An Empirical Analysis of Web Storage and Its Applications to Web Tracking

ZUBAIR AHMAD, SAMUELE CASARIN, and STEFANO CALZAVARA, Università Ca' Foscari Venezia, Italy

In this article, we present a large-scale 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 10k. 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. We then deep dive in web tracking as a prominent case study: our analysis shows that web storage is not yet as popular as cookies for tracking purposes; however, taint tracking is useful to detect potential new trackers not included in standard filter lists. Moreover, we observe that many websites do not comply with the General Data Protection Regulation directives when it comes to their use of web storage.

CCS Concepts: • **Security and privacy** → *Web application security*; • **Information systems** → *Web applications;*

Additional Key Words and Phrases: JavaScript, taint analysis, web storage, web tracking

ACM Reference format:
Zubair Ahmad, Samuele Casarin, and Stefano Calzavara. 2023. An Empirical Analysis of Web Storage and Its Applications to Web Tracking. *ACM Trans. Web* 18, 1, Article 7 (October 2023), 28 pages.
https://doi.org/10.1145/3623382

1 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 are 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 [4]) suffers from significant shortcomings: cookies are limited in size, have an unconventional semantics, and are inconvenient to access programmatically [8]. The HTML5 standard thus introduced the Web Storage API [2], which retains the intuitive flavor of cookies while addressing their most relevant shortcomings. In particular, the Web Storage API offers a

This article is an extended version of a workshop paper titled "What Storage? An Empirical Analysis of Web Storage in the Wild" by the same authors. The original paper was accepted and presented at MadWeb 2022, co-located with NDSS 2022.
This research was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union–NextGenerationEU.
Authors' address: Z. Ahmad, S. Casarin, and S. Calzavara, Università Ca' Foscari Venezia, Italy; e-mails: zubair.ahmad@unive.it, samuele.casarin@unive.it, stefano.calzavara@unive.it.

1559-1131/2023/10-ART7
https://doi.org/10.1145/3623382

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 granting much enlarged storage capacity with respect to cookies.

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 received just limited attention by the security and privacy community so far. This is concerning because the web storage functionality is reminiscent of traditional cookies, hence it can be employed to implement web authentication [7] or to track users across third parties [9], all uses that deserve careful scrutiny. In the present article, we aim 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 article we make the following contributions:

(1) We implement a dynamic taint tracking engine for JavaScript based on Jalangi [29], and we configure it to detect information flows involving the Web Storage API. The use of taint tracking is necessary because web storage information is accessed and manipulated by JavaScript, hence one has to track how values read from/written to the web storage propagate through the control flow of web applications (Section 3).
(2) We perform an empirical study to collect information flows on 10k 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. Specifically, we classify flows based on relevant security and privacy properties: confidentiality, integrity, confinement, web tracking potential, and persistence (Section 4).
(3) We deep dive in web tracking as a prominent case study. In particular, we first investigate the effectiveness of filter lists at detecting tracking via the web storage, we then analyze the privacy implications of the detected tracking flows, and we finally evaluate the compliance of web storage uses in the wild against the key **General Data Protection Regulation (GDPR)** directives (Section 5).

In the end, 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. Specifically, we measure that 59% of the information flows involving the web storage can be attributed to known trackers detected by popular filter lists. Although filter lists are too coarse grained to reason about tracking at the flow level, we show that 28% of the detected information flows have tracking potential, because they include potential user identifiers according to existing heuristics. Moreover, we show that taint tracking is useful to detect potential new trackers not included in existing filter lists: indeed, our information flow analysis can identify 266 flows with tracking potential which escape filter list detection, leading to the identification of 90 new (potentially) tracking scripts. Our analysis also shows that cross-site tracking can readily happen in 44% of the domains where we identified any use of web storage; however, our data do not show significant abuses of first-party content for cross-site tracking as identified by recent research on cookies [20, 23]. Finally, we observe that many websites do not comply with the GDPR directives when it comes to their use of web storage; in particular, we identify several violations to the consent and transparency dimensions of GDPR. This motivates the need for further research on the security and privacy implications of web storage content.

2 BACKGROUND

In the following, we provide a brief review of the technical ingredients required to understand this article. We assume familiarity with the basic functionality of the web platform, such as basics of the HTTP protocol, HTML, and JavaScript.

2.1 Same Origin Policy and Web Storage

The **Same Origin Policy (SOP)** is the baseline defense mechanism of web browsers, which enforces a strict separation between content served by different *origins*—that is, 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. Note that 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.

