detectable Difference: API Usage										
WebSocket	○	○	○	○	○	○	○	○	○	○
Payment API	○	○	○	○	○	○	○	○	○	○
Detectable Difference: Page Content										
Performance API Empty Page	●	●	●	●	●	●	●	●	●	●
Performance XSS Auditor	●	●	●	●	●	●	●	●	●	●
Cache	●	●	●	●	●	●	○	●	●	●
Frame Count	●	●	●	●	●	●	○	●	●	●
Media Dimensions	●	●	●	●	●	●	●	●	●	●
Media Duration	○	○	○	○	○	○	○	○	○	○
Id Attribute	○	○	○	○	○	○	○	○	○	○
CSS Property	●	●	●	●	●	●	●	●	●	●
Detectable Difference: Header										
SRI Error	●	●	●	●	●	●	●	○	○	○
Performance API Download	●	●	○	○	○	○	○	●	○	○
Performance API CORP	●	●	●	●	●	●	●	●	●	○
COOP Leak	○	○	○	○	○	○	○	○	○	○
Performance API XFO	●	●	●	●	●	●	●	●	●	●
CSP Directive	○	○	○	○	○	○	○	○	○	○
CORP	●	●	●	●	●	●	●	●	●	○
CORB	●	●	●	●	●	●	●	●	●	●
ContentDocument XFO	○	○	○	○	○	○	○	○	○	○
Download Detection	●	●	○	○	○	○	○	●	○	○
∑ Attackable (max. 34)	24	24	22	22	22	22	19	22	20	18

Table 4: Evaluation results of 10 iOS browsers. All browsers based on new versions of WebKit mostly behave identically except for Firefox Focus. Older version closely match the desktop Safari behavior (cf. Tables 3).