

Finding All Cross-Site Needles in the DOM Stack: A Comprehensive Methodology for the Automatic XS-Leak Detection in Web Browsers

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

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

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

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

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

ABSTRACT

Cross-Site Leaks (XS-Leaks) are a class of vulnerabilities that allow a web attacker to infer user state from a target web application cross-origin. Fixing XS-Leaks is a cat-and-mouse game: once a published vulnerability is fixed, a variant is discovered. To end this game, we propose a methodology to find *all* leak techniques for a given state-dependent resource and a set of inclusion method. We translate a website's DOM at runtime into a directed graph. We execute this translation twice, once for each state. The outputs are two slightly different graphs. We then get the set of *all* leak techniques by computing these two graphs' differences. The remaining nodes and edges differ between the two states, and the corresponding DOM properties and objects can be observed cross-origin.

We implemented AUTOLEAK, our open-source solution for automatically detecting known and yet unknown XS-Leaks in *web browsers* and *websites*. For our systematic study, we focus on XS-Leak test cases for *web browsers* with detectable differences induced by HTTP headers. We created and evaluated a total of 151 776 test cases in Chrome, Firefox, and Safari. AUTOLEAK executed them automatically without human interaction and identified up to 8 403 leak techniques per test case. On top, AUTOLEAK's systematic evaluation uncovers 5 novel classes of XS-Leaks based on leak techniques that allow detecting novel HTTP headers cross-origin. We show the applicability of our methodology on 24 web sites in the Tranco Top 50 and uncovered XS-Leaks in 20 of them.

CCS CONCEPTS

• Security and privacy → Web application security; Browser security; Web protocol security.

KEYWORDS

XS-Leaks, Browsers, Web Security, AutoLeak, Graphs, Privacy

ACM Reference Format:

Dominik Trevor Noß, Lukas Knittel, Christian Mainka, Marcus Niemiets, and Jörg Schwenk. 2023. Finding All Cross-Site Needles in the DOM Stack: A Comprehensive Methodology for the Automatic XS-Leak Detection in Web Browsers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*, November 26–30, 2023, Copenhagen, Denmark. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3576915.3616598>

1 INTRODUCTION

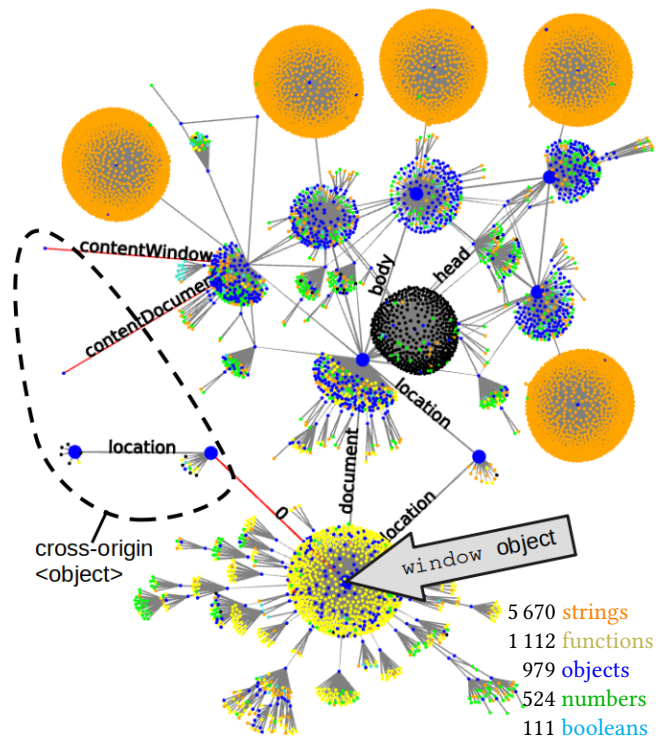


Figure 1: Firefox DOM graph of a web page with only two tags: `<script>` and `<object>`, which is used as cross-origin inclusion method (red edges). The graph consists of 11 427 properties (edges) and 8 823 objects (nodes), each of which could function as a leak technique.



This work is licensed under a Creative Commons Attribution-ShareAlike International 4.0 License.

XS-Leaks. Client-side vulnerabilities that allow collecting privacy-sensitive information about cross-origin web resources are called *XS-Leaks*. For state-dependent resources, different states of a web application result in *differences* in the HTTP responses (e.g., in status codes, headers, or the body). Attackers can not detect these differences cross-origin due to the Same-Origin Policy (SOP). However, XS-Leaks use side-channel information, called *leak techniques*, from browser features that is not considered security-critical and, consequentially, not blocked by the SOP. If a leak technique for a state-dependant difference exists, this is a *detectable difference*. Differences may be detectable only for certain *inclusion methods* (e.g., `iframe`, `object`, `embed`) because they impose different restrictions on the SOP [31]. Attackers can use detectable differences to construct XS-Leaks to gather information about a user’s state on a target website.

Example: Fixing and breaking HotCRP. The web application HotCRP allows users to download a specific file only if they have the required access rights. An error event is thrown if the user’s authentication state does not allow accessing this state-dependent resource. Sudhodanan et al. [35] showed this event could be detected cross-origin, using a specific detectable difference: the status code. Subsequently, the reported XS-Leak was fixed in HotCRP [1]. Using our methodology, implemented in AUTOLEAK, we found a new XS-Leak in the fixed version of HotCRP. The detectable difference used is the presence or absence of the `Content-Disposition` header. Similarly to Sudhodanan et al., it allows an attacker to determine whether a victim has access to a specific submission on HotCRP. In our finding, an attacker website can use an `iframe` as the *inclusion method* for the cross-origin content to iterate over all submissions to deanonymize reviewers or authors.

State of Research. For XS-Leaks, there are two possible research directions. The first direction is to analyze real-world web applications like HotCRP. This has been done by manually creating several test cases, with different inclusion methods, and a manually crafted list of possible leak technique. Since authentication and authorization mechanisms differ between web applications, manually prepared scripts are necessary to access the different states with the same browser. For example, Sudhodanan et al. [35] prepared scripts for 62 web pages and used a list of 40 leak techniques to analyze them (including HotCRP), and Rautenstrauch et al. [30] used 34 leak techniques and manually signed in on 100 websites for their login analysis. The list of leak techniques may be incomplete, so an existing XS-Leak may be overlooked, leading to a significant false negative rate. This gap motivates our first research question:

RQ 1: Given a state-dependent resource, is there a systematic way to automatically find *all* leak techniques for all known inclusion methods?

We answer this research question in the affirmative: using EXTRACTOR to get the DOM graph and COMPARATOR to compare multiple DOM graphs as sketched below, we get a complete list of *all* leak techniques for a given state-dependent resource and a set of inclusion method in the cross-origin DOM. For example, Rautenstrauch et al. [30] concentrate on the 34 most promising leak techniques that allow for fast computations and large-scale (browser and web application) evaluations. In contrast to this, we treat each DOM object and property as a *possible* leak technique; depending on the

test case, up to 8 403 leak techniques could thus be identified (Table 2). Also, instead of finding bugs in specific web applications, we focus on the second, more generic line of XS-Leak research: finding leak techniques in web browsers. For this, we must construct lab-based test cases, which include a state-dependent resource and an inclusion method. In contrast to prior work, we do not need to specify leak techniques for these test cases since they are generated by AUTOLEAK. While there is only a finite number of inclusion methods, an infinite number of state-dependent resources can be constructed. We need a carefully chosen, finite subset of state-dependent resources for browser tests. This motivates our second research question:

RQ 2: Is it possible to systematically construct lab-based state-dependent resources to automatically find *all* leak techniques for a given set of inclusion methods in a given web browser?

We try to answer this question for leak techniques induced by two types of detectable differences. First, we created test cases for detectable differences induced by (the presence or absence of) single HTTP headers. This includes, amongst others, the novel XS-Leak in HotCRP induced by the `Content-Disposition` header as described above. For this purpose, we collected a list of 119 HTTP headers, resulting in 77 112 test cases due to their combination with different inclusion methods and file types. Second, we created 50 544 test cases for detectable differences induced by 13 HTTP status codes. Another interesting approach for creating test cases was recently published in [30].

Methodology: Comparison of DOM Graphs. We use a novel transformation to map a web browser’s actual DOM and JavaScript environment, which depends on the loaded resource, to a directed graph, which we call *DOM graph*. DOM graphs are huge. Figure 1 depicts a DOM graph of a simple webpage (containing only two HTML tags) in Firefox, visualized with FM-3 in Tulip [4]. Surprisingly, even this simple webpage has 8 823 nodes and 11 427 edges. Creating DOM graphs is far from straightforward: The nodes (objects) in the DOM do not have unique names but are addressed through edge paths. The edges (properties) have unique labels, but these labels may occur several times at different edges. We invented a unique naming scheme for nodes, the *shortest path labeling*, and a reliable method to identify loops. We construct each graph by using a breadth-first traversal of the actual DOM, starting with `window` as the root object. Our novel methodology to automatically list *all* cross-origin observable leak techniques, for a given state-dependent resource and a set of inclusion method, is based on the comparison of two such DOM graphs:

- (1) Our JavaScript-based EXTRACTOR runs in the execution context of an empty web page. It embeds the target state-dependent resource *sdr* using an inclusion method *i*. EXTRACTOR creates a DOM graph containing all accessible objects and properties. In particular, the DOM graph contains those nodes and edges that are observable cross-origin under the given inclusion method *i*.
- (2) We run EXTRACTOR twice: first with *sdr* in state 0, and then in state 1. These runs result in two DOM graphs.
- (3) By running a graph difference algorithm implemented in COMPARATOR, the identical (isomorphic) parts of both graphs are

removed, leaving only the parts in which the graphs differ. The remaining nodes are both observable cross-origin, and differ between the two states, and therefore are potential leak techniques.

Empirical Study. Our method is empirical and results can be reproduced with our open-source tool AUTOLEAK. We have condensed our findings in Table 2, where for each triple, consisting of a detectable difference, a specific web browser, and an inclusion method, we list the number of leak techniques and details on the root causes. For example, if we use the presence or absence of the Content-Disposition header as detectable difference, and an <object> element as inclusion method, in Safari there are 8403 leak techniques. On top, we identified 5 novel *classes* of XS-Leaks that detect the existence of HTTP headers that any previous research has not described.

