Understanding the Use of Web Storage in Real-World Web Applications

Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics

Computer Science Master's Thesis
Academic Year 2020-2021

Graduand Samuele Casarin
Supervisor Prof. Stefano Calzavara
Co-Supervisor Dott. Pietro Ferrara

Acknowledgments

I would like to thank my supervisor, prof. Stefano Calzavara, for the great scientific and moral support he gave me to complete this project.
I thank Pietro Ferrara, Ph.D., for giving me a direction for the formalization of the taint tracking approach.
I thank Zubair Ahmad, Ph.D. student, project partner, with whom I shared many moments of study and friendship, for helping me in the search for related work.
And thanks to my parents, who have always supported me during my studies and for the gift of life.

Abstract

Formerly born as a simple system for the exchange of public documents, over the time the Web has become one of the main services of the Internet, and it is still evolving into an increasingly sophisticated platform. As the complexity of this structure grows, more and more attention is required to ensure that web applications meet their security and privacy requirements. The advent of HTML5 brought many changes to the client-side environment, one of which is the introduction of Web Storage, a feature that allows web applications to store data in the user's browser. In this thesis we perform, to our knowledge, the first empirical analysis of the use of web storage in the wild. We leverage dynamic taint tracking at the level of JavaScript to collect explicit flows of information involving web storage in the Tranco Top 5k sites. Afterwards, we perform an automated classification of the detected information flows to shed light on the key characteristics of web storage. Our analysis shows that web storage is routinely accessed by third parties, including known web trackers, who are particularly eager to have both read and write access to persistent web storage information. This motivates the need for further research on the security and privacy implications of web storage content.

Keywords Web Storage, Taint Analysis, JavaScript

Contents

1 Introduction **1**

2 Background **2**
2.1 Basics of the Web . . . 3
2.1.1 Cookies . . . 3
2.1.2 JavaScript . . . 4
2.1.3 Same Origin Policy . . . 4
2.2 Web Storage . . . 5
2.3 Information Flow Control . . . 7
2.3.1 Explicit flows . . . 7
2.3.2 Implicit flows . . . 8
2.4 Jalangi . . . 10

3 Dynamic Taint Tracking **12**
3.1 Overview . . . 13
3.2 Technical Details . . . 15
3.2.1 Core JavaScript . . . 15
3.2.2 Taint model . . . 17
3.2.3 Abstract machine . . . 18
3.2.4 Generating instructions . . . 18
3.3 Example . . . 25
3.3.1 Error handling . . . 27

4 Web Measurement **29**
4.1 Methodology . . . 30
4.1.1 Sources and sinks . . . 30
4.1.2 Web crawling . . . 32
4.1.3 Flow classification . . . 33
4.2 Measurement Results . . . 35

5 Related work **38**

6 Conclusion **40**

Chapter 1

Introduction

In recent years, the Web has seen an exponential growth of the number of applications that rely on JavaScript to handle a substantial part of their logic in the browser. In fact, JavaScript is easy to learn, is very well supported, and gives web users an improved experience than traditional server-side applications, which need to reload the page to refresh data. 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, suffers from significant shortcomings: cookies are limited in size, have an unconventional semantics and are inconvenient to access programmatically. The HTML5 standard thus introduced the Web Storage API, a client-side data storage mechanism which retains the intuitive flavour of cookies, while addressing their most relevant drawbacks.
Although the Web Storage API has been around for a few years now and is fully supported by all major web browsers, anecdotal evidence based on previous web measurements suggests that web storage is still far from the popularity of cookies. Remarkably, contrary to cookies, web storage also received only limited attention by the security and privacy community so far. This is concerning, because the web storage functionality is reminiscent of traditional first-party cookies, hence it can be employed to implement web authentication [6] or to track users across third parties [8], all uses that deserve careful scrutiny. In the present thesis, we take a first step to improve our understanding of the usage of web storage in the wild. In particular, we perform an *empirical analysis* of web storage information set by popular websites based on dynamic taint tracking and an automated classification of the collected information flows. Our analysis uncovers several uses of web storage in the wild, for which we discuss relevant security and privacy implications.

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

1. We implement a dynamic taint tracking engine for JavaScript inspired by Ichnaea [19] and we configure it to detect information flows involving the Web Storage API (Section 3).

2. We perform a large-scale measurement to collect information flows and shed light on the key characteristics of the use of web storage in the wild. Our analysis is based on an automated classification of the detected information flows along different axes (Section 4).

A research article out from this thesis has been accepted at MADWeb 2022 [1].

Chapter 2

Background

In this chapter, we provide a brief review of the technical ingredients required to understand the present thesis.
First, we recap the fundamentals of the Web: the HTTP(S) protocol, cookies, JavaScript, and, regarding security, the Same Origin Policy.
Then, we present web storage, the central element of our thesis. Specifically, we define its characteristics and and compare them with respect to cookies.
Afterwards, we exhibit the most important findings in the field of information-flow control, which we intensely leverage for our analysis approach.
Finally, we present an overview of Jalangi, the framework on which we build a tool for the dynamic analysis of JavaScript based on code instrumentation.

2.1 Basics of the Web

The Web is a huge network of resources of any kind, from simple documents to images and sounds, whose transmission over the Internet is possible by means of *Hyper-Text Transfer Protocol* (HTTP) [22], a reliable client-server protocol. The concept behind this protocol is very simple: (*i*) the client sends a request message to the server looking for a resource, then (*ii*) the server replies to the client with a response message, which includes the data corresponding to the desired resource. Note that it is always the client who initiates communication with the server and never the opposite.
Strictly speaking, HTTP is not a *secure* protocol: in fact, a network attacker, i.e. an attacker who has access to the victim's network, could easily intercept all data passing over the wire and also alter the message traffic between the server and the client. At the beginning of the Web, an attack of this kind did not cause critical consequences due to the platform's original purpose; but then, with the introduction of dynamic web pages and web applications, the need to protect user data has arisen. For this reason, another protocol called *HTTP Secure* (HTTPS) [24] has been made available. HTTPS has the same rules as HTTP but relies on the *Secure Socket Layer* (SSL) or *Transport-Level Security* (TLS) protocol for the encryption of the communication channel.
Typically, users navigate the Web through a *web browser*, or simply browser. Popular modern browsers (e.g., Google Chrome and Mozilla Firefox) are large software products composed of many components, among which the HTTP client, that is constantly updated to the most recent standard specifications.

2.1.1 Cookies

One significant property of HTTP(S) is that of being a *stateless* protocol, i.e., the server does not keep any information generated from previous requests and therefore each request is independent of the other. Once again, this characteristic of HTTP was not a limitation, until session management became essential to keep a user logged in, or to add items to a shopping cart on an e-commerce site, and so on.
To overcome this lack, HTTP supports *cookies* [4], i.e., small strings of data, created by the server, stored in the browser, and then sent back to the server on each subsequent request. Cookies are structured as key-value pairs and may specify some parameters that control their behavior from different perspectives, including *lifetime* (how long is the cookie available?) and *scope* (in what context is the cookie available?).
As for lifetime, we make distinction between *persistent cookies*, i.e., cookies whose expiration time is determined by the application and therefore survive the execution of the browser, and *session cookies*, i.e., cookies that expire when the browser (or tab) is closed. The expiration time is determined by the `Expires` and `Max-Age` parameters.
Instead, as far as scope is concerned, the browser determines whether to send a cookie to the server by comparing the domain: in particular, a cookie is sent together with a request only if the domain of the page corresponds to the domain of the cookie or to one of its subdomains. However, the `Domain`, `Path`, and `Secure` parameters can be set in order to further restrict the cookie scope to a given domain, path, and protocol (only HTTPS or also HTTP?), respectively.

