The Leaky Web: Automated Discovery of Cross-Site Information Leaks in Browsers and the Web

Jannis Rautenstrauch, Giancarlo Pellegrino, Ben Stock
CISPA Helmholtz Center for Information Security
{jannis.rautenstrauch,pellegrino,stock}@cispa.de

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179311

***Abstract*—When browsing the web, none of us want sites to infer which other sites we may have visited before or are logged in to. However, attacker-controlled sites may infer this state through browser side-channels dubbed Cross-Site Leaks (XS-Leaks). Although these issues have been known since the 2000s, prior reports mostly found individual instances of issues rather than systematically studying the problem space. Further, actual impact in the wild often remained opaque.**

To address these open problems, we develop the first automated framework to systematically discover observation channels in browsers. In doing so, we detect and characterize 280 observation channels that leak information cross-site in the engines of Chromium, Firefox, and Safari, which include many variations of supposedly fixed leaks. Atop this framework, we create an automatic pipeline to find XS-Leaks in real-world websites. With this pipeline, we conduct the largest to-date study on XS-Leak prevalence in the wild by performing *visit inference* and a newly proposed variant *cookie acceptance inference* attack on the Tranco Top10K. In addition, we test 100 websites for the classic XS-Leak attack vector of *login detection*.

Our results show that XS-Leaks pose a significant threat to the web ecosystem as at least 15%, 34%, and 77% of all tested sites are vulnerable to the three attacks. Also, we present substantial implementation differences between the browsers resulting in differing attack surfaces that matter in the wild. To ensure browser vendors and web developers alike can check their applications for XS-Leaks, we open-source our framework and include an extensive discussion on countermeasures to get rid of XS-Leaks in the near future and ensure new features in browsers do not introduce new XS-Leaks.

I. Introduction

Every day we perform numerous activities online that we do not want to be publicly known. One would expect that only the website and its partners know about these activities. However, privacy-invasive leaks of information to other websites opened in the same browser have existed since the dawn of the web and are known as Cross-Site leaks (XS-Leaks) [58]. These leaks are a never-ending problem for both websites and browsers. Websites try to mitigate high-impact information leaks on their site [35]. However, due to browser differences, the complexity of modern websites, and ever newly discovered observation channels, these fixes are usually incomplete, and information still leaks in other places on the site [34], [64].

Although recent works provided the first steps into a more systematic study of XS-Leaks by introducing formal models fitting all known XS-Leaks [31], [66] and evaluating all known leaks in different web browsers [31], they did not solve the problem that all prior works are only manually discovering individual XS-Leak instances. The focus on individual XS-Leaks is insufficient to create a shared understanding of XS-Leaks in the web ecosystem. Furthermore, it often leads to incomplete fixes of both XS-Leaks on websites and bugs in browsers as only the reported test cases are validated. Additionally, the current model is purely reactive instead of preemptive, and many XS-Leaks are only discovered years after a feature was introduced.

What is needed is a systematic testing framework to focus on the bigger picture of XS-Leaks in the web ecosystem. The framework should be comprehensive and explainable, allowing for complete fixes and mitigations. In addition, it should be easily extensible, making it possible to use it preemptively for new features. Furthermore, a measurement of how often different XS-Leaks occur in the wild is needed. Such a measurement shows how big of a problem XS-Leaks are and gives priorities to browser vendors in which leaks to fix first.

In this paper, we propose the first systematic framework to automatically discover possible cross-site information leakage, called *observation channels*, in browsers without a priori knowledge of XS-Leaks and browser behavior. The main insight of this framework is that instead of manually searching for single response pairs that can be distinguished cross-site by a browser API, one can systematically observe the browser behavior for thousands of cross-site responses and dozens of browser APIs. Then, we automatically summarize which response information is distinguishable by each API by relying on binary decision trees, which allow for easy comprehension by humans. We implemented a prototype implementation of our framework. With it, we discovered 280 observation channels that leak cross-site information in the engines of Chromium, Firefox, and Safari (we use Playwright to test the underlying engines, which are dubbed Chromium, Firefox, and WebKit there). The summaries show which information leaks through each channel. They characterize the exact behavior of known channels such as *image-event handlers* [22], reveal major differences between the browser families, and show that thought-to-be-fixed observation channels such as mediaError [1], [2] still leak information.

To show the impact the discovered channels have in the wild and rank them by severity, we perform the largest to-date study on the prevalence of XS-Leaks by testing how often *visit inference* using the discovered channels works on the Tranco Top10K [32]. We also find that the classical *visit inference* attack is overhauled as most websites cannot be used without

© 2023, Jannis Rautenstrauch. Under license to IEEE.
DOI 10.1109/SP46215.2023.00067

accepting cookies first and propose a new attack variant called *cookie acceptance inference*. We find 15% of all tested sites vulnerable to visit inference and 34% of all tested sites vulnerable to cookie acceptance inference. In addition, we run a small-scale semi-manual *login detection* experiment on the top 100 sites where we could successfully log in. We find 77% of sites vulnerable to this more sophisticated attack. These results highlight that XS-Leaks constitute a significant threat to the web ecosystem. The results also show that the browser differences discovered in the observation channels matter as many websites are only vulnerable in some browsers. To guide the way to an XS-Leak-free future, we include an extensive discussion on possible countermeasures and open-source our tools such that browser vendors and website developers can minimize XS-Leaks issues. Furthermore, at the time of this writing, we are discussing with the affected vendors how they can incorporate our tests to avoid leaks in their products.

To sum up, our paper makes the following contributions:

- We propose a generalized concept of observation channels that models cross-site information leakage in browsers without the need for state-dependent URLs (Section III).
- We propose the first framework for the automatic discovery and characterization of cross-site information leakage in browsers. We create a prototype implementation for all major browsers and use it to discover 280 observation channels that leak information in the engines of Chromium, Firefox, and Safari (Section III).
- We perform the largest to-date study on the prevalence of XS-Leaks in the wild for *visit inference* attacks. Moreover, we introduce a new, more realistic attack variant *cookie acceptance inference*. We show that XS-Leaks are a significant threat to the web, with 15% and 34% of top 10k sites being vulnerable to the two attacks. Additionally, we perform the first post-SameSite lax study on *login detection* through XS-Leaks showing that 77/100 top-ranked sites are vulnerable (Section IV).
- We share the collected insights that the observed differences between browsers matter for users' privacy and that the current focus on single responses is a misleading route to achieving security and uniformity in browsers. In addition, we discuss countermeasures against XS-Leaks and suggest a way to minimize XS-Leaks (Section V).
- We make our tools [54] available for review and to foster future research and enable web developers and browser vendors to search for XS-Leaks issues.

II. Background: XS-Leaks

The goal of every XS-Leak attack is to steal user information cross-site. The threat model considered is a web attacker [4], and the attacker only controls how a target resource is included and how they observe the browser. The targeted server controls the HTTP response r, and the browser the victim uses determines the observed result. To achieve this goal, the attacker has to find *one* URL belonging to the target site that returns two different responses (r_1, r_2) depending

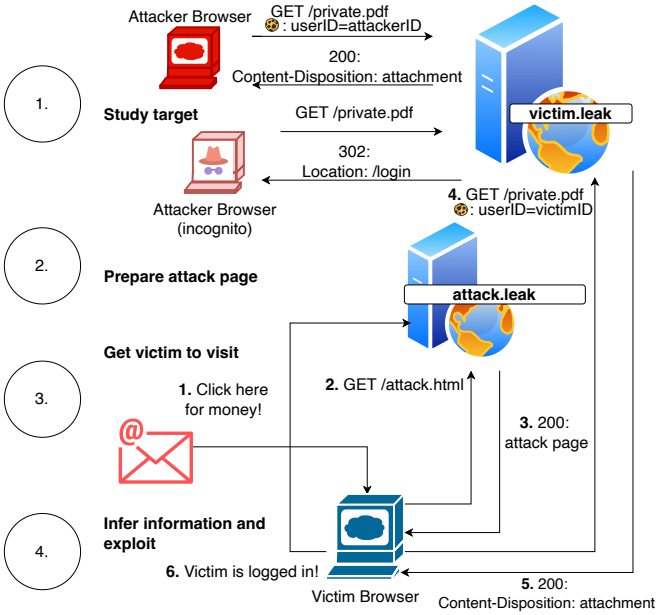

Fig. 1: *Steps of an XS-Leak attack.*

on user state information transmitted to the site that can be distinguished. For this, attackers have to go through several steps illustrated in Fig. 1.

1) The attacker first chooses a target site (victim.leak) and creates at least two different states. The created states can be anything and correspond to the information the attacker wants to steal. For example, they can be logged-in state, visited state, or anonymous state. In this example, the attacker uses logged-in state, and anonymous state. The attacker crawls URLs on the target site in all created states and collects the responses. The goal of the attacker is to find state-dependent URLs (SD-URLs). We define SD-URLs as URLs that deterministically deliver different HTTP responses (i.e., different status codes, header values, or body contents) for different visitor states. This definition is in line with prior work [63].
2) Based on the collected responses, the attacker uses their knowledge of browser behavior to choose the target URLs and the inclusion and observation method for the attack. The attacker then creates a suitable attack page and hosts it on a controlled site (attack.leak).
3) The attacker lures victims into visiting the attacker's site by targeted phishing or similar.
4) When a victim visits the attacker's site, the victim's browser requests the target URL, including the victim's state information on the target site The target site then creates a response corresponding to the victim's state. It is important to note that the attacker site cannot directly access the returned response due to the same-origin policy. However, depending on the chosen observation channel, it can infer some information about the returned response. For example, suppose the inclusion method is IFrame, the observation method is accessing the origin

attribute, and the response has a *Content-Disposition: attachment* header. In that case, the response triggers a download in the victim's browser, which results in the origin attribute being accessible by the attacker's site as it points back to the attacker's site. If the response does not trigger a download, trying to access the origin attribute results in a DOMException due to the same-origin policy as it points to the target site. Based on the collected information in step 1, the site infers the victim's state as it knows the expected observations for different states and sends this information to the attacker's server.