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. Whereas 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 next: local storage can be used just by replacing `sessionStorage` with `localStorage` in the method call.

```
1  sessionStorage.setItem('name', 'alice');
2  var n = sessionStorage.getItem('name');
3  // the 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.

2.2 Web Tracking

Web tracking operates by setting unique identifiers into client-side storage, such as cookies and web storage, and using them to correlate navigation behaviors of web clients. It can be broadly categorized in two classes, with different goals and privacy implications [24]:

- In *same-site tracking*, the tracking script is loaded into the web page by means of a `<script>` tag, hence it inherits the origin of the embedding web page. This implies that the script sets tracking information that is under the control of the web page rather than the tracker. If the tracking script is also loaded in a different page, it has no visibility of any identifiers set in the first page, hence the tracker has no immediate way to learn navigation patterns across different websites. This form of tracking is normally used for analytics and statistics collection; however, recent work also investigated possible abuses of first-party identifiers, such as *cookie syncing* [20] and *UID smuggling* [23].
- In *cross-site tracking*, the tracking script is loaded into the web page by means of an `<iframe>` tag, hence it maintains the tracker's origin. Since tracking information set by the script remains under the control of the tracker, the tracker can detect navigation profiles across websites. Indeed, if the tracking script is also loaded in a different page, its requests may attach the tracker-controlled identifier to pinpoint the client. This form of tracking is thus

normally used, such as for advertisement, where the tracker may construct navigation profiles to provide personalized ads with higher revenues.

Although browsers are generally lenient toward same-site tracking, several browsers such as Mozilla Firefox and Brave now implement countermeasures such as *partitioned storage* to prevent cross-site tracking [23]. In this work, we perform our analysis on top of Google Chrome, which does not implement any form of tracking protection by default at the time of writing. The privacy implications drawn from our study thus directly apply to Google Chrome users, who account for more than 60% of the web users as of August 2023.[1]

2.3 General Data Protection Regulation

To bring data protection laws in line across member states, GDPR [1] was presented in January 2012, approved in May 2016 and enforced in May 2018. GDPR has several implications for web services and is therefore supposed to impact the technical design of websites, which data they collect, and how they inform users about their practices. Importantly, Article 3 of GDPR regulates any service that is served in the European Union or monitors the behavior of the users within there, even when there is no legal representation of the service provider therein.

We consider two key requirements of GDPR as relevant to our study: *consent* and *transparency*. In legal literature, the regulation is covered in further detail [25].

2.3.1 Consent. According to Article 6 of GDPR, the processing of personal data is only legitimate if "the data subject has given consent to the processing of his or her personal data for one or more specific purposes." Consent, in turn, is explained in Article 4 as "any freely given, specific, informed and unambiguous indication of the data subject's wishes by which he or she, by a statement or by a clear affirmative action, signifies agreement to the processing of personal data relating to him or her." From a technical perspective, consent on the Web is normally granted by means of *cookie banners*—that is, small banners rendered via JavaScript to provide information about the nature and purposes of the data collected by means of cookies; cookies should only be set in the user's browser after approval is granted by clicking over the banner [6, 11]. Note that some cookies are exempt from this requirement, most notably those cookies which are strictly required for website functionality (e.g., to implement web sessions). Although the popular term *cookie banner* might suggest that consent should only be granted for cookies, this is not the case: GDPR applies to all types of client-side storage, including web storage [21, 25].

2.3.2 Transparency. Article 12 of GDPR states that any service provider who processes/controls personal data should inform the data subject about the fact and present the information in "a concise, transparent, intelligible, and easily accessible form, using clear and plain language." Article 13 more specifically lists what information needs to be provided. This includes contact data, the purposes and legal basis for the processing, and the data subject's rights regarding their personal data, such as the rights to access, rectification, or deletion. From a technical perspective, these requirements make it necessary for every website operating in the European Union to have a *privacy policy* and existing privacy policies must be updated to comply with the new transparency requirements.

3 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. Taint tracking is a standard dynamic approach to information flow control [28], which we use to detect how information read from/written to the web storage

[1] https://gs.statcounter.com/browser-market-share

propagates in live websites. After reviewing the motivations and high-level ideas of the proposed solution, we discuss the key technical details of our implementation.

3.1 Motivation 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—that is, dataflows that either start by reading from the web storage or end by writing into the web storage. Flows of the former type may breach the confidentiality of web storage content, whereas 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 [28, 32]. 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', 'https://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 article, hence we prefer a dynamic analysis that is naturally resilient to obfuscated/minified code that may occur in the wild.

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 [26]. In particular, we leverage the Jalangi framework for JavaScript instrumentation [29]. 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 [19]. Unfortunately, since Ichnaea is not publicly available and is only semi-formally described in the original paper, we had to engineer our own solution, which we detail in this section. The key idea of the proposed analysis approach is that 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. To exemplify, Figure 1 shows the abstract machine code generated for the previous example; we only show the instructions generated for lines 1, 2, and 5 of the original code because lines 3 and 4 are uninteresting for taint tracking purposes.

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

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

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

3.2 Analysis Specification

3.2.1 Core JavaScript. To provide a formal description of the taint tracking engine, we consider a core subset of non-strict ECMAScript 5.1 [12]. Such subset, whose grammar is given in Figure 2, is subject to many simplifying assumptions and excludes many of the language features (e.g., exception handling) and syntactic sugars described in the full ECMAScript specification. Nevertheless, it suffices to provide the key ideas of our taint tracking approach. Additional JavaScript features not included in our core subset but supported by our implementation are briefly discussed later.

The considered subset of JavaScript supports all the literals of primitive values and the `null` keyword. Objects are initialized through object expressions by providing the list of key-value pairs for

Id	::=	`x` | `foo` | ...	(identifier)
Num	::=	`0` | `1` | `-1` | `0.1` | ...	(numeric literal)
Str	::=	`""` | `"hello"` | ...	(string literal)
Bool	::=	`true` | `false`	(boolean literal)
Lit	::=	*Num* | *Str* | *Bool* | `null`	(literal)
Obj	::=	`{` *Id* `:` *Expr* `,` ⋯ `,` *Id* `:` *Expr* `}`	(object)
Fun	::=	`function (` *Id* `,` ⋯ `,` *Id* `) {` *Stmt* `}`	(function)
uop	::=	`!` | `typeof` | ...	(unary operator)
bop	::=	`+` | `*` | ...	(binary operator)
Expr	::=	*Lit*	(literal expression)
	|	*Obj*	(object expression)
	|	*Fun*	(function expression)
	|	*uop Expr*	(unary operation)
	|	*Expr bop Expr*	(binary operation)
	|	*Id*	(variable read)
	|	*Expr* `[` *Expr* `]`	(property read)
	|	*Expr* `(` *Expr* `,` ⋯ `,` *Expr* `)`	(function call)
Stmt	::=	`var` *Id* `=` *Expr*	(variable declaration)
	|	*Id* `=` *Expr*	(variable assignment)
	|	*Expr* `[` *Expr* `]` `=` *Expr*	(property write)
	|	`return` *Expr*	(return statement)
	|	*Stmt* `;` *Stmt*	(sequence)

Fig. 2. Syntax of a core subset of JavaScript.

data properties. Similarly, functions are anonymous and specified through function expressions by supplying the list of formal arguments and the body. Like in traditional core programming languages, we also assume a set of unary and binary operators such that their evaluation does not have side effects and does not involve any implicit type conversion of its operands (e.g., in JavaScript, the evaluation of `"obj:" +` requires the object to be converted into a string). Our core subset of JavaScript supports variable declarations to introduce bindings in the local scope, variable reads to load their value by means of the corresponding identifier, and variable assignments to store a given value. Object properties are accessed for reading and writing as well, but only using the bracket notation *Expr*`[`*Expr*`]`, in contrast to the full language that also supports the dot notation *Expr*`.`*Id*. Furthermore, although variable and property writes are expressions in JavaScript, here they are just statements.

Finally, the full JavaScript language provides three different semantics of invocation: (i) call, (ii) method call, and (iii) constructor invocation. To complicate things, there exist numerous situations in which a function is implicitly executed by the JavaScript engine, including implicit type conversions, accessing accessor properties, asynchronous operations, and so on. For simplicity, here we just consider regular function calls as the simplest form of invocation. Moreover, we assume that functions must be invoked with a number of actual arguments equal to the number of formal arguments and that invocations must terminate with a `return` statement, differently from the real-world language in which functions can be invoked with a number of actual arguments lesser or greater than the number of formal arguments and an invocation may exit without executing a `return` statement.

3.2.2 Abstract Machine. The abstract machine manipulates a stack of abstract values reflecting the taint of values in the stack of the JavaScript program while also maintaining maps that

ℓ	::=	$(Str, Str, Str*)$	(label)
τ	::=	$\{\ell_1, ..., \ell_n\} \mid \emptyset$	(taint, abstract value)
Inst	::=	`push`(τ)	(push constant value onto stack)
	|	`pop`()	(pop value from stack)
	|	`initvar`(*Id*)	(pop value, initialize variable with it)
	|	`readvar`(*Id*)	(push current value of variable)
	|	`writevar`(*Id*)	(write value at top of stack into variable)
	|	`initproperty`(*Id*, *Id*)	(pop value, initialize object property with it)
	|	`readproperty`(*Id*, *Id*)	(push current value of object property)
	|	`writeproperty`(*Id*, *Id*)	(write value at top of stack into object property)
	|	`join`()	(pop two values, join, push result)

Fig. 3. Instruction set of the abstract machine.

associate abstract values to variables and object properties. With the aim of identifying the different types of sources that influence concrete values during the script execution (local storage, network, etc.), we model abstract values with *label sets* τ. Accordingly, the empty label set annotates a non-tainted concrete value, whereas a non-empty label set annotates a concrete value that is tainted by the labels it contains. A label is a triple $\ell = (type, loc, extras)$ that represents the operation that led to its creation, where $type \in Str$ is the type of label, which identifies a specific type of operation, $loc \in Str$ represents the code location where the operation was executed, and $extras \in Str^*$ is a sequence of extra information about that operation. For example, the label (`"localStorage.getItem"`, `"https://foo.com/index.js:15:48"`, ⟨`"theme"`, `"dark"`⟩) represents a call to the `getItem` method of `localStorage`, located at row 15 and column 48 of https://foo.com/index.js, with extra information about the key of the item being accessed, `"theme"`, and its value, `"dark"`. We use labels to represent also sinks in the taint analysis. When a source generates a value, we put a new label representing that operation into the set that annotates such value; afterward, when a tainted value (i.e., a value annotated with a non-empty label set τ) reaches a sink, we record the information flow as a pair (τ, s), where s is a label representing the sink.

Figure 3 shows the list of instructions supported by our abstract machine. The instructions for manipulating the stack of abstract values (`push`, `pop`), accessing the maps of abstract values associated to local variables (`initvar`, `readvar`, `writevar`), and accessing object properties (`initproperty`, `readproperty`, `writeproperty`) are common between our tool and Ichnaea [19]. We point out that the identifiers that refer to variables and properties in the abstract machine are fully qualified names, which are fresh and unique along the whole script execution. In addition, our abstract machine supports the `join` instruction, which extracts two abstract values from the top of the stack and then pushes their join (in terms of label sets, their union). We need this instruction for our treatment of native functions, whose internal behavior is not observable because their code is not instrumented by Jalangi.