2.1.2 JavaScript

Modern web applications have the ability to delegate part of their business logic to the browser; this is possible by means of JavaScript (JS), the *de facto* standard scripting language of the Web. It is a dialect of the general-purpose ECMAScript (ES) programming language [11], which extends it with browser-specific features, among which the Document Object Model (DOM) and the Browser Object Model (BOM) that are respectively collections of object-oriented APIs which enable to control the elements in the page and the browser on the fly.
JavaScript is an *interpreted*, *dynamically-typed*, *object-oriented* programming language, that uses an *event-driven* programming model to deal with the asynchronous nature of the Web: web applications can set callback functions to be invoked automatically when an event occurs in the page, e.g., when a button in the page is pressed.
At the time of this thesis, the most popular browsers support the 13th version of ECMAScript (ES2022). This version includes a lot of additional features including syntactic with respect to early releases of the language, but still remains backwards compatible with the latter.
More details about the syntax and semantics of JavaScript will be provided along the thesis.

2.1.3 Same Origin Policy

The Same Origin Policy (SOP) is the baseline defense mechanism of web browsers, which enforces a strict separation between content served by different *origins*, i.e., combinations of protocol, host and port. For example, scripts running in a page fetched from `https://www.foo.com` cannot access the DOM of a page fetched from `https://www.bar.com`. SOP mediates both read and write accesses, thus acting as the security cornerstone to grant confidentiality and integrity on the Web. However, when a page at `https://www.foo.com` includes a script from a different origin like `https://www.bar.com`, the script inherits the origin of `https://www.foo.com` and is executed with the corresponding privileges.
The scoping rules for cookies constitute a *relaxed variant* of the traditional SOP: the origin is defined in terms of the domain, while the protocol and port are not considered by default.
Cookies are also accessible via JavaScript by reading and writing the property `document.cookie`, provided that the `HTTPOnly` parameter is not explicitly set by the server. In this case, the traditional SOP dictates the access from JavaScript to those cookies: the browser forbids any access to `document.cookie` from a cross-origin site, even if it is within the same domain of the cookie. For example, `https://bar.foo.com` cannot access `document.cookie` of `https://baz.foo.com`, despite the domain of the cookie is `*.foo.com`.

2.2 Web Storage

Many modern web applications still take advantage of cookies for client-side data persistence. Although this standard mechanism is simple to use and works out of the box, it has several limitations:

- since cookies are included in many requests to the server, the frequent transmission of large cookies may have a negative impact on the performance of the website;
- cookies are limited in size (4 KB per cookie);
- there is no standard facility to parse the value of `document.cookie`, therefore cookies are inconvenient to access via JavaScript;
- even if the user uses only secure HTTPS connections, a cookie may have been set using an insecure connection, thus violating the integrity of cookie's data (for example, a network attacker could seize the moment the user is browsing using an insecure connection to inject a cookie with malicious code that will be executed in the user's browser) [6].

Introduced among the innovations of the HTML5 standard, the Web Storage API [33] is a key-value data storage framework, which is proposed as a solid alternative to cookies, while maintaining the same degree of simplicity.
The Web Storage API offers two different services, called *local storage* and *session storage* respectively. Both types of storage have the same characteristics, with the only difference being related to the expiration of the stored content, that recalls the one between persistent cookies and session cookies. While content in the local storage persists indefinitely, content in the session storage is purged when the browser (or tab) is closed.
Web storage is only accessible through JavaScript and its content is never transmitted to the server, thus leaving the web application full control over how stored data is used. Moreover, the access to the store is protected by traditional SOP. For example, a script executing in a page fetched from `https://foo.com` cannot access the storage of `https://bar.com`; however, a script fetched from `https://foo.com` running in a page with origin `https://bar.com` have complete control of the storage of the latter.
The Web Storage API provides a convenient set of methods with a clean semantics to handle the key-value data structure, of which the following are the main ones:

- `getItem`(*key*) returns the value of the item associated to the given *key*;
- `setItem`(*key*, *value*) assigns *value* to the item with the given *key* (it creates a new item if such item does not exist);
- `removeItem`(*key*) deletes the item with the given *key*.

In particular, the stored keys and values are strings; when setting values of other type, those are implicitly converted to strings.
Last but not least, web storage has a greater capacity than cookies (5 MB), which allows one to store larger amounts of data.
We show an example use of session storage below: local storage can be used just by replacing `sessionStorage` with `localStorage`.

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

The most popular browsers support both web storage and cookies, and the decision of web applications to employ the first, the other, or a combination of them is almost philosophical.
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.

2.3 Information Flow Control

Traditional methods of securing secrets in computer science, such as access control and encryption, check whether a given user or program has the least privileges to access that sensitive information. However, these methods do not provide any guarantee over how secrets are used by the programs that handle them.
Information Flow Control (IFC) is a technique to ensure that information transfers performed by a program, or more generally an information system, do not violate the security policy [13].
In a typical model, the attacker has access to the source code of a monitored program. Here, each value is annotated with a *security label*, that determines whether it is safe to release the information to the attacker. In particular, labels are partially ordered in a *lattice* [10], meaning that any subset of them *joins* with a unique *least upper bound* and *meets* with a unique *greatest lower bound*.
In what follows, let H (for "High") be the label representing a secret information, whose leak implies a confidentiality violation, and L (for "Low") be instead the label associated to a public information, that does not compromise security.
The goal of information flow control is ensuring that data read from a secret (H) *source* does not affect data released through a public (L) *sink*. Generally, data with a given security label cannot be disclosed through a sink with a lower security label. The security policy defines the selection of undesired source-to-sink flows.
While we consider only confidentiality, the above argument can be extended in order to protect integrity of information. In this case, data coming from an *untrusted* (H) source cannot be written to a *trusted* (L) sink.
Various security policies and monitoring strategies for information flow control have been proposed in literature [25]. We exhibit some of the most important results.

2.3.1 Explicit flows

Let `source` be a source function that returns an H-labeled number, and `sink` be an L-labeled sink function that outputs the value passed as an argument, disclosing it.
Consider the following JavaScript program:

```
1 var secret = source();
2 var disguised = secret * 5;
3 sink(disguised); // Security violation!
```

After executing the program, it is enough for an attacker to divide the resulting value by 5 to read the secret information.
This kind of confidentiality violation occurs because an information generated by a secret (H) source is explicitly transferred to a public (L) sink by means of data dependencies, i.e., evaluation of expressions, reading and writing of variables, and more; for this reason, these information transfers are called *explicit flows*.
The most practical monitoring strategy that captures explicit flows is *taint tracking*. The concept behind this technique is that security labels represent the *taintedness* of values, which propagates through the execution of operations. In particular, a secret (H) value is said to be *tainted*, while a public (L) value is also called *untainted*; furthermore, a variable or object property is said to be tainted

if the value it stores is tainted. The thumb rule for *taint propagation* is that, if a value is tainted, then the result of an operation that depends on it is tainted as well. Concretely, the security label of a value is the *least upper bound* between the security labels of all the values that influence it.
Taint tracking enforces a security property for programs called *explicit secrecy* [27]: the execution of a program which types this condition cannot leak any secret (H, tainted) information through explicit flows.
Due to its practicality, taint tracking finds great popularity in many real-world applications, including security analysis [28, 2]. Some programming languages, like Perl[1], include a built-in taint checking mechanism, while for Ruby[2] it has recently been removed.
Let us go back to the previous program and discuss how taint tracking ensures confidentiality. At line 1, the `secret` variable is tainted. Then, at line 2, the value of `disguised` depends on the tainted value of `secret`, thus the taint of the latter propagates to the former. Finally, when the tainted value reaches the sink at line 3, a security error is raised and the secret is not disclosed to the attacker. The transfer of information from `source()` to `sink()` is an explicit flow that violates the security policy.
The above is an example of application in which taint tracking is effective in order to avoid a confidentiality violation. However, there exist some programs that does not hold explicit secrecy, and thus are stopped, but are safe in practice, like the following example:

```
1 var secret = source();
2 var disguised = secret - secret;
3 sink(disguised); // Security violation!
```

This program does not pose a threat, because the value of `disguised` at line 2 is always equal to zero, but a simple taint tracking monitor still reports the danger due to the data dependency of `disguised` on `secret`.

