What Storage? An Empirical Analysis of Web Storage in the Wild

Zubair Ahmad
Univ. Ca' Foscari Venezia
zubair.ahmad@unive.it

Samuele Casarin
Univ. Ca' Foscari Venezia
862789@stud.unive.it

Stefano Calzavara
Univ. Ca' Foscari Venezia & OWASP
stefano.calzavara@unive.it

***Abstract*—In this paper we perform the first empirical analysis of the use of web storage in the wild. By using dynamic taint tracking at the level of JavaScript and by performing an automated classification of the detected information flows, we shed light on the key characteristics of web storage uses in the Tranco Top 5k. Our analysis shows that web storage is routinely accessed by third parties, including known web trackers, who are particularly eager to have both read and write access to persistent web storage information. This motivates the need for further research on the security and privacy implications of web storage content.**

I. Introduction

Modern web applications increasingly make use of JavaScript to provide an improved user experience, similar to traditional desktop applications. The more the web application logic is pushed from the server to the client, however, the more sensitive data have to be handled at the client side rather than at the server side. Unfortunately, the traditional approach to handle client-side storage on the Web, i.e., HTTP cookies [3], suffers from significant shortcomings: cookies are limited in size, have an unconventional semantics and are inconvenient to access programmatically. The HTML5 standard thus introduced the Web Storage API [1], which retains the intuitive flavour of cookies, while addressing their most relevant shortcomings. In particular, the Web Storage API offers a simple key-value view of client-side storage: web applications can store in the browser a value v bound to a key k and later retrieve v again by means of k. This approach offers a convenient programming abstraction with a clean semantics, while offering much enlarged storage capabilities.

Although the Web Storage API has been around for a few years now and is fully supported by all major web browsers, anecdotal evidence based on previous web measurements suggests that web storage is still far from the popularity of cookies. Remarkably, contrary to cookies, web storage also received only limited attention by the security and privacy community so far. This is concerning, because the web storage functionality is reminiscent of traditional first-party cookies, hence it can be employed to implement web authentication [5] or to track users across third parties [6], all uses that deserve careful scrutiny. In the present paper, we take a first step to improve our understanding of the usage of web storage in the wild. In particular, we perform an *empirical analysis* of web storage information set by popular websites based on dynamic taint tracking and an automated classification of the collected information flows. Our analysis uncovers several uses of web storage in the wild, for which we discuss relevant security and privacy implications.

Contributions: To sum up, in the present paper, we make the following contributions:

1) We implement a dynamic taint tracking engine for JavaScript based on Jalangi [16] and we configure it to detect information flows involving the Web Storage API (Section III).
2) We perform an empirical study to collect information flows on live sites and shed light on the key characteristics of the use of web storage in the wild. Our analysis is based on an automated classification of the detected information flows along different axes (Section IV).

Remarkably, our analysis shows that web storage is routinely accessed by third parties, including known web trackers, who are particularly eager to have both read and write access to persistent web storage information. This motivates the need for further research on the security and privacy implications of web storage content by our community.

II. Background

We provide below a brief review of the technical ingredients required to understand the present paper. We

Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2022
28 April 2022, San Diego, CA, USA
ISBN 1-891562-78-9
https://dx.doi.org/10.14722/madweb.2022.23005
www.ndss-symposium.org

assume familiarity with the basic functionality of the web platform, e.g., basics of the HTTP protocol, HTML and JavaScript.

A. Same Origin Policy

The Same Origin Policy (SOP) is the baseline defense mechanism of web browsers, which enforces a strict separation between content served by different *origins*, i.e., combinations of protocol, host and port. For example, scripts running in a page fetched from https://www.foo.com cannot access the DOM of a page fetched from https://www.bar.com. SOP mediates both read and write accesses, thus acting as the security cornerstone to grant confidentiality and integrity on the Web. However, when a page at https://www.foo.com includes a script from a different origin like https://www.bar.com, the script inherits the origin of https://www.foo.com and is executed with the corresponding privileges.