3.2.3 Instruction Generation. The idea behind taint tracking is to propagate the taint of the sub-expressions to the result upon expression evaluation. We hold the invariant that the last values pushed onto the stack represent the taint of the previously calculated sub-expressions, hence we generate instructions that push a number of abstract values on top of the stack equal to the number of sub-expressions. At the end of the evaluation, at the top of the stack there will be the value reflecting the taint of the whole expression. Taints are then propagated across statements (e.g., assignments).

Figure 4 shows how abstract machine instructions are generated for each type of operation specified in the core subset of JavaScript. For each expression and statement, we formalize a simple

Operation	Callbacks	Explanation
Expressions (*Expr*)		
$l \in Lit$	**post**(v_l) EMIT(push($\emptyset$))	Literals are never tainted.
$o \in Obj$ $o \equiv \{p_1 : e_1, \cdots, p_n : e_n\}$	**post**(v_o) EMIT(initproperty($oid(v_o), p_n$)) $\cdots$ EMIT(initproperty($oid(v_o), p_1$)) EMIT(push($\emptyset$))	Initialize properties using the n topmost values of the stack. The object itself is not tainted.
$f \in Fun$	**post**(v_f) EMIT(push($\emptyset$))	Functions are never tainted.
$uop\ e$	**post**(uop, v_e) *do nothing*	The taint of the result corresponds to the taint of the operand at the top of stack, hence nothing to do.
$e_1\ bop\ e_2$	**post**(v_1, bop, v_2) EMIT(join())	The taints of the two operands at the top of the stack propagate to the result.
$name \in Id$	**post**($name$) EMIT(readvar($name$))	Push the taint associated to the variable *name*.
$e_o[e_p]$	**post**(v_o, v_p) EMIT(pop()) EMIT(pop()) EMIT(readproperty($oid(v_o), offset(v_p)$))	Pop the taints of the property name and the object; then, push the taint of the property v_p of v_o.

$e_f(e_1, \cdots, e_n)$	**pre**($v_f, v_1, \cdots, v_n$)
	if v_f *is user-defined function* **then**
	ENTER-USER-FUNCTION($v_1, \cdots, v_n$)
	$v_f \equiv$ `function (`arg_1`, ···, `arg_n`) { ··· }`
	EMIT(initvar(arg_n))
	$\cdots$
	EMIT(initvar(arg_1))
	else
	ENTER-NATIVE-FUNCTION($v_1, \cdots, v_n$)
	end if
	
	post($v_f, v_1, \cdots, v_n, v_r$)
	if v_f *is user-defined function* **then**
	LEAVE-USER-FUNCTION(v_r)
	EMIT(pop())
	EMIT(readvar("_ret_"))
	else
	LEAVE-NATIVE-FUNCTION(v_r)
	end if
		pre: If v_f is a user-defined function, initialize the formal arguments using the n topmost values of the stack. The procedures ENTER-USER-FUNCTION and ENTER-NATIVE-FUNCTION are described in Figure 5 and explained later.
		
		post: If v_f is a user-defined function, discard the called function and read the special `_ret_` variable to load the returned value onto the stack. The procedures LEAVE-USER-FUNCTION and LEAVE-NATIVE-FUNCTION are described in Figure 5 and explained later. The parameter v_r contains the concrete value returned by the function invocation.

Statements (*Stmt*)		
`var` $name$ `=` e	**post**($name, v_e$) EMIT(initvar($name$))	Initialize the variable *name* with the value on top of the stack.
$name$ `=` e	**post**($name, v_e$) EMIT(writevar($name$)) EMIT(pop())	Assign the value on top of the stack to the variable *name*; then, discard the assigned value.
$e_o[e_p]$ `=` e	**post**(v_o, v_p, v_e) EMIT(writeproperty($oid(v_o), offset(v_p)$)) EMIT(pop()) EMIT(pop()) EMIT(pop())	Assign the value on top of the stack to the property v_p of v_o; then, discard the assigned value and the taints of the property name and the object.
`return` e	**post**(v_e) EMIT(writevar("_ret_")) EMIT(pop())	Assign the value on top of the stack to the special `_ret_` variable for communicating the return value to the caller; then, discard the assigned value.

Fig. 4. Rules for generating abstract machine instructions from our core subset of JavaScript; we assume that instructions for the gray-colored expressions have already been issued (i.e., may already have affected the stack).

model of the callbacks implemented in Jalangi for taint tracking purposes [29]. Jalangi allows one to define callbacks both before and after execution of operations: we denote these callbacks with pre() and post(), respectively. The callbacks have access to different parameters depending on the performed operation—for example, callbacks for binary expressions have access to both the operand and the values of the evaluated sub-expressions. For most of the core language constructs, we do not need the full generality of the Jalangi approach and we just define the post() callback. The figure is described in detail in the rest of this section.

Basic Operations. Literal and function expressions are constant values in the source code, which generally do not represent sensitive information; accordingly, a push($\emptyset$) instruction is issued for

the abstract machine, indicating that such values are not tainted. Objects, instead, are more complex because object expressions define a number of properties, let us say n, whose values have been pushed onto the stack just before evaluating the object itself; those n properties must be initialized in reverse order using the n topmost values of the stack, and therefore we emit n `initproperty` instructions.

The taint of the result of unary operations corresponds to the taint of the single operand at the top of the stack, hence it is not necessary to issue any instructions for the abstract machine. Instead, in the case of binary operations, the taint of the two operands must propagate to the result; therefore, we emit a `join()` instruction to merge the two topmost taints into a single one.

Variables and object properties are both considered containers of data, so their taint is associated to the value they carry. A variable is declared in the abstract machine as a result of generating an `initvar(`*name*`)` instruction. Afterward, it can be accessed by issuing `readvar(`*name*`)` for reading and `writevar(`*name*`)` for writing. The *name* parameter includes both the variable name and the identifier of the activation frame where the variable has been declared. Analogously, the access to the property v_p of an object v_o is possible by emitting a `readproperty(`$oid(v_o)$, $offset(v_p)$`)` to get the assigned value and `writeproperty(`$oid(v_o)$, $offset(v_p)$`)` to put another one. In this case, *oid* and *offset* are mappings from JavaScript objects and property values to their corresponding unique identifiers in the abstract machine. It is also worth noting that `writevar` and `writeproperty` do not extract from the stack the value they store, hence those instructions must be followed by `pop()`; furthermore, `readproperty` and `writeproperty` do not pull out of the stack the taints for the object and the property name, so once again a `pop()` instruction must be issued twice.

Function Calls. Handling function calls is the most challenging part of the analysis because the JavaScript specification defines a significant amount of *native* functions—that is, built-in functions available in the browser which are not amenable for instrumentation because their internals are invisible to Jalangi. Ichnaea bridges the lack of information on data dependencies with the help of manually crafted models for specific native functions, which generate instructions for the abstract machine that mimic their internals. However, this strategy is only effective for the supported functions, thus it cannot easily scale to the whole JavaScript standard library and is also hard to maintain in terms of engineering effort.

Since we do not have access to Ichnaea and its models of native functions, we design a different solution based on standard concepts, which is not as precise as Ichnaea's manually crafted models but works for any invocation of both user-defined and native functions, thus being more general and easier to maintain. Essentially, since operations performed within a native function are unknown, in such a case we determine an *over-approximation* of the taint for the returned value. Along with the existing assumptions, we suppose that the instrumented code allows us to intercept the entry into and exit from a function invocation, be it user defined or native, except for the case of the invocation of a native function by another native function, because such calls are not observable by Jalangi due to lack of instrumentation. With this in mind, let us examine all the possible types of function calls.

The simplest instance is the one in which *a user-defined function invokes another user-defined function*: in this case, when the function is called, the stack contains the taints of n actual arguments as the n topmost values. Hence, first of all, the formal arguments must be initialized in reverse order by generating n `initvar` instructions. Then, upon reaching a `return` *e* statement, the taint for the value of *e* on top of the stack is stored in a special variable, called `_ret_`, with the `writevar("_ret_")` instruction, and is subsequently pulled out of the stack with `pop()`. After the call, the top value of the stack is the taint for the called function, so it is discarded with `pop()`, and finally the value associated to the special `_ret_` variable is read by emitting a `readvar("_ret_")` instruction, to communicate the returned value to the caller.

When *a user-defined function invokes a native function*, the taints for n actual arguments combine into a single value as a result of generating $n - 1$ `join` instructions; the obtained value on top of the stack is the over-approximated taint for the result of the invocation. If no argument has been passed, we push $\emptyset$ onto the stack. Such taint is then recursively joined with the taints associated to the properties of objects passed as an argument because the values of these properties may influence the result as well. At the end, the taint for the result is stored into the special `_ret_` variable if the returned value is primitive, and otherwise we recursively propagate it to the properties of the returned object and replace the taint associated to the `_ret_` variable with $\emptyset$, thus preserving the rule that objects are never tainted. The recursion through objects for updating the taint of their properties must take into account the possibility of cyclic references, such as in case of an object referring to itself; to avoid infinite recursion, we process an object only if *it has not been visited yet* in a single traversal.

Finally, we discuss the case in which *a native function invokes a user-defined function*. This may happen in the case of *higher-order* native functions—that is, functions that accept another function as an argument. An example is the `Array.prototype.map` native function, which progressively applies a given callback to all the elements of an array and then creates a new array with the corresponding results. Note that we can observe this kind of invocation because we are able to detect when the execution enters a user-defined function, knowing that the last observed operation is a native function call. In this case, we duplicate the topmost value of the stack (i.e., the resulting taint of the native function) for each primitive value passed as an argument while issuing `push(`$\emptyset$`)` for each passed object. This means that the taints for the arguments may depend on each of the arguments passed to the native caller. Duplication is achieved using an auxiliary `_arg_` variable, which is written once and read as many times as necessary. On the return, we load the taint for the `_ret_` variable and perform a weak update of the native function's resulting taint by joining these two values: in fact, the user-defined callback may have returned a value annotated with a new label, which we have to consider for the final result of the native function call or the arguments of another invocation of the callback.

Our approach is formally described in Figure 5: we define a collection of procedures that emit instructions for the abstract machine whenever the execution enters and leaves a user-defined or native function, in accordance with the preceding rules. In particular, we call from inside user-defined functions the Enter-User-Function procedure before initializing formal arguments and the Leave-User-Function procedure before returning to the caller, and we call the Enter-Native-Function and Leave-Native-Function procedures respectively before and after the invocation of native functions, since we are able to observe the entry and exit of this kind of invocation only from the caller (see Figure 4). The behavior of these procedures is stateful with respect to a stack of abstract activation frames, which reflects the nature of invoked functions in the runtime call stack of being user defined (with the `"USER"` string) or native (with the `"NATIVE"` string). We clarify that such abstract call stack is just an auxiliary data structure for generating abstract machine instructions, and hence the abstract machine is totally independent from it. It is possible to manipulate the abstract call stack with the following standard procedures: Push-Frame pushes an abstract frame onto the stack, Pop-Frame removes an abstract frame from the stack, and Top-Frame gets the abstract frame at the top of the stack without removing it.

3.2.4 Additional JavaScript Features. The JavaScript standard specification includes a lot of additional features that we have not covered in the formal discussion. For the majority of them, including arrays, getters and setters, dynamic code evaluation (`eval()`), and arguments in function calls, we borrow the treatment from Ichnaea and so we refer readers to the work of Karim et al. [19] for further details. Instead, we briefly explain why and how we behave differently with respect to one important feature of the language: exception handling.