2.3.2 Implicit flows

In general, tracking explicit flows is not sufficient to guarantee language-based confidentiality. As an example, look at the following program:

```
1 var secret = source();
2 var disguised = 0;
3 if (secret < 50) {
4     disguised = 1;
5 }
6 sink(disguised);
```

Obviously, the above program does not allow the attacker to directly disclose the secret, since this time there is no explicit data dependency between `secret` and `disguised`. Despite this, the execution of this program provides a valuable information about the value of `secret`: if the output is 1, then `secret` is lesser than 50, otherwise it is greater or equal to 50. By smartly changing the number to compare in the conditional at line 3, it is therefore possible to exfiltrate the whole

[1] `https://perldoc.perl.org/perlsec`
[2] `https://www.ruby-lang.org/it/`

value of `secret` in a number of executions that is logarithmic with respect to the size of the secret.
The idea behind this kind of confidentiality attack is to manipulate the control flow using secret (H) information, while setting public (L) variables that let the attacker deduce some details of the secret from the execution path. These information flows are known as *implicit flows*.
There exist many policy models which track implicit flows. Observable secrecy [3] ensures that no information from a secret source reaches a public sink through both explicit and implicit flows; nonetheless, an attacker is still able to leak the secret among different executions of the program, distinguishing the runs where something is returned from those where a security error is raised. Non-interference [14] prevents this kind of information leak by interrupting the execution whenever a public (L) variable is updated in a secret (H) context; anyway, it is usually too strict to be applied in practice. Hence, some models introduce "escape hatches" that allow the *declassification* of secret information to public one [26]. In particular, the security policy establishes what, who, where, and when information can be declassified in order to not violate confidentiality.
Implicit flows are just an instance of *covert channels* [20], i.e., information channels whose main purpose is not information transfer. Attackers can use these channels to deduce secret information by observing certain system behaviors that depend on it - e.g., the termination or non-termination of the program, its execution time, and more.

2.4 Jalangi

While living in its heyday for the undisputed popularity, the JavaScript language is infamous for many of its weird features, which often lead to unexpected issues. This further justifies the need to develop techniques and tools to detect bugs and vulnerabilities in web applications.
Static analysis of JavaScript is demanding, due to the large amount of dynamic features of the language. Noteworthy examples of tools for this kind of analysis are TAJS [18] and ACTARUS [15].
As for dynamic analysis, a popular approach for many tools is to modify a browser. While this strategy has the advantage of being very efficient in terms of performance, it is hard to keep up with the fast browser development. As if that were not enough, JavaScript is now also used in other contexts outside of browsers, such as servers (e.g., Node.js[3]), mobile applications (e.g., Apache Cordova[4] apps), and desktop applications (e.g., Electron[5] apps), which limit the portability of the approach.
Conversely, other solutions have been designed with browser independence in mind. One of these is Jalangi [29], a dynamic analysis framework for JavaScript.
The first version of Jalangi is still available[6], but no longer maintained. It supports two basic mechanisms: selective record-replay and shadow values. The former allows one to *record* the execution of a part of the program and *replay* it at a later time; the latter is a boxing facility for storing information on any kind of value.
Jalangi2[7] is the current version of the Jalangi framework. It removes both selective record-replay and shadow values, while maintaining all the main features to perform a generic analysis of JavaScript. Since we are using this version for our project, in this thesis we specifically mean Jalangi2 when we refer to Jalangi.
Jalangi operates via a source-to-source transformation, in which all the main instructions of the source code are wrapped within specific callbacks: a process known as *code instrumentation*. The so-called instrumented code preserves the semantics of JavaScript, while allowing analysis developers to customize the callbacks to passively track different information at runtime.
Let us make an example. Consider the JavaScript statement `y = x + 2`, that applies the binary "+" operator to the value for the `x` variable and the number `2` and then assigns the result to the `y` variable. Note that we cannot say whether the "+" operator computes the sum of two numbers or the concatenation of two strings, because its semantics depends on the type of both the operands. A simplified instrumentation of that code could be the following:

```
y = Write("y", Binary("+", Read("x", x), Literal(2))).
```

Here, executing the various operations with a regular JavaScript engine, first the `Read` callback is invoked, passing the name and the value for the `x` variable, followed by the call to the `Literal` callback for the literal number `2`. Then, the above values are passed as arguments to the invocation of the `Binary` callback, that applies the "+" operator on them. Finally, the `Write` callback is invoked when assigning the result of the operation to the `y` variable.
In the real instrumented code, each callback is invoked with an additional hard-

[3] https://nodejs.org/
[4] https://cordova.apache.org/
[5] https://www.electronjs.org/
[6] https://github.com/SRA-SiliconValley/jalangi
[7] https://github.com/Samsung/jalangi2