B. Web Storage

The Web Storage API offers two different facilities, called *local storage* and *session storage* respectively. Both types of storage are protected by SOP and have similar access interfaces, with the most notable difference being related to the expiration of the stored content. While content in the local storage persists indefinitely, content in the session storage is purged when the browser (or tab) is closed. We show an example use of session storage below: local storage can be used just by replacing `sessionStorage` with `localStorage`.

```
1 sessionStorage.setItem('name', 'alice');
2 var n = sessionStorage.getItem('name');
3 // next line prints "My name is alice"
4 console.log("My name is " + n);
```

In the following we use the term "web storage" to refer to both local storage and session storage when the distinction is immaterial to the discussion. Similarly, in the textual discussion, we just write `setItem` or `getItem` to abstract from the specific web storage object where the method is invoked.

III. Dynamic Taint Tracking

We present the dynamic taint tracking engine that we developed to study the most prominent uses of web storage in the wild. After reviewing the motivations and high-level ideas of the proposed solution, we discuss the key technical details of our implementation.

A. Motivations and Overview

Contrary to cookies, which are normally set via HTTP headers and then automatically attached by the browser to specific network requests, web storage can only be read and written via JavaScript. This means that one cannot monitor the use of web storage just by inspecting network traffic, but has to deal with the complexity of JavaScript to reconstruct valuable information. In particular, we are interested in detecting *information flows* involving the Web Storage API, i.e., data flows that (i) start by reading from the web storage, or (ii) end by writing into the web storage. Flows of the former type may breach the confidentiality of web storage content, while flows of the latter type may compromise its integrity. We thus leverage *dynamic taint tracking* as a standard approach for our principled investigation.

Taint tracking captures *explicit flows* of information, i.e., data dependencies rather than control flow dependencies, which often leads to increased practicality [15], [18]. Taint tracking operates by introducing a taint when reading from a sensitive *source* of information and propagating it across different operations, until it reaches a *sink*, where a potential security issue is detected. For example, consider the following code snippet:

```
1 var n = sessionStorage.getItem('ccn');
2 var s = "Credit card number is: " + n;
3 var xhr = new XMLHttpRequest();
4 xhr.open('GET', '//foo.com/leak.php');
5 xhr.send(s);
```

Here, the variable `n` is tainted at line 1 after reading from the session storage. The variable `s` is then tainted at line 2, because it is computed by concatenating a tainted value to an untainted string, hence the value of `s` depends from tainted data. Finally, at line 5 we detect that a tainted value is written into a sink, thus detecting a potential confidentiality violation.

Note that detection is performed at runtime, because JavaScript is a challenging language for static analysis. In particular, we target an automated measurement on live sites in this paper, hence we prefer a dynamic analysis which is naturally resilient to obfuscated/minified code that may occur in the wild.

B. Technical Details

Our dynamic taint tracking engine is a complex yet relatively standard solution based on existing technologies and the extensive research line on information flow control [14]. In particular, we leverage the Jalangi framework for JavaScript instrumentation [16]. Jalangi operates via a source-to-source transformation, aware of all the dynamic features of JavaScript, which inserts callbacks for all the main operations performed by the JavaScript interpreter. Analysis developers can thus customize the callbacks to track different information at runtime and use it appropriately.

The implementation of our dynamic taint tracking engine follows the approach proposed in Ichnaea [12].

```
1   // Line 1
2   readvar('sessionStorage');          // push taint (false) for variable 'sessionStorage'
3   readproperty('obj2', 'getItem');    // push taint (false) for property 'getItem' of 'obj2'
4                                       // note: 'obj2' is the object ID of 'sessionStorage'
5   push(false);                        // push taint (false) for literal 'ccn'
6   push(false);                        // push taint (false) for receiver object 'obj2'
7   // native function call
8   pop();                              // pop taint (false) for receiver object obj2'
9   pop();                              // pop taint (false) for first argument (string 'ccn')
10  pop();                              // pop taint (false) for the called function
11  // since getItem is a source, taint the top of the stack and propagate it to variable 'n'
12  readret();                          // push taint (true) for the return value
13  writevar('n');                      // store taint (true) for variable 'n' (without pop)
14  pop();                              // pop taint (true) at the end of expression
15
16  // Line 2
17  push(false);                        // push taint (false) for string 'Credit card number is: '
18  readvar('n');                       // push taint (true) for variable 'n'
19  // the taint is propagated to the top of the stack (result of '+') and into variable 's'
20  binaryop('+');                      // apply binary '+' operator and propagate taint (true)
21  writevar('s');                      // store taint (true) for variable 's' (without pop)
22  pop();                              // pop taint (true) at the end of expression
23
24  // Line 5
25  readvar('xhr');                     // push taint (false) for variable 'xhr'
26  readproperty('obj19', 'send');      // push taint (false) for property 'send' of 'obj19'
27                                      // note: 'obj19' is the object ID of 'XMLHttpRequest'
28  readvar('s');                       // push taint (true) for variable 's'
29  push(false);                        // push taint (false) for receiver object 'obj19'
30  // native function call
31  pop();                              // pop taint (false) for receiver object 'obj19'
32  // the taint reaches a network sink, hence the tool logs the information flow
33  pop();                              // pop taint (true) for first argument (variable 's')
34  pop();                              // pop taint (false) for the called function
35  readret();                          // push taint (false) for the return value
36  pop();                              // pop taint (false) at the end of expression
```