```
Require: a_1, ··· , a_n Actual arguments
  procedure ENTER-USER-FUNCTION(a_1, ··· , a_n)
    if TOP-FRAME() = "NATIVE" then
      EMIT(writevar("_arg_"))
      for i ← 1..n do
        if a_i is primitive then
          EMIT(readvar("_arg_"))
        else if a_i is object then
          EMIT(push(∅))
        end if
      end for
    end if
    PUSH-FRAME("USER")
  end procedure

Require: r Return value
  procedure LEAVE-USER-FUNCTION(r)
    POP-FRAME()
    if TOP-FRAME() = "NATIVE" then
      EMIT(readvar("_ret_"))
      EMIT(join())
    end if
  end procedure

Require: a_1, ··· , a_n Actual arguments
  procedure ENTER-NATIVE-FUNCTION(a_1, ··· , a_n)
    if n = 0 then
      EMIT(push(∅))
    else
      for n − 1 times do
        EMIT(join())
      end for
    end if
    for i ← 1..n do
      if a_i is object then
        OBJ-TAINT(a_i)
        EMIT(join())
      end if
    end for
    PUSH-FRAME("NATIVE")
  end procedure

Require: r Return value
  procedure LEAVE-NATIVE-FUNCTION(r)
    POP-FRAME()
    EMIT(writevar("_ret_"))
    EMIT(pop())
    if r is object then
      OBJ-PROPAGATE(r)
      EMIT(push(∅))
      EMIT(writevar("_ret_"))
      EMIT(pop())
    end if
  end procedure
```

```
Require: o The object to get the taint from
  procedure OBJ-TAINT(o)
    EMIT(push(∅))
    if o has not been visited yet then
      Let o ≡ {p_1 : v_1, ··· , p_n : v_n}
      for i ← 1..n do
        if v_i is primitive then
          EMIT(readproperty(oid(o), offset(p_i)))
        else if v_i is object then
          OBJ-TAINT(v_i)
        end if
        EMIT(join())
      end for
    end if
  end procedure