Contributions. We make the following contributions:

- We are the first to present a comprehensive methodology to systematically detect *all* leak techniques, for a given state-dependent resource and a set of inclusion method, by creating DOM graphs for each state and comparing them (Section 4). We describe a novel algorithm to create *DOM graphs* from the actual DOM of a web page.
- We provide AUTOLEAK, a framework for creating test cases, graph-based snapshots of a browser’s DOM, identifying pairwise differences in DOM graphs, and automatically generating PoC exploits (see Section 5).
- We evaluate a test suite of 151 776 test cases, of which 16 028 produce leak techniques (Section 6). On top, we identified 5 novel classes of XS-Leaks. Since the results are reproducible, researchers and analysts can use AUTOLEAK in continuous integration processes to verify their fixes for XS-Leaks, or if novel XS-Leaks are introduced together with new browser features.
- We show that AUTOLEAK can also be used to analyze real-world web applications by evaluating the Tranco Top 50 websites and applying it to HotCRP, YouTube, and Slack (see Section 7).

In the interest of open science, AUTOLEAK and its source code, including the data set of all differences within the DOMs, are available.¹ We reported all new leak techniques to Google, Mozilla, and Apple (see Section 6.7).

2 BACKGROUND

2.1 XS-Leaks

An XS-Leak can be described as a function $xsl(sdr, i, t)$ [25]. It takes three arguments and outputs a single bit b' . Its first argument is a state-dependent resource, which is a URL to some web resource that differs in certain aspects based on the user’s state. The second argument is the inclusion method i , a method to trigger a cross-origin request to a specific state-dependent resource. The third argument is the leak technique. Its basic idea is to infer the user’s state. The output of the $xsl()$ function corresponds to the user’s state if the given leak technique is exploitable with inclusion method i .

¹<https://autoleak.org/>

Running Example: An XS-Leak based on the X-Frame-Options header. For a better understanding, we introduce a running example which we will reference throughout the paper (cf. Figure 3, Figure 4). Suppose an attacker wants to infer whether a victim is currently signed-in into an account at example.com. The X-Frame-Options (XFO) header XS-Leak works as follows:

- We use `https://myaccount.example.com` as the state-dependent resource. If the victim is signed in, the web server delivers the resource with the HTTP response header `X-Frame-Options: DENY`. Otherwise, the header is absent.
- The user’s set of states is: $s \in \{\text{signed in, signed out}\}$
- The attacker uses an <object> element as the inclusion method: `<object data='https://myaccount.example.com/'>`.
- In Firefox, there are several leak techniques which can be detected with AUTOLEAK:
 - if the user is **logged in**, an error event is thrown on the <object>, the number of subframes is `window.length==0`, and `window[0]` is **undefined**.
 - If the user is **logged out**, a load event is thrown, the number of subframes is `window.length==1` and a cross-origin window handle is reachable via `window[0]`.

2.2 Graphs & Labeled Directed Multigraphs

Let us briefly recall the definition of a directed graph.

Definition (Directed Graph): A *directed graph* $G = (N, E)$ consists of nodes $n \in N$ and edges $e \in E \subseteq N \times N$. Directed graphs correspond to our naive intuition about the Document Object Model (DOM): each object is a node and we can go from one node to another by following labeled directed edges, which are the DOM properties. For example, we can access the location object either using `window.location` or `window.document.location`. Unfortunately, the DOM cannot be mapped to such a simple structure: There are parallel edges in the DOM (e.g., `firstChild` and `firstElementChild`), and there are multiple loops in the DOM (e.g., the `window`, `frames`, `top`, `parent` all connect `window` with itself). Thereby, we need a more complex structure.

Definition (Labeled Directed Multigraph): A *multigraph* is a graph that may have more than one edge connecting two nodes and may have multiple loop edges on single nodes. A *labeled directed multigraph* (LDMG) is a multigraph with labeled nodes and edges. Formally, an LDMG is an 8-tuple $G = (N, E, s, t, \Sigma_N, \Sigma_E, \ell_N, \ell_E)$ where N is the set of nodes, E is the set of edges, $s : E \rightarrow N$ is a map that assigns a start node to each edge, and $t : E \rightarrow N$ assigns the target node. Σ_N and Σ_E are the sets of labels for nodes and edges, and ℓ_N and ℓ_E are the functions assigning these labels. Although this definition is mathematically complex, LDMGs can be efficiently generated and stored. Graph algorithms can be adapted to LDMGs, and allow for further analysis.

3 SYSTEMATIZATION OF KNOWN XS-LEAK ANALYSES

We shed light on prior work on XS-Leaks. Table 1 gives an overview.

Tool-based Vulnerability Detection. One of our paper’s main contributions is a systematic and automatic tool-based approach for identifying XS-Leaks at scale. Prior work conducted either manual vulnerability detection or also used tool-supported analyses.

Table 1: We compare research close to our work and identify a gap in large-scale detectable differences within HTTP headers and new leak techniques.

Name	Paper	Tool	Investigation Target:		Detectable Differences			New IM	New LT
			Websites	Browser	Header	Body	Status Code		
Scriptless Attacks	Heiderich et al. [17]				⊙	⊙		⊙	⊙
Unexpected Dangers	Lekies et al. [26]	⚡	⬆		⊙	⊙			⊙
SOP Evaluation	Schwenk et al. [31]	⚡		⬆	⊙	⊙		⊙	
Leaky Images	Staicu and Pradel [33]		⬆					⊙	⊙
Cosi	Sudhodanan et al. [35]	⚡	⬆	⬆	⊙	⊙	⬆	⊙	
Oh, the Places	Janc and West [21]			⊙				⊙	
Pool-Party	Snyder et al. [32]			⊙		⊙			⊙
Leakuidator	Zaheri and Curtmola [43]	🛡	⊙				⊙	⊙	
XSinator	Knittel et al. [25]	⚡		⬆	⊙	⊙	⊙	⊙	⊙
Same-Site Cookies	Khodayari and Pellegrino [23]		⊙			⊙			⊙
SoK: XS-Leaks	Van Goethem et al. [36]	🛡		⊙	⊙	⊙	⊙		⊙
Leaky Web	Rautenstrauch et al. [30]	⚡	⬆	⬆	⊙	⊙	⬆		⊙
AUTOLEAK		⚡	⊙	⬆	⬆	⊙	⬆		⬆

⚡ Offensive tool 🛡 Defensive tool ⬆ Large Scale Analysis & Contributions ⊙ Contributions

Roughly every second paper listed in Table 1 analyzed XS-Leak manually [17, 21, 23, 32, 33]. These works concentrate on finding severe issues in specifications or conducting semi-automated and manual security evaluations. Lekies et al. [26] was, to the best of our knowledge, the first work which automatically detected such information disclosure leaks. The authors implemented a Chrome extension that automatically requests each state-dependent resource twice, the first time including and the second time excluding attaching cookies to the request. Sudhodanan et al. [35] implemented BASTA-COSI which crawls websites and automatically generates XS-Leak attack exploits based on known XS-Leaks. Schwenk et al. [31] and Knittel et al. [25] implemented a browser evaluation test bed as a website. Their tools used a set of hard-coded leak techniques while AUTOLEAK autonomously discovers new leak techniques. Zaheri and Curtmola [43] and Van Goethem et al. [36] implemented a tool-based mitigation for XS-Leaks.

Investigation Target: Website vs. Browser. XS-Leak research methodologies distinguish between their target. Some works mainly target the detection of issues in websites and web applications [23, 26, 30, 33, 35, 43]. Others focus on identifying browser-related issues [21, 23, 25, 30–32, 35, 36]. When correlating the browser target with the offensive tool-based approach, three works provided a website evaluation test bed for browsers [25, 30, 31]. Note that BASTA-COSI [35] is a tool to detect XS-Leak attacks in websites. Heiderich et al. [17] targeted neither websites nor browsers but revealed mistakes in the HTML specifications.

Detectable Differences. A state-dependent resource can have a detectable difference in any part of the HTTP response. Therefore, we distinguish in which HTTP part prior work identified them.

HTTP Header. With AUTOLEAK, we specifically address XS-Leaks that detect HTTP-Headers. We identified several prior work that deals with detectable differences in HTTP-Headers [17, 25, 26, 30, 31, 35, 36]. These works have mainly put their focus on previously known XS-Leak-Headers instead of systematically investigating a broader set. Schwenk et al. [31], for example, only considered

CORS HTTP-Headers. The HTTP header column in Table 1 emphasizes the gap. A comprehensive investigation of HTTP headers as detectable differences was not yet considered.

HTTP Body. Most of the related work in the field of XS-Leaks has their major contribution in the detection of new XS-Leaks in the HTTP body [17, 23, 25, 26, 30–33, 36]. Finding detectable differences in this HTTP part is the most complicated because the combination and nesting of HTML elements lead to an enormous number of possible HTTP bodies. In contrast to prior work, we decided to put our major effort into the large-scale analysis of HTTP headers.

Status Codes. Status codes in HTTP responses are a well-known possible cause for XS-Leaks and have been analyzed by Knittel et al. [25], Rautenstrauch et al. [30], Sudhodanan et al. [35], Van Goethem et al. [36], Zaheri and Curtmola [43]. They are a convenient root cause for XS-Leaks. Two works [30, 35] have intensively investigated them at scale. Since there is a finite set of HTTP status codes, we included them in our methodology for completeness.

New Inclusion Methods. A large body of research used existing XS-Leaks and created new variants. The main recipe is to use the underlying leak techniques of a known XS-Leak, but combine it with different inclusion methods. For example, Schwenk et al. [31] found a new login oracle XS-Leaks in Edge by using the `<link>` tag as the inclusion method. Sudhodanan et al. [35] systematically combined known leak techniques with a large body of inclusion methods based on JavaScript event handlers in their attack classes. Zaheri and Curtmola [43] used `<video>`, `<audio>`, and `<object>` tags in their XS-Leak variants. We conclude that extending the number of inclusion methods with known leak techniques is a reasonable but well-studied approach.