Fig. 1. Example of abstract machine code for taint tracking

In particular, the instrumented JavaScript preserves the semantics of the original JavaScript, while running an *abstract machine* to track information flows in parallel. The abstract machine manipulates a stack of *abstract values* that reflect the taints of values on the runtime stack of the original JavaScript program, while also maintaining maps that associate abstract values with local variables and object properties. We do not present a full formal description of the abstract machine for space reasons (and also because it largely follows Ichnaea [12]), but we provide an example to clarify how it operates. Figure 1 shows the abstract machine code generated for the previous example; we only show the instructions generated for lines 1, 2, 5 of the original code, because lines 3 and 4 are uninteresting for taint tracking purposes.

Some of the fundamental instructions of the abstract machine are employed here: `push` and `pop` to access the stack, `binaryop` to propagate taints in binary operations, `readvar` and `writevar` to load and store the taintedness of variables, `readproperty` and `writeproperty` to load and store the taintedness of object properties, and `readret` to load the taintedness of the return value of a function call. At line 1 of the original code, after calling the function `getItem`, which is a source, the instruction `readret` pushes `true` onto the stack to track that the return value is tainted. At line 2, the '+' binary operator combines a literal value, that is always untainted, with the tainted value generated at the previous line; in this case, the taints of the operands, which are the two topmost elements of the stack, are joined and propagated to the result of the expression, hence `true` is pushed on top of the stack to track that such value is tainted. Finally, at line 5, the tainted value computed at line 2 is passed as argument when invoking the function `send`, which has been defined as a sink, therefore the tool reports the information flow from the function call to `getItem` at line 1 to this network sink.

IV. Web Measurement

We now explain how we performed our empirical measurement in the wild and we report on the most relevant findings of our study.

TABLE I. LIST OF SOURCES AND SINKS USED FOR TAINT TRACKING

Class		Details
Sources	Cookies	`document.cookie`
	Current URL	`document.URL`, `location`, `window.location`, `document.location`
	Navigator	`navigator.geolocation`, `navigator.language`, `navigator.platform`, `navigator.userAgent`
	Network	`XMLHttpRequest` (input)
	Web storage	`localStorage.getItem`, `sessionStorage.getItem`
Sinks	Cookies	`document.cookie`
	Network	`XMLHttpRequest` (output), `navigator.sendBeacon`, `src` attribute of HTML element
	Web storage	`localStorage.setItem`, `sessionStorage.setItem`

A. Methodology

We use the developed dynamic taint tracking engine to automatically identify information flows involving the Web Storage API in the top 5k domains of the Tranco list [13] generated on December 14th, 2021.[1] More formally, an information flow involves the Web Storage API if and only if: (i) it starts from a call to the `getItem` method and ends into a sink, or (ii) it starts from a source and ends into a call to the `setItem` method. We refer to the former as *confidentiality flows* and to the latter as *integrity flows*. Table I reports the different sources and sinks considered in our analysis, largely inspired by previous web measurements based on information flow control [7], [17]. Note that the web storage was largely ignored as source or sink in previous work, to the best of our knowledge.