The impact of a successful XS-Leak highly depends on the targeted site and the targeted state information leaked. It can reach from history sniffing and login detection (e.g., used to perform more targeted XSS or CSRF attacks) over targeted tracking and advertisements (e.g., based on your inferred age or gender) to deanonymization (victim is the owner of a specific account). These attacks are especially critical on privacy-sensitive sites, such as adultery sites, where the gained information could be used for blackmailing. In addition, in oppressive countries, an attacker could be a state actor trying to identify people that visited forbidden websites.

III. Observation Channels in Browsers

To exploit an XS-Leak on a website, an attacker needs to know of a way in a browser that can distinguish between a response pair observed on the website. In the past, detecting such ways has been manual. This section develops a systematic approach for automatically discovering and characterizing observation channels in every browser.

A. Browser Observation Function and Observation Channels

The definition of XS-Leaks in the previous section is too complex to be a good abstraction for comprehensively analyzing information leaks in browsers. While it is necessary to create state on a website and find an SD-URL that returns a distinguishable response pair to find XS-Leaks on a website, it is enough to find arbitrary responses that result in different observations in a browser to show that a browser leaks any information. In the following, we define all terms necessary to model information leakage in browsers using the concept of browser observation functions and observation channels.

- **Inclusion method i:** an inclusion method, such as *image* or *fetch*, instructs a browser to perform a request to a server in a specific way. Every browser has a finite set I of inclusion methods that can be used to initiate requests.
- **Observation method m:** an observation method, such as accessing the width of an object reference or obtaining the current geolocation, observes information about the state of a browser in a given moment. Every browser has a finite set M of observation methods that give away information about the current state of the browser.
- **HTTP response r:** an HTTP response is generated by a server when a request is received. An HTTP response contains a status code and optionally headers, such as Content-Type, and a body. The set R represents all possible HTTP responses, and as the headers and body can be arbitrary bytes, this set is infinite.
- **Browser observation function:** A function $bo(i, m, r) = o$ that given an inclusion method, observation method, and HTTP response returns an observation $o \in O$ in a browser. The response belongs to the request initiated by the inclusion method, and the observation method executes after the browser has fully processed the response. This function can differ in various browsers and change with every browser version.

In principle, every observation method can be combined with every inclusion method. However, dependencies exist, and many observation methods behave differently based on the inclusion method. For example, accessing the width property of an element only works if the inclusion method targets an HTML element with a width property such as *image*. To account for this and ease presentation, we define the concept of an observation channel as the combination of an inclusion method and an observation method, i.e., $oc_{xy} = (i_x, m_y)$.

With the above definitions, we can distinguish between observation channels that leak cross-site information and others that do not. Given an observation channel oc_{xy} in a browser B, if two cross-site responses (r_1, r_2) result in two different observations $(o_1 = bo(oc_{xy}, r_1), o_2 = bo(oc_{xy}, r_2))$, this constitutes a distinguishable response pair in B for the given observation function. If at least one such pair exists, the observation channel oc_{xy} leaks information about responses cross-site, and we call it a working channel. We note that not every working channel necessarily poses risk to users and perform experiments to rate their criticalness in Section IV.

B. Conceptual Overview

In the following, we describe the general methodology to automatically discover working observation channels.

1) Test Generation: We aim to find all possible observation channels in browsers that leak information. Thus, compared to finding XS-Leaks on real websites where one only controls the observation channel, we also control the responses delivered to the browser. As we control all inputs of the browser observation function, we can fully compute it. Then, we can automatically determine all tuples of (oc, r_1, r_2) that result in different observations, as this means that a browser leaks information cross-site for this observation channel.

First, one has to create independent sets of inclusion methods, observation methods, and responses. These sets can be separately created by consulting the browser and HTML standard documentation, prior research, and other investigation. Then, our framework automatically creates and executes one test for every combination of inclusion method, observation method, and response. Every test consists of visiting a site that requests a cross-site URL that returns the specified response according to the given inclusion method and saving the outcome of the given observation method.

2) Summarizing Results: After collecting all results, they must be processed to examine which observation channels leak information. If at least two groups of responses exist that result

in different observations for a given observation channel in a browser, this observation channel leaks information cross-site, i.e., is a working channel. As the approach assumes that the complete combination of responses and observation channels were tested, a *unique* statement on the collected observations shows whether an observation channel leaks information in the given response space.

However, the result that $500{,}000$ responses resulted in observation *a* and $100{,}000$ responses resulted in observation *b* does not provide any meaningful insights apart from that the observation channel leaks information. We postulate that a human-understandable summary is necessary to understand the exact nature of the different outcomes and the real-world XS-Leak potential of each working observation channel. These summaries can help uncover bugs in implementations and unintentional loopholes in the HTML standard.

C. Implementation and Instantiation

This section describes which observation channels and responses we tested and why. Also, we describe the tools used to perform the tests and how to automatically create human-understandable summaries from the results.

1) Generated Tests: We based the sets of inclusion methods, observation methods, and responses on previous works [31], [58], [63] and own research. We implemented twenty inclusion methods found in previous works. These include *image*, *fetch* in different configurations, and *window.open*. In principle, every browser API can act as an observation method. However, many APIs, such as checking the current geolocation, are likely not influenced by the factors controlled in the browser observation function experiment and thus not included in our prototype implementation. We identified and implemented 34 observation methods, such as *events-fired* and *width*, reported as leaking information in the past or related to known methods. A complete list and corresponding code of the implemented inclusion methods and observation methods are available online [54]. The set of possible HTTP responses is infinite; thus, it is impossible to test it comprehensively. However, past research and own experiments indicate that only a couple of response properties influence XS-Leak behavior. We used ten properties to vary responses, such as status code, body content, and headers such as X-Frame-Options, with between 2 and 63 values each. All properties and values are shown in Table III in the appendix. The total combination of properties results in a response space of $1{,}886{,}976$ responses. However, we discovered through an iterative design process that many status codes behave the same across all tested browsers (e.g., all tested 5XX codes). Thus, we created 13 groups of status codes to reduce the number of tests. With this optimization, the testable response space decreases to a size of $359{,}424$.

All 34 implemented observation methods are independent of each other. Independence means that the execution of one observation method on a page does not influence the outcome of other observation methods on the same page, e.g., measuring the height of an element does not influence its width. Hence, we can execute all observation methods for one inclusion method at once for efficiency reasons.

We enumerated both the inclusion methods and the response space. We implemented an observation page generator, responsible for delivering the observation pages that include a URL using the specified inclusion method and then executing all observation methods, and an echo application, responsible for delivering all the requested responses. Both applications use Django [15], and we deploy them using uWSGI [65] for reliability and HTTPS support, which is necessary for response features such as COOP [69]. The two applications can adapt to the future as one can easily add new responses, inclusion methods, and observation methods by adding small code snippets as described in our README [54]. For this, it is unnecessary to understand the rest of the framework or know whether the added methods are prone to XS-Leaks.

2) Tested Browsers: We used Playwright [41] as the automation tool to control browsers to visit all the observation pages. Prior work showed that in the context of XS-Leaks, almost no differences between browsers of the same engine exist [31]. For example, MicrosoftEdge, Chrome, and Chromium all use Chromium as the base and behave the same for most observation methods. Thus, we tested the three browsers available with Playwright (1.18.1) by default: Chromium (99), Firefox (95), and WebKit[1] (15.4). The Playwright browser versions slightly differ from the default configurations. For example, pop-ups are allowed, and several features that interfere with automation are disabled. Currently, this results in the COOP [69] header being deactivated in Firefox and WebKit. While it is suboptimal to get results that do not perfectly mirror the experience of the browser for *every* user, this is bound to happen in any case in light of configurable settings (e.g., blocking of all third-party cookies) or browser extensions. We discovered differences between headful and headless modes through our iterative design process. As we aim to evaluate XS-Leaks relevant for average users, we tested headful browsers. We tested Chromium and Firefox on a Linux server. We discovered several issues with the WebKit version on Linux, so we tested WebKit on x86 iMacs and MacBooks.

3) Test Sequence: The general sequence of the tests is: The automated browser has the inputs (n, m) and visits a URL following the pattern https://observer.tld/n/?url=https://echo.tld/m/, the observation page causes the browser to request the response r_m from the echo application according to the inclusion method i_n, and after the response is received all observation methods are executed. Finally, we save the results in a DB and continue with the following test by either incrementing n or m. One needs to give the browser enough time to run all 34 observation methods and fully process the response. On a website, one often relies on the *load* event to start running code after a site has fully loaded. However, in some instances, e.g., videos or CORB [70] errors, the *load* event is fired early and cannot be relied on. In other cases, e.g., for some invalid

[1]WebKit is a custom browser by Playwright using the trunk build of the WebKit engine before it is used in Apple Safari, see https://playwright.dev/docs/browsers#webkit

HTTP responses, the tested browsers will not fire any *load* event at all. Thus, we need to rely on a timing-based method to decide when to run the browser observation function. In the current implementation, every test takes an average of 1.7s. In total, the number of tests to cover the complete combinatorial space is: $733{,}224{,}960 (= 20i * 34m * 359{,}424r * 3b)$. As all observation methods for one inclusion method execute in the same request, we have to perform $21{,}565{,}440$ browser visits.

4) Normalization and Outlier Removal: One thing to note is that the different channels have a differing number of possible observations. Many have a binary outcome, e.g., image inclusions always fire a load event or an error event. Others have theoretically infinitely many possible observations, such as checking the width of an image inclusion. However, within the constructed response space, only a few different observations are observed as we only test with one example image. Nevertheless, three observation methods have a large number of outcomes within our tests. For example, the observation method *securitypolicyviolation* has thousands of observations, as the observation includes the violating URL, and every test has a different URL. Therefore, for all channels using such observation methods, we smoothed the responses by replacing the varying part with a static string to only have structurally identical differences before creating the summaries.

Notably, modeling browser behavior assumes that the browser observation function is deterministic and is only influenced by the observation channel and the HTTP response. However, through our iterative design process, we discovered this is not always the case. Other factors, such as browser randomness and timeout issues, can influence the observations. Thus, we decided to run all tests *twice* to check the stability of every channel. In total, we ran $718{,}848 (= 359{,}424r * 2)$ tests for each channel in each browser. We defined *unstable channels* as channels where more than one percent, i.e., $3{,}594$, of all test pairs had different observations. We removed these unstable channels to prevent noise from affecting the results and establish a lower bound of working channels.