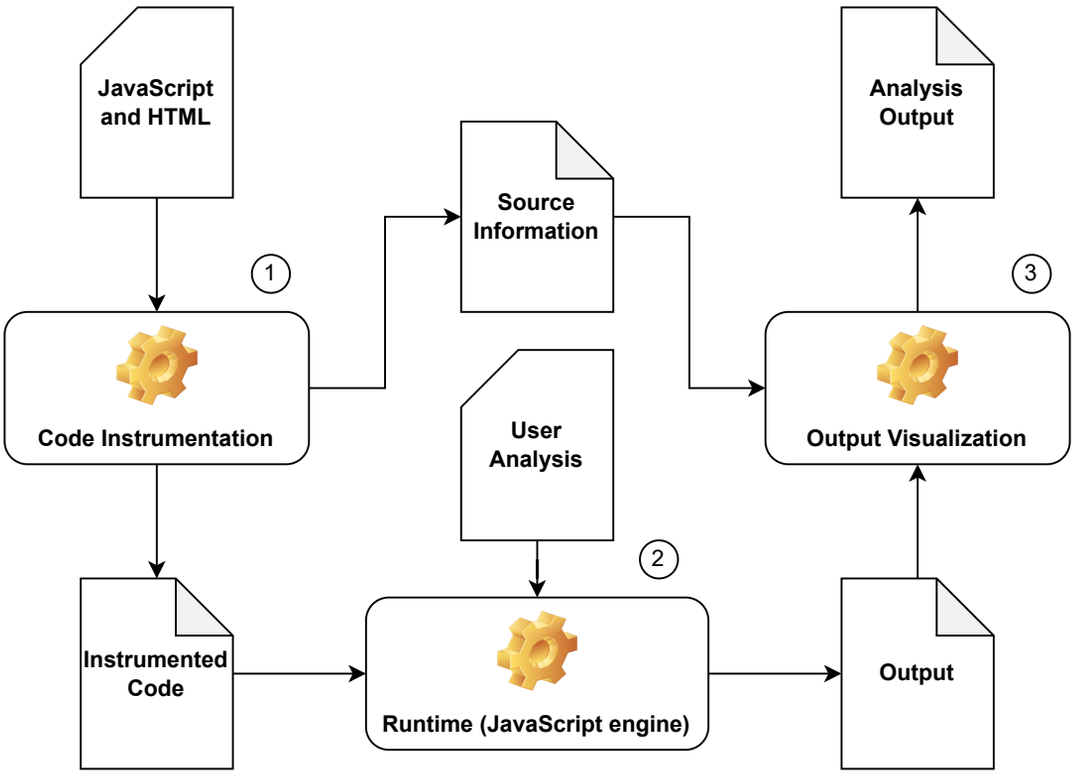

Figure 2.1: Overview of the Jalangi2 process

coded number, called Instruction ID (IID), which uniquely identifies an instruction within the script: in particular, Jalangi associates such identifiers to the location of the corresponding operations in the original source code.
In addition, Jalangi treats in a special way the `eval()` function invocation and the `new Function()` constructor, which evaluate a string as JavaScript code: in these cases, such code is instrumented during the execution.
Another key feature of Jalangi is *shadow memory*. This mechanism maintains shadow objects associated to runtime *objects* and *activation frames* on which it is possible to store and retrieve context-dependent information. In particular, the access to shadow objects respects the JavaScript rules of prototype-based inheritance for runtime objects and lexical scoping for activation frames.
An overview of the Jalangi2 process is depicted in Figure 2.1. First of all, Jalangi produces an instrumented version of HTML and JavaScript files and a source information database, which contains the mapping between IIDs and locations ①. Then, the instrumented code is executed with a regular JavaScript engine together with a user-defined analysis, that defines the various Jalangi2 callbacks ②. Finally, the runtime output and the source information can be combined in order to visualize the final analysis output ③.

Chapter 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. After reviewing the motivations and high-level ideas of the proposed solution, we discuss the key technical details of our implementation. Our approach is largely inspired by that of Ichnaea [19], a state-of-the-art taint tracking tool based on Jalangi [29], but with some changes in cases of inability to observe the running operations. Finally, we provide an example of how our engine works on a simple JavaScript program.

3.1 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.
This fundamental aspect of web storage led us to the decision of leveraging an information flow control strategy in order to answer our questions. In particular, we are interested in detecting information flows involving the Web Storage API, i.e., data flows that (*i*) start by reading from or (*ii*) end by writing into web storage. These flows are interesting from a security and privacy perspective, because flows of the former type may violate the confidentiality of Web Storage content, carrying out a data leak, while flows of the latter type may breach its integrity, with the effect of tampering the stored data.
Among the various strategies of information flow control proposed in literature, we leverage taint tracking for our study. As well as for practicality, this selection is mainly supported by a recent research conducted by Staicu et al. [31], in which they conclude that the majority of security-relevant information flows in real-world JavaScript is composed by explicit flows; thus, taint tracking is sufficient to study most of the security scenarios in the wild.
Furthermore, the detection is performed at runtime, because JavaScript is a challenging language for static analysis. In particular, we target a large-scale measurement in this paper, hence we prefer a dynamic analysis which 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 [25]. In particular, the implementation of our engine follows the approach proposed in Ichnaea [19], a state-of-the-art taint tracking tool for JavaScript. The main reason behind the choice of this tool lies in the fact that is *platform-independent*, as a natural consequence of being based on code instrumentation: in fact, the instrumented code allows one to track the taint without the need to use a modified version of the browser [7] or JavaScript engine [30].
The high-level idea behind Ichnaea is to use the instrumented code to generate instructions for an abstract machine while executing the original code, so that the state of the abstract machine reflects the taintedness of the concrete execution state. Note that, since the abstract machine isolates abstract values from the concrete environment, this approach has also the advantage of keeping the semantics of JavaScript unchanged. For example, this is not true for techniques that rely on boxing to track the taintedness of primitive values [9].
Our engine, as well as Ichnaea, leverages the Jalangi framework [29] for JavaScript instrumentation, which inserts callbacks for each of the main operations performed by the JavaScript interpreter, aware of all the dynamic features of the language. Furthermore, it makes use of shadow memory to associate activation frames and objects with unique identifiers, so as to properly access variables and properties in a given context, respecting the rules of prototype-based inheritance and lexical scoping.
Figure 3.1 gives an informal idea of the approach. First of all, Jalangi instruments the original JavaScript source code and appends the engine code and a *taint specification*, that defines the list of sources and sinks of the analysis, to the

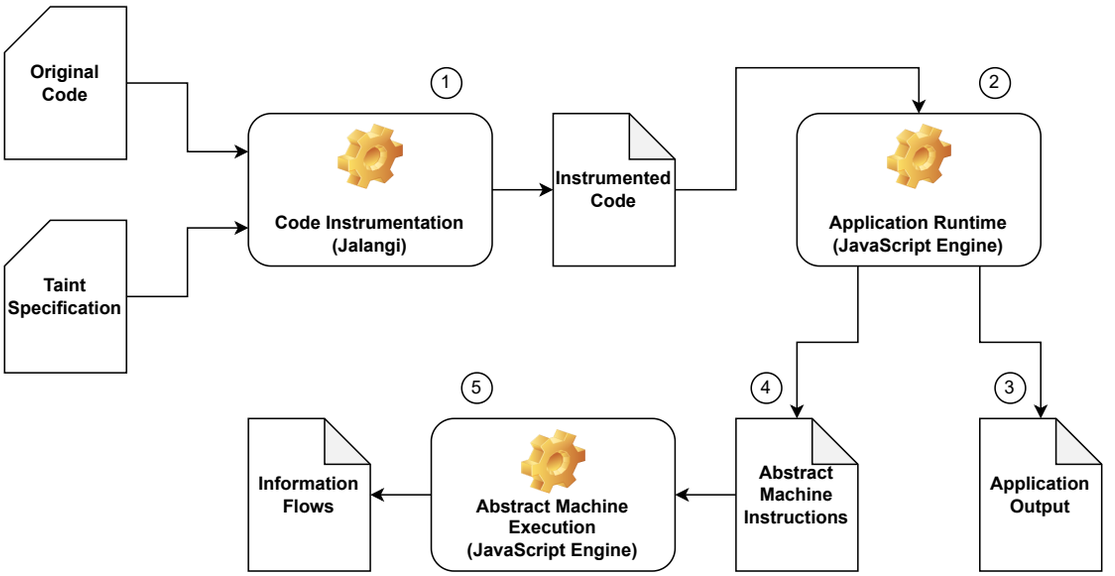

Figure 3.1: Overview of the approach

resulting instrumented code ①. The execution of the instrumented code with an unmodified JavaScript engine ② keeps the same behaviour as the original code ③, while emitting instructions for an abstract machine ④. Finally, the execution of such instructions with a JavaScript engine generates a list of information flows ⑤, according to the given taint specification.

3.2 Technical Details

As anticipated, Ichnaea employs code instrumentation to emit instructions for an abstract machine, whose state expresses the taintedness of concrete values.
In this section, we provide the formal specification of Ichnaea's abstract machine, which is also the core of our taint tracking engine. Afterwards, we exhibit a variant set of rules for generating abstract machine instructions, in order to solve problems of the original rules in some particular cases. We refer readers to [19] for more details about the original design of Ichnaea.