We use the Puppeteer library[2] to drive our instrumented browser to each domain in the Tranco list, leaving 60 seconds to render the HTML content after connecting. For each domain, we leverage taint tracking to collect all the information flows involving the Web Storage API. To better understand the use of web storage, we then perform an automated classification of the collected information flows. This is a non-trivial task, that we dealt with after a preliminary manual investigation to understand the nature of the collected data. In particular, we categorize the flows along different axes, all fully amenable to automation, as described in the following.

a) Confinement: A first relevant aspect we investigate is related to the origins involved in the flows. We say that a flow is *internal* if and only if it is confined within a single origin. In other words, these flows do not include network sources or sinks (cf. Table I), unless network communication only involves the same origin where the flow is detected. The other flows, which we call *external*, are more interesting from a security and privacy perspective, because they involve third parties. For example, a page at https://www.foo.com may include a script from https://www.bar.com, which reads the content of the local storage and sends it to https://www.bar.com, thus potentially leaking sensitive information from https://www.foo.com. The reason why we define confinement at the origin level, rather than at the site[3] level, is that web storage content is origin-scoped and thus subject to SOP.

b) Tracking: Tracking is one of the driving forces of the web ecosystem and it is extremely common in the wild. We call a *tracking flow* any information flow that starts from a source, or ends into a sink, located in a script served by a known web tracker. To reconstruct this information, we leverage the fact that the instrumentation performed by Jalangi keeps track of the URL from which each script was downloaded. By matching this script URL against popular filter lists like EasyList and EasyPrivacy [9], we can detect the involvement of known web trackers in the identified web storage accesses.

c) Persistence: A last relevant aspect is the *persistence* of the information involved in the flow. Though both local storage and session storage can store arbitrary information, the content of local storage may persist indefinitely. Persistence may have important implications on both security and privacy. For example, the local storage may become a source of persistent XSS [19] and may potentially enable perpetual tracking of web users. For each flow, we thus track the type of the involved web storage. Notice that the same flow may involve both the local storage and the session storage, e.g., because local storage and session storage information is combined together before network communication.

B. Measurement Results

Overall, our crawler successfully accessed and instrumented JavaScript code on 3,324 domains, detecting 5,187 information flows involving the Web Storage API on 837 domains (25%). These include a significant number of flows where the web storage acts as both source and sink, which we filter out because they are confined to the Web Storage API and thus have limited security and privacy implications. After filtering, we are left with 3,207 flows on 651 domains, including 531 confidentiality flows and 2,676 integrity flows.

[1] https://tranco-list.eu/list/NXVW

[2] https://pptr.dev/

[3] A site is defined as an effective top-level domain (eTLD) + 1. For example, foo.example.com and baz.example.com belong to the same site example.com.

TABLE II. SOURCES AND SINKS INVOLVED IN CONFIDENTIALITY AND INTEGRITY FLOWS

	Class	#flows	#domains
Confid.	Cookies	202	66
	Network	329	139
Integrity	Cookies	410	72
	Current URL	1,582	353
	Navigator	979	204
	Network	913	238

Table II reports a first breakdown of the detected flows in terms of the involved sources and sinks.[4] As we can see, a significant number of flows involves network sources or sinks: this happened for 329 confidentiality flows (62%) and 913 integrity flows (34%).

We now focus on a more fine-grained classification of the detected flows, as we described in the previous section. Overall, we found that 1,053 flows (33%) are *external*, i.e., a significant amount of the flows related to the Web Storage API also involve an origin different from the origin of the page where the flow was detected. Moreover, 2,276 flows (71%) are related to *tracking*, i.e., the majority of the detected flows can be attributed to known trackers included in popular filter lists. Finally, 2,487 flows (78%) only make use of local storage, 708 flows (22%) only make use of session storage and 12 flows make use of both. All this information suggests that a common use case of web storage is *persistent web tracking, possibly performed by third parties*.

To provide further insights on the use of web storage in the wild, we also investigate potential correlations between the different axes considered in our classification. The results of our analysis are shown in Table III. The table supports the following selected observations:

- Confidentiality flows are roughly equally split between internal and external flows, while integrity flows are mostly internal (70%). This shows that it is much more common to send web storage information to third parties, rather than having third parties write information in the web storage.
- The majority of the confidentiality flows can be attributed to trackers (65%). Remarkably, however, even a higher percentage of integrity flows can be attributed to trackers (72%). Indeed, the table also shows that the majority of tracking flows are integrity flows (85%). This suggests that trackers routinely both read and write web storage information in the wild.
- External flows are more likely to be confidentiality flows than internal flows (25% vs. 12%) and the very large majority of the external flows can be attributed to trackers (78%). Moreover, tracking flows are more likely to be external than non-tracking flows (36% vs. 25%). This suggests that trackers normally send web storage information to third parties.
- Tracking flows operate on local storage more frequently than non-tracking flows (81% vs. 69%). Moreover, flows involving the session storage are more likely to be internal than flows involving the local storage (83% vs. 63%). This suggests that local storage is the prime target of trackers and session storage is largely dedicated to internal use within a single origin.
- Finally, we observe that confidentiality flows are more likely to operate on local storage than integrity flows (88% vs. 76%), just like external flows involve local storage more frequently than internal flows (88% vs. 73%). This shows that the persistent information saved in the local storage is often the target of information leaks, likely towards third parties.