5) Summaries: As explained above, we create human-understandable summaries of the observation channels. These summaries visualize which response properties are responsible for which outcome. We use decision trees to visualize the relevant response properties in a browser. Decision trees are a well-known machine learning technique that produces easy-to-understand summaries for humans. In addition, they are good at removing unnecessary attributes and can handle some amount of noise in the data. We use the H2O random forests implementation [23] to build decision trees as it is a performant library that natively handles categorical data. Later, we convert the trees to PDFs using Python AnyTree [6] and Graphviz [20] for manual analysis and automatically discover groups of channels that behave the same.

D. Test Results

1) General: The total time taken for the experiment was 13 days for Chromium and Firefox on a Linux server with 100 browsers in parallel and 47 days for WebKit on three Apple machines with a total of 15 browsers in parallel.

The number of all tested observation channels is $2{,}040 (= 20i * 34m * 3b)$. Out of these, 410 observation channels had more than one observation. This result might seem high. However, many observation channels are intentional loopholes of the same-origin policy, such as receiving postMessages from IFrames. We tested every combination of inclusion method and observation method. Thus, many observation channels, such as checking the duration attribute on an image, should indeed not work and always result in the same observation, often *undefined*, *null* or similar default values. Thus, having many non-working observation channels is expected too.

Out of the 410 observation channels with more than one observation, 358 channels have less than 1% of tests with different results with a mean of 0.12% of differing tests. The other 52 channels are unstable with a mean of 4.28% of differing tests. After removing the test results of all tests with differing results in the two repetitions and tests with infrequent observations (less than 32), a lower bound of 280 working observation channels remain. By reporting this lower bound, we are conservative in our assessment of the attack surface in modern browsers.

The remaining 280 observation channels are roughly uniformly distributed over the three browsers (97 in Chromium, 94 in Firefox, 89 in WebKit). Disregarding the browsers, 114 unique channels exist. 72 exist in all three browsers, 22 exist in two browsers, and 20 only exist in one browser. Most of the 20 used inclusion methods leak information in all three browsers. One inclusion method *double-script* does not leak information in Firefox as it relies on an issue not existing in Firefox. Two inclusion methods, *embed-img, link-prefetch*, do not work in WebKit as they are not supported. In addition, three fetch configurations using *cors* only work in Firefox as they always throw a *CORS* error in the other two browsers. Four of the 34 observation methods did not leak any information in the experiments (el-blur, sheet, paused, fetch-events). The remaining ones mostly leak information in all browsers. Notable exceptions are the *history.length* and *windowHeight* method that only leak information in Chromium.

A rigorous distinction between new and known channels is challenging largely because previous reports used inconsistent classifications and incomplete descriptions of the discovered XS-Leaks. Instead, our focus is on modeling the capabilities of each channel, i.e., what information it leaks, and our decision tree summaries described in the following precisely model such capabilities. Regardless, we report several newly identified XS-Leaks opportunities, including new channels and channels thought to be fixed in Section III-D4.

2) Decision Tree Example: Fig. 2 presents the decision tree for the observation channel *image-height* in Firefox. This channel can leak the height of a rendered image in a browser. In our response space, the outcome is either 50 or the size of the broken image icon. However, it is not trivial to decide which responses result in which outcome, and this differs in browsers. Therefore, to formalize the notion of a successful

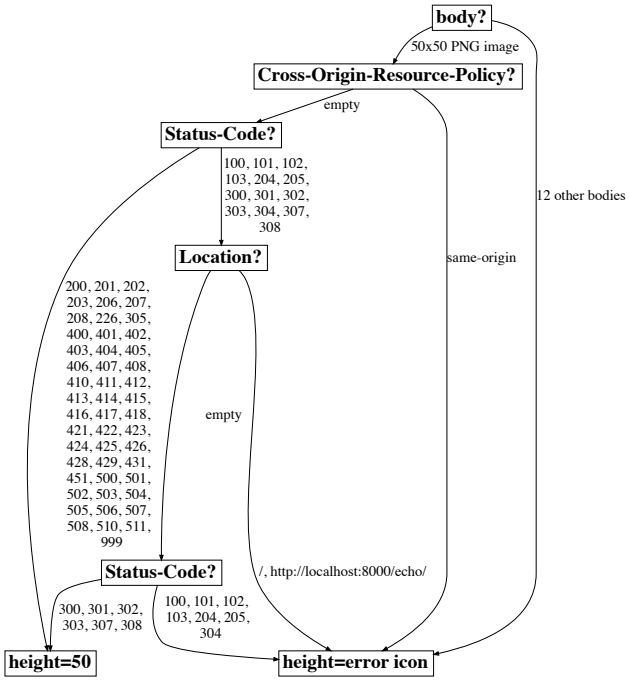

Fig. 2: *Decision tree for observation channel* image-height *(Firefox).*

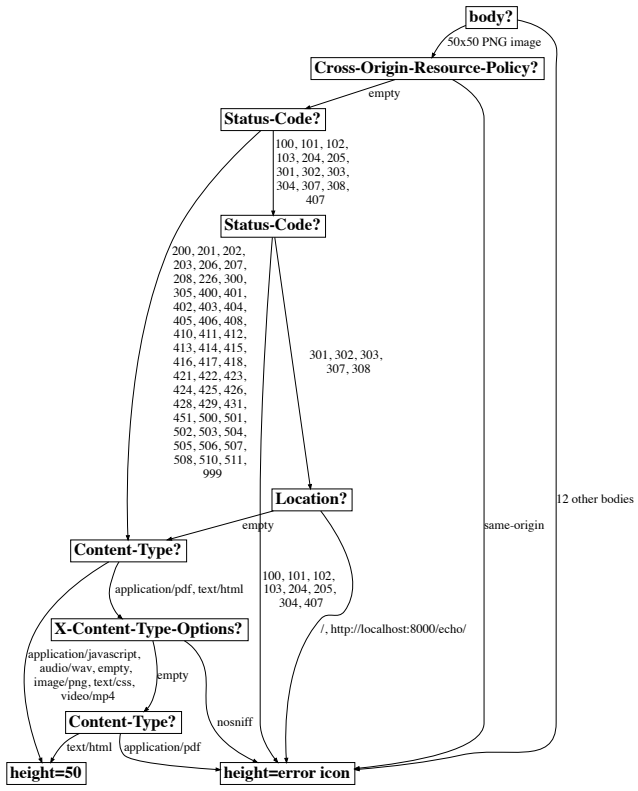

Fig. 3: *Decision tree for observation channel* image-height *(Chromium).*

image rendering in a browser, one can analyze the created decision trees. With a given response, one can follow the paths of the decision tree belonging to the currently investigated observation channel and obtain the observation. Without a response, the trees are analyzed by investigating every path to decide whether there are interesting patterns. As an example, consider the following response *body: image body, status code: 300, location: http://localhost:8000/echo/*. We start with the root node. This node instructs us to check the response's body content. We continue to the left as the response's body is a valid image. Otherwise, we would have already reached a leaf node with the outcome of the broken image icon height. The next node splits on the Cross-Origin-Resource-Policy (CORP) header. As the response has no CORP header, we continue the path on the left. Then, we check the status code. Status code 300 belongs to the right, and we continue there. The next node checks the location header. As there is a location header, we continue on the right and reach a leaf node. The outcome is the size of a broken image icon, as this response redirects to a non-image resource.

3) Browser Comparisons: As mentioned earlier, not a single observation channel is identical between all browsers. Here, we highlight some of the differences we found. Fig. 3 presents the decision trees created for *image-height* in Chromium. Comparing it to the Firefox tree in Fig. 2, one can see several differences. In general, both browsers observe the height of the image (50) for a successful image rendering and the height of the browser's broken image icon otherwise (24 in Firefox, 16 in Chromium). However, the definitions of a successful rendering are different. To summarize the differences, both browsers only render an image if the body contains a valid image and no CORP header disallows this. Also, they always fail for the status codes *100, 101, 102, 103, 204, 205, and 304*. However, Chromium additionally always fails for the status code *407*. They both do not render the image if the status code is a redirection code *301, 302, 303, 307, 308* and a valid Location header redirects to a non-image location. Firefox additionally accepts code *300* for redirections. In Chromium, the rendering also fails if the Content-Type header is *application/pdf* or if the X-Content-Type-Options header is set and the Content-Type header is *text/html*. The Cross-Origin Read Blocking (CORB) implementation of Chromium explains this behavior as Chromium replaces images with an empty body in these cases, and empty bodies are not valid images [5]. The summaries created for WebKit are larger as WebKit also renders videos [7] and PDFs in image tags. This results in another outcome (100, the example video's height) and more complicated rules as the default height of the rendered PDF is also 50. Another difference in these summaries is that WebKit additionally redirects responses with status code *305*, but not *300*, and does not fail for *205*.

The general patterns observed in this example regarding allowed status codes or content types apply to all observation channels. However, additional differences exist for many observation channels and studying the created summaries uncovered more insights. One of the reoccurring patterns was differences in status code handling. For example, for Content-Disposition responses, Chromium and WebKit allow status

codes *204* and *205*, but Firefox does not. For media resources, Chromium only allows code *200*, Firefox allows all *2XX* codes except *204* and *205*, and WebKit has unique results for code *206*. Other patterns relate to headers. For *link-stylesheet* inclusions, Firefox performs strict MIME type checking and only allows responses with Content-Type *text/css* or empty, Chromium and WebKit do not restrict on the Content-Type.