3.2.1 Core JavaScript

Figure 3.2 depicts the grammar for a core subset of non-strict ECMAScript 5.1 [11], which we consider in the formal description of the taint tracking engine.
The proposed subset of JavaScript is subject to some simplifying assumptions and excludes most of the control-flow constructs, syntactic sugars, and singular features of the full, standard language.
First of all, the language supports all the literal primitive values, including numbers, strings, booleans, `undefined` and `null`.
Objects are dynamic dictionaries of key-value pairs, called *properties*. Generally, there exist two types of properties: *data properties* and *accessor properties*. A property of the former type is defined with a *value*, while a property of the latter type specifies at least one of two functions, called *getter* and *setter*, that are invoked automatically by the JavaScript engine when the property is accessed for reading and writing respectively. In the core subset, objects can be constructed through object literals by specifying the list of initial properties; moreover, we initially consider only data properties.
Functions are special objects of the language, and consist of a list of *formal arguments*, and a body of JavaScript code. Differently from the full language, in which functions can be defined with a name, we assume that all functions are anonymous.
We support all ECMAScript 5.1 operators, but with some limitations: we accept unary and binary operations involving objects, provided that their evaluation has no side effects and does not require the implicit conversion to primitive values (e.g., the evaluation of `"obj:"+{}` requires to convert the object into a string).
Variables can be declared with an initial value, read and written. Object properties can be accessed for reading and writing as well, but only using the more generic *bracket notation* (*Expr* `[` *Expr* `]`), in contrast to the full language that also support the *dot notation* (*Expr* `.` *Id*). Furthermore, variable and property writes are simple statements, while in the standard JavaScript they are side-effectful expressions.
The full JavaScript provides three different semantics of invocation: (*i*) function call, (*ii*) method call, and (*iii*) constructor invocation. For simplicity, let us just consider normal function calls to be the most generic form of invocation of the three. In addition, we distinguish between function and procedure calls: in the first case the evaluation produces a value that can be used in an expression, while in the other case any potential returned value is discarded. Finally, for ease of exposition, 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 standard language in which functions can be invoked with a number of actual arguments lesser or greater than the number

Id	::=	...	(identifier)
Num	::=	`NaN` | `Infinity` | ...	(numeric literal)
Str	::=	`""` | ...	(string literal)
Bool	::=	`true` | `false`	(boolean literal)
Prim	::=	*Num* | *Str* | *Bool* | `undefined` | `null`	(primitive literal)
Obj	::=	{ *Id* `:` *Expr* `,` $\cdots$ `,` *Id* `:` *Expr* }	(object literal)
Fun	::=	`function (` *Id* `,` $\cdots$ `,` *Id* `) {` *Stmt* }	(function literal)
uop	::=	`!` | `typeof` | ...	(unary operator)
bop	::=	`+` | `*` | ...	(binary operator)
Expr	::=	*Prim*	(primitive literal expression)
	|	*Obj*	(object literal expression)
	|	*Fun*	(function expression)
	|	*uop Expr*	(unary operation)
	|	*Expr bop Expr*	(binary operation)
	|	*Id*	(variable read)
	|	*Expr* `[` *Expr* `]`	(property read)
	|	*Expr* `(` *Expr* `,` $\cdots$ `,` *Expr* `)`	(function call)
Stmt	::=	`var` *Id* = *Expr*	(variable declaration)
	|	*Id* = *Expr*	(variable assignment)
	|	*Expr* `[` *Expr* `]` = *Expr*	(property write)
	|	*Expr* `(` *Expr* `,` $\cdots$ `,` *Expr* `)`	(procedure call)
	|	`return` *Expr*	(return statement)
	|	*Stmt* `;` *Stmt*	(sequence)

Figure 3.2: Syntax for a core subset of JavaScript

τ	::=	$\{\cdots\}$ | `false`	(taint, abstract value)
V	::=	Id	(variable name)
O	::=	Id	(object identifier)
P	::=	Id	(property name)
$Inst$	::=	`push`(τ)	(push constant value onto stack)
	|	`pop`	(pop value from stack)
	|	`unaryop`(uop)	(pop value, apply unary operator, push result)
	|	`binaryop`(bop)	(pop two values, apply binary operator, push result)
	|	`initvar`(V)	(pop value, initialize variable with it)
	|	`readvar`(V)	(push current value of variable)
	|	`writevar`(V)	(write value at top of stack into variable)
	|	`initproperty`(O, P)	(pop value, initialize object property with it)
	|	`readproperty`(O, P)	(push current value of object property)
	|	`writeproperty`(O, P)	(write value at top of stack into object property)
	|	`join`	(pop two values, join, push result)

Figure 3.3: Instruction set of the abstract machine

of formal arguments and an invocation may exit without executing a `return` statement.

3.2.2 Taint model

The simplest way to represent the taintedness of values is through a binary state: *tainted* or *not tainted*. However, this pair of abstract values does not provide any information about the operations that introduced the taint. This is a problem for our project, because we need to distinguish a value influenced by web storage from one influenced by another type of source.

In order to overcome this limitation, we represent operations as *labels*. A label is a triple (t, l, e) where $t \in Str$ is the *type* of label, which identifies a specific type of operation, $l \in Str$ represents the code *location* where the operation was executed, and $e \in Str^*$ is a sequence of *extra information* about that operation.

For example, (`"localStorage.getItem"`, `"https://foo.com/index.js:15:48"`, ⟨`"theme"`, `"dark"`⟩) is a label representing the call to the `getItem` method of `localStorage`, located at position `15:48` (row and column, respectively) of `https://foo.com/index.js`, with extra information about the key of the item being accessed - `"theme"` - and its value - `"dark"`.

At this point, we model the taints of values with sets of labels, which we arrange in a lattice by means of the inclusion relation $\subseteq$. As a consequence, a subset of these *abstract values* joins through the set union operator $\cup$ and meets through the set intersection operator $\cap$. The infimum value of the lattice, or bottom $\bot$, coincides with the empty set $\varnothing$ and annotates a non-tainted concrete value. Conversely, a non-empty set of labels annotates a concrete value which is tainted by those labels.

Labels represent both sources and sinks of the taint analysis. When a source generates a value, we put a new label representing that operation into the set which annotates such value; afterwards, when a tainted value, i.e., a value annotated with a non-empty set of labels τ, reaches a sink, we record the information flow as a pair (τ, s), where s is a label representing the sink.

3.2.3 Abstract machine

The abstract machine manipulates a stack of abstract values (τ) that reflect the taints of values in the runtime stack of the original JavaScript program, while also maintaining maps that associate taints with local variables and object properties. Within the abstract machine, the empty set of labels is represented with `false`. Moreover, variables (V), objects (O), and properties (P) are uniformly distinguished by unique identifiers (Id).
Figure 3.3 shows the list of abstract machine instructions. The stack-based architecture of the abstract machine implies the natural definition of two basic instructions: `push`(τ), that inserts the abstract value τ onto the stack, and `pop`, that discards the topmost value of the stack. The `unaryop`(uop) and `binaryop`(bop) instructions pop one or two values from the stack respectively, apply to these a specific function for the given operator and push the resulting value onto the stack. The `initvar`(v), `readvar`(v), and `writevar`(v) instructions handle the taint for the variable v using the map associated to the activation frame in which v has been defined. Similarly, the `initproperty`(o, p), `readproperty`(o, p), and `writeproperty`(o, p) instructions control the taint for the property p using the map associated to the object identified by o. In particular: `initvar`(v) and `initproperty`(o, p) pop a value from the stack and initialize a new entry in the corresponding map with the popped value; `readvar`(v) and `readproperty`(o, p) push the value currently bound to the variable v or property p onto the stack; finally, `writevar`(v) and `writeproperty`(o, p) stores the top value of the stack in the map entry for the variable v or property p, without extracting it from the stack.
In addition to the original instructions listed above, our abstract machine defines the `join` instruction, which extracts two abstract values from the top of the stack and then pushes their least upper bound, computed by applying the join operator on them. We need this instruction to model taint propagation for a generic operation, where the actual one is unknown - specifically, in the case of non-instrumented code.

3.2.4 Generating instructions

The general principle for taint tracking is the following: when evaluating an expression, the taint of the operands must propagate to the result. In practice, we assume that the last values pushed onto the stack represent the taint of the previously calculated sub-expressions. Hence, we generate instructions that involves a number of abstract values on top of the stack equal to the number of operands of the expression. Eventually, at the top of the stack there will be the value reflecting the taintedness of the whole operation.
Primitive, object, and function literals are constant values in the source code, which generally do not represent sensitive information; accordingly, a `push(false)` instruction is issued for the abstract machine, indicating that such values are not tainted. However, object literals define a number of initial properties, let us say n, whose values have been pushed onto the stack just before evaluating the literal expression; those n properties must be initialized in reverse order using the n topmost values of the stack, therefore we emit n `initproperty` instructions.
The taint propagation of unary and binary operations is performed by the ad-hoc `unaryop` and `binaryop` instructions, respectively.