New Leak Techniques. Previous works identified new leak techniques manually. Even if their research was tool-supported, these tools did not reveal new leak techniques. For example, Knittel et al. [25] revealed new leak techniques that abuse global limits with WebSocket. The leak techniques were identified based on a formal model and then integrated into their evaluation testbed. The

manual identification was yet the preferred method to identify new leak techniques in prior research [17, 26, 30, 32, 33]. In summary, we see a gap in the systematic and automatic detection of new leak techniques.

4 METHODOLOGY TO FIND XS-LEAK

At the core of our methodology is the comparison of DOM graphs. It is applied to different test scenarios, depending on whether we want to mass-test web browser implementations or analyze web pages on the Internet. To find all leak techniques for a given inclusion method and state-dependent resource, two DOM graphs are generated, and the difference between these two graphs is computed.

4.1 Extracting DOM Graphs

There is no simple method to map the JavaScript DOM of a web page to an LDMG $G = (N, E, s, t, \Sigma_N, \Sigma_E, \ell_N, \ell_E)$, and unfortunately, the browser's built-in JavaScript does not provide a method to serialize and export the entire DOM graph at once. One can enumerate and traverse each object's properties. We start with the root element of the DOM and insert a root node n_0 labeled with the empty string as the first node of the graph. The root node n_0 is added as the first node to the list \mathcal{L} of nodes to be processed. We then extract all root node properties n_0 and add an edge for each. For each edge e , we add (e, n_0) to the function s . We label each edge with a complex label that includes the property's name and attributes (e.g., being frozen, being extensible, etc.). Each edge e is directed from n_0 to another DOM object O_i . It may be a new node n_i or already contained in the DOM graph. Each such edge e is added to the set E . To detect loops and multiple paths, we test each newly found node n_i for equality against all previously found nodes (see the Section 5.2 for implementation details). Only if n_i is different from all previous nodes, it is added to the set N and appended to the list \mathcal{L} . An entry (e, n_i) is added to function t , and n_i is labeled with the path from the root node to n_i . If n_i is equal to any previous node n , an entry (e, n) is added to t . New nodes are assigned complex labels consisting of multiple components: (1) an ID label containing a string that describes the shortest path to the node, (2) a type label with the data type of the node (e.g., string, Boolean, function, object), and a value label. Once all properties e of n_0 are processed, n_0 is removed from the list \mathcal{L} . The next node n_1 from \mathcal{L} is then processed in the same way as described above for n_0 . Thereby, for each node in the list, all edges are detected. This process expands the graph, and then, the node is removed. The graph extraction algorithm terminates when the node list \mathcal{L} is empty. Details of the algorithm are described in Section 5.2.

4.2 Computing the Difference of Two DOM Graphs

The algorithm for computing the difference between two LDMGs G_0 and G_1 (cf. Figure 3(a)) starts with the root node n_0 , performs a breadth-first search in both LDMGs in parallel. All outgoing edges of n_0 , called the *out-star* of n_0 , are examined in both graphs. Two edges e, e' of the out-star are identical if they have the same label ($l_E(e) = l_E(e')$) and the same target node ($t(e) = t(e')$). Identical edges are removed from both graphs. Two nodes n, n' are identical if they are isolated after removing identical edges (i.e., they have

neither incoming nor outgoing edges) and if they have the same label. As a result, we get two graphs G'_0 and G'_1 (cf. Figure 4): G'_0 is the graph G_0 with all edges and nodes identical to G_1 removed, and G'_1 is the graph G_1 with all edges and nodes identical to G_0 removed. Consequently, if two DOM graphs are isomorphic, the difference graphs will be empty. This algorithm is efficient as it effectively is the synchronized depth-first traversal of the two graphs, which runs in linear complexity with worst-case performance $O(|V| + |E|)$.

4.3 How to Find New XS-Leaks in Browsers

For a given state-dependent resource and a set of inclusion methods, we aim to *automatically* find *all* leak techniques in web browsers, both known and novel. More precisely, we want to construct functions $xsl(sdr, i, t)$ in a systematic manner, which returns different bits depending on the state of sdr . For the inclusion methods i , this construction is straightforward: the HTML standard only defines a limited number of them for cross-origin resources, which we collected. The input set of leak techniques t are the DOM objects and properties which correspond to the nodes and edges in the difference graphs G'_0 and G'_1 . We get them by running EXTRACTOR and COMPARATOR. This is efficient and usable since we get the complete input set for t with a single run of these tools, automated for each possible inclusion method i .

Only one problem remains to be solved: how do we systematically generate a finite list of state-dependent resources sdr ? This problem is not universally solvable: by nesting arbitrary HTML elements in arbitrary depth, we can generate an infinite list of possible test cases. The selection of a suitable subset from this infinite number of test cases is a community effort, for instance, the most typical HTML nesting scenarios. A single research paper cannot cover this systematically. Therefore, we take another approach. Since only a few prior works investigated the interaction of HTTP headers with XS-Leaks (cf. Table 1), our goal is to fill this gap with a comprehensive large-scale analysis of the influence of HTTP headers on different inclusion methods. Our running example (cf. Section 2) exemplifies that these headers may induce a difference in the web browser's rendering of an HTTP response and that these differences can be found *somewhere* in the DOM. If such a difference can be identified cross-origin, we identified that as a new leak technique. We keep the HTML part of our test cases simple: one state-dependent resource sdr is included with one inclusion method i , without any HTML nesting. Thus, we can derive a systematic way to build our sdr test cases. We start with a list of HTTP headers that may eventually influence the rendering of sdr . Each of these headers typically has a limited list of parameter values. There are exceptions, such as the CSP policy which requires a complex grammar. For each meaningful header-parameter combination, we design a state-dependent resource in two variants. In the first variant, the header is present with the selected parameter. In the second variant, the header is absent. We additionally constructed test cases for status codes, file types, and defensive headers (c.f. Section 6).

4.4 How to Find New XS-Leaks in Web Applications

Since each real-world web application has unique characteristics, a manual step is always necessary: we must determine the state-dependent resource and the two or more states we want to test. In [35], 62 scripts were manually created to be able to test the 62 web applications automatically. We proceed similarly and use two *different browser profiles* for each web application to access the same state-dependent resource. One browser profile can access the state-dependent resource in state 0 and the other in state 1. Our test cases involve different inclusion methods. For each inclusion method, we use EXTRACTOR and COMPARATOR to generate the difference between the two DOM graphs. For demonstrating the feasibility of detecting XS-Leaks in websites with AUTOLEAK, we chose three fundamentally different, high-level target applications: YouTube, Slack, and HotCRP. We found XS-Leaks in all of them, see Section 7.

4.5 False Positives and False Negatives

AUTOLEAK has a low False Negative rate, given that it finds XS-Leak described in previous research, and that it uses all DOM objects and properties. However, it may miss differences that are observable through other means than the DOM tree (e.g., the operating system or cache usage [13, 36]), global limits or timing-based side-channels. True Positives are checked for validity by automatically generating PoC exploits. Some parts of the DOM change inherently and are irrelevant to XS-Leaks: E.g., `performane.timeOrigin` contains a timestamp that changes each and every time a website is loaded. We implemented an ignore-list of these ever-changing variables, effectively removing these False Positives before they are displayed to the analyst. This list was created by comparing the two DOM graphs of the same document before and after a reload.

5 AUTOLEAK: DETECTING XS-LEAKS AUTOMATICALLY

AUTOLEAK consists of four main components: a) a test runner for automating graph creation en masse (AUTOMATOR), b) a JavaScript crawler for extracting DOM graphs (EXTRACTOR), c) a framework for storing and comparing differences in graphs (COMPARATOR), d) and an exploit generator (POCGENERATOR). For a better understanding of these components and their interaction with each other, we use our running example from Section 2.

5.1 Automator: Automated Test-Case Execution

AUTOMATOR is a test-runner for automatically generating large quantities of DOM graphs for our test cases. For this purpose, AUTOMATOR instruments and uses various browsers. We need to evaluate many test cases to find XS-Leaks based on HTTP response headers. AUTOMATOR consists of an HTTP server to deliver customizable HTTP responses, a Celery [7] task queue and state management in MongoDB [8]. AUTOMATOR loads the test cases from a config file. After the analyst confirms a selection, AUTOMATOR queues the tasks and processes them until finished. Each test case takes 25 seconds on average, from the instantiating of the browser process, opening the URL, loading the state-depending resource, extracting the graph of the website, closing and resetting the browser, switching state,

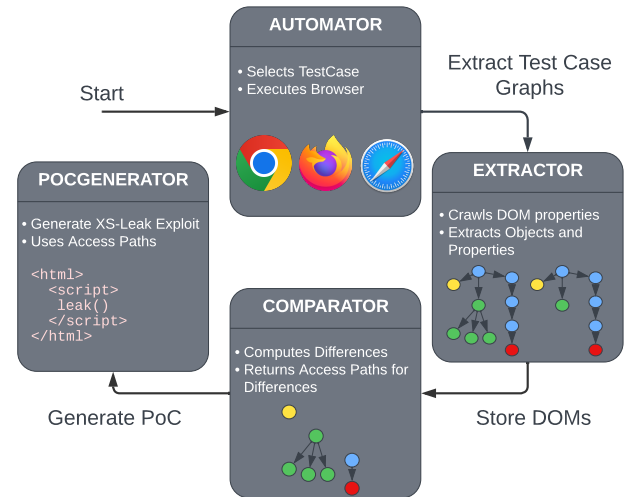


Figure 2: AUTOLEAK’s architecture. Four components interact to detect XS-Leaks automatically. AUTOMATOR provides and runs the test cases. Each test case consists of identical websites except for one feature, e.g., an HTTP header or the status code. EXTRACTOR traverses the DOMs for both cases. COMPARATOR estimates the difference between the DOMs. POCGENERATOR can then generate exploits.

opening a new browser, creating the second graph, running the comparison algorithm and the storing the results in the database. Each test case consists of an inclusion method (e.g., `<object data=j...>`), a detectable difference (e.g., X-Frame-Options: DENY versus no header), a data type (e.g., a JPEG image), and a browser (e.g., Firefox). To minimize the noise generated by the test harness, we entirely keep track of the test case on the server side, meaning the state-dependent URL is constant and will return different HTTP responses depending on its server-side state. AUTOMATOR uses Playwright to automate headless browsers and dynamically injects the code of EXTRACTOR via the DevTools protocol, so no additional `<script>` element is needed.