4) Browser Bugs: We manually analyzed all decision trees belonging to the 280 working observation channels. For the channels not existing in all browsers, we investigated the reason for not existing, i.e., whether they are unsupported or should not leak any information. In addition, for all channels working in more than one browser, we opened the decision trees next to each other and visually compared the possible outcomes and paths leading to them. If they differed, we iteratively distinguished between expected differences, such as the CORB blob existing in Fig. 3 but not in Fig. 2, and unexpected differences. To perform the classification, we consulted specifications and browser documentation. If the behavior of a browser broke any specification or could cause trouble to users without being intended, we reported it to the affected vendor. In total, we reported 11 bugs (including 3 CVEs). Several of the discovered bugs are special cases of already known and thought to be fixed bugs, highlighting the need for a more systematic and comprehensive approach to studying observation channels. Vendors can quickly discover the code locations related to a leak using our tools. Furthermore, they can use them to double-check their fixes by re-generating the trees after implementing a patch. For example, in 2018, the mediaError property was shown to leak too much cross-origin information [3], and this was fixed in both Firefox [2] and Chromium [1]. However, we discovered that the implemented fixes are incomplete. In Firefox, the fix is only applied to cross-site pages and not to same-site, cross-origin pages [46] (CVE-2022-34477). In Chromium, there are still more than the allowed two observations, as status codes *100 - 103, and 407* and responses with a *CORP* header result in a unique error message [44]. Other issues are that Firefox leaks the CSP *frame-ancestor* status of a response by throwing a violation on the parent frame [52] (CVE-2022-22745), and that Firefox leaks that a server-side redirect occurred, including same-origin redirects [53] (CVE-2022-36316) and several other bugs [43], [45], [47]–[51] We note that additional findings might be in the summaries if analyzed by browser vendors. Therefore, we got into contact with browser vendors and released our tools.

E. Response Distinguishing Oracle

With the created summaries, one can investigate all working observation channels and understand their root causes. However, one often wants to know which observation channels can distinguish two given responses in a browser and is not interested in why they can be distinguished. Manually going through all summaries to get the outcome of two given responses is a tiresome and error-prone task. For this reason, we created the response distinguishing oracle. Given two responses, it displays the channels that distinguish them alongside the outcomes for each response.

We have created a graphical version of the response distinguishing oracle shown in Fig. 5 in the appendix. Here, users can configure the two responses using drop-down menus and click the *Distinguish!* button. The tool will then display all observation channels that distinguish the two configured responses. This tool can be used by developers, browser vendors, and security researchers. In the example screenshot, we configured the responses to only be different in the status code (*200* and *404* respectively) and have empty values in all other response properties. The output shows that several observation channels can distinguish such responses in all three browsers. However, while some channels work in all three browsers (e.g., *link-stylesheet-events-fired*), others only work in some browsers (e.g., *embed-events-fired* only works in Firefox for these two responses).

Additionally, the oracle can be used as part of automatic tools scanning for XS-Leaks on websites. Given two responses belonging to the same URL in two different states, the oracle can guide which observation channels can distinguish them. We investigated such guidance in-depth in the next section.

Here, the two responses observed in the wild first have to be mapped to the covered response space. This mapping is done by dropping all uncovered response properties and transforming the values of the other properties to their closest relative in the response space using custom mapping functions. This mapping is necessary as it is unlikely to observe the exact responses covered in the response space in the wild. For example, the date header changes constantly and should be irrelevant. Also, consider the Content-Disposition header [36]. In the response space, the header can be absent, meaning no download triggers, or have the value *attachment* which usually triggers a download. In the wild, this header can also contain the value *inline*, have a filename specified after the value, or contain any other string. For our purposes, it is only important whether the header will trigger a download or not. Currently, the header always triggers a download unless it starts with *inline*. Thus, we can map values starting with *inline* to empty and every other value to *attachment* without losing accuracy.

IV. XS-Leaks in the Wild

As the previous section showed, browsers still have many leaky observation channels that can distinguish between responses. However, to pose a problem for the web ecosystem, websites that deliver such responses for different user states have to exist. We do not know how often these observation channels would work in the wild. Without this knowledge, one cannot understand how much of an issue XS-Leaks are for the web ecosystem. In this section, we investigate this question by scanning popular websites from the Tranco Top10K [32] for XS-Leak issues in three different attack modes of varying complexity: *login detection*, *visit inference*, and *cookie acceptance inference*. The results indicate which observation channels are particularly dangerous and can guide future action to eliminate XS-Leaks from the web.

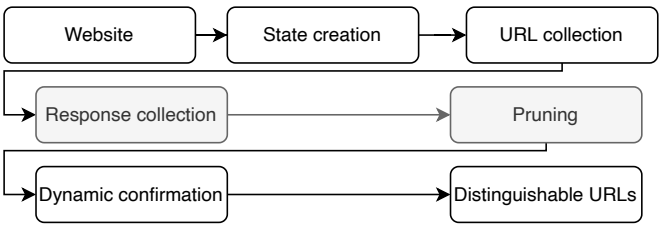

Fig. 4: *Overview of the does-it-leak pipeline.*

A. General Approach

In the previous section, we controlled all three inputs to the browser observation function to comprehensively investigate which observation channels leak information in browsers. In this section, we perform realistic XS-Leaks attacks against real websites. For an XS-Leak to occur in the wild, a URL must return two responses (r_1, r_2) distinguishable by a browser observation channel that depends on some user state (s_1, s_2). To investigate the issue, we create different user states on websites and then observe the outcomes of many observation channels to different URLs found on websites.

Fig. 4 illustrates the general approach to detecting XS-Leaks in the wild. First, we select a target website as the input. We then discover and create at least two suitable states on the site. State information to be distinguished can be anything, e.g., an anonymous visitor, a returning visitor, a logged-in user, or an admin user. Depending on the states one wants to distinguish, the corresponding attacks are known by different names, such as *visit inference*, *login detection*, or *targeted deanonymization*, and are of varying complexity. Then, we collect URLs on the site as potential candidates that might leak the state information. Many URLs belong to static resources, and we do not want to test them. Thus, we collect the responses for all target URLs by performing top-level requests to them in all created states. Then, the number of URLs to be tested can be reduced by a static pruning step using the collected responses. In this step, we discard URLs whose responses did not differ in any properties relevant to a known observation channel or URLs that differed in a way that is not distinguishable by any known observation channels. Nevertheless, the pruning step is insufficient as websites can serve different responses to cross-site requests in contrast to the collected same-site responses. The difference in responses could be due to randomness, SameSite cookies, Fetch metadata, or browser detection. Thus, a dynamic confirmation step tests all remaining target URLs using suitable observation channels. We perform this step several times to cope with noise in the responses. Finally, the outcome of this pipeline is all pairs of observation channels and URLs that can reliably distinguish the target states on each website. This allows us to provide a lower bound of *confirmed* XS-Leaks in the wild.

B. Login Detection

One of the classical threats of XS-Leaks is the inference of a login into a certain site. However, automatically registering accounts and logging in *at scale* is non-trivial in practice. While prior work indicated some success [16], [28], the rates are too low for conducting a meaningful large-scale analysis. However, for an adversary, taking the step of registering accounts and determining potential login detection side channels on *a specific site* is trivial. Therefore, our first experiment is manual in nature and focusses on the top 100 sites that allow for anyone to register (ranked *1* to *338*).

For these sites, we crawled up to $1{,}000$ URLs per site, excluding common logout URLs (e.g., log out, sign out, invalidate, etc.)) to minimize the chance of destroying our session during the experiment, and compared the two states *visited* and *logged-in*. For potentially leaking URLs, we only use those that come from the same party (we start from the same-party mapping of Steffens *et al.* [61] to build our mapping), e.g., inferring logins to Youtube is also feasible through URLs on google.com.

Out of the 100 sites, 77 are vulnerable, including top-rated sites such as google.com, facebook.com, twitter.com. 71 are vulnerable in Chromium, and 74 are vulnerable in Firefox. Many popular sites use SameSite cookies to protect themselves against attacks such as CSRF. However, almost no site uses the *COOP* header or the most secure value of *Strict* for SameSite, and thus most sites are still vulnerable to *window.open*-based attacks. Some sites might use user- or session-specific URLs to defend against XS-Leaks and similar attacks [60]. As a result, an attacker cannot find URLs to attack other users but only URLs to attack themselves. To ensure we are not reporting such cases as vulnerable, we investigated the source of the vulnerable URLs. On 71 sites vulnerable URLs came from both the *visited* and the *logged-in* state, on 2 sites the vulnerable URLs only came from the *visited* state, and on 4 sites all vulnerable URLs came from the *logged-in* state. For these 4 sites, we manually inspected these URLs to check if they were attacker-guessable. None of the URLs contained any session identifiers, making all of them guessable.

C. Visit Inference

Login detection can only be run in a small scale given the lack of properly functioning automation. However, an attacker may learn sensitive information about their victim merely from detecting if a site had been visited before, e.g., adult content. This attack is classically known as *history sniffing*, e.g., through detecting the color of links pointing towards the pages in question [8]. However, we aim to investigate which of the discovered channels can be used in the wild to still leak visits even in light of browser vendors' pushes to eradicating leaking channels from their products. To avoid confusion with prior attacks, we refer to this attack as *visit inference*.

1) Experimental settings: We test the Tranco Top10K[2] [32] as a representative sample of popular websites that real attackers might target. We used Playwright automation (v1.18.1) to control browsers and use the same observation channels considered in the previous section. We create the two states *anonymous*, i.e., a fresh browser context that we reset between

[2]Generated on 24 April 2022, available at https://tranco-list.eu/list/LY5Y4

each test, and *visited*, i.e., a browser profile that visited the *landing page* of the currently tested site to model the visit inference attack. Our test setup performs *site visit inference* but can be extended to *URL visit inference*. The *anonymous* state always exists. We consider the *visited* state successful if a load event fires within 30 seconds on the landing page.

We used the Chromium browser for the URL collection step. First, we visited the landing page of each site. We stayed until the load event fired or for a maximum of 30 seconds. We recorded all outgoing requests, for example, included images or fetch requests, and extracted all hyperlinks on the site.

We then performed the response collecting step in Chromium. Here, we limited the response collection to 500 URLs and one hour for each site. We use the response distinguishing oracle from the previous section for the pruning step of the URLs. However, to later compare the results between the different browsers and not conflate these results with potential artifacts of the response distinguishing oracle, we test every pair of inclusion method and URL that remains in at least one browser in all browsers.