Variables and object properties are both considered containers of data, so their taint is associated to the value they carry. A variable v is declared in the abstract machine as a result of generating an `initvar`(v) instruction. Afterwards, it can be accessed by issuing `readvar`(v) for reading and `writevar`(v) for writing. The v 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 p of an object o is possible by emitting a `readproperty`($oid(o)$, $offset(p)$) to get the assigned value and `writeproperty`($oid(o)$, $offset(p)$) to put another one. In this case, *oid* is a mapping from a JavaScript object to the corresponding unique identifier (O) in the abstract machine, while *offset* maps a JavaScript expression to a property identifier (P). It is also worth to notice that `writevar` and `writeproperty` do not extract from the stack the value they store, thus those instructions must be followed by a `pop`; moreover, `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.
The last basic operation that we discuss is user-defined function (procedure) calls. User-defined functions have the significant characteristic of being instrumented, which enables to accurately observe how the taint propagates through the various operations. We assume that, 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, in order to communicate the returned value to the caller - in the case of a procedure call, which does not involve returning a result, this last step is skipped.
Figure 3.4 exhibits the comprehensive list of abstract machine instructions for each type of operation in the core subset of JavaScript.
In the next steps, we will further develop the design of our taint tracking engine, in order to support most of the non-trivial features of the full ECMAScript 5.1 language. For many of these, our approach will differ from that employed in Ichnaea.

Generic function invocation

In addition to function *calls* in the strict sense, intended as the explicit application of a function by the user, there exist numerous situations in which a function is implicitly executed by the JavaScript engine. We mention some of these cases:

- **object-to-primitive conversions**: some operators and native functions require their parameters to accept primitive values, but if an object is used as a parameter where a primitive is expected (e.g., the `5+{}` expression), the JavaScript engine invokes the object's `valueOf` or `toString` method to convert it to a primitive;

- **getters and setters**: these functions are invoked automatically when an accessor property of an object is read and written;

Operation	Generated instructions	Justification
Expressions (*Expr*)		
$l \in Prim$	`push(false)`	Primitive literals are never tainted.
$o \in Obj$ $o \equiv \{p_1 : e_1, \cdots, p_n : e_n\}$	`initproperty`($oid(o), p_n$) $\cdots$ `initproperty`($oid(o), p_1$) `push(false)`	Initialize properties using the n topmost values of the stack. The object literal itself is not tainted.
$f \in Fun$	`push(false)`	Function literals are never tainted.
uop e	`unaryop`(*uop*)	Apply unary operator to the top value of the stack.
e_1 *bop* e_2	`binaryop`(*bop*)	Apply binary operator to the two topmost values of the stack.
$v \in Id$	`readvar`(v)	Push the value for the variable v.
`this`	`push(false)`	`this` always points to an object; objects are never tainted.
o`[`p`]`	`pop` ×2 `readproperty`($oid(o), offset(p)$)	Extract the property name and the object; then, push the value for the property p of o.
e`(`e_1`,`$\cdots$`,`e_n`)`	`pop` `readvar("_ret_")`	Discard the called function; then, read the special `_ret_` variable to load the value returned by the callee.
Statements (*Stmt*)		
`var` v `=` e	`initvar`(v)	Initialize v with the value on top of the stack.
v `=` e	`writevar`(v) `pop`	Assign the value on top of the stack to v; then, discard the assigned value.
o`[`p`]` `=` e	`writeproperty`($oid(o), offset(p)$) `pop` ×3	Assign the value on top of the stack to the property p of o; then, discard the assigned value, the property name, and the object.
e`(`e_1`,`$\cdots$`,`e_n`)`	`pop`	Discard the called function; do not load any value returned by the procedure.
f`(`v_1`,`$\cdots$`,`v_n`)`	`initvar`(v_n) $\cdots$ `initvar`(v_1)	Initialize the formal arguments using the n topmost values of the stack.
`return` e	`writevar("_ret_")` `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.

Figure 3.4: Rules for generating abstract machine instructions; we assume that instructions for gray-colored expressions have already been issued

- **asyncronous callbacks**: users can register callback functions that the JavaScript engine invokes whenever a certain event occurs on the page.

We identify both explicit and implicit function calls with the term "*invocation*".
Moreover, the JavaScript specification defines a significant amount of standard functions, which are most often built into the browser; as a consequence, the code for these functions is not available directly. This is problematic, because such code is not instrumented, therefore we cannot track the taint when it is executed. We refer to these as *native functions*.
Further complexity is given by *higher-order* native functions, i.e., that accept another function as an argument or return a function. It is a typical pattern in JavaScript applications to apply common native methods to user-defined callbacks, that let users define part of their behavior. During execution, the native function could invoke the callback multiple times and with different parameters, whose taint depends on its hidden operations. An example is the `Array.prototype.map` native function, that progressively applies the callback to all the elements of an array and then creates a new array with the corresponding results.
Ichnaea bridges the lack of information on data dependencies with the help of manually crafted models for specific native functions, which generate the necessary instructions for the abstract machine. However, this strategy is only effective for the supported functions, thus it cannot be easily scaled to the whole JavaScript standard library.
We design a different solution, based on standard concepts, that does not perform precise taint tracking like Ichnaea's models, but works for any invocation of both user-defined and native functions. This solution considers all the cases described above, especially the possible alternation between user-defined and native function invocations, and also does not override any of the previously defined rules.
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. In particular, during the execution of a native function, we hold the invariant that the topmost value of the stack represents the taint for the result of the invocation.
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 invocation of a native function by another native function, because this kind of operation is performed inside the black box of the latter and therefore is not observable. With this in mind, let us examine all the possible cases in which a function of one type invokes another function of the other type.
The simplest instance is the one in which **a user-defined function invokes another user-defined function**: in this case, no further abstract machine instruction is generated, in addition to those already issued.
In the remaining cases, we have to generate instructions to pass arguments before the execution of the function begins, and to receive the value returned by the callee upon the termination.
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 `false` 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 too. The recursion must take into account

the possibility of cyclic references between objects - in the simplest case, when an object refers to itself; thus, we apply recursion on an object *if it has not been visited yet* in the current traversal. At the end, the taint for the result is stored into the special `_ret_` variable if the returned value is primitive, otherwise we recursively propagate it to the properties of the returned object and replace the taint associated to the `_ret_` variable with `false`, thus preserving the assumption that objects are never tainted.
Finally, we discuss the instance in which **a native function invokes a user-defined function**. 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(false)` 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, that 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, that 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 formalized in Figure 3.5: we define a collection of *macros*, i.e., procedures that expand to instructions for the abstract machine whenever the execution enters and leaves a user-defined or native function, with respect to the above rules. The behavior of macros depends on an *abstract call stack*, which summarizes an activation frame in terms of the type of the invoked function that led to its creation: `"USER"` or `"NATIVE"`, standing for user-defined and native function, respectively. The primitives for the abstract call stack are Push-Frame, Pop-Frame, and Top-Frame, with the expected semantics.