Challenge: State Switching without Cookies. To find the leak techniques for a given state-dependent resource, it must be opened in a browser twice - once in each state. In between, the state must be switched. In real-world applications, the state usually depends on the HTTP session cookie. The session cookie is a secret value that the browser appends to HTTP requests to authenticate to the server. However, *sdrs* can depend on other factors, such as the IP address. An XS-Leak could leak whether the user is part of a private network by testing a *sdr* of an intranet website. *sdrs* with IP-based geoblocking can leak the geographic location of the user. Also, there are more possibilities besides the cookie storage to store client secrets (e.g., localStorage, WebSQL) and to transfer them to the server (e.g., through WebSocket connections or GET parameters). Consequently, we constructed AUTOMATOR to be agnostic of the state-switching mechanism. The tool manages the state of each *sdr* completely server-side and switches between each call of EXTRACTOR. This way, we also find leak technique for differences

that don't work with cookies, such as when the server disallows the sending of cookies via CORS.

Running Example in AUTOMATOR. AUTOLEAK provides a web interface on <https://autoleak.org>. It provides a pre-configured list of inclusion methods, detectable differences, file types, and browsers. AUTOLEAK triggers the AUTOMATOR component. Although AUTOMATOR's architecture allows running thousands of test cases automatically, we keep this example minimal. We select `<object>` as inclusion method and `X-Frame-Options` as detectable difference. We set the file type to `html` and select Firefox as the browser under test. Pressing the `run`-button automatically instantiates a headless browser instance on our server. The browser visits a specific test-case URL, which invokes the EXTRACTOR component. Afterward, it visits the same URL a second time, but this time, AUTOMATOR changed the state of the state-dependent URL: the document is served with `X-Frame-Options: DENY`, which represents the detectable difference. Again, it invokes the EXTRACTOR.

5.2 Extractor: Extracting DOM Graphs

EXTRACTOR solves the challenge of translating the DOM structure of a given test page into an external mathematical object, a labeled directed multigraph (LDMG, Section 2.2). Since EXTRACTOR is a JavaScript program, it has the same capabilities that regular users (or attackers) have. Thus the exported DOM LDMG exactly maps the view of web attackers on included target resources. EXTRACTOR enumerates JavaScript properties and uses them to traverse the DOM from object to object. Starting with the root object, EXTRACTOR executes a breadth-first traversal of the DOM objects. The collected information is stored as nodes, edges, and their assigned labels of a labeled directed multigraph. For each object, it collects information, for example, whether the object is *frozen* or *extensible*, the result of applying different serializations and the object's *prototype* chain. All this information is stored along with the object node in the LDMG. Thus, the *labels* of the nodes represent complex data structures on their own. EXTRACTOR discovers each object's properties via various methods, like `for(x in obj)`, `Object.getOwnPropertyDescriptors(obj)`, `Object.keys(obj)`. Each property is stored as an edge in the LDMG, labeled with its name. After collecting data on the objects and all their properties, the neighbor objects are queued for traversal. For detecting cycles and loops, each newly discovered object is compared to all previously visited objects using both `Object.is(obj1, obj2)` and `===`. The crawling ends when the queue is empty. EXTRACTOR can receive callback functions for each new node or edge. For example, it can add event listeners to all nodes or instantiate the performance API (see example below). This expansion is executed *before* the crawling process starts. Afterward, EXTRACTOR does not modify the DOM anymore. The collected graph data is uploaded from the browser to a server so that COMPARATOR can further process them.

Challenge: Evading DOM Pollution. EXTRACTOR is itself a JavaScript program, so there is the danger of *polluting* the DOM through modification made by EXTRACTOR. The extractor must not be able to access a reference to itself. If it were to, the extractor risks recursion by iterating over its own queue entries, thus introducing noise into the measurements. Therefore, EXTRACTOR is implemented as an anonymous function. Anonymous functions

are nameless and have their own scope for local variables. Only local variables are used because they are not visible in the DOM: `(function () { x = 5 })()` would create the global variable `window.x` and would therefore pollute the DOM. Instead, we use `(function () { var x = 5 })()`, and this local variable does not appear outside of the scope of the function. For more information on global and local variables, see [20].

Challenge: Reliable execution of EXTRACTOR. In JavaScript, accessing an object's property can fail for many reasons, such as being restricted by the SOP, invalid usage of bound methods, or browser bugs. Thus, the extractor extensively uses `try` and `catch` to create and extract the DOM graph reliably. This approach allows collecting and storing every error message as node or edge properties so they can be part of the subsequent analysis even if aspects of the extraction failed.

```
1 XSL_im = document.createElement('object')
2 XSL_im.data = 'https://test.com/testcase'
3 XSL_run = () => {
4   document.body.appendChild(window.XSL_im)
5 }
```

Listing 1: Testcase for the inclusion method `<object>` that includes the state-dependent resource.

Running Example in EXTRACTOR. EXTRACTOR generates two DOM graphs for each test case that AUTOMATOR processes. The HTML source code for both calls, as provided by AUTOMATOR, is depicted in Listing 1. Note that Listing 1 does not simply use an HTML `<object data="..">` element. We choose to generate all test cases with JavaScript. This design choice allows EXTRACTOR to crawl the DOM before executing the inclusion method (`XSL_run()`) and add all possible event listeners to all nodes. For example, EXTRACTOR adds `load` and `error` listeners to the `<object>`. Afterward, EXTRACTOR executes the inclusion method (`XSL_run()`) and crawls the DOM a second time, and extracts all nodes and edges. The result is depicted in Figure 3 and stored in a database on our server so that COMPARATOR can further process them. For the running example, AUTOMATOR invokes EXTRACTOR to create two graphs: one with 11 427 edges and 8 823 nodes, and another one with 11 403 edges and 8 804 nodes.

5.3 Comparator: Differences in DOM Graphs

COMPARATOR can identify differences in two DOM graphs. The intuition behind COMPARATOR is that it removes everything the DOM graphs have in common, starting from the root.

Algorithm. The algorithm is described in Section 4.2. Here, we only mention details that are important for the interworking of the different components of AUTOLEAK. Edges that are different in the two graphs are kept. For each such edge, the *common shortest path* to it is assigned as an additional attribute and is stored in a list called *roots of differences*. This list contains the first contact points with differing graph components but not the subsequent edges. We discovered that these points of divergence are helpful for summarizing differences between graphs, and are later used in POCCGENERATOR. In the end, the BFS has deleted every edge that

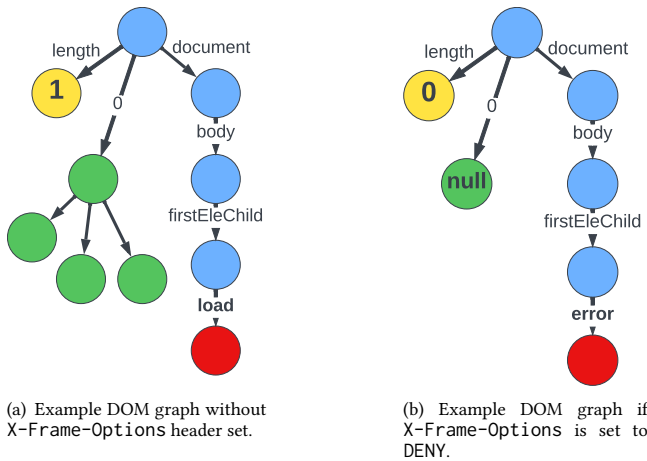


Figure 3: Our example extracts two different DOM graphs depending on the URL’s state. Without X-Frame-Options, the object’s content is accessible (green nodes) and `window.length=1`. If set to DENY, the object throws an error event instead of load and `window.length=0`.

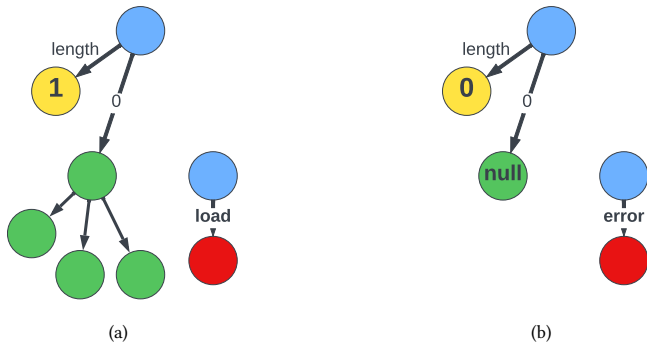


Figure 4: COMPARATOR computes differences between graphs. The left side shows G'_0 , the differences of Figure 3(a) when removing everything identical to Figure 3(b). The right side shows G'_1 .

identical in both graphs, and kept all the differences, including a list detailing where differences begin. The remaining graphs, which may now be unconnected, are returned and the results can be analyzed. We implemented this algorithm in Python using the MultiDiGraph implementation from NetworkX [15].

Challenge: Shortest Paths Variation. The common shortest path attribute is useful for understanding graph differences. When describing differences in DOM graphs, it is natural to locate differences using the shortest path from the Global Object (GO). However, the shortest paths to corresponding nodes for interesting edges can be different from one DOM graph to another, because extra edges in one DOM graph may provide a local shortcut. This occurs quite often in practice, for example, if a website uses an `iframe` to include a cross-origin website. Then, `window[0]` usually points to a cross-origin window handle and is mostly inaccessible.

Thus, the shortest path to the `iframe` itself is `window.document.body.firstChild`. If the `iframe` fails to load the resource and defaults to an `about:blank` page, `window[0]` is same-origin. Now, the shortest path to the `iframe`-element itself is `window[0].framingElement`. Consequently, if we would only compare a list of shortest paths to every node in both graphs, most of them would be false positive differences. To ensure that we can uniformly locate differing edges, we use the common shortest path concept - a shortest path that exists in every graph. This new label is computed during the synchronized breadth-first traversal. The common shortest path in our example is `window.document.body.firstChild`, since the `iframe` can be reached via this path in both cases (cross-origin `iframe` loaded vs. cross-origin loading failed), and there is no shorter path available in both DOM graphs.

Challenge: Filtering Expected Changes. Certain parts of the DOM depend on the time or the URL, for example, `performance.timeOrigin` and `document.lastModified`. To reduce noise and highlight only relevant differences, we filter out irrelevant changes which do not depend on the inclusion method or state. These differences were identified by comparing the DOM graph of two identical pages (which are, therefore, in the same state).