We must dynamically confirm all remaining *(inclusion method, URL)* pairs in the corresponding browser due to the challenges mentioned above, such as SameSite cookies. We visit every pair up to five times to minimize the probability of false positives due to server-side randomness. If there is no difference in all observation methods, we abort early. Otherwise, we repeat the test. If there was a difference five times, we distinguish between systematic or random differences. For example, if the frame count in one state is always 0 and always larger than 0 in the other state, we consider this a systematic difference. On the other hand, if the frame count of both states is always different, but sometimes it is 0 in one state and sometimes in the other, we consider this a random difference and discard it. We limit the tests to a maximum of 25 URLs for each inclusion method and a maximum of 3 hours per site.

To not overload the sites with requests and minimize the chance of getting blocked, we perform at most one concurrent request and one request per second for each site. The tests are only performed for Chromium and Firefox as we cannot run WebKit on our Linux server to test up to 100 sites in parallel. We open-source our pipeline such that developers can test their site, and other researchers can benefit from it. The limits can be changed, and other state information can be provided.

2) Results: We successfully crawled $8{,}355$ sites out of the Tranco Top10K, which is in line with prior work [61]. On the other sites, the crawl failed due to various issues such as DNS lookup errors (625), timeouts (436), or certificate errors (271). For the $8{,}355$ sites, we collected a total of $1{,}982{,}223$ URLs with a median of 183 URLs per site with a minimum of one URL and a maximum of $6{,}721$ URLs.

We collected response data for a median of 183 URLs per site in the response collection step. The basic pruning step described above reduced the median number of URLs to 26. A total of 413 sites have zero URLs left after the basic pruning step. With the response distinguishing oracle, the median number of URLs that have to be tested for any inclusion method reduces to 23 URLs. This reduction sounds low at first. However, the median number of tested inclusion-methods-URL pairs reduces to 75 compared to a total of $520 (= 26\text{URLs} * 20i)$ pairs without the response distinguishing oracle. With the additional limit of at most 25 URLs tested for each inclusion method, we test a median of 59 inclusion-method-URL pairs in both browsers.

Observation channels		Vulnerable sites			
Inclusion Method	Observation Method	Both	Only C	Only FF	Either
window.open	length	221	337	187	745
iframe-csp	length	44	77	98	219
iframe	length	35	82	99	216
script	events-fired	3	35	59	97
fetch-creds-cors	performanceAPI	0	0	96	96
object	events-fired	3	10	64	77
embed	events-fired	1	7	58	66
iframe-csp	events-fired	3	16	46	65
link-stylesheet	events-fired	1	49	8	58
fetch-creds-cors-integrity	performanceAPI	0	0	58	58
iframe-csp	el-securitypolicyviolation	14	14	23	51
script	performanceAPI	17	4	29	50
	win.performanceAPI	2	12	34	48
iframe-csp	origin	2	12	34	48
	window.name	2	12	34	48
script	el-error	0	5	43	48
	CSS2Properties	2	12	34	48
iframe-csp	contentDocument	2	12	34	48
	el-message	4	14	27	45
iframe	el-message	2	17	26	45

TABLE I: *Top 20 observation channels in the wild*

As mentioned above, we test every *(inclusion method, URL)* pair in both browsers to ensure that artifacts of the response distinguishing oracle do not influence the reported differences observed in the wild. We start the dynamic confirmation step on $7{,}856$ sites. In total, our pipeline executed $3{,}521{,}427$ dynamic tests. The early abort is highly effective, as $2{,}436{,}935$ tests start in the first phase and only $344{,}082$ remain for a second run and $227{,}329$ tests in the fifth run. The complete pipeline from URL collection to dynamic confirmation used up to 100 browsers in parallel and took 7 days and 6 hours.

Many tested URLs belong to third parties, as most websites include resources and hyperlinks from many vendors. While visiting most websites also sets cookies for several third-party domains, and these URLs could be used for XS-Leaks, these domains often are included by several first-party sites (e.g., Google's DoubleClick). Hence, we cannot necessarily say that a specific site was visited before, but possibly only one in a set. Therefore, to be conservative in our analysis and to avoid false positives, we limit our analysis to same-site (based on the public suffix list [38]) URLs only.

After limiting the analysis to same-site URLs, a total of $1{,}291$ sites have distinguishable URLs, i.e., 15% of all tested sites. This number might seem low in comparison to the login detection experiment. However, many sites deliver entirely different experiences for logged-in users, whereas refreshing a site as a logged-out user mostly returns the same content. Out of all vulnerable sites, only 363 sites are distinguishable in both browsers, 490 sites are only vulnerable in Firefox, and 438 sites are only vulnerable in Chromium. While some of these differences can be explained by web servers performing browser detection and only serving vulnerable responses to one of them, many are caused by the browser differences discovered in the previous section. Table I shows the 20

channels that worked the most often, split by the two browsers, highlighting the differing severity of channels as the top 3 are responsible for most of the leaks. We note that several working channels did not leak any information in our experiments. The best-working channel *window.open-length* worked on a total of 745 sites. We explain the success of this channel by the fact that the inclusion method *window.open* is the only one that works when the state defining cookies have a SameSite value of *Lax* and that sites often change the number of included frames based on the user state. Of these, 585 sites are only vulnerable to *window.open*, highlighting the need to reconsider whether the often recommended value of *Lax* is secure enough and whether browsers should leak the number of frames in a document cross-site. For most other channels, more sites are vulnerable only in Firefox. The different SameSite defaults can partly explain it. Chromium defaults to *Lax* and only accepts *None* with a *Secure* flag. Firefox currently still defaults to *None* and allows *None* without a *Secure* flag. Another informative example is *link-stylesheet-events-fired* that worked 50 times in Chromium and only nine times in Firefox. Here, Firefox performs strict MIME type checking and can only distinguish between valid and invalid stylesheets, whereas Chromium can distinguish between responses with success status codes and ones without. As these results show, numerous XS-Leak attacks work in the wild, and there are notable differences between Chromium and Firefox. This finding highlights that browser vendors need a proper testbed to get rid of XS-Leaks in the future and cannot rely on isolated bug reports.

D. Cookie Acceptance Inference

In addition to the *visit inference*, we further introduce a variant called *cookie acceptance inference*. On today's web, users are frequently faced with banners such as *accept cookies* or *agree to our terms to continue* [14], [39]. Thus, it is not unrealistic to assume that when actually visting a site, users will interact with these dialogs, and we emulate it in this attack. Furthermore, this attack is more robust in the wild as it only identifies users that interacted with the target site and none that accidentally visited, allowing the attacker to run several sequential tests without corrupting the victim's state.

1) Experimental Settings: We use the same general settings as in the visit inference experiment. In addition, to the previous two states, we create the *accepted* state. This state represents a user that visited a site and interacted with it by accepting all cookies. For this, we built a simple script that first detects all elements that one has to click to use a site without distraction, such as *accept cookies*, *continue*, *ok*. For this, we use 93 locators [42], manually extracted from the top 250 websites. Then, we automatically try to click on all detected elements. We consider the *accepted* state successful, if we visit the landing page, at least one *target locator* is clicked successfully, and then a change in the cookies on the site, i.e., new cookies, removed cookies, or changed values, is observed. Otherwise, we record that the state could not be reached and do not test the site. Many websites have more than one way to deal with these banners, such as *reject all cookies* or *individualize choices*, and there might be ways to distinguish these options. The idea of our tool is to choose the easiest option, as previous research has shown that most users tend to choose the easiest option [39]. We believe the easiest option is often *accept all* [14], [39]. Note that we only used Chromium for this experiment given technical issues with the automated clicking in Firefox through Playwright.

Observation channels		Vulnerable sites	
Inclusion Method	Observation Method	Visit & Acceptance	Only Acceptance
window.open	length	247	665
iframe-csp	el-securitypolicyviolation	13	182
	length	47	81
iframe	length	43	80
	el-blur	4	52
iframe-csp	el-blur	1	52
iframe	el-message	6	35
iframe-csp	el-message	5	33
window.open	el-message	4	28
embed	el-blur	0	32
object	el-blur	0	29
link-stylesheet	events-fired	16	12
embed	el-message	1	17
script	events-fired	13	5
window.open	origin	12	1
	CSS2Properties	12	1
	win.performanceAPI	12	1
	window.name	12	1
iframe-csp	events-fired	8	3
object	performanceAPI	5	6

TABLE II: *Top 20 observation channels by attack type (Chromium)*

2) Results: Out of the 7,856 sites that had at least one URL after pruning, for 3,160 sites, we successfully reached the *accept* state. The other sites had the following issues. On 726 sites, a locator was found and clicked, but no change in the cookies was observed. On 3,970 site no locator was detected. Still, 1,059 changed their cookies without us clicking anything. With this in mind, we have to note that the success heuristic is not fool-proof, as many sites change their cookies by themselves, and we cannot guarantee that our click caused the observed change in the cookies.

Out of the 3,160 sites where we reached the *accept state*, visits to 348 sites could already be discovered through the basic visit inference attack. With the addition of cookie acceptance inference, we could identify an additional 749 sites as vulnerable, increasing the number of vulnerable sites to 34%. These numbers show the importance of our more realistic attack variant, as more than twice as many sites are vulnerable. Out of these, on 123 sites, all three states could be distinguished from each other, and 12 sites were only vulnerable in the visit inference case.

Table II presents the working observation channels for the cookie acceptance inference attack compared to the visit inference attack. Most channels increased the number of sites where they worked. However, the increase is not uniformly distributed as it is related to how the cookie banners accepted by our module are usually implemented. Many cookie banners are implemented as a frame, so the *length* method often changes by one if cookies are accepted. Another noteworthy increase is the *el-securitypolicyviolation* method. This method's number increases as many sites redirect to another origin for the cookie acceptance check (e.g., https://consent.site.tld). Another interesting increase is for the *el-blur* method.

This method often works as the cookie banner is autofocused, meaning that a blur event on the observation page framing a site with a cookie banner is fired.

V. Discussion

In this section, we first identify key insights derived from our work. We then discuss limitations and countermeasures, and end with our ethical considerations and how our research results can help secure browsers in the future.