Require: o The object to which to propagate the taint (from the special
  _ret_ variable)
  procedure OBJ-PROPAGATE(o)
    if o has not been visited yet then
      Let o ≡ {p_1 : v_1, ··· , p_n : v_n}
      for i ← 1..n do
        if v_i is primitive then
          EMIT(readproperty(oid(o), offset(p_i)))
          EMIT(readvar("_ret_"))
          EMIT(join())
          EMIT(writeproperty(oid(o), offset(p_i)))
          EMIT(pop())
        else if v_i is object then
          OBJ-PROPAGATE(v_i)
        end if
      end for
    end if
  end procedure
```

Fig. 5. Procedures for generic function invocations, describing our treatment of native and user-defined functions

When a function executes a `throw` statement, the JavaScript engine interrupts that function and returns to the caller recursively, until it hits a `try-catch` statement or the stack of activation frames is empty. In the first base case, the variable in the `catch` clause is filled with the parameter of the `throw` statement. It is well specified that Ichnaea, as well as our tool, passes the taint of such parameter to the variable in the `catch` clause through another special variable called `_throw_`, similar to the treatment of values returned by functions. However, the description of Ichnaea does not mention anything about cleaning up the stack of abstract values from intermediate values of interrupted functions. This detail is important for the correctness of the taint tracking engine: if not considered properly, the state of the abstract machine may get out of sync with respect to the concrete state every time an exception occurs, leading to erroneous analysis results. To remedy this problem, we extend our model of abstract activation frame with an additional numeric

information, which we call *frame pointer*. This value corresponds to the height of the stack of abstract values at the time of creation of the activation frame—that is, when the JavaScript engine enters the last invoked function. When the running function gets interrupted exceptionally, a number of pop() instructions is emitted until the height of the stack of abstract values is equal to the corresponding frame pointer. This way, no more taints associated to intermediate values of the interrupted function are present in the stack of abstract values, hence the alignment between the abstract and the concrete state is preserved.

4 EMPIRICAL WEB STORAGE ANALYSIS

We now explain how we performed our empirical measurement of the use of web storage in the wild and report on the most relevant findings of our study, based on an automated classification of the detected information flows involving the web storage from a security and privacy perspective.

4.1 Methodology

We use the developed dynamic taint tracking engine to automatically identify information flows involving the Web Storage API in the top 10k domains of the Tranco list [22] generated on April 17, 2023.[2] 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*, thus taking the web storage perspective. Table 1 reports the different sources and sinks considered in our analysis, largely inspired by previous web measurements based on information flow control [10, 30]. Note that the web storage was largely ignored as source or sink in previous work, to the best of our knowledge. Besides detecting the flows, we use our taint tracking engine to log all the calls to `setItem`, saving the key, the value, and the script that performed the call. This is useful to understand the popularity of web storage in the wild and investigate other use cases that do not fit our information flow pattern (see the privacy analysis in Section 5.2).

We use the Puppeteer library[3] to drive our instrumented browser (Google Chrome) to each domain in the Tranco list, leaving 120 seconds to render the HTML content after connecting. For each correctly accessed domain, we leverage taint tracking to collect all the information flows involving the Web Storage API on all the frames within the web page. The use of a relatively high timeout of 120 seconds is intended to give to our dynamic analysis a reasonable amount of time to discover useful information flows during JavaScript execution. 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.

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 1), unless network communication only involves the same origin where the flow was 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 that reads the content of the local storage and sends it to https://www.bar.com, thus potentially leaking sensitive information

[2] https://tranco-list.eu/list/GZ7NK
[3] https://pptr.dev/

Table 1. List of Sources and Sinks Used in Our Taint Tracking Engine

	Class	Details	Tracked Information
Sources	Cookies	`document.cookie`	Read value and script URL

	Current URL	`document.URL`
		`location`
		`document.location`
		`window.location`
			Read value and script URL

	Navigator	`navigator.geolocation`
		`navigator.language`
		`navigator.platform`
		`navigator.userAgent`
			Read value and script URL

	Network	`XMLHttpRequest` (input)	Input URL and script URL
	Web storage	`localStorage.getItem` `sessionStorage.getItem`	Read value, key and script URL
Sinks	Cookies	`document.cookie`	Written value and script URL
	Network	`XMLHttpRequest` (output) `navigator.sendBeacon` `src` attribute of HTML element	Output URL and script URL
	Web storage	`localStorage.setItem` `sessionStorage.setItem`	Written value, key, and script URL

from https://www.foo.com. The reason we define confinement at the origin level, rather than at the site[4] level, is that web storage content is origin scoped and thus subject to SOP.

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 [13], we can detect the involvement of known web trackers in the identified web storage accesses. Note that although filter lists are not perfect [14, 17], they are actively maintained by their communities and routinely used both in web privacy measurements and browser extensions such as Ghostery.[5] In Section 5, we further investigate tracking flows to better assess their privacy implications.

Persistence. A last relevant aspect is the *persistence* of the information involved in the flow. Although 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 [33] 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, for example, because local storage and session storage information is combined together before network communication.

[4]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).
[5]https://www.ghostery.com/

Table 2. Sources and Sinks Involved in Confidentiality and Integrity Flows

	Class	#Flows	#Domains
Confidentiality	Cookies	444	123
	Network	390	224
Integrity	Cookies	1,225	426
	Current URL	910	305
	Navigator	224	113
	Network	314	226

4.2 Measurement Results

Overall, our crawler successfully accessed and instrumented JavaScript code on 7,179 domains, with a success rate of the instrumentation process of around 89% (other failures were attributed to connection timeouts and generic errors). At the end of the crawling, we detected 19,290 information flows involving the Web Storage API on 1,887 domains (26%). These included a significant number of flows where the web storage acts as both source and sink, which we filtered out because they are confined to the Web Storage API and thus have limited security and privacy implications. Moreover, we removed all the flows where the source and the sink were located in different scripts, as we manually verified that most such cases are false positives arising from the complexity of performing accurate taint tracking on real-world JavaScript. After filtering, we were left with 3,402 information flows on 995 domains (14%), including 834 confidentiality flows (25%) and 2,568 integrity flows (75%). Overall, we detected 69,262 calls to `setItem` on 3,884 domains (54%). This means that although web storage is used on more than half of the domains that we crawled, the number of cases where some relevant information flow is found is much lower. This is related to the fact that web storage can be used for generic reasons which have nothing to do with our list of sources and sinks, such as to store timestamps, language preferences, or any other type of client-side information.

Table 2 reports a first breakdown of the detected flows in terms of the involved sources and sinks.[6] As we can see, 390 confidentiality flows (47%) involve a network sink. This already suggests that privacy might be a concern, since it is common for web storage information to be communicated over the network. We then focus on a more fine-grained classification of the detected flows, as we described in the previous section. Overall, we observe that 647 flows (19%) are *external*—that is, 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, 2,003 flows (59%) are related to *tracking*—that is, the majority of the detected flows can be attributed to scripts downloaded from known trackers included in popular filter lists. Finally, 2,490 flows (73%) only make use of local storage, 865 flows (25%) only make use of session storage and just 47 flows make use of both. All this combined information preliminarily suggests that a common use case of web storage is *persistent web tracking via the local storage, possibly involving 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 visualized as a heatmap in Table 3. The table supports the following selected observations:

- Confidentiality flows are roughly equally split between internal and external flows, whereas integrity flows are mostly internal (89%). This shows that it is more common to send web

[6]The sum of the integrity flows exceeds 2,568, as 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 3. Classification of the Detected Information Flows

	Confidentiality	Integrity	Internal	External	Tracking	Non-Tracking	Local	Session	Both
Confidentiality			462 (55%)	372 (45%)	507 (61%)	327 (39%)	768 (92%)	53 (6%)	13 (2%)
Integrity			2,293 (89%)	275 (11%)	1,496 (58%)	1,072 (42%)	1,722 (67%)	812 (32%)	34 (1%)
Internal	462 (17%)	2,293 (83%)			1,615 (59%)	1,140 (41%)	1,936 (70%)	783 (28%)	36 (1%)
External	372 (57%)	275 (43%)			388 (60%)	259 (40%)	554 (86%)	82 (13%)	11 (2%)
Tracking	507 (25%)	1,496 (75%)	1,615 (81%)	388 (19%)			1,454 (73%)	514 (26%)	35 (2%)
Non-Tracking	327 (23%)	1,072 (77%)	1,140 (81%)	259 (19%)			1,036 (74%)	351 (25%)	12 (1%)
Local	768 (31%)	1,722 (69%)	1,936 (78%)	554 (22%)	1,454 (58%)	1,036 (42%)			
Session	53 (6%)	812 (94%)	783 (91%)	82 (9%)	514 (59%)	351 (41%)			
Both	13 (28%)	34 (72%)	36 (77%)	11 (23%)	35 (74%)	12 (26%)			

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 (61%). Remarkably, however, roughly the same percentage of integrity flows can similarly be attributed to trackers (58%). Indeed, the table also shows that the majority of the tracking flows are integrity flows (75%). 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 (57% vs. 17%), and the majority of the external flows can be attributed to trackers (60%). Moreover, a significant percentage of the tracking flows are external (19%). This suggests that trackers may send web storage information to third parties.
- Tracking flows normally operate on local storage (73%) rather than session storage. Moreover, flows involving the session storage are more likely to be internal than flows involving the local storage (91% vs. 78%). This suggests that local storage is the prime target of trackers, whereas 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 (92% vs. 67%), just like external flows involve local storage more frequently than internal flows (86% vs. 70%). This shows that the persistent information saved in the local storage is often the target of information leaks, likely toward third parties.

Table 4. Additional Breakdown of the External Information Flows

	Same Site	Cross Site
Confidentiality	91 (24%)	281 (76%)
Integrity	28 (10%)	247 (90%)
Tracking	68 (18%)	320 (82%)
Non-Tracking	51 (20%)	208 (80%)
Local	110 (20%)	444 (80%)
Session	7 (9%)	75 (91%)
Both	2 (18%)	9 (82%)