To further shed light on the security and privacy implications of web storage in the wild, we also perform an additional classification of the detected *external* information flows, i.e., information flows involving two different origins. In particular, we analyze how many such flows are still within the same site and how many are cross site. This is an interesting information, because different domains under the same site normally belong to the same owner, i.e., the entity who performed the domain registration, hence same-site external flows are likely less significant from a security and privacy perspective. The results of our analysis are shown in Table IV. They highlight that the very large majority of the external flows are cross-site and this observation is uniform across all classes of external flows. This further confirms the relevance of our findings.

The last analysis we carry out estimates how many information flows are introduced by *libraries*. These flows are particularly interesting, because libraries are normally used by multiple pages, hence the analysis of a single library may shed light on the behavior of multiple pages. To identify libraries, we look for duplicate flows within different domains and we aggregate them based on the script URL information provided by Jalangi. Specifically, we use the script URL of the source for the integrity flows and the script URL of the sink for the confidentiality flows. Table V reports information on the top 10 most popular libraries, based on the number of domains where an information flow was detected. As we can see, the large majority of these libraries (8 out of 10) is related to web tracking and the most popular library is used for tracking on 66 domains, i.e., roughly

[4]The sum of the integrity flows exceeds 2,676, because a flow may involve multiple sources. In this case, the same flow is counted on two different rows of the table, e.g., Cookies and Network.

TABLE III. CLASSIFICATION OF THE DETECTED INFORMATION FLOWS

	Confid.	Integrity	Internal	External	Tracking	Non-Tracking	Local	Session	Both
Confid.	-	-	268 (50%)	263 (50%)	343 (65%)	188 (35%)	464 (88%)	55 (10%)	12 (2%)
Integrity	-	-	1,886 (70%)	790 (30%)	1,933 (72%)	743 (28%)	2,203 (76%)	653 (24%)	0 (0%)
Internal	268 (12%)	1,886 (88%)	-	-	1,452 (67%)	702 (33%)	1,564 (73%)	586 (27%)	4 (0%)
External	263 (25%)	790 (75%)	-	-	824 (78%)	229 (22%)	923 (88%)	122 (12%)	8 (0%)
Tracking	343 (15%)	1,933 (85%)	1,452 (64%)	824 (36%)	-	-	1,845 (81%)	422 (19%)	9 (0%)
Non-Tracking	188 (20%)	743 (80%)	702 (75%)	229 (25%)	-	-	642 (69%)	286 (31%)	3 (0%)
Local	464 (19%)	2,023 (81%)	1,564 (63%)	923 (37%)	1,845 (74%)	642 (26%)	-	-	-
Session	55 (8%)	653 (92%)	586 (83%)	122 (17%)	422 (60%)	286 (40%)	-	-	-
Both	12 (100%)	0 (0%)	4 (33%)	8 (67%)	9 (75%)	3 (25%)	-	-	-

TABLE IV. ADDITIONAL BREAKDOWN OF THE EXTERNAL INFORMATION FLOWS

	Same Site	Cross Site
Confid.	22 (8%)	241 (92%)
Integrity	30 (4%)	760 (96%)
Tracking	15 (2%)	809 (98%)
Non-Tracking	37 (16%)	192 (84%)
Local	33 (4%)	890 (96%)
Session	19 (16%)	103 (84%)
Both	0 (0%)	8 (100%)

10% of the domains where we identified an information flow involving the Web Storage API.