A. Key Insights

Still, plenty of possibilities exist to leak information cross-site using a plethora of different observation channels in all major browsers. Some of these channels are known exceptions of the same-origin policy, such as receiving postMessages from an included IFrame. However, many of these channels have strange edge-cases. Furthermore, many new channels and bugs are slight adjustments of previously reported and allegedly fixed problems. As prior work always focussed on single isolated response pairs instead of systematic testing over large response spaces, all these cases were previously missed. These cases highlight the necessity for a framework like ours, which allows for systematic testing.

We highlight that XS-Leaks are prevalent on the web and a considerable threat to the web ecosystem. We tested the three attack modes *visit inference*, *cookie acceptance inference*, and *login detection* and, even with limited number of tested URLs, could find 15%, 34%, and 77% of tested sites vulnerable, respectively. These results suggest that it is easy to find vulnerable URLs for single sites for an attacker. In fact, for 350 (27%) sites vulnerable to visit inference, the homepage itself was vulnerable. Common patterns on websites include showing information, such as a welcome banner, in a frame and only for the first visit of a site or redirecting requests without cookies to first set cookies and then repeat the request.

Another insight is that there are substantial differences between the different browser implementations. For example, not a single observation channel worked the same for any two of Chromium, Firefox, and WebKit. Some of these differences are due to missing or deactivated features in browsers, such as CORB only existing in Chromium and link-prefetch being disabled in WebKit. Most, however, are due to previously missed edge cases. Examples include the treatment of status codes such as *204, 205, 300, and 407* or the priority of different contradicting response properties. The results for *visit inference* show that less than half of all vulnerable websites are vulnerable in both tested browsers. This result shows that the differences in the browsers really matter and that the unification of edge-case behavior could greatly reduce the total attack surface. Some differences between browsers are due to conscious decisions of browser vendors, such as the differing SameSite default setting in browsers. If no SameSite value is set for a cookie, in the tested versions, Chromium defaults to *Lax* whereas Firefox defaults to *None*. All inclusion methods except for *window.open* only work with SameSite *None*, as otherwise, no cookies are send for subresources.

B. Limitations

We aim to establish a lower bound of working observation channels in browsers and show that XS-Leaks constitute a significant threat to the web ecosystem to increase awareness and future mitigation of XS-Leaks. We do not aim to cover every possible observation channel in each browser or find every XS-Leak on each tested website.

We limited the set of tested browsers to recent versions of Chromium, Firefox, and WebKit given their significant market share of over 85% in 2022 [10]. Hence, we might have missed additional channels in less popular browsers. We limited the set of observation channels and the tested response space so that the testing stays feasible while covering as many potential cases as possible. While most channels can be executed stealthily in the background, the best working channels based on *window.open* usually require user interaction. In addition, the opened windows can be spotted by an attentive visitor in the absence of browser bugs, such as pop-unders, limiting the impact of these channels. We also limited the crawling time and depth and limited ourselves to the three discussed attacks. Thus, only because we did not report a site as prone to XS-Leaks does not mean that the site is not vulnerable.

For the reported XS-Leaks in the wild, we limited ourselves to same-site URLs to ensure that the tested site causes the leak and uniquely identifies the site visit. However, most websites include a plethora of third-party URLs. Thus, the URLs of these parties might also leak information. When testing these cross-site URLs, we found an additional $1{,}664$ sites vulnerable in the *visit inference* experiment. However, as such URLs may also be influenced by visiting pages on *other* first-party sites, we excluded them from further consideration, therefore likely underreporting real-world findings.

Additionally, not every site vulnerable in only one browser has to be caused by a browser difference. The websites may also implement browser switches based on the User-Agent header, and only serve problematic responses for some browsers. Also, while we made sure to make false positives as unlikely as possible by repeating every test five times, we cannot ensure that no false positives exist in the data that randomly differed in all five repetitions.

C. Countermeasures

Although XS-Leaks have been known for over 20 years, many countermeasures were only introduced recently. For many years, only two methods existed for websites to be secure, and both are impractical. The first is to return similar responses for all states that result in the same observations. Such behavior, however, is infeasible for every non-trivial website. The second is to use session-specific URLs, which would destroy many legitimate features, such as link sharing.

Over the years, many security headers were introduced against various attacks that can also mitigate some XS-Leaks. These include *X-Frame-Options* and *CSP's frame-ancestors directive* that stops XS-Leaks using the IFrame inclusion method. *CORP* that stops XS-Leaks using various inclusion methods such as image or video. *COOP* that stops XS-Leaks

using the *window.open* inclusion method. These, however, often restrict a site's legitimate functionality, such as being used in a mashup. In addition, it is of utmost importance that these headers are consistently deployed for all states as otherwise, their presence or absence can often be detected. As prior work has shown, this either does not occur consistently across all pages on a site [11] or may be influenced by client characteristics such as the geo location [55].

An orthogonal approach to changing the responses is making the requests indistinguishable or giving the server more information about requests so that the server can deny dubious requests. One drastic method to stop many XS-Leaks is completely blocking third-party cookies implemented in browsers like Safari or Brave [9], [67]. Another method is partitioning the cookies by top-level site as recently deployed by Firefox [37]. The most widely used browser, Chrome, currently does not do either but plans to take steps by 2024 [19]. It is important to note that these approaches do not stop leaks using the *window.open* method nor same-site attackers [59].

Another promising method is the SameSite flag of cookies. Cookies with a value of *Lax* are not sent with any cross-site requests apart from top-level get requests such as issued by *window.open*. The more secure setting of *Strict* even blocks cookies on *window.open* and could block all XS-Leaks using cookies as the state channel. However, both secure settings also destroy legitimate use cases. Thus, almost no site uses the most secure setting of *Strict* [30] and many sites only protect some of their cookies with *Lax* to hinder CSRF attacks. At the same time, they explicitly set other cookies to *None* to have greater functionality, often enabling XS-Leaks even in browsers that use the new default of *Lax*.

The non-cookie-based approach is to add request headers that give servers more information about the context of the request. The first two headers that did this were the *Referer* and the *Origin* header. However, these are not attached to *every* request making it difficult for a server to rely on them. A new addition is the set of Fetch Metadata headers that only contain more coarse-grained information, such as whether it is a cross-site request and whether the response is used as an image or script. A strict policy could block XS-Leaks, e.g., disallowing cross-origin image loading. However, this cannot protect legacy browsers (which do not send the headers) and aggressive blocking of cross-origin embedding is likely infeasible for *all* resources that might leak.

Another approach to counter XS-Leaks is reducing the observation methods' power. One could, for example, restrict currently allowed same-origin bypasses such as being allowed to access the *length* property of cross-origin window objects. Lastly, browser features such as CORB and removal of edge-cases in favor of the most secure browser can reduce the available attack surface of XS-Leaks in browsers.

We stress that currently, XS-Leaks are an ecosystem problem, and not a single entity is responsible alone. While many defenses exist, it is difficult for a website to be free of XS-Leaks. The browser support of different defenses varies, many defenses interfere with legitimate usage, and other defenses are opt-in for websites and challenging to deploy. It is, therefore, imperative that operators can assess their risk and mitigate specific leaks, which is why we make our tools available [54].

D. Ethical Considerations

This work deals with security issues in browsers and on websites. We responsibly disclosed all security-critical bugs found in the process of this work to the affected browser vendors. In addition, we contacted the three leading browser vendors to discuss the general methodology with them. The discussions are currently ongoing.

While testing real websites, we followed best practices to not put real users at risk or inconvenience. We only attacked accounts and sessions we created for these experiments and limited ourselves to a maximum of one request per second for every site and a maximum of a few thousand requests per site. We discovered many sites to be vulnerable to *history inference*, *cookie acceptance inference*, and *login detection*. However, the impact of all these attacks depends on the exact security needs of the site; what is worse, sites may even not care about their users' privacy regarding these attacks. Given that prior work in vulnerability notifications [62] has had limited impact when disclosing problems to vast amounts of operators, we rather decided to discuss with browser vendors to help them close the leak channels in the first place.

E. Going Forward

We open-source our tools to foster future research and help developers and browser vendors alike to secure their products. The response distinguishing oracle can be used as an educational and awareness tool for developers. Instead of requiring in-depth knowledge about XS-Leaks, developers can simply provide two responses from any of their endpoints, and our tools present them with all channels that can distinguish it in any browser. In addition, the does-it-leak pipeline could be bundled with a web vulnerability scanner to scan websites for XS-Leak issues automatically. While these tools could be used for malicious purposes, attackers only need to find a single vulnerable URL on a site, which is often possible without advanced tools. Finding all vulnerable URLs is more helpful for defenders as they can then correctly secure their site.

Our results show that the past focus on single response pairs left many edge-case leaks and browser differences undiscovered. The focus on single responses is problematic for browser vendors using single responses for regression tests and browser standardization projects such as web-platform-tests [71] that only use a couple of responses to test each standard and thus over-report conformity between browsers. In the future, browser vendors and browser test organizations can switch from this single isolated responses model to a new model where they test many responses from a vast response space. They then can find edge-cases and differences before they reach users where they might have severe privacy implications. It is hard to a priori see which implications a change in a browser brings. Every new feature introduced in a browser can change the browser observation function and

create new leaks. When browser vendors test a vast response space before the roll-out of new features, they could have more confidence that they do not unintentionally introduce new leaks or make existing leaks more dangerous. Also, standardization bodies and browser vendors can unify edge-case behavior by agreeing on the most secure implementation for every browser difference, decreasing the overall attack surface.

VI. Related work

In this section, we survey related works in the areas of XS-Leaks, history sniffing, and browser testing.

A. State of XS-Leaks

In 2021, Knittel *et al.* proposed a formal model for XS-Leaks and manually discovered several new leaks using it. Furthermore, they automatically evaluated 56 browser configurations against their list of leaks and found many differences, and thus proposed a new mitigation technique of changing the browser behavior [31]. In 2022, Van Goethem *et al.* extended this formal model with the concept of components and a thorough evaluation of currently available defenses [66]. In 2020, Sudhodanan *et al.* were the first to test many known XS-Leak methods in three mainstream browsers, including newly detected variants. In addition, they manually created accounts on 58 tested websites and tested them for several attacks such as login detection and account type identification [63]. The general problem of XS-Leaks has been known since the early 2000s when Felten and Schneider described access detection attacks exploiting cache behavior via a timing side-channel [17]. Since then many different leak channels were discovered [12], [21], [22], [24], [25]. Recently the XS-Leaks wiki project tried to group all the leak channels and provide a central place of information [58].