Challenge: Condensing Massive Tables into Tags. The mass comparison of DOM graphs produces large amounts of information, as each pair of graphs can result in thousands of differences. Our large scale browser study consisting of 151 776 test cases produced 303 552 graphs, which consists of a total of 1 865 021 895 properties and 1 411 433 830 objects. We created a rule set for assigning tags to condense and summarize the complex changes between graphs. Tagging rules are regular expressions executed on the roots of change.

Running Example in COMPARATOR. Figure 4 shows a simplified version of COMPARATOR’s result for the two graphs of the running example (cf. Figure 3). There are several differences: `length` property differs in value, one graph contains a `window` under `0`, and one does not, and different events are caught. Out of the collective 22 828 edges of both graphs, COMPARATOR computes a precise list of 208 unique paths that have changed depending on whether X-Frame-Options is set. Manually reviewing each differing DOM path is labor-intensive. Instead, to condense complexity, we use the feature of COMPARATOR to summarize the roots of differing graph components. For Firefox, there are 15 roots of change. Ultimately, COMPARATOR comprehensibly summarizes the original list of 208 differing DOM paths with 6 tags: **elemdims**, **events**, **onerror**, **onload**, **window[0]**, and **windowlength**.

5.4 PoCGenerator: Automatic Exploit Generation

POCGENERATOR uses information from COMPARATOR to automatically create a leak technique: (1) the common path to access the node that differs. (2) the values of difference. Together with the inclusion method, POCGENERATOR builds an HTML file that contains the resulting XS-Leak as JavaScript. In our running example, COMPARATOR identified multiple differences and POCGENERATOR used the `length` difference to create the code depicted in Listing 2.


```
1 XSL_im = document.createElement('object')
2 XSL_im.data = 'https://test.com/testcase'
3 XSL_run = () => {
4   document.body.appendChild(window.XSL_im)
5 }
6 XSL_leak = () => {
7   if(window['length'] === 0){
8     console.log('State 0') // X-Frame-Option header set
9   }else{
10    console.log('State 1') // X-Frame-Option header NOT set
11  }
12 }
```

Listing 2: Generated exploit from POCGENERATOR.

6 BROWSER EVALUATION: DETECTING NOVEL XS-LEAKS AUTOMATICALLY

AUTOLEAK automatically finds *all* leak techniques for a given state-dependent resource and a set of inclusion methods usable to create XS-Leaks. We extended known classes with new leak techniques (Table 2) and identify five entirely new XS-Leaks (Table 3).

6.1 AutoLeak Setup

A test case consists of an inclusion method, a difference we want to test, and a browser. The difference consists of header, status code, and body. We created a configuration file composed of the following testing parameters:

Inclusion methods. We collected known inclusion methods from existing research and standards [25, 35, 36]. Furthermore, we evaluated HTML tags and JavaScript APIs that allow fetching resources by considering the community-based HTTPLeaks list [16].

Headers. There are too many HTTP header and value combinations to test them all in a reasonable time. Furthermore, most of them are not processed by the browser, for example, the Server header. Therefore, they are unlikely to cause detectable differences in the browser. In order to make a suitable selection, we chose two approaches to build a representative set of headers. First, we collected headers that previous research showed to be detectable cross-origin (cf. Section 6.2). Second, we extend this set by extracting 500 most frequently used headers and value combinations that the HTTP Archive collected [3], a list of HTTP headers from the Firefox source code [29], Web Platform Tests [37], and MDN Web Docs [10] (cf. Section 6.3). In a first run, we tested all headers with a single file type with every inclusion method in all browsers. We then reduced the number down to the most promising 119 headers so that we could test them with every file type in a reasonable time.

Body. Some XS-Leaks only work when a specific file type is returned. We created minimal response bodies for the most used file types. In total, we chose nine file types. (1) *html* and (2) *json* are essential file types on the web and are often treated differently by the browser (e.g., MIME sniffing, Cross-Origin Read Blocking (CORB)). We included basic (3) *css* and (4) *js* files because there exist known XS-Leaks [26] and they are parsed by the browser when included with the corresponding inclusion method. (5) *gif*, (6) *wav*, and (7) *pdf* are interesting because they can be opened directly in the browser, which might cause XS-Leaks. We also added two additional HTML variants to cover known XS-Leaks: (7) one with an

iframe tag and (8) one with a focusable input element. Furthermore, we added a test case for (9) an empty response body.

Status codes. We selected the ten most used status codes collected by the HTTP Archive [3] and added those where known XS-Leaks exist [35]. Although AUTOLEAK allows us to easily extend this number, we chose to keep our test set small. We summarized the results of all 50 544 status code tests on <https://autoleak.org/>.

Browsers. We ran the tests in three major browsers: Chrome (🦊) v. 109, Firefox (🦀) v. 107, and Safari (🍏) v. 16.4. Previous XS-Leak research shows that browsers that share the same underlying browser engine exhibit the same XS-Leak behavior [25]. For example, browsers like Edge, Opera, Brave, and Vivaldi are all based on Chromium and showed very similar results in our initial tests.

6.2 Extending Known XS-Leaks

To prove the effectiveness of our tool, we verified that AUTOLEAK can detect known XS-Leaks. We collected headers where known XS-Leaks exist. These leaks allow an attacker to detect if the header is present in a cross-origin response. For some headers it is also possible to distinguish between different header values. This can be useful if the header is present in both cases. We created a configuration file based on these headers and let AUTOLEAK evaluate them automatically. We chose to run them in combination with all implemented inclusion methods, file types, and in 🦊, 🦀, and 🍏. Table 2 shows the results of AUTOLEAK for this configuration. Due to the lack of space, we show results for the file type *html*. Our website provides the complete list for all 16 028 test cases.

Old Bugs – New Insights. Our tool was able to reproduce known results, see Table 2. While these leaks have been studied by previous research we identified new edge cases and leak techniques because of our graph based approach.

The *CSP header directives* can be probed with the iframe attribute *cs* [42]. This type of leak only works if an attacker can detect that an error page is rendered in an iframe. Previously, attackers used to reload the iframe with a hash and count the *onload* events. This technique was fixed in 🦊 [18]. Our tool revealed that the patch only addressed a single leak technique. There is still a difference in the history API when reloading the iframe. Thus, the XS-Leak was not entirely fixed. The deprecated *afterScriptExecute* and *beforeScriptExecute* document *event handlers* do not fire when a script request is blocked in 🦀, for example, by the Cross-Origin Resource Policy (CORP) header. IFRAME tags do not resize when the response is blocked by *X-Frame-Options*. On the other hand, the object tag adapts its visual size depending on whether the content can be loaded.

6.3 Five New Detectable Headers

We could identify 5 novel XS-Leaks that detect headers that were not known by any prior research. Table 3 shows our findings.

Accept-Ranges: bytes This response header signals to the browser that the server supports partial requests. AUTOLEAK discovered that in 🦊 this header can be detected cross-origin for *wav* files. In an audio or video element, multiple requests are issued only when the header is set, which causes a difference in the Performance API. This XS-Leak is a handy example for the need of

Table 2: AUTOLEAK detects all leak techniques for a given test case. For example, the first row shows that one test case for the Content-Disposition-header in Chrome using the inclusion method *embed* has 6 669 leak techniques. These are distributed among the three DOM-sub parts events, perftentries, and window[0].

Inclusion Method(s)	Details	#Leak Techniques
Header: Content-Disposition: attachment		
🔗 embed	events, perftentries, window[0]	6669
🔗 iframe, object	contentDocument, contentWindow, events, perftentries, window[0]	6671
🔗 windowOpen	popup	6449
🔗 embed, object	elemdims, events, window[0], windowlength	136
🔗 iframe	contentDocument, contentWindow, perftentries, window[0]	7084
🔗 windowOpen	events, onfocus, popup	6923
🔗 embed	events, perftentries, window[0]	8391
🔗 iframe	contentDocument, contentWindow, events, perftentries, window[0]	8393
🔗 object	contentDocument, contentWindow, elemdims, events, perftentries, window[0]	8403
🔗 windowOpen	popup	8115
Header: Content-Security-Policy: default-src 'self';		
🔗 iframeCSPHashreload	history	2
Header: Content-Security-Policy: frame-ancestors 'self';		
🔗 object	contentDocument, contentWindow, elemdims, events, onerror, onload, perftentries, window[0]	6751
🔗 embed	elemdims, events, window[0], windowlength	136
🔗 iframe	events	100
🔗 object	elemdims, events, onerror, onload, window[0], windowlength	212
🔗 embed, iframe, object	perftentries	134
Header: Cross-Origin-Opener-Policy: same-origin		
🔗 windowOpen	popup	7
🔗 windowOpen	popup	6725
Header: Cross-Origin-Resource-Policy: same-origin		
🔗 fetch	error, fetchresponse	65
🔗 script, stylesheet	events, onerror, onload	164
🔗 embed	elemdims, events, window[0], windowlength	136
🔗 fetch	error, fetchresponse	68
🔗 object	elemdims, events, onerror, onload, window[0], windowlength	212
🔗 script	events, onafterscriptexecute, onbeforescriptexecute, onerror, onload	877
🔗 audio, image, video	perftentries	134
🔗 fetch	error, fetchresponse, perftentries	198
🔗 script	events, onerror, onload, perftentries	415
🔗 stylesheet	perftentries, styleSheets	151
Header: Timing-Allow-Origin: *		
🔗 embed, iframe, object	perftentries	10
🔗 audio, embed, favicon, fetch, iframe, image, object, script, stylesheet, video	perftentries	10
🔗 import	perftentries	8
🔗 audio, embed, fetch, iframe, image, object, script, stylesheet, video	perftentries	7
Header: X-Content-Type-Options: nosniff		
🔗 script	events, onafterscriptexecute, onbeforescriptexecute, onerror, onload	877
🔗 script	events, onerror, onload	281
Header: X-Frame-Options: DENY		
🔗 object	contentDocument, contentWindow, elemdims, events, onerror, onload, perftentries, window[0]	6751
🔗 embed	elemdims, events, window[0], windowlength	136
🔗 object	elemdims, events, onerror, onload, window[0], windowlength	212
🔗 embed, iframe, object	perftentries	134

our systematic evaluation with AUTOLEAK – it only appears in this niche combination of inclusion method and file type.