V. RELATED WORK

Many papers presented dynamic analysis techniques for JavaScript, we refer to [2] for a recent survey on the topic. Despite the popularity of JavaScript analyses, however, we are not aware of prior empirical studies on the use of web storage in the wild. A notable exception is a study carried out by Belloro and Mylonas in 2018 [4]. In their work, the authors analyzed how less known client-side storage mechanisms like web storage, IndexedDB and Web SQL Database (now deprecated) were used for web tracking on popular sites and questioned the lack of user control over the locally stored data. In our work, instead, we take a holistic view of web storage and we carry out a systematic analysis of its use in the wild, based on an automated categorization of the detected information flows along different axes. Indeed, our study is based on a dynamic information flow analysis, which minimizes false positives and provides meaningful semantics to different usages of the Web Storage API. Their work, in turn, uses a lightweight static analysis which only detects API calls, hence cannot discriminate between actual information leaks and simple "feature detection" libraries like Modernizr.[5]

Chen and Kapravelos presented a taint tracking engine called Mystique and used it to track information leakage from browser extensions [7]. Mystique was applied to a total of 181,683 browser extensions, detecting 3,686 extensions leaking private information. In later work, Mystique was also used to investigate the leakage of first-party cookies to third-party cookies for web tracking [6]. In particular, the authors estimated that around 57% of the sites in the Alexa top 10k include at least one cookie containing a unique user identifier which is exchanged with multiple third parties.

Sjosten et al. proposed EssentialFP, a principled approach to the dynamic detection of browser fingerprinting [17]. EssentialFP is based on dynamic analysis and in particular on an extension of JSFlow [10]. To capture the essence of fingerprinting, EssentialFP relies on an extensive list of browser-specific sources and looks for information flows ending in known network sinks. The efficacy of EssentialFP was illustrated through an empirical study based on two classes of web pages: fingerprinting pages (authentication, bot detection and more) and non-fingerprinting pages (analytics, polyfills, advertisement).

Karim et al. implemented a platform-independent dynamic taint analysis tool for JavaScript, called Ichnaea [12]. They encoded the taint propagation logic as instructions for an abstract machine, so as to leverage an existing JavaScript instrumentation framework called Jalangi [16]. To evaluate Ichnaea, the authors applied it to a Tizen web application to detect privacy leaks and identified flows of tainted input data to sensitive sinks in Node.js modules, thus detecting both known and unknown vulnerabilities. Our implementation follows the approach proposed in Ichnaea with few modifications, yet it is targeted to a different application scenario.

Staicu et al. performed an empirical investigation of information flows in existing JavaScript code [18]. Their study accounted for both explicit and implicit information flows, concluding that explicit flows are by far the most prevalent in the wild and the additional runtime overhead required to track implicit flows may be unjustified. Their analysis is also based on Jalangi [16], which the authors used to implement a dynamic information flow tracker inspired to JSFlow [10].

VI. CONCLUSION

In this paper, we performed a first empirical analysis of the use of web storage in the wild, based on dynamic taint tracking and an automated classification of the detected information flows. Our analysis showed that

[5]https://modernizr.com/

TABLE V. MOST POPULAR LIBRARIES INTRODUCING INFORMATION FLOWS

Library	#flows	#domains	Tracking?
https://static.chartbeat.com/js/chartbeat.js	132	66	✓
https://mc.yandex.ru/metrika/tag.js	228	34	✓
https://fast.wistia.com/assets/external/E-v1.js	106	26	✗
https://pagead2.googlesyndication.com/pagead/managed/js/adsense/m202112060101/show_ads_impl_with_ama.js	124	25	✓
https://quantcast.mgr.consensu.org/tcfv2/cmp2.js	24	24	✗
https://static.chartbeat.com/js/chartbeat_video.js	42	20	✓
https://az416426.vo.msecnd.net/scripts/a/ai.0.js	22	19	✓
https://cdn.izooto.com/scripts/sdk/izooto.js	42	16	✓
https://bat.bing.com/bat.js	34	15	✓
https://cdn.pdst.fm/ping.min.js	17	15	✓

web storage is routinely accessed by third parties, including known web trackers, who are particularly eager to have both read and write access to persistent web storage information. This motivates the need for further research on the security and privacy implications of web storage content, that we plan to pursue as a follow-up work of this preliminary investigation. Constructive solutions designed to prevent web storage abuses would be particularly worth investigating, e.g., the recently proposed "page-length storage" approach designed to mitigate the effects of stateful web tracking [11].