Table 5. Most Popular Libraries Introducing Information Flows in the Crawled Domains

Library	#Flows	#Domains	Tracker?
https://bat.bing.com/bat.js	322	142	✓
https://mc.yandex.ru/metrika/tag.js	108	76	✓
https://cdn.cxense.com/sp1.html	39	39	✓
https://top-fwz1.mail.ru/js/code.js	78	38	✓
https://s.pinimg.com/ct/lib/main.da2a1c8f.js	29	28	✗
https://mc.yandex.ru/metrika/watch.js	20	19	✓
https://sofire.bdstatic.com/js/dfxaf3-635b4cd6.js	47	13	✓
https://j.6sc.co/6si.min.js	13	13	✓
https://script.4dex.io/localstore.js	10	10	✓
https://assets.ubembed.com/universalscript/releases/v0.180.0/bundle.js	9	9	✗

The Tracker? column shows whether the domain providing the library is included in a filter list.

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—that is, 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 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 4. 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, including tracking flows (82%). 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 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 5 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 majority of these libraries (8 out of 10) are related to web tracking and the most popular library is used for tracking on 142 domains. Overall, we identify 331 distinct domains (33% of the domains with some information flow) making use of at least one of the libraries reported in the table.

4.3 Accuracy of Dynamic Taint Tracking

Our taint tracking engine may suffer from both false positives and false negatives. False positives may be introduced, for example, as the result of the over-approximation introduced by the invocation of native functions which are not instrumented by our implementation. False negatives may also occur for generic reasons, such as dynamic code generation that accidentally escapes Jalangi. We consider false positives particularly harmful in our context because our work draws conclusions based on the detected information flows.

To estimate the prevalence of false positives, we perform a manual investigation on a random subset of 212 flows from 66 domains. In particular, for each detected flow, we first check whether the value read from the source occurs as a substring of the value written to the sink: we consider these cases as true positives because there is clear evidence that some information was transferred from the source to the sink. Of course, this simple recipe cannot account for all possible cases, such as the value read from the source may be encoded or encrypted before reaching the sink, hence we integrate our analysis with an in-depth manual inspection. In total, we identify 197 true positives (93%), most often thanks to the simple recipe based on substring match, hence we are confident about the accuracy of our findings.

5 CASE STUDY: WEB TRACKING

Since we showed that the majority of the information flows involving the web storage can be attributed to tracking purposes, we now perform an additional investigation of this important use case of web storage. In particular, we first carry out a systematic evaluation of the preliminary tracking detection heuristics based on filter lists, which we compare against a different detection approach proposed in the literature [9]. Based on the results of this analysis, we improve tracking detection by combining the two techniques to minimize false positives and deep dive into the final set of tracking flows. We first analyze such flows to understand their privacy implications and finally further inspect them to identify GDPR violations on popular sites.

5.1 Effectiveness of Filter Lists

Filter lists are a state-of-the-art tool to mitigate the dangers of web tracking; however, prior research identified relevant shortcomings in their construction and maintenance [14, 17]. In particular, filter lists may suffer from both false positives (incorrect inclusion of non-trackers) and false negatives (incomplete coverage of existing trackers). Moreover, filter lists only allow for a coarse-grained detection of tracking flows because all the flows created by a script downloaded from a known tracker are marked as tracking flows in our web measurement. This is sub-optimal, as even known trackers may use the web storage for different purposes. We thus consider an alternative approach to detect tracking behavior and we compare it against the use of filter lists to collect additional insights.

5.1.1 Methodology. To better understand the effectiveness of filter lists and their impact on the results of our web measurement, we consider an alternative approach to detect tracking flows based on *semantic information*—that is, web storage items must contain uniquely identifying information to be used for tracking. For example, a tracker may set a local storage item `lang` with value `en` to track the language of the user; however, this information cannot be used for tracking individuals—long, uniquely identifying information is rather required. In particular, we adapt the tracking detection heuristics proposed for first-party cookies in the work of Chen et al. [9] to the web storage setting, as the tracking techniques available for first-party cookies and web storage are similar, except for the inclusion of an improvement proposed in recent work [23]. According

Table 6. Comparison of the Classification Approaches for Information Flows (Confusion Matrix)

	Script URL in Filter List	**Script URL Not in Filter List**
Heuristics Marked Flow as Tracking	TP: 949	FN: 266
Heuristics Marked Flow as Non-Tracking	FP: 1,054	TN: 1,133

to the proposed heuristics, a web storage item (k, v) may be used for tracking if and only if all the following conditions hold:

(1) The item is set in the local storage—that is, it persists when closing the browser;
(2) The length of the unquoted value v is at least 8 characters;
(3) A web storage item (k, v') with the same key k is set when visiting the same web page from a different browser instance—that is, a fresh browser without any web storage information;
(4) The values v and v' are "significantly different,"—that is, the web storage item allows one to readily tell apart different browser instances; and
(5) The web storage item (k, v) persists with the same value v when the page is reloaded.

The notion of "significantly different" values is also taken from the work of Chen et al. [9]. Specifically, values are pre-processed to remove all the timestamps occurring therein and to recursively strip their longest common sub-sequence, until the longest common sub-sequence includes at most two characters. The similarity score of the residual values is then computed using the Ratcliff-Obershelp algorithm, setting a threshold of 66%. Values whose similarity score is below the threshold are considered significantly different. The last condition was proposed in the work of Randall et al. [23] to ensure that ephemeral information whose scope is limited to a single page request is not incorrectly marked as useful for tracking.

5.1.2 Results. We now have two different approaches to detect tracking flows: one based on filter lists and another one based on existing heuristics capturing necessary conditions for tracking. We compare the performance of two approaches in practice by means of Table 6—that is, a 2×2 matrix capturing how each information flow is classified by the two methods. If we consider the heuristic approach as the ground truth, because it captures reasonable requirements for web tracking, the table is effectively a *confusion matrix* reporting on true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). The resulting values of the true-positive rate and true-negative rate coming from the use of filter lists are 0.78 and 0.52, respectively. The high true-positive rate reflects a low number of false negatives and means that filter lists are very effective at detecting tracking flows; however, the low true-negative rate reflects a high number of false positives, which implies that filter lists are too coarse grained in practice to support classifications at the flow granularity. In particular, this suggests that it is common practice even for scripts downloaded from a known tracker to introduce information flows that are not actually used for tracking.

Nevertheless, from a privacy perspective, classifying scripts is arguably more interesting than classifying information flows. We then carry out a similar analysis but with a different level of granularity—that is, working on script URLs rather than on information flows. We first collect all the script URLs in our dataset of flows and mark all of them as tracking or non-tracking, depending on whether they occur in a filter list or not. We then perform a different labeling, based on whether the script URL was ever found to be responsible for the introduction of at least one tracking flow on some web page according to the heuristics adapted from Chen et al. [9]. The results are shown in Table 7, leading to a true-positive rate and a true-negative rate of 0.82 and 0.51, respectively. This result is quite interesting because it suggests two key observations. First, although the true-positive rate is even higher than before, there are a non-negligible number of tracking scripts (90)

Table 7. Comparison of the Classification Approaches for Scripts (Confusion Matrix)

	Script URL in Filter List	Script URL Not in Filter List
Heuristics Marked Script as Tracking	TP: 397	FN: 90
Heuristics Marked Script as Non-Tracking	FP: 337	TN: 344

that are not detected by filter lists, hence the use of dynamic taint tracking may be useful to uncover new potential trackers. Moreover, the low true-negative rate still reflects a significant number of false positives, which tells us that many web trackers already included in filter lists do not yet use the web storage for tracking purposes—that is, cookie-based tracking is likely still more prevalent in the wild than tracking via the web storage. Note that this does not tell that filter lists are wrong, just that tracking does not (yet) happen via the web storage.

Although a full investigation of the 90 potentially tracking scripts evading detection by filter lists is difficult and goes beyond the scope of the present article, we manually inspected some of the domains serving such scripts and it was easy to find several examples of content used for analytics and tracking. A particularly interesting example is a script served by a sub-domain of rockerbox.com, an analytics service advertising on its homepage the implementation of custom tracking techniques designed to counter third-party cookie blocking and the Intelligent Tracking Protection system by Apple. We conjecture that other services might leverage web storage to bypass protection against more known tracking vectors like cookies, and we plan to investigate this further as future work.

5.1.3 Conclusion. To conclude, we investigated two techniques to detect tracking flows (filter lists and heuristics), yet none of them turned out to be perfect. In the following, we use the term *tracking flows* to refer to those flows which are marked as tracking according to both techniques, so as to minimize false positives. In particular, observe that the combination of the two techniques ensures that (i) the information flow was introduced by a script downloaded from a known tracker and (ii) the script involves a local storage item containing a potential user identifier. Based on this new definition, we identify 949 tracking flows (28%) spread across 397 domains (40% of the domains with some information flow). This confirms that web storage is often used for tracking purposes in the wild, although the introduction of the additional check on user identifiers roughly halves the number of tracking flows identified in practice.

5.2 Privacy Implications