The observation channels considered in this work fit into the proposed formal model. We generalize it by removing the need to manually choose distinguishable response pairs and show that one can systematically test a response space and summarize the result instead. Our framework is the first that can automatically find new information leaks in browsers. Moreover, our results of large-scale real-world analyses not only show that attackers can infer login, visit, or cookie accepted states in a large body of sites, but also enables to identify the most critical bugs. This way, browser vendors can prioritize fixing efforts based on our real-world findings.

B. History Sniffing

The web community has known history sniffing attacks since the 2000s. Since then many methods were discovered, fixed, and re-discovered over time [8], [13], [27], [29], [40], [57], [68]. Most of the works on history sniffing focused on leaking information from the browser history storage, e.g., using the color of visited links, and did not include requests to the target sites. However, Sanchez-Rola *et al.* showed that it is also possible to detect that a user visited a site by sending requests to a target site and timing the results. Such detection works because many sites set cookies when visited, and the responses differ based on the cookies attached to requests [56].

Our real-world test builds upon this insight and uses visit inference as the primary example attack to study how big of an issue XS-Leaks are for the web ecosystem. However, nowadays, it is not possible to use many sites without first accepting cookies. Thus we extended the visit inference attack to the *cookie acceptance inference* attack, where we also interacted with the sites by clicking on every accept button.

C. Browser Testing

Browsers are regularly tested for functionality and security. The web-platform-tests project hosts an extensive collection of tests to ensure specification conformance and compatibility between browsers [71]. In 2015, Hothersall-Thomas *et al.* presented BrowserAudit, a test suite to check various security features in browsers such as CSP and CORS [26]. In 2018, Franken *et al.* studied whether third-party cookie blocking policies block all third-party cookies [18]. In 2019, Luo *et al.* created a test suite for security features such as CSP and HSTS and used it to study their evolution in mobile browsers [33].

All these works specify the correct behavior of each test upfront and mostly rely on hand-crafted tests in the orders of hundreds. In contrast, our pipeline observes behavior for millions of automated tests, and we later analyze the results by comparing the created decision tree summaries.

VII. Conclusion

XS-Leaks have been known for years, yet still new instances frequently appear. To make significant leaps in the arms race of finding and patching them, we introduced the first framework to automatically discover and characterize cross-site information leaks in browsers. A key aspect of our approach is to use decision trees to generate explainable summaries of the root causes of the leaks. We discovered 280 information leaking channels in the engines of Chromium, Firefox, and Safari. While analyzing the generated descriptions, we found 11 bugs, including 3 CVEs, in browsers, several of which were thought to be fixed. Furthermore, we uncovered that more than previously thought flaws are specific to individual engines.

To show that such information leaks and the differences between the browsers impact users' privacy, we performed three case studies finding XS-Leaks on real websites. Our visit inference and cookie acceptance attacks showed 15% and 34% of sites being vulnerable, respectively, even with a shallow crawl. Furthermore, our login detection study on 100 top-ranked sites showed that 77 of them were vulnerable through XS-Leaks. These findings underline the importance of being able to detect leak channels in a systematic way.

With our discussion of current countermeasures, we hope to spark a new discussion between browser vendors and specifications bodies to make the web more secure by default and show a realm of promising future research directions. We open-source our tools [54] such that web developers can ensure their own site is XS-Leaks free. Further, at the time of this writing, we are discussing with browser vendors how to best integrate our pipeline in their development processes.

Acknowledgment

We thank our anonymous shepherd and the reviewers for their valuable feedback.

This work was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science. This work received funding from the European Union's Horizon 2020 research and innovation programme under the TESTABLE project (grant agreement 101019206).

Availability

Code for all experiments is available online: https://github.com/cispa/xs-observations

Data is available on request.

References

[1] G. Acar. "1450853 - (CVE-2020-15666) MediaError message property leaks cross-origin response status." (2018), [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1450853.

[2] G. Acar. "828265 - MediaError message property leaks cross-origin response status." (2018), [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=828265.

[3] G. Acar, D. Y. Huang, F. Li, A. Narayanan, and N. Feamster, "Web-based Attacks to Discover and Control Local IoT Devices," in *Workshop on IoT Security and Privacy*, 2018. DOI: 10.1145/3229565.3229568.

[4] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a Formal Foundation of Web Security," in *IEEE Computer Security Foundations Symposium*, 2010. DOI: 10.1109/CSF.2010.27.

[5] L. Anforowicz. "More CORB-protected MIME types - adding protected types one-by-one. · Issue #860 · whatwg/fetch," GitHub. (2019), [Online]. Available: https://github.com/whatwg/fetch/issues/860.

[6] *Any Python Tree Data*, 2022. [Online]. Available: https://anytree.readthedocs.io/en/latest/.

[7] Apple. "Delivering Video Content for Safari." (2022), [Online]. Available: https://developer.apple.com/documentation/webkit/delivering_video_content_for_safari.

[8] D. Baron. "Preventing attacks on a user's history through CSS :visited selectors." (2010), [Online]. Available: https://dbaron.org/mozilla/visited-privacy.

[9] Brave. "OK Google, don't delay real browser privacy until 2022," Brave Browser. (), [Online]. Available: https://brave.com/ok-google/.

[10] "Browser Market Share Worldwide," StatCounter Global Stats. (2022), [Online]. Available: https://gs.statcounter.com/browser-market-share.

[11] S. Calzavara, T. Urban, D. Tatang, M. Steffens, and B. Stock, "Reining in the Web's Inconsistencies with Site Policy," in *Network and Distributed System Security Symposium*, 2021. DOI: 10.14722/ndss.2021.23091.

[12] M. Cardwell. "Abusing HTTP Status Codes to Expose Private Information," Grepular. (2011), [Online]. Available: https://www.grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information.

[13] A. Dabrowski, G. Merzdovnik, N. Kommenda, and E. Weippl, "Browser History Stealing with Captive Wi-Fi Portals," in *IEEE Security and Privacy Workshops*, 2016. DOI: 10.1109/SPW.2016.42.

[14] M. Degeling, C. Utz, C. Lentzsch, H. Hosseini, F. Schaub, and T. Holz, "We Value Your Privacy ... Now Take Some Cookies: Measuring the GDPR's Impact on Web Privacy," in *Network and Distributed System Security Symposium*, 2019. DOI: 10.14722/ndss.2019.23378.

[15] *Django*, version 4.0.3, 2022. [Online]. Available: https://www.djangoproject.com/.

[16] K. Drakonakis, S. Ioannidis, and J. Polakis, "The Cookie Hunter: Automated Black-box Auditing for Web Authentication and Authorization Flaws," in *ACM SIGSAC Conference on Computer and Communications Security*, 2020. DOI: 10.1145/3372297.3417869.

[17] E. W. Felten and M. A. Schneider, "Timing attacks on Web privacy," in *ACM Conference on Computer and Communications Security*, 2000. DOI: 10.1145/352600.352606.

[18] G. Franken, T. V. Goethem, and W. Joosen, "Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies," in *USENIX Security Symposium*, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/franken.

[19] Google. "Expanding testing for the Privacy Sandbox for the Web," Google. (2022), [Online]. Available: https://blog.google/products/chrome/update-testing-privacy-sandbox-web/.

[20] "Graphviz," Graphviz. (2022), [Online]. Available: https://graphviz.org/.

[21] J. Grossman. "I Know What Websites You Are Logged-In To (Login-Detection via CSRF)," WhiteHat Security. (2012), [Online]. Available: https://web.archive.org/web/20160317054027/https://www.whitehatsec.com/blog/i-know-what-websites-you-are-logged-in-to-login-detection-via-csrf/.

[22] J. Grossman. "Login Detection, whose problem is it?" (2008), [Online]. Available: https://blog.jeremiahgrossman.com/2008/03/login-detection-whose-problem-is-it.html.

[23] *H2O: Distributed Random Forest (DRF)*, 2022. [Online]. Available: https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/drf.html.

[24] R. Hansen. "Detecting States of Authentication With Protected Images," ha.ckers. (2006), [Online]. Available: https://web.archive.org/web/20150417095319/http://ha.ckers.org/blog/20061108/detecting-states-of-authentication-with-protected-images/.

[25] E. Homakov. "313737 - Disclose domain of redirect destination taking adventadge of CSP." (2013), [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=313737.

[26] C. Hothersall-Thomas, S. Maffeis, and C. Novakovic, "BrowserAudit: Automated testing of browser security features," in *International Symposium on Software Testing and Analysis*, 2015. DOI: 10.1145/2771783.2771789.

[27] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in JavaScript web applications," in *ACM Conference on Computer and Communications Security*, 2010. DOI: 10.1145/1866307.1866339.

[28] H. Jonker, S. Karsch, B. Krumnow, and M. Sleegers, "Shepherd: A Generic Approach to Automating Website Login," in *Workshop on Measurements, Attacks, and Defenses for the Web*, 2020. DOI: 10.14722/madweb.2020.23008.

[29] S. Karami, P. Ilia, and J. Polakis, "Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage," in *Network and Distributed System Security Symposium*, 2021. DOI: 10.14722/ndss.2021.23104.

[30] S. Khodayari and G. Pellegrino, "The State of the Same-Site: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies," in *IEEE Symposium on Security and Privacy*, 2022. DOI: 10.1109/SP46214.2022.9833637.

[31] L. Knittel, C. Mainka, M. Niemietz, D. Trevor Noß, and J. Schwenk, "XSinator.com: From a Formal Model to the Automatic Evaluation of Cross-Site Leaks in Web Browsers," in *ACM SIGSAC Conference on Computer and Communications Security*, 2021. DOI: 10.1145/3460120.3484739.

[32] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen, "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation," in *Network and Distributed System Security Symposium*, 2019. DOI: 10.14722/ndss.2019.23386.

[33] M. Luo, P. Laperdrix, N. Honarmand, and N. Nikiforakis, "Time Does Not Heal All Wounds: A Longitudinal Analysis of Security-Mechanism Support in Mobile Browsers," in *Network and Distributed System Security Symposium*, 2019. DOI: 10.14722/ndss.2019.23149.