Moreover, we plan to reuse known heuristics from the literature [6], [9] to detect personally identifiable information and better investigate the real-world privacy implications of the detected tracking flows. We also want to take a more in-depth look into the most popular libraries introducing information flows involving the Web Storage API, given the impact that libraries may have. It would be particularly interesting to analyze the documentation of known trackers to collect insights about their use of web storage and investigate their compliance with regulations like GDPR [8]. Finally, we would like to further refine our classification of information flows to account for common use cases that we anticipate, e.g., web authentication and browser fingerprinting. Digging into selected use cases may be helpful to provide additional insights of the uses and abuses of web storage in the wild.

REFERENCES

[1] "Web storage," https://html.spec.whatwg.org/multipage/webstorage.html.

[2] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C. Staicu, "A survey of dynamic analysis and test generation for javascript," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 66:1–66:36, 2017. [Online]. Available: https://doi.org/10.1145/3106739

[3] A. Barth, "HTTP state management mechanism," https://datatracker.ietf.org/doc/html/rfc6265.

[4] S. Belloro and A. Mylonas, "I know what you did last summer: New persistent tracking mechanisms in the wild," *IEEE Access*, vol. 6, pp. 52 779–52 792, 2018. [Online]. Available: https://doi.org/10.1109/ACCESS.2018.2869251

[5] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta, "Surviving the web: A journey into web session security," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 13:1–13:34, 2017. [Online]. Available: https://doi.org/10.1145/3038923

[6] Q. Chen, P. Ilia, M. Polychronakis, and A. Kapravelos, "Cookie swap party: Abusing first-party cookies for web tracking," in *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, J. Leskovec, M. Grobelnik, M. Najork, J. Tang, and L. Zia, Eds. ACM / IW3C2, 2021, pp. 2117–2129. [Online]. Available: https://doi.org/10.1145/3442381.3449837

[7] Q. Chen and A. Kapravelos, "Mystique: Uncovering information leakage from browser extensions," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 1687–1700. [Online]. Available: https://doi.org/10.1145/3243734.3243823

[8] 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 *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/we-value-your-privacy-now-take-some-cookies-measuring-the-gdprs-impact-on-web-privacy/

[9] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 1388–1401. [Online]. Available: https://doi.org/10.1145/2976749.2978313

[10] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "Jsflow: tracking information flow in javascript and its apis," in *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, Y. Cho, S. Y. Shin, S. Kim, C. Hung, and J. Hong, Eds. ACM, 2014, pp. 1663–1671. [Online]. Available: https://doi.org/10.1145/2554850.2554909

[11] J. Jueckstock, P. Snyder, S. Sarker, A. Kapravelos, and B. Livshits, "There's no trick, its just a simple trick: A web-compat and privacy improving approach to third-party web storage," *CoRR*, vol. abs/2011.01267, 2020. [Online]. Available: https://arxiv.org/abs/2011.01267

[12] R. Karim, F. Tip, A. Sochurková, and K. Sen, "Platform-independent dynamic taint analysis for javascript," *IEEE Trans. Software Eng.*, vol. 46, no. 12, pp. 1364–1379, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2018.2878020

[13] V. L. Pochat, T. van Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27,*

2019. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/tranco-a-research-oriented-top-sites-ranking-hardened-against-manipulation/

[14] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, 2003. [Online]. Available: https://doi.org/10.1109/JSAC.2002.806121

[15] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld, "Explicit secrecy: A policy for taint tracking," in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 2016, pp. 15–30. [Online]. Available: https://doi.org/10.1109/EuroSP.2016.14

[16] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs, "Jalangi: a selective record-replay and dynamic analysis framework for javascript," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 488–498. [Online]. Available: https://doi.org/10.1145/2491411.2491447

[17] A. Sjösten, D. Hedin, and A. Sabelfeld, "Essentialfp: Exposing the essence of browser fingerprinting," in *IEEE European Symposium on Security and Privacy Workshops, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 2021, pp. 32–48. [Online]. Available: https://doi.org/10.1109/EuroSPW54576.2021.00011

[18] C. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, "An empirical study of information flows in real-world javascript," *CoRR*, vol. abs/1906.11507, 2019. [Online]. Available: http://arxiv.org/abs/1906.11507

[19] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/dont-trust-the-locals-investigating-the-prevalence-of-persistent-client-side-cross-site-scripting-in-the-wild/