We now perform a systematic privacy analysis of the identified uses of web storage in the wild. We first focus on the uses of local storage in a third-party position (i.e., within an iframe). We identify 1,708 domains setting at least one key in the local storage within an iframe, including 1,338 domains storing a tracking item according to our detection heuristics. This amounts to 44% of the domains where we found a call to `setItem`. Since such local storage items are set in a tracker-controlled position, they can be readily leveraged for cross-site tracking, thus allowing the owner of the domain loaded in the iframe to build a precise navigation profile of the user across different sites. Table 8 reports the top domains serving content in iframes setting a tracking item in the local storage, sorted by decreasing number of domains embedding scripts from them. Despite the potential tracking capabilities of such domains, we observe that the majority of them are not marked as trackers according to filter lists. Indeed, if we additionally checked whether the domain loaded in the iframe occurs in a filter list, the number of domains susceptible to cross-site tracking would drop just to 202 (5%), which is somewhat concerning.

We then move to the analysis of the uses of local storage in a first-party position (i.e., within the top-level page). We expect most of these flows to be attributed to fewer privacy invasive activities

Table 8. Domains with the Highest Potential for Cross-Site Tracking, Sorted by Decreasing Number of Embedding Domains

Embedded Domain	#Domains	Tracker?
www.google.com	745	✗
www.youtube.com	185	✗
ads.pubmatic.com	44	✓
player.vimeo.com	41	✗
eus.rubiconproject.com	41	✓
www.recaptcha.net	38	✗
ls.hit.gemius.pl	35	✓
cdn-gl.imrworldwide.com	34	✓
consent-pref.trustarc.com	29	✗
geo.captcha-delivery.com	23	✗

The Tracker? column shows whether the domain is included in a filter list or not.

than the previous ones, such as site analytics and statistics collection, which do not enable cross-site tracking. Nevertheless, recent literature identified abuses of first-party items as well [20, 23]. For these cases, it is important to understand how local storage items flow across different parties, which we can do by leveraging our taint tracking approach. In particular, we identify 315 tracking flows leading to network sinks across 91 domains (i.e., around 33% of the tracking flows involve network communication). The majority of such flows are indeed immediately compliant with our intuition of same-site tracking—that is, the flow is directed to the same site that originally set the tracking item: this is the case for 283 flows (90%). The other 32 flows on 16 domains are more interesting from a privacy perspective because they might reveal *cookie syncing* practices [20], which we should rather call *web storage syncing* in our case. This happens when a script loaded from some tracker sets a user identifier in the local storage that is later communicated to a different party. Still, we manually vet all these cases and observe that the change of site never implies a change of organization—for example, an item set from bdstatic.com is communicated to baidu.com, but both domains are owned by the same company (Baidu).

To conclude, our investigation shows that the use of web storage in the wild has significant privacy implications because it may allow cross-site tracking (much as third-party cookies) in 44% of the domains where we found a call to `setItem`. However, we did not detect any abuse of first-party storage items, which are just used for same-site tracking according to our collected data. This means that users of privacy-aware browsers implementing partitioned storage, such as Mozilla Firefox and Brave, do not suffer major privacy threats from the use of web storage. We leave the investigation of more sophisticated privacy threats such as UID smuggling [23] to future work, since its measurement is more complicated to carry out and its popularity was proved to be still limited in practice in recent studies.

5.3 Web Storage and GDPR

We finally investigate to which extent web storage information in the wild is compliant with GDPR. Cookies and web storage play very similar roles in practice and are subject to the same regulations; however, web storage is arguably less known than cookies and received less attention by the research community, hence it deserves careful scrutiny. We particularly investigate the consent and transparency dimensions of GDPR.

Table 9. Presence of Cookie Banners in the Wild

	Includes Cookie Banner	No Cookie Banner
Includes Tracking Flows	39	60
No Tracking Flows	49	51

5.3.1 Web Storage and Consent. A first insight on whether the use of web storage in the wild complies with the consent dimension of GDPR already comes from the results of our previous experiment in Section 4. In particular, our web measurement identified tracking flows across 397 different domains of Tranco. Since our crawler does not interact with web pages in any way, and in particular it does not click through cookie banners, this immediately tells us that 397 domains (10% of the domains making a call to `setItem`) exhibit tracking behavior through the web storage although the user never granted her consent. These cases are arguably violations to GDPR, as tracking flows are not strictly needed for website functionality.

To further investigate the consent dimension of GDPR, we also carry out an additional experiment. Starting from the top 10k domains in the Tranco list, we identify all the domains including a tracking flow and all the domains without any tracking flow based on our measurement. We then sample 100 domains from each of the two categories and manually investigate their homepages to check for the presence of cookie banners, successfully accessing 199 of them. Table 9 shows how such domains are distributed based on the presence of cookie banners and tracking flows:

- Thirty-nine domains include both a tracking flow and a cookie banner. These domains may provide valuable information to their users via cookie banners; however, they may also perform tracking via the web storage even before the cookie banner is clicked, hence they are not fully compliant with GDPR.
- Sixty domains include a tracking flow but do not have a cookie banner at all. These cases are concerning, as they arguably represent violations to the consent dimension of GDPR. Interestingly, 15 domains out of 60 are Chinese (25%) and sit out of the European Union.
- Forty-nine domains have a cookie banner but do not include any tracking flow. These domains are either compliant with GDPR or just do not use the web storage for tracking purposes. We manually confirmed that 19 out of such domains populate the web storage after clicking through the cookie banner.
- Fifty-one domains include neither a tracking flow nor a cookie banner. These cases do not perform any form of tracking via the web storage and therefore do not need to explain its use and ask for consent through a cookie banner.

These numbers suggest that many uses of web storage in the wild are not yet compliant with the consent dimension of GDPR, as we identify from 60 to 99 violations in 199 domains, based on whether we want to optimistically consider the presence of a cookie banner alone as sufficient for GDPR compliance. The amount of violations thus ranges from 30% to 50% approximately, depending on how we count violations.

5.3.2 Web Storage and Transparency. We finally investigate to which extent web storage is accounted for in existing privacy policies, as detailed next.

Dataset Construction. We automatically scraped the privacy policies of the websites where web storage is used for setting tracking information. We used the tool developed in the work of Hosseini et al. [16] to automatically scan websites for the links to their privacy policies. This led to a corpus of 420 policies downloaded from 397 domains. For our current study, we only consider privacy policies written in English, as this makes it feasible to manually confirm the correctness of our

Table 10. Keywords Searched in the Privacy Policies and Cookie Banners, Defining Policy Categories

Category	Keywords
Cookies	"cookie," "cookies"
Web Storage	"local storage," "localstorage" "session storage," "sessionstorage" "web storage," "webstorage"
General Terms	"similar technology," "similar technologies" " tracking technology," "tracking technologies" "other technology," "other technologies"

findings. To extract the text of the privacy policies from the original raw dataset of HTML pages, we used boilerpipe,[7] a plain-text-from HTML extraction library, to remove unnecessary HTML like menus, footers, and scripts. We finally scrutinized all the extracted textual versions of the privacy policies to remove the unavoidable errors of an automated scraping pipeline like the one in the work of Hosseini et al. [16]. In the end, we were left with a dataset of 373 privacy policies from 234 domains (i.e., roughly 89% of the extracted policies turned out to be correct). The filtered-out cases include a few privacy policies which do not contain any textual content or are written in a language other than English, for which we cannot confirm correctness.

Policy Analysis. To investigate whether the collected policies are transparent with respect to the use of web storage, we search for specific keywords in the extracted text. We specifically look for the strings "local storage," "session storage," and "web storage," possibly without the whitespace. We also look for the strings "similar technology," "other technology," "tracking technology," and their plural versions because we empirically observed that websites sometimes generalize over cookies by using such terminology. Table 10 reports the full set of keywords that we look for in the collected policies. Based on the keywords therein, each policy can be assigned to one or more categories: "Cookies," "Web Storage," and "General Terms." We carefully scrutinize all the policies to ensure that the automated assignment of categories performed by our analysis script is correct.

Figure 6 shows how the collected policies are categorized according to the considered keywords. As we can see, only 26 privacy policies (7%) explicitly mention specific keywords related to the web storage, which shows that the majority of the websites using web storage for tracking purposes are not fully transparent with respect to its adoption. If we take a more relaxed approach and rather focus on policies including general terms which might be associated to the web storage by experienced users, we instead find 196 such policies (52%). This is somewhat reassuring; however, we observe that these policies do not necessarily provide a clear understanding of data collection, for example, because users would not know the details of the involved tracking technologies and may face challenges at clearing the corresponding client-side data. Nevertheless, we also observe that references to cookies are much more popular in privacy policies than references to the web storage because 309 policies mention cookies at least once (82%). To confirm the validity of our results, we perform a random sampling of 10 policies where a keyword was found and 10 policies where no keyword was found to confirm correctness based on our human understanding of the policies.

Cookie Banners. To further shed light on the transparency dimensions of GDPR, we also perform the classification proposed for privacy policies on the set of the 88 collected cookie banners. The

[7] https://github.com/slaveofcode/boilerpipe3

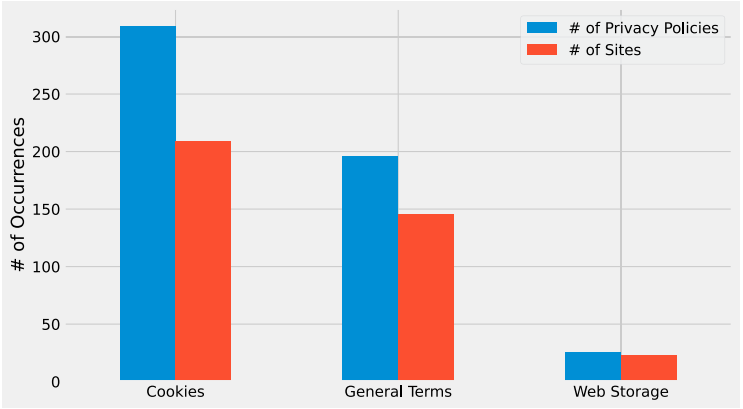

Fig. 6. Comparison of keywords presence in privacy policies.

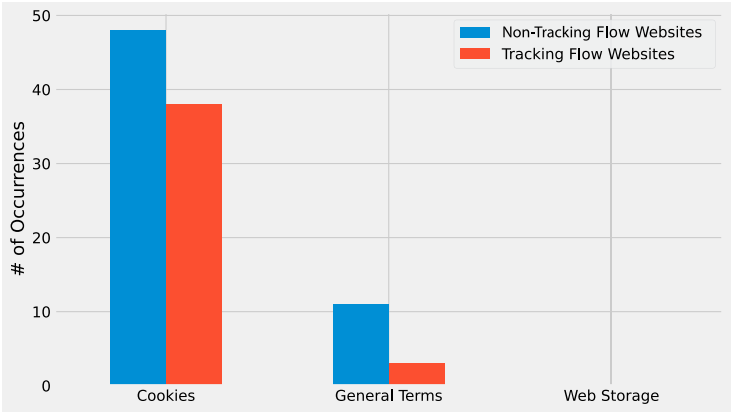

Fig. 7. Comparison of keywords presence in cookie banners.

results are shown in Figure 7: it is remarkable that not even a single cookie banner ever explicitly mentions the web storage. Moreover, even references to general terms that might refer to the web storage are uncommon in cookie banners. This is concerning because we observed that at least 39 domains shipping the cookie banners (43%) use the web storage for tracking purposes, and this number may further increase when clicking through the cookie banners.