Error handling

We now provide the core language with the ability to raise exceptions using the `throw` statement and handle them with the `try`, `catch`, and `finally` construct. When a function invocation throws an error, the JavaScript engine interrupts the execution of such function and passes the control to the first `catch` block in the call stack, or terminates the script execution whether it does not exist. In addition, it removes from the runtime stack all and only the values of the interrupted invocations.
Ichnaea communicates the taint of a thrown error via the special `_throw_` variable, in a way similar to returning the result of an invocation with the `return` statement. However, it does not keep information about which taints in the stack are associated to values of a given activation frame. As such, the insufficient or excessive removal of elements from the stack causes the abstract machine state to be misaligned from the concrete state.
While following the same approach for communicating the taint of the error, we evolve the strategy described in the previous argumentation to overcome the latter problem. In particular, we extend abstract activation frames with an additional information, that we call *frame pointer* - borrowing the name from the world of binary programs. In our sense of the term, the frame pointer of an abstract frame

```
Input: 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(false))
        end if
      end for
    end if
    PUSH-FRAME("USER")
  end procedure

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

Input: a_1, ··· , a_n Actual arguments
  procedure ENTER-NATIVE-FUNCTION(a_1, ··· , a_n)
    if n = 0 then
      EMIT(push(false))
    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

Input: 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(false))
      EMIT(writevar("_ret_"))
      EMIT(pop)
    end if
  end procedure
```

```
Input: o The object to get the taint from
  procedure OBJ-TAINT(o)
    EMIT(push(false))
    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

Input: 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
```

Figure 3.5: Macros for generic function invocations

indicates the height of the abstract value stack from which the taints of the values produced in that frame are pushed.
The frame pointer acts as a logical barrier between abstract values of different frames. When a user-defined is invoked, the frame pointer in the pushed abstract frame is equal to the taint stack height, minus the number of actual arguments; on the invocation of a native function, such number is equal to the taint stack height, minus the taint of the resulting value.
Whenever a function invocation exits due to an error, an ad-hoc macro emits a succession of `pop` instructions until the taint stack height and the frame pointer of the top abstract frame are equal, in order to discard all the values produced in that frame. The macro also takes action on the function that catches the error, because the control flow has changed. At the end, the abstract frame is popped from the abstract call stack.
In this way, the right number of abstract values is always removed from the stack in case of error, and the state of the abstract machine persists in being aligned with the concrete execution state.

Other JavaScript features

The JavaScript standard specification includes a lot of additional constructs and semantics that we have not covered in the formal discussion. We briefly explain how we handle some of them. Unless otherwise indicated, we follow the approach proposed in Ichnaea.
In JavaScript, arrays are a special type of objects, where indexes are properties; like Ichnaea, we issue the same instructions to access their elements as we generate for objects.
When an accessor property of an object is read or written, the JavaScript engine invokes the associated getter or setter function, respectively. In such a case, we emit instructions for a function invocation, rather than treating the operation as a normal property read/write access.
In function calls, one can pass a number of actual arguments different from the number of formal arguments. If fewer actual arguments than formal ones are specified, the JavaScript engine fills the missing formal arguments with `undefined`, and consequently we generate `push(false)`. Otherwise, if there are more actual arguments than expected, they can be accessed through an array-like object, called `arguments`. In particular, a key characteristic of such object is that the values of formal arguments always reflect the values of the corresponding elements of `arguments`. Thus, we store the taints for the properties of `arguments` and access them uniformly every time we access a formal argument.
Finally, there exist some JavaScript objects that interact directly with the JavaScript engine and exhibit a bundle of interface methods to control them. Among others, Document Object Model (DOM) objects represent and manipulate the HTML elements in the page, while XMLHttpRequest (XHR) objects enable to perform on-the-fly HTTP requests to the server. Differently from Ichnaea, which indiscriminately treats all objects as containers of data, we associate each of these objects to a single, field-insensitive taint and generate instructions for the abstract machine as if they were primitive values.

3.3 Example

In the last section of this chapter, we help the reader understand how our taint tracking engine works.
For our demonstration, we consider the following JavaScript program, which has no particular purpose, other than the expository one:

```
1 var s = secret();
2 var v = [31];
3 v[1] = v[0] * s;
4 var r = v.reduce(function (acc, cur) { return acc + cur; }, 0);
5 disclose(r);
```

Both of them are native functions. At line 1, the call to the native `secret` function returns a confidential number, which is assigned to the `s` variable. At line 2, a literal array with a single element `31` is stored into the `v` variable. At line 3, we write at index `1` of the array in `v` the multiplication between the element at index `0` of the same object and the value of `s`. At line 4, we invoke the native `reduce` function to the array in `v` and store the result in `r`. Finally, at line 5, we tell the world the value of `r` by calling the `disclose` function.
For each line of the toy script, we explain which instructions are generated for the abstract machine. Here, we assume that the `secret` function is an hypothetical source of the analysis, and the `disclose` function is a sink. Due to the JavaScript's behavior to move all declarations to the top of the function code, better known as *hoisting*, we assume that the instructions to initialize the local variables have been already issued.
Let us start with instructions for line 1. Since we call a native function, we expand the Enter-Native-Function and Leave-Native-Function macros on function entry and exit, respectively. The former just emits a `push(false)` instruction, because there are no arguments, while the latter generates the instructions to communicate such taint to the caller. Between the two macro expansions, the resulting taint on top of the stack, which was initially untainted (`false`), becomes tainted (for sake of simplicity, `true`), due to the fact that the invoked `secret` function is a source of the analysis. At the end, we emit instructions to read the resulting taint and store it into the `s` variable.

```
1  readvar("frame1:secret")        // push taint (false) for variable "secret"
2  // -- begin Enter-Native-Function macro expansion --
3  push(false)
4  // -- end Enter-Native-Function macro expansion --
5  // taint the top of the stack
6  // -- begin Leave-Native-Function macro expansion --
7  writevar("_ret_")               // store taint (true) for return value
8  pop                             // discard taint (true) of return value
9  // -- end Leave-Native-Function macro expansion --
10 pop                             // discard taint (false) of function
11 readvar("_ret_")                // push taint (true) for return value
12 writevar("frame1:s")            // store taint (true) for variable "s"
13 pop                             // discard taint (true) of assigned value
```

At line 2, we treat an array literal as an object literal, so we initialize the single `0` property of the object by generating an `initproperty` instruction, and then store the taint of the array itself in the `v` variable as before. We identify the array instance with `"obj4"` and represent it with `<obj4>`.

```
push(false)                      // push taint (false) for literal 31
initproperty("obj4", "0")        // init property "0" of <obj4>
push(false)                      // push taint (false) for array literal
writevar("frame1:v")             // store taint (false) for variable "v"
pop                              // discard taint (false) of assigned value
```

The instructions issued for line 3 reflect the taints for a property read, a variable read, a binary operation, and a property write statement. In particular, the `*` binary operation joins an untainted value with a tainted one, and therefore propagates the `true` taint to the result value.

```
readvar("frame1:v")              // push taint (false) for variable "v"
push(false)                      // push taint (false) for literal 1
readvar("frame1:v")              // push taint (false) for variable "v"
push(false)                      // push taint (false) for literal 0
pop                              // discard taint (false) of literal 0
pop                              // discard taint (false) of <obj4>
readproperty("obj4", "0")        // push taint (false) for prop. "0" of <obj4>
readvar("frame1:s")              // push taint (true) for variable "s"
binaryop("*")                    // apply "*" binary operator
writeproperty("obj4", "1")       // store taint (true) for prop. "1" of <obj4>
pop                              // discard taint (true) of assigned value
pop                              // discard taint (false) of literal 1
pop                              // discard taint (false) of <obj4>
```