[34] R. Masas. "Mapping Communication Between Facebook Accounts Using a Browser-Based Side Channel Attack," Imperva. (2019), [Online]. Available: https://www.imperva.com/blog/mapping-communication-between-facebook-accounts-using-a-browser-based-side-channel-attack/.

[35] R. Masas. "Patched Facebook Vulnerability Could Have Exposed Private Information About You and Your Friends," Imperva. (2018), [Online]. Available: https://www.imperva.com/blog/facebook-privacy-bug/.

[36] MDN. "Content-Disposition." (2022), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Disposition.

[37] Mozilla. "Firefox Rolls Out Total Cookie Protection By Default To All Users." (2022), [Online]. Available: https://blog.mozilla.org/en/products/firefox/firefox-rolls-out-total-cookie-protection-by-default-to-all-users-worldwide/.

[38] Mozilla. "Public Suffix List." (2022), [Online]. Available: https://publicsuffix.org/.

[39] M. Nouwens, I. Liccardi, M. Veale, D. Karger, and L. Kagal, "Dark Patterns after the GDPR: Scraping Consent Pop-ups and Demonstrating their Influence," in *Conference on Human Factors in Computing Systems*, 2020. DOI: 10.1145/3313831.3376321.

[40] L. Olejnik, C. Castelluccia, and A. Janc, "Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns," in *HotPETs*, 2012. [Online]. Available: https://hal.inria.fr/hal-00747841.

[41] Playwright. "Fast and reliable end-to-end testing for modern web apps." (2022), [Online]. Available: https://playwright.dev/.

[42] Playwright. "Locators." (2022), [Online]. Available: https://playwright.dev/docs/locators.

[43] J. Rautenstrauch. "1251534 - Security: CSP matching algorithm does not ignore paths for client-side redirections." (2021), [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=1251534.

[44] J. Rautenstrauch. "1251921 - Security: MediaError messages still leak cross-origin informatio." (2021), [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=1251921.

[45] J. Rautenstrauch. "1260366 - Security: X-Frame-Options and CSP: Frame-ancestor information leaks cross-origin using object tag." (2021), [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=1260366.

[46] J. Rautenstrauch. "1731614 - MediaError message property leaks information on cross-origin same-site pages." (2021), [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1731614.

[47] J. Rautenstrauch. "1732012 - X-Frame-Options is ignored on redirection status-codes (without a location set)." (2021), [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1732012.

[48] J. Rautenstrauch. "1732069 - Sec-Fetch-Site inconsistent on localhost/IPs." (2021), [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1732069.

[49] J. Rautenstrauch. "1732106 - Cross-Origin-Resource-Policy incorrectly applied on object and embed tags." (2021), [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1732106.

[50] J. Rautenstrauch. "1732141 - Request loads forever if code is 101 or 304 and ct=application/pdf." (2021), [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1732141.

[51] J. Rautenstrauch. "1732199 - Infinite reload of 201, 203, 204 responses." (2021), [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1732199.

[52] J. Rautenstrauch. "1735856 - Securitypolicyviolation leaks cross-origin information for frame-ancestors violations." (2021), [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1735856.

[53] J. Rautenstrauch. "1768583 - Fetch requests with mode cors and credentials leak whether the request redirected or not via performanceAPI." (2022), [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1768583.

[54] J. Rautenstrauch. "Code for all experiments conducted in this paper." (2022), [Online]. Available: https://github.com/cispa/xs-observations.

[55] S. Roth, S. Calzavara, M. Wilhelm, A. Rabitti, and B. Stock, "The Security Lottery: Measuring Client-Side Web Security Inconsistencies," in *USENIX Security Symposium*, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/roth.

[56] I. Sanchez-Rola, D. Balzarotti, and I. Santos, "BakingTimer: Privacy analysis of server-side request processing time," in *Annual Computer Security Applications Conference*, 2019. DOI: 10.1145/3359789.3359803.

[57] M. Smith, C. Disselkoen, S. Narayan, F. Brown, and D. Stefan, "Browser history re:visited," in *Workshop on Offensive Technologies*, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/smith.

[58] M. Sousa *et al.* "XS-Leaks Wiki." (2020), [Online]. Available: https://xsleaks.dev/.

[59] M. Squarcina, M. Tempesta, L. Veronese, S. Calzavara, and M. Maffei, "Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web," in *USENIX Security Symposium*, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/squarcina.

[60] C.-A. Staicu and M. Pradel, "Leaky Images: Targeted Privacy Attacks in the Web," in *USENIX Security Symposium*, 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/staicu.

[61] M. Steffens, M. Musch, M. Johns, and B. Stock, "Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI," in *Network and Distributed System Security Symposium*, 2021. DOI: 10.14722/ndss.2021.24028.

[62] B. Stock, G. Pellegrino, F. Li, M. Backes, and C. Rossow, "Didn't You Hear Me? - Towards More Successful Web Vulnerability Notifications," in *Network and Distributed System Security Symposium*, 2018. DOI: 10.14722/ndss.2018.23171.

[63] A. Sudhodanan, S. Khodayari, and J. Caballero, "Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks," in *Network and Distributed System Security Symposium*, 2020. DOI: 10.14722/ndss.2020.24278.

[64] terjanq. "Mass XS-Search using Cache Attack." (2019), [Online]. Available: https://terjanq.github.io/Bug-Bounty/Google/cache-attack-06jd2d2mz2r0/index.html.

[65] *uWSGI*, version 2.0.20, 2021. [Online]. Available: https://uwsgi-docs.readthedocs.io/en/latest/.

[66] T. Van Goethem, G. Franken, I. Sanchez-Rola, D. Dworken, and W. Joosen, "SoK: Exploring Current and Future Research Directions on XS-Leaks through an Extended Formal Model," in *ACM Symposium on Information, Computer and Communications Security*, 2022. DOI: 10.1145/3488932.3517416.

[67] WebKit. "Full Third-Party Cookie Blocking and More," WebKit. (), [Online]. Available: https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/.

[68] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson, "I Still Know What You Visited Last Summer: Leaking Browsing History via User Interaction and Side Channel Attacks," in *IEEE Symposium on Security and Privacy*, 2011. DOI: 10.1109/SP.2011.23.

[69] WHATWG. "Cross-Origin-Opener-Policy." (2022), [Online]. Available: https://html.spec.whatwg.org/multipage/origin.html#cross-origin-opener-policies.

[70] WHATWG. "Fetch Standard CORB." (2022), [Online]. Available: https://fetch.spec.whatwg.org/#corb.

[71] WPT. "Web-platform-tests documentation." (2022), [Online]. Available: https://web-platform-tests.org/.

APPENDIX

Property	Count	Options
Status-Code	63	100, 101, 102, 103, 200, 201, 202, 203, 204, 205, 206, 207, 208, 226, 300, 301, 302, 303, 304, 305, 307, 308, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 421, 422, 423, 424, 425, 426, 428, 429, 431, 451, 500, 501, 502, 503, 504, 505, 506, 507, 508, 510, 511, 999
Body	13	HTML with one frame, HTML with two frames, HTML that sends postMessage, HTML with meta_refresh, HTML that opens paymentAPI, CSS that sets h1 color to blue, Invalid JavaScript, JavaScript that sets a variable, 50x50 PNG image, 100x100 mp4 video with duration 2s, WAV audio file with duration 1s, PDF, empty
Content-Type	8	text/html, text/css, application/javascript, video/mp4, audio/wav, image/png, application/pdf, empty
Content-Disposition	2	attachment, empty
Location	3	http://localhost:8000, /, empty
X-Frame-Options	2	deny, empty
X-Content-Type-Options	2	nosniff, empty
Cross-Origin-Resource-Policy	2	same-origin, empty
Cross-Origin-Opener-Policy	2	same-origin, empty
Content-Security-Policy	3	frame-ancestors 'self', default-src 'self', empty

TABLE III: *Considered properties and options of the response space*

	Response 1	Response 2
Status-Code	200	404
Body-Content	empty	empty
Content-Type	empty	empty
X-Content-Type-Options	empty	empty
X-Frame-Options	empty	empty
Content-Disposition	empty	empty
Location	empty	empty
Cross-Origin-Opener-Policy	empty	empty
Cross-Origin-Resource-Policy	empty	empty
Content-Security-Policy	empty	empty

Distinguish!

Inclusion method	Observation method	Browser	Observation 1	Observation 2
embed	events-fired	firefox	load	uncalled
link-stylesheet	events-fired	chromium	load	error
link-stylesheet	events-fired	firefox	load	error
link-stylesheet	events-fired	webkit	load	error
link-stylesheet	performanceAPI.smooth	webkit	{'size': {'transferSize': None, 'decodedBodySize': None, 'encodedBodySize': None}, 'timing': {'duration': >0, 'fetchStart': >0, 'redirectEnd': 0, 'redirectStart': 0, 'secureConnectionStart': 0}, 'initiatorType': 'link', 'nextHopProtocol': 'http/1.1'}	{'size': {'transferSize': None, 'decodedBodySize': None, 'encodedBodySize': None}, 'timing': {'duration': >0, 'fetchStart': >0, 'redirectEnd': 0, 'redirectStart': 0, 'secureConnectionStart': 0}, 'initiatorType': 'link', 'nextHopProtocol': ''}
object	contentDocument	chromium	js-null	{'location': {'hash': '', 'host': '', 'href': 'about:blank', 'port': '', 'origin': 'null', 'search': '', 'hostname': '', 'pathname': 'blank', 'protocol': 'about:', 'ancestorOrigins': {'0': 'http://localhost:8001'}}}
object	contentDocument	webkit	js-null	{'location': {'hash': '', 'host': '', 'href': 'about:blank', 'port': '', 'origin': 'null', 'search': '', 'hostname': '', 'pathname': 'blank', 'protocol': 'about:', 'ancestorOrigins': {'0': 'http://localhost:8001'}}}
object	events-fired	chromium	load	error
object	events-fired	firefox	load	error
object	events-fired	webkit	load	uncalled

See more...

Fig. 5: *Screenshot of the response distinguishing oracle.*