Allow-CSP-From Header Detection. If a site is rendered in an iframe, it can agree to let the parent origin set Content Security Policy (CSP) directives using the *csp* iframe attribute [39]. This leak is very similar to the CSP directive leak (cf. Section 6.2). As seen in Table 3 it can be detected if the attacker is allowed to set a CSP.

Cache-Control: no-store. The Cache-Control HTTP header is used to specify browser caching policies [9]. For example, the no-store directive prevents browsers from caching the response. AUTOLEAK found that for file types other than JSON and HTML it is possible to detect if the header is present in the response. This detection is possible since no performance API entry is created if the header is present.

Refresh Header Leak. The Refresh header is a non-standard HTTP header instructing the browser to refresh the page after a given time interval. It is mostly used with meta tags in HTML to

reload the page, but it can also be used as a header. Table 3 shows that it is possible to detect if a page is redirected in this way. This can be accomplished in several ways, for example, by listening for the load event of an *<iframe>*.

Server-Timing The Server-Timing header allows developers to expose server timing metrics via the Performance API. In 🌀, it can be detected for a few file types if included with embed, iframe, or object. Note that this violates the specification, as server-timing metrics should only be exposed for same-origin requests [14]. Not only can an attacker probe if the header is set, but also get the exact server timing information.

6.4 Evaluation Insights

Our header analysis revealed exciting insights.

Browser engines. The three major browser engines, Blink in 🌀, Gecko in 🍉, and WebKit in 🍎, overall exhibit similar behavior.

Table 3: AUTOLEAK identified five new XS-Leaks. Each row represents the results of one or more test cases. Results have been merged where applicable. See Section 6.3 for details.

Inclusion Method(s)	File Type	Details	#Leak Techniques
1. Header: Accept-Ranges:bytes			
🔒 audio, video	wav	perfenries	278
2. Header: Allow-CSP-From:*			
🔒 iframeCSP	wav	perfenries	124
🔒 iframeCSP	pdf	contentDocument, contentWindow, events, perfenries, window[0]	6671
🔒 iframeCSPHashreload	css, empty, gif, html, js, json, text	history	2
🔒 iframeCSPHashreload	wav	history, perfenries	126
🔒 iframeCSPHashreload	pdf	contentDocument, contentWindow, elemonload, events, perfenries, window[0]	6707
3. Header: Cache-Control:no-store			
🔒 fetch	css, gif, js, text, wav	perfenries	124
4. Header: Refresh:0; url=https://www.example.com/			
🔒 embed, iframe, object	css, empty, gif, html, js, json, text	events, onload	117
🔒 embed, iframe, object	wav	events, onload, perfenries	241
🔒 embed, object	css, empty, html, js, json, text	events, onload	129
🔒 iframe	css, empty, gif, html, js, json, text	events, onload	129
🔒 embed, iframe, object	css, empty, gif, html, js, json, text	events, onload	123
5. Header: Server-Timing:cache;desc="Cache Read";dur=23.2			
🔒 embed, iframe, object	css, gif, html, js, json, text	perfenries	70

Nevertheless, our results show that there exist small differences in which leak techniques and inclusion methods can be used for a given XS-Leak. For example, inclusion methods that allow embedding, like `iframe`, `embed`, and `object`, surprisingly react slightly differently across browsers when loading cross-origin resources. Overall, from our 5040 test cases with response body HTML, 🟡 is vulnerable in 677 cases, 🔒 in 530, and 🟢 in 403 cases. These results can partly be attributed to the fact that 🔒 implements more features, like the `iframe CSP` attribute, that increase the attack surface. For 🟡, we found that the implementation of the Performance API is responsible for a large number of vulnerabilities.

Softening the SOP. An attacker can detect the presence of headers that soften the SOP for a given origin. If wildcard (*) or origin reflection is used, the attacker’s page can detect this behavior: `TimingAllowOrigin: *` adds additional timing measurements to Performance API entries for cross-origin requests to

the resource. By checking for them, the header can be detected. `AccessControlAllowOrigin: *` allows any website to read the content of a response cross-origin, otherwise prohibited by the SOP. This header disallows the use of credentials such as cookies with wildcard origins [2]. Detecting this header can result in an XS-Leak if the state is dependent on different aspects such as the IP (e.g., to detect geo-blocking or VPN usage). `AllowCSPFrom: *` can be detected because a cross-origin page can test if it is allowed to set CSP directives for an `iframe` (cf. Section 6.3). These headers soften the SOP and create new accessible properties, which can be probed.

Hardening the SOP. Security headers, like XFO, XCTO, COOP, CORP and CSP harden the SOP by restricting resources from loading with certain inclusion method (cf. Section 6.6). These headers can be detected because the included resource throws an error, or DOM properties that are usually accessible, are missing or changed.

Defaulting to `about:blank`. When a header causes an error, framing elements often default to `about:blank`, a special null origin with no content. These fallback documents are usually accessible to the parent. This causes thousands of differences in the DOM graph because either the frame can be traversed or it is cross-origin.

Performance API Leaks. The Performance API [38] is very prone to side-channel attacks [25, 42]. For every request, one performance entry should be created. However, sometimes no resource-timing entry is generated in one of the two states. This leak is very common in 🟡 and especially in 🟢. Other smaller bugs expose small details about a cross-origin response. For example, the `nextHopProtocol` property of a Performance API entry shows the network protocol used [11]. In 🟢 a redirect can be detected if it causes the protocol to change (e.g., from `http/1.1` to `http/2`). In other browsers, the property is empty for cross-origin resources.

6.5 Evaluating Inclusion Method Protections


Website owners can deploy security headers to block requests for certain inclusion methods to prevent XS-Leaks:

- Framing Protections, like *X-Frame-Options* and *CSP: frame-ancestors*, should protect against XS-Leaks using framing elements as inclusion method (e.g., `iframe`, `object`, `embed`).
- *CORP* protects resources from inclusion methods that issue `no-cors` requests (e.g., `script`, `img`, `link`, `video`, `audio`, `fetch`)
- *Cross-Origin Opener Policy (COOP)* protects resources from `window.open`. COOP prevents other websites from gaining arbitrary window references.

To test the effectiveness of these security headers, we created a new set of test cases by combining them with all findings from our previous evaluations. These tests closely resemble real-world targets that often deploy at least some security headers.

Download Detection. As seen in Table 2, downloads triggered by the `Content-Disposition: attachment` header can be detected cross-origin. However, our tests show that even when *X-Frame-Options* or *CSP: frame-ancestors* is set, framing elements in 🟡 and 🟢 default to `about:blank`, when resource triggered a download. Also, this behavior is true for `window.open` when the resource is COOP protected; the empty origin is still accessible in all browsers. This is rather problematic as there is currently no way to protect against this leak solely with browser features.



Insufficient Patch. Previously, a bug in the Performance API was reported to Chromium developers. It allowed an attacker to differentiate between response status codes with an object tag [41]. However, the tests with security headers show that if the page uses framing protections, this patch is insufficient. The resource-timing entry is not created when the status code is 200 and the page is blocked by, for instance, XFO.



Inconsistent Frame Blocking. While testing `X-Frame-Options: deny` and `Content-Security-Policy: frame-ancestors 'none'`; we found an interesting *negative* side effect. The two headers can be distinguished in an `<iframe>` in . If the `Content-Security-Policy` blocks cross-site framing, the browser throws a load event on the `<iframe>` element. If the blocking is caused by `X-Frame-Options: deny`, no error is thrown. These experiments demonstrate the strength of AUTOLEAK: the tests can easily be adapted to new scenarios and extended with new headers.

6.6 Discussing Further Mitigations

Cookies are probably the most frequently used technology to authenticate the state-dependent URLs. Recently, they have gained significant security improvements that also affect XS-Leaks.

SameSite Cookies. The `SameSite` cookie attribute restricts sending cookies to the server in cross-site requests. It defines three different values: `None`, `Lax`, and `Strict`, which Khodayari and Pellegriano [23] analyzed recently. Cookies with `SameSite=None` will be attached to every cross-site request, regardless of the inclusion method. `Lax` cookies are only sent during top-level navigation that do not use `POST` (e.g., `window.open`). In the `Strict` case, the cookies is never attached to cross-site HTTP requests. Thus, their responses are state-less and prevent XS-Leaks.

Partitioned Cookies. Partitioned cookies are another XS-Leak mitigation for cookie-based state-dependent resources. It automatically binds all third-party cookies to the top-level site. If an attacker's site requests a state-dependent resource cross-site, it will only be sent with the third-party cookies that are bound to the attacker's top-site cookie jar. Usually, partitioned cookies are an effective countermeasure against many XS-Leaks, since they are not representing a victim's authenticated state on any meaningful website. However, if `window.open` is used as an inclusion method, first-party cookies are still attached to the request. Currently, partitioned cookies are implemented in  as opt-in, called *Cookies Having Independent Partitioned State (CHIPS)* [28], and in  as default, called *Total Cookie Protection* [12].

Blocking All Third-Party Cookies In newer versions of , the Intelligent Tracking Prevention (ITP) blocks all third-party cookies by default [40]. This blocking prevents all XS-Leaks except for those based on pop-ups. Similarly, in  Incognito Mode, all third-party cookies are blocked.

6.7 Responsible Disclosure Process

We identified previously unknown leak techniques and reported those to browser vendors. We took special care when we identified bugs that allowed us to leak differences that were previously unknown, see Table 3. The Chromium team acknowledged `Cache-Control:no-store` (Section 6.3) and `Insufficient Patch` (Section 6.6). We reported the `Inconsistent Frame Blocking` problem

(Section 6.6) to Firefox. Webkit acknowledged and fixed the `Server-Timing` leak (Section 6.3). When we identified bug patterns, such as bugs in the Performance API in Webkit, we reported those all at once. Firefox signaled that they can not fix these problems easily. We consulted browser vendors about addressing minor bugs, quirks, and new leak techniques for XS-Leaks. They suggested extending `Web Platform Tests` [37]. AUTOLEAK provides a complete list of paths that enable XS-Leaks. This is a first step to systematically remove XS-Leaks from the web platform.