6 RELATED WORK

Many papers presented dynamic analysis techniques for JavaScript, we refer to the work of Andreasen et al. [3] 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 in 2018 by Belloro and Mylonas [5]. 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 carry out a systematic analysis of its use in the wild, based on an automated categorization of the detected information flows along different axes. Moreover, 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 that only detects API calls and therefore cannot discriminate between actual information leaks and simple "feature detection" libraries like Modernizr.[8] Of course, their analysis did not take a look into GDPR compliance for web storage content, as GDPR was not yet in effect when their study was performed.

Chen and Kapravelos [10] presented a taint tracking engine called *Mystique* and used it to track information leakage from browser extensions. 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 [9]. 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 that is exchanged with multiple third parties. In our work, we reuse their heuristics for detecting user identifiers in the web storage setting.

Sjösten et al. [30] proposed EssentialFP, a principled approach to the dynamic detection of browser fingerprinting. EssentialFP is based on dynamic analysis and particularly on an extension of JSFlow [15]. 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. [19] implemented a platform-independent dynamic taint analysis tool for JavaScript, called *Ichnaea*. They encoded the taint propagation logic as instructions for an abstract machine, so as to leverage an existing JavaScript instrumentation framework called *Jalangi* [29]. 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 some modifications, yet it is targeted to a different application scenario. Unfortunately, we could not reuse Ichnaea directly in our analysis because it is not publicly available.

Staicu et al. [32] performed an empirical investigation of information flows in existing JavaScript code. 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 [29], which the authors used to implement a dynamic information flow tracker inspired to JSFlow [15]. Based on their investigation, we decided to only track explicit flows in our analysis.

Previous research explored various techniques to detect and counter web tracking [9, 14, 24, 27]. Our work mainly focuses on understanding and exploring the usage of web storage in the wild, covering different use cases (including tracking). This means that our work does not investigate tracking as deeply as previous work on the topic; however, existing work on tracking does not look into web storage as thoroughly as we do here. Cookie banners and privacy policies have been widely studied in literature, covering different axes of GDPR [6, 11, 31]. Different sets of tools are being used to collect and pre-process privacy policies from websites to study cookies compliance with GDPR, whereas our work investigates to which extent web storage information in the wild is compliant with GDPR.

7 CONCLUSION

In this article, 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

[8] https://modernizr.com/

analysis showed 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. A further investigation on web tracking highlighted that web storage is not yet as popular as cookies for tracking purposes; however, taint tracking is useful to detect potential new trackers not included in standard filter lists. It is also concerning that many websites do not make use of web storage according to the GDPR regulations, based on our empirical investigation in the wild. The findings of our work thus motivate the need for further research on the security and privacy implications of web storage content.

As future work, constructive solutions designed to prevent web storage abuses would be particularly worth investigating, such as the recently proposed "page-length storage" approach designed to mitigate the effects of stateful web tracking [18]. 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 in practice. Finally, we would like to further refine our classification of information flows to account for common use cases that we anticipate, such as 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.

ACKNOWLEDGMENTS

We would like to thank Hamza Rouhani for his contribution to this project as part of his internship at our university. We also thank the anonymous MadWeb'22 reviewers for their useful feedback on the original, shorter version of this article and the ACM TWEB reviewers for their excellent feedback on the extended version.

REFERENCES

[1] European Parliament. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation).

[2] HTML. n.d. Web Storage. Retrieved September 15, 2023 from https://html.spec.whatwg.org/multipage/webstorage.html

[3] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A survey of dynamic analysis and test generation for JavaScript. *ACM Comput. Surv.* 50, 5 (2017), Article 66, 36 pages. https://doi.org/10.1145/3106739

[4] Adam Barth. n.d. HTTP State Management Mechanism. Retrieved September 15, 2023 from https://datatracker.ietf.org/doc/html/rfc6265

[5] Stefano Belloro and Alexios Mylonas. 2018. I know what you did last summer: New persistent tracking mechanisms in the wild. *IEEE Access* 6 (2018), 52779–52792. https://doi.org/10.1109/ACCESS.2018.2869251

[6] Dino Bollinger, Karel Kubicek, Carlos Cotrini, and David Basin. 2022. Automating cookie consent and GDPR violation detection. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*.

[7] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. 2017. Surviving the web: A journey into web session security. *ACM Comput. Surv.* 50, 1 (2017), Article 13, 34 pages. https://doi.org/10.1145/3038923

[8] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2018. Dr Cookie and Mr Token—Web session implementations and how to live with them. In *Proceedings of the Second Italian Conference on Cyber Security*, Elena Ferrari, Marco Baldi, and Roberto Baldoni (Eds.). CEUR Workshop Proceedings, Vol. 2058. CEUR, 1–10. http://ceur-ws.org/Vol-2058/paper-02.pdf

[9] Quan Chen, Panagiotis Ilia, Michalis Polychronakis, and Alexandros Kapravelos. 2021. Cookie swap party: Abusing first-party cookies for web tracking. In *Proceedings of The Web Conference 2021 (WWW '21)*. ACM, New York, NY, 2117–2129. https://doi.org/10.1145/3442381.3449837

[10] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, 1687–1700. https://doi.org/10.1145/3243734.3243823

[11] Martin Degeling, Christine Utz, Christopher Lentzsch, Henry Hosseini, Florian Schaub, and Thorsten Holz. 2019. We value your privacy . . . now take some cookies: Measuring the GDPR's impact on web privacy. In *Proceedings of the*

26th Annual Network and Distributed System Security Symposium (NDSS '19). https://www.ndss-symposium.org/ndss-paper/we-value-your-privacy-now-take-some-cookies-measuring-the-gdprs-impact-on-web-privacy/

[12] ECMA International. 2011. ECMAScript Language Specification. Retrieved September 15, 2023 from https://262.ecma-international.org/5.1/

[13] Steven Englehardt and Arvind Narayanan. 2016. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, 1388–1401. https://doi.org/10.1145/2976749.2978313

[14] Imane Fouad, Nataliia Bielova, Arnaud Legout, and Natasa Sarafijanovic-Djukic. 2020. Missed by filter lists: Detecting unknown third-party trackers with invisible pixels. *Proc. Priv. Enhancing Technol.* 2020, 2 (2020), 499–518. https://doi.org/10.2478/popets-2020-0038

[15] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the Symposium on Applied Computing (SAC '14)*. ACM, New York, NY, 1663–1671. https://doi.org/10.1145/2554850.2554909

[16] Henry Hosseini, Martin Degeling, Christine Utz, and Thomas Hupperich. 2021. Unifying privacy policy detection. *Proc. Priv. Enhancing Technol.* 2021, 4 (2021), 480–499. https://doi.org/10.2478/popets-2021-0081

[17] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. 2017. The ad wars: Retrospective measurement and analysis of anti-adblock filter lists. In *Proceedings of the 2017 Internet Measurement Conference (IMC '17)*. ACM, New York, NY, 171–183. https://doi.org/10.1145/3131365.3131387

[18] Jordan Jueckstock, Peter Snyder, Shaown Sarker, Alexandros Kapravelos, and Benjamin Livshits. 2020. There's no trick, its just a simple trick: A web-compat and privacy improving approach to third-party web storage. *CoRR abs/2011.01267* (2020). https://arxiv.org/abs/2011.01267

[19] Rezwana Karim, Frank Tip, Alena Sochurková, and Koushik Sen. 2020. Platform-independent dynamic taint analysis for JavaScript. *IEEE Trans. Softw. Eng.* 46, 12 (2020), 1364–1379. https://doi.org/10.1109/TSE.2018.2878020

[20] Panagiotis Papadopoulos, Nicolas Kourtellis, and Evangelos P. Markatos. 2019. Cookie synchronization: Everything you always wanted to know but were afraid to ask. In *Proceedings of The World Wide Web Conference (WWW '19)*. ACM, New York, NY, 1432–1442. https://doi.org/10.1145/3308558.3313542

[21] European Parliament. 2002. Directive 2002/58/EC of the European Parliament and of the Council of 12 July 2002 concerning the processing of personal data and the protection of privacy in the electronic communications sector, Official Journal of the European Communities.

[22] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. 2019. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS '19)*. https://www.ndss-symposium.org/ndss-paper/tranco-a-research-oriented-top-sites-ranking-hardened-against-manipulation/

[23] Audrey Randall, Peter Snyder, Alisha Ukani, Alex C. Snoeren, Geoffrey M. Voelker, Stefan Savage, and Aaron Schulman. 2022. Measuring UID smuggling in the wild. In *Proceedings of the 22nd ACM Internet Measurement Conference (IMC '22)*. ACM, New York, NY, 230–243. https://doi.org/10.1145/3517745.3561415

[24] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. 2012. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. 155–168. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/roesner

[25] D. Rücker and T. Kugler. 2017. *New European General Data Protection Regulation*, C. H. Beck, Hart, Nomos.

[26] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[27] Iskander Sánchez-Rola, Matteo Dell'Amico, Davide Balzarotti, Pierre-Antoine Vervier, and Leyla Bilge. 2022. Journey to the center of the cookie ecosystem: Unraveling actors' roles and relationships. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP '22)*. IEEE, Los Alamitos, CA, 1990–2004. https://doi.org/10.1109/SP40001.2021.9796062

[28] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. 2016. Explicit secrecy: A policy for taint tracking. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P '16)*. IEEE, Los Alamitos, CA, 15–30. https://doi.org/10.1109/EuroSP.2016.14

[29] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '13)*. ACM, New York, NY, 488–498. https://doi.org/10.1145/2491411.2491447

[30] Alexander Sjösten, Daniel Hedin, and Andrei Sabelfeld. 2021. EssentialFP: Exposing the essence of browser fingerprinting. In *Proceedings of the IEEE European Symposium on Security and Privacy Workshops, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, Los Alamitos, CA, 32–48. https://doi.org/10.1109/EuroSPW54576.2021.00011

[31] Mukund Srinath, Shomir Wilson, and C. Lee Giles. 2021. Privacy at scale: Introducing the PrivaSeer corpus of web privacy policies. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (ACL/IJCNLP '21): Volume 1: Long Papers*. 6829–6839. https://doi.org/10.18653/v1/2021.acl-long.532

[32] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. 2019. An empirical study of information flows in real-world JavaScript. *CoRR abs/1906.11507* (2019). http://arxiv.org/abs/1906.11507

[33] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS '19)*. https://www.ndss-symposium.org/ndss-paper/dont-trust-the-locals-investigating-the-prevalence-of-persistent-client-side-cross-site-scripting-in-the-wild/

Received 28 September 2022; revised 22 May 2023; accepted 19 August 2023