7 WEBSITE EXPLOITABILITY

The following section highlights the prevalence of XS-Leaks in web applications. AUTOLEAK can also be used to detect XS-Leaks in real websites, given a state and URL. First, we scanned the Tranco Top 50 for XS-Leaks that allow login detection. Next, we highlight three more severe leaks in YouTube, Slack, and HotCRP.

7.1 Tranco Top 50 Login Detection

We used AUTOLEAK to find and exploit XS-Leaks that allow login detection in the Tranco Top 50 websites. We manually registered an account on each website that allowed account creation without the need for a foreign phone number or a required payment method. We made exceptions for websites where we had private accounts. On 11 domains, no website was reachable (e.g., `akamaiedge.net`), so we skipped them. In total, we could authenticate to 24 out of the Tranco Top 50 websites. To find a suitable state-dependent resource using this account, we used three different pages per website: (1) the website's startpage, (2) the login page, and (3) the account/profile page. We prepared two Chrome browser profiles for the state-dependent resource in state 0 (logged in) resp. state 1 (logged out). Afterward, we used AUTOLEAK to automatically find all leak techniques for our set of inclusion methods on that state-dependent resource.

Results. We found XS-Leaks on 20 of 24 websites within the Tranco Top 50. Among our three tested state-dependent resource, the login page was the most successful: we identified XS-Leaks in 19 out of 24 login pages. In contrast, the start page had the least issues. Only 11 of them were vulnerable. Websites often redirect logged-in users away from the login page, which leads to more, possibly detectable, differences. Similarly, we observed that in the unauthenticated state, users are often redirected to the login page when trying to access the account page. The most effective inclusion method was `windowOpen`, which worked on 14 websites. Interestingly, websites that were resilient to XS-Leak attacks are often single-page applications. These websites serve a single HTML file and fetch user-specific resources using JavaScript. With this approach, the responses for the two states do not differ. We reported the issues to the website vendors.

7.2 Real-World Case-Studies

We want to exemplify that XS-Leaks are not limited in finding login states. Thereby, we used AUTOLEAK to find XS-Leaks for very different states in three case studies. In Youtube Studio, we can identify a specific user. In Slack, we can detect if a user is a member of a specific workspace. Finally, we found XS-Leaks on HotCRP that

Table 4: Tranco Top 50 (6JL4X, 05 April 2023) evaluation with AUTOLEAK. We could log in on 24 websites, where 20 of them were vulnerable ● to login detection on at least one of the three tested pages. Only 4 websites were not vulnerable ○.

Rank	Website	Vuln?	Number of Leak Techniques per Inclusion Method on the ...		
			...Start Page	...Login Page	...Profile Page
1	google.com	●	○	● iframeCSPHashreload: 2, object: 136, objecthashreload: 136, windowOpen: 27	● object: 11059, objecthashreload: 11186, windowOpen: 60
2	facebook.com	●	○	● windowOpen: 7	○
4	youtube.com	●	○	● iframeCSPHashreload: 2, object: 136, objecthashreload: 136, stylesheet: 180, script: 180, preloadScript: 180, preloadStyle: 180	○
6	microsoft.com	●	○	● windowOpen: 79	● iframe: 127, iframeCSPHashreload: 2, iframeHashreload: 164, object: 10960, objecthashreload: 11087, embed: 127, embedHashreload: 164
7	twitter.com	○	○	○	○
10	instagram.com	●	○	● windowOpen: 7	○
11	apple.com	●	○	● windowOpen: 27	○
13	linkedin.com	●	● iframe: 1, iframeHashreload: 1, object: 1, objecthashreload: 1, embed: 1, embedHashreload: 1	● iframe: 1, iframeHashreload: 1, object: 1, objecthashreload: 1, embed: 1, embedHashreload: 1	● object: 4, objecthashreload: 4, embed: 223, stylesheet: 180, script: 180, windowOpen: 27, preloadScript: 180, preloadStyle: 180
15	wikipedia.org	○	○	○	○
16	live.com	●	● iframe: 27, iframeHashreload: 27, object: 11086, objecthashreload: 11213, embed: 250, embedHashreload: 335, windowOpen: 34	● iframe: 127, iframeCSPHashreload: 2, iframeHashreload: 206, object: 10960, objecthashreload: 10964, embed: 127, embedHashreload: 206	● iframeCSPHashreload: 2, iframeHashreload: 164, object: 10960, objecthashreload: 11087, embed: 127, embedHashreload: 164
18	amazon.com	●	● windowOpen: 54	○	● windowOpen: 53
24	yahoo.com	●	● iframe: 1, iframeHashreload: 2, object: 1, objecthashreload: 1, embed: 1, embedHashreload: 2, windowOpen: 237	● windowOpen: 209	○
26	googletagmanager.com	●	● stylesheet: 180, script: 180, preloadScript: 180, preloadStyle: 180	● stylesheet: 180, script: 180, preloadScript: 180, preloadStyle: 180	● stylesheet: 180, script: 180, preloadScript: 180, preloadStyle: 180
27	bing.com	●	● iframe: 27, iframeHashreload: 27, object: 27, objecthashreload: 27, embed: 27, embedHashreload: 27, windowOpen: 27	● iframe: 127, iframeCSPHashreload: 2, iframeHashreload: 206, object: 10960, objecthashreload: 10964, embed: 127, embedHashreload: 206	● iframe: 27, iframeHashreload: 27, object: 27, objecthashreload: 27, embed: 27, embedHashreload: 27, windowOpen: 27
32	office.com	●	● windowOpen: 79	● windowOpen: 79	● windowOpen: 79
34	github.com	○	○	○	○
35	reddit.com	●	● windowOpen: 80	● windowOpen: 105	● windowOpen: 27
36	pinterest.com	●	● iframe: 1, iframeHashreload: 5, object: 1, objecthashreload: 1, embed: 1, embedHashreload: 5, windowOpen: 53	● iframe: 4, iframeHashreload: 8, object: 1, objecthashreload: 1, embed: 4, embedHashreload: 8, audio: 1, video: 1, windowOpen: 107	● iframe: 1, iframeHashreload: 2, object: 1, objecthashreload: 1, embed: 1, embedHashreload: 2, windowOpen: 107
37	wordpress.org	●	○	● windowOpen: 79	○
38	whatsapp.com	○	○	○	○
42	fastly.net	●	● windowOpen: 28	● windowOpen: 258	● windowOpen: 258
44	zoom.us	●	○	● windowOpen: 105	● windowOpen: 105
47	adobe.com	●	○	● windowOpen: 105	● windowOpen: 105
50	vimeo.com	●	● windowOpen: 232	● windowOpen: 232	● windowOpen: 205
Σ	24	20	11	19	14

Legend: ● Vulnerable ○ Not Vulnerable

allows us to de-anonymize reviewers or authors. For each website, we identified the state-dependent resource manually.

YouTube Studio. YouTube is a feature-rich web application open to all Internet users. It is owned by Google, one of the leading companies in web technology and, subsequently, in web security. Therefore, we assumed YouTube is a high-value target, employing all countermeasures to prevent XS-Leaks. We identified an XS-Leak on YouTube Studio that leaks if a victim visiting the attacker’s website is the owner of a specific YouTube channel. The YouTube channel page may reveal personal information, such as contact email or other social media accounts. The XS-Leak exploits the state-dependent URL `studio.youtube.com/channel/<id>`. This website returns status code 200 if the visitor is the channel’s owner corresponding to the specified channel id. Otherwise, the status code is 403. Attackers can differentiate between these by including the state-dependent URL in an object tag and counting the created resource-timing entries. Note that YouTube returns `X-Frame-Options: deny` in both cases to forbid other pages from embedding this page. Surprisingly, this does not protect against this XS-Leak, as shown in Section 6.6.

Slack. We selected Slack as an example of a web application to handle closed user groups. Members of an organization using Slack should only be visible to other members of the same organization. On Slack, we found an XS-Leak that leaks whether a registered user can access a specific workspace. Attackers could use this XS-Leak to identify individuals of a specific organization. AUTOLEAK identified the issue on the state-dependent URL `<workspace>.slack.com/terms-of-service`. Only Slack users who belong to that specific `<workspace>` can access this link. The redirect can not be detected in an easy way, however, AUTOLEAK reported a difference in the `window.length` property after the redirect.

HotCRP. HotCRP is a web application specifically designed to provide double anonymity for reviewing scientific papers: reviewers should not be able to detect the identity of the authors of a paper, and authors should not be able to determine who reviewed their paper. AUTOLEAK found a XS-Leak on HotCRP, which allows an attacker to learn if a victim has access to a specific submission id. Only authors and reviewers can download the submission PDF. The state-dependent URL for this XS-Leak is `<conf>.hotcrp.com/ search?p=<id>&[...]`. HotCRP provides a detectable difference with

Content-Disposition: attachment; header. It is present if the user has access to the submission. Otherwise, it is absent. AUTOLEAK identified the following leak technique: either `window[0].location == "about:blank"` or the attempted access to this property raises an error.

8 RELATED WORK

Table 1 shows a comparison of AUTOLEAK to prior work. In this section, we summarize additional related work for DOM fingerprinting and graph-based approaches.

Starov and Nikiforakis [34] applied XHOUND on the most popular Google Chrome extensions and found out that most of them are fingerprintable due to changes within the DOM. Karami et al. [22] implemented a *parallel DOM* which separates the DOM modification made by browser extensions from the code attempting to exploit said modification for fingerprinting purposes.

Barth et al. [5] analyzed cross-origin JavaScript capability leaks. They modified source code of the Safari browser to annotate JavaScript objects with its origin to detect leaks using singular heap graphs. In contrast, we built a graph extraction browser-agnostic in pure JavaScript, and rendering the need for memory addresses obsolete by using graph comparison instead. By using Chromium's code base, Li et al. [27] constructed JSgraph as a forensic engine for a post-mortem reconstruction of web attacks. A graph-based machine learning approach to ad and tracker blocking was presented by Iqbal et al. [19]. They used ADGRAPH as a tool to construct a graph-based representation by instrumenting Chromium's rendering engine. Bau et al. [6] proposed an approach for ad and tracker blocking by using machine learning, including a graph-based approach to show how scripts are loaded into the DOM. Khodayari and Pellegrino [24] constructed DOM graphs (Hybrid Property Graphs) by systematically extending known DOM clobbering exploits.

9 CONCLUSION

We present AUTOLEAK, a novel methodology to identify XS-Leaks from the DOM systematically. We used a graph-based approach to automatically find *all* leak techniques for a given state-dependent resource and set of inclusion methods. AUTOLEAK automatically executed 151 776 test cases with Chrome, Firefox, and Safari. It identified 16 028 test cases with XS-Leaks.

Lessons Learned. Previous research on XS-Leaks systematically analyzed HTTP/HTML configurations and produced many test cases. However, a single proof-of-concept was described for each XS-Leak. It was left to browser vendors to mitigate similar vulnerabilities. This paper describes a methodology for finding *all* leak techniques for each test instance. This methodology can help browser vendors learn the problem's exact scope. Using a graph algorithm, we could detect all state-dependent differences in the DOM of target applications. Thus we could identify *all* possible leak techniques which could be exploited in an XS-Leak attack.

Future Research Directions. Besides the DOM, other components in the browser exist where including a state-dependent resource may lead to detectable difference [36]. Differences may occur in the HTTP cache, socket pool, OS layer, or any other component involved in processing or fetching the cross-origin resource. An attacker may detect these differences. Our graph-based approach

shows that a systematic analysis can be done for the DOM component. Investigating other components would be a good step towards an XS-Leak-free web, but may come with its own challenges. For a complete analysis, more complex test cases are needed. However, it is unclear if generating a finite list of useful test cases efficiently is possible since some XS-Leaks may occur only in niche scenarios

ACKNOWLEDGMENT

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. Lukas Knittel was supported by the research project "North-Rhine Westphalian Experts in Research on Digitalization (NERD II)", sponsored by the state of North Rhine-Westfalia – NERD II 005-2201-0014. Dominik Noß was supported by the German Federal Ministry of Economics and Technology (BMWi) project "Industrie 4.0 Recht-Testbed" (13I40V002C) and by the research project "MITSicherheit.NRW" funded by the European Regional Development Fund North Rhine-Westphalia (EFRE.NRW).

REFERENCES

- [1] 2019. Attempt to plug an information leak represented by http status. <https://github.com/kohler/hotcrp/commit/406a966aad00a762460fbc62cfb04a7532fc9fbd>
- [2] 2022. Fetch Standard, CORS protocol and credentials. <https://fetch.spec.whatwg.org/#cors-protocol-and-credentials>
- [3] 2022. The HTTP archive. <https://httparchive.org/>
- [4] David Auber, Daniel Archambault, Romain Bourqui, Maylis Delest, Jonathan Dubois, Antoine Lambert, Patrick Mary, Morgan Mathiaut, Guy Melançon, Bruno Pinaud, Benjamin Renoust, and Jason Vallet. 2017. TULIP 5. In *Encyclopedia of Social Network Analysis and Mining*, Reda Alhajj and Jon Rokne (Eds.). Springer, 1–28. https://doi.org/10.1007/978-1-4614-7163-9_315-1
- [5] Adam Barth, Joel Weinberger, and Dawn Song. 2009. Cross-Origin Javascript Capability Leaks: Detection, Exploitation, and Defense. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Montreal, Canada) (SSYM'09). USENIX Association, USA, 187–198.
- [6] Jason Bau, Jonathan Mayer, Hristo Paskov, and John C Mitchell. 2013. A promising direction for web tracking countermeasures. *Proceedings of W2SP* (2013).
- [7] Celery. 2023. Celery: Distributed task queue. <https://github.com/celery/celery>
- [8] Mongo DB. 2023. Mongo DB Website. <https://www.mongodb.com/>
- [9] MDN Web Docs. 2022. HTTP Headers: Cache-Control. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control>.
- [10] MDN Web Docs. 2022. MDN Web Docs. <https://developer.mozilla.org/>.
- [11] MDN Web Docs. 2022. PerformanceResourceTiming nextHopProtocol. <https://developer.mozilla.org/en-US/docs/Web/API/PerformanceResourceTiming/nextHopProtocol>.
- [12] MDN Web Docs. 2023. State Partitioning. https://developer.mozilla.org/en-US/docs/Web/Privacy/State_Partitioning.
- [13] Edward W. Felten and Michael A. Schneider. 2000. Timing attacks on Web privacy. In *Conference on Computer and Communications Security*.
- [14] Ilya Grigorik and Charles Vazac. 2022. *Server Timing*. W3C Working Draft. W3C. <https://www.w3.org/TR/server-timing/#privacy-and-security>.
- [15] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11 – 15.
- [16] Mario Heiderich. 2020. HTTPLeaks. <https://github.com/cure53/HTTPLeaks>.
- [17] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. 2012. Scriptless Attacks: Stealing the Pie without Touching the Sill. In *ACM SIGSAC Conference on Computer and Communications Security* (2012). ACM, ACM Press, 760–771. <https://doi.org/10.1145/2382196.2382276>
- [18] Luan Herrera. 2021. Guessing the URL a cross-origin iframe was redirected to by listening to the load event. <https://crbug.com/1248444>.
- [19] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. 2018. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. <https://doi.org/10.48550/ARXIV.1805.09155>
- [20] Travi J. 2012. What does it mean global namespace would be polluted? <https://stackoverflow.com/questions/8862665/what-does-it-mean-global-namespace-would-be-polluted/13352212>.
- [21] Artur Janc and Mike West. 2020. Oh, the Places You'll Go! Finding Our Way Back from the Web Platform's Ill-conceived Jaunts. In *2020 IEEE European Symposium*

- on *Security and Privacy Workshops (EuroS&PW)* (Genoa, Italy). IEEE, IEEE, 673–680. <https://doi.org/10.1109/eurospw51379.2020.00096>
- [22] Soroush Karami, Faezeh Kalantari, Mehrmoosh Zaeifi, Xavier J. Maso, Erik Tricket, Panagiotis Ilia, Yan Shoshitaishvili, Adam Doupé, and Jason Polakis. 2022. Unleash the Simulacrum: Shifting Browser Realities for Robust Extension-Fingerprinting Prevention. In *USENIX Security Symposium*. USENIX Association.
- [23] Soheil Khodayari and Giancarlo Pellegrino. 2022. The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society. <https://publications.cispa.saarland/3504/>
- [24] Soheil Khodayari and Giancarlo Pellegrino. 2023. It's (DOM) Clobbering Time: Attack Techniques, Prevalence, and Defenses. *IEEE Symposium on Security and Privacy (S&P) (2023)*.
- [25] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. 2021. XSinator. com: From a Formal Model to the Automatic Evaluation of Cross-Site Leaks in Web Browsers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1771–1788.
- [26] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. 2015. The Unexpected Dangers of Dynamic JavaScript. In *USENIX Security Symposium (2015)*. USENIX Association, 723. <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies>
- [27] Bo Li, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. 2018. JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_07B-4_Li_paper.pdf
- [28] Milica Mihajlija. 2022. Cookies having independent partitioned state (CHIPS). <https://developer.chrome.com/docs/privacy-sandbox/chips/>
- [29] Mozilla. 2022. Firefox Source Code. <https://hg.mozilla.org/>
- [30] Jannis Rautenstrauch, Giancarlo Pellegrino, and Ben Stock. 2023. The Leaky Web: Automated Discovery of Cross-Site Information Leaks in Browsers and the Web. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society.
- [31] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. 2017. Same-Origin Policy: Evaluation in Modern Browsers. In *USENIX Security Symposium*. USENIX Association, 713–727. <https://doi.org/10.5555/3241189.3241245>
- [32] Peter Snyder, Soroush Karami, Benjamin Livshits, and Hamed Haddadi. 2023. Pool-Party: Exploiting Browser Resource Pools as Side-Channels for Web Tracking. In *32th USENIX Security Symposium (USENIX Security 23)*.
- [33] Cristian-Alexandru Staicu and Michael Pradel. 2019. Leaky Images: Targeted Privacy Attacks in the Web. In *USENIX Security Symposium*. USENIX Association, 923–939.
- [34] Aleksii Starov and Nick Nikiforakis. 2017. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *2017 IEEE Symposium on Security and Privacy (SP)*. 941–956. <https://doi.org/10.1109/SP.2017.18>
- [35] Avinash Sudhodanan, Soheil Khodayari, and Juan Caballero. 2020. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In *Network and Distributed System Security Symposium (San Diego, CA)*. Internet Society. <https://doi.org/10.14722/ndss.2020.24278>
- [36] Tom Van Goethem, Gertjan Franken, Iskander Sanchez-Rola, David Dworken, and Wouter Joosen. 2022. SoK: Exploring Current and Future Research Directions on XS-Leaks through an Extended Formal Model. In *ACM Asia Conference on Computer and Communications Security (ASIACCS) (New York, NY, USA, 2022-05-30) (ASIA CCS '22)*. ACM Press, 784–798. <https://doi.org/10.1145/3488932.3517416>
- [37] W3C. 2022. The web-platform-tests Project. <https://wpt.fyi/>
- [38] Yoav Weiss and Noam Rosenthal. 2022. *Resource Timing Level 2*. W3C Working Draft. W3C. <https://www.w3.org/TR/2022/WD-resource-timing-2-20220706/>
- [39] Mike West. 2021. *Content Security Policy: Embedded Enforcement*. W3C Editor's Draft. W3C. <https://w3c.github.io/webappsec-cspeel/>
- [40] John Wilander. 2019. Preventing Tracking Prevention Tracking. <https://webkit.org/blog/9661/preventing-tracking-prevention-tracking/>
- [41] Takashi Yoneuchi. 2019. Issue 1038036: Security: Cross-Origin (Partial) Status Code Leakage. <https://crbug.com/1038036>
- [42] Takashi Yoneuchi. 2019. XS-Leak with Resource Timing API and CSP Embedded Enforcement. <https://crbug.com/1105875>
- [43] Mojtaba Zaheri and Reza Curtmola. 2021. Leakuidator: Leaky Resource Attacks and Countermeasures. In *Security and Privacy in Communication Networks - 17th EAI International Conference, SecureComm 2021, Proceedings (2021)*. Springer Science and Business Media Deutschland GmbH, 143–163. https://doi.org/10.1007/978-3-030-90022-9_8