At line 4, we focus on instructions generated for the call to the `reduce` function. `array.reduce` is an higher-order, native method of arrays, that is equivalent to *fold* in functional programming: given an array, a callback, and an initial value, it invokes the callback with the first element of the array and the initial value, then uses its result as the initial value in the callback applied to the next element, and so on; the returned value is the aggregate value of all elements in the array.
In the expansion of the ENTER-NATIVE-FUNCTION macro, the OBJ-TAINT macro is expanded to push onto the stack the over-approximated taint for the properties of the received array - that is, `<obj4>`. Since the taint for the second element of the array is `true`, the resulting taint is also `true`.
Then, `reduce` invokes the user-defined callback with the initial value and the first element of the array as actual arguments, that respectively initialize the `acc` and `cur` variables in the activation frame with identifier `"frame2"`. As an effect of expanding the ENTER-USER-FUNCTION macro, the taint for each of these arguments is equal to the native caller's resulting taint.
When the execution of the callback ends, the LEAVE-USER-FUNCTION macro expands to instructions that push the taint for the returned value onto the stack and join it with the resulting taint of the native function.
The user-defined callback is invoked another time by the native function in order to aggregate the second element of the array. The generated instructions are the same, but now the local variables belongs to another activation frame, that we identify with `"frame3"`.
At the termination of the call, we expand the LEAVE-NATIVE-FUNCTION macro, that makes available the resulting taint to the caller.

```
readvar("frame1:v")              // push taint (false) for variable "v"
push(false)                      // push taint (false) for literal "reduce"
pop                              // discard taint (false) of literal "reduce"
pop                              // discard taint (false) of <obj4>
readproperty("obj4", "reduce")   // push taint (false) for method "reduce"
push(false)                      // push taint (false) for function literal
push(false)                      // push taint (false) for literal 0
// -- begin Enter-Native-Function macro expansion --
join                             // join taints of arguments
// expand Obj-Taint(<obj4>)      // push taint (true) for props. of <obj4>
```

```
11 join                          // ... join with taints of arguments
12 // -- end Enter-Native-Function macro expansion --
13 // -- begin Enter-User-Function macro expansion --
14 writevar("_arg_")              // store taint (true) for arguments
15 readvar("_arg_")               // push taint (true) for first argument
16 readvar("_arg_")               // push taint (true) for second argument
17 // -- end Enter-User-Function macro expansion --
18 initvar("frame2:cur")          // init first formal argument
19 initvar("frame2:acc")          // init second formal argument
20 readvar("frame2:acc")          // push taint (true) for variable "acc"
21 readvar("frame2:cur")          // push taint (true) for variable "cur"
22 binaryop("+")                  // apply "+" binary operator
23 writevar("_ret_")              // store taint (true) for return value
24 pop                           // discard taint (true) for return value
25 // -- begin Leave-User-Function macro expansion --
26 readvar("_ret_")               // push taint (true) for return value
27 join                          // ... join with taint of resulting value
28 // -- end Leave-User-Function macro expansion --
29 // repeat instructions at lines 13-28 using variables in "frame3"
30 // -- begin Leave-Native-Function macro expansion --
31 writevar("_ret_")              // store taint (true) for return value
32 pop                           // discard taint (true) of return value
33 // -- end Leave-Native-Function macro expansion --
34 pop                           // discard taint (false) of function
35 readvar("_ret_")               // push taint (true) for return value
36 writevar("frame1:r")           // store taint (true) into variable "r"
37 pop                           // discard taint (true) of assigned value
```

Finally, at line 5, the tainted (`true`) value of the `r` variable, which is the result of the native function call, is passed as an argument to the `disclose` function, that is, a sink of the analysis. Hence, our engine logs the information flow.

```
1  readvar("frame1:disclose")     // push taint (false) for variable "disclose"
2  readvar("frame1:r")            // push taint (false) for variable "r"
3  // -- begin Enter-Native-Function macro expansion --
4  // -- end Enter-Native-Function macro expansion --
5  // log the information flow
6  // -- begin Leave-Native-Function macro expansion --
7  writevar("_ret_")              // store taint (true) for return value
8  pop                           // discard taint (true) of return value
9  // -- end Leave-Native-Function macro expansion --
10 pop                           // discard taint (false) of function
```

3.3.1 Error handling

We also propose a simple example of how our engine handles errors.
Consider the following JavaScript program:

```
1  function g() {
2      throw new Error();
3  }
4
5  function f() {
6      return 7 + g();
7  }
8
9  var result;
10 try {
11     result = 5 + f();
12 } catch (e) {
13     result = -1;
14 }
```

Starting from line 11, a `push(false)` instruction is generated for the abstract machine to indicate the taint for the literal `5`, followed by issuing the instructions for the call to the user-defined `f` function. The invocation causes the insertion of an abstract frame in the abstract call stack, whose frame pointer is set to 2, because the two elements further down the abstract stack are the taints associated to the literal `5` and the `f` function. Similarly, inside the `f` function, another abstract frame is pushed for the call to the `g` function, whose frame pointer is set to 4: in fact, additionally to the previous elements, the taints for the literal `7` and the `g` function have been pushed onto the stack, for a total of 4 elements.
The call to `g` raises an exception with the `throw` statement, which stores the taint for the `Error` object into the special `_throw_` variable. Since the execution exits `g` without leaving any pending expression, we just discard the abstract frame for the call to `g` out of the abstract call stack. Instead, when exiting `f`, we generate two `pop` instructions in order to bring the stack height (currently, 4) back to the topmost frame pointer, i.e., 2: the two discarded elements were the taints for the literal `7` and the `g` function. Then, we discard the abstract frame of the call to `f`. Finally, the expression which invoked `f` is wrapped within a `try..catch` block, thus the taints for `5` and `f` are discarded in a similar way as before, and the taint for the `_throw_` variable is written to the `e` variable.
Below, we show the generated instructions for the abstract machine. Note that we comment the stack height (`H =` n) and the occurrences in which the extended abstract call stack is pushed (`Push-Frame`) and popped (`Pop-Frame`). For ease of exposition, we also simulate the construction of the `Error` object with the `push(false)` instruction.

```
1  push(false)                    // initial value for variable "result" (H = 1)
2  initvar("frame1:result")       // init variable "result" (H = 0)
3  push(false)                    // push taint for literal 5 (H = 1)
4  readvar("frame1:f")            // push taint for variable "f" (H = 2)
5  // -- begin Enter-User-Function macro expansion --
6  // Push-Frame("USER", 2)       // push frame with frame pointer = 2 for call to f
7  // -- end Enter-User-Function macro expansion --
8  push(false)                    // push taint for literal 7 (H = 3)
9  readvar("frame2:g")            // push taint for variable "g" (H = 4)
10 // -- begin Enter-User-Function macro expansion --
11 // Push-Frame("USER", 4)       // push frame with frame pointer = 4 for call to g
12 // -- end Enter-User-Function macro expansion --
13 push(false)                    // simulate taint for Error object (H = 5)
14 writevar("_throw_")            // store taint for raised error
15 pop                            // pop taint for raised error (H = 4)
16 // -- begin exceptional Leave-User-Function macro expansion --
17 // Pop-Frame()                 // discard frame of call to g
18 // -- end exceptional Leave-User-Function macro expansion --
19 // -- begin exceptional Leave-User-Function macro expansion --
20 pop                            // discard taint of function literal (H = 3)
21 pop                            // discard taint of literal 7 (H = 2)
22 // Pop-Frame()                 // discard frame of call to f
23 // -- end exceptional Leave-User-Function macro expansion --
24 // -- begin "catch" statement macro expansion --
25 pop                            // discard taint of function literal (H = 1)
26 pop                            // discard taint of literal 5 (H = 0)
27 // -- end "catch" statement macro expansion --
28 readvar("_throw_")              // push taint for raised error (H = 1)
29 writevar("frame1:e")            // store taint into catch variable "e"
30 pop                            // discard taint of assigned value (H = 0)
31 push(false)                    // push taint for literal -1 (H = 1)
32 writevar("frame1:result")       // store taint into "result" variable
33 pop                            // discard taint of assigned value (H = 0)
```

Chapter 4

Web Measurement

We now explain how we performed our large-scale measurement in the wild and we report on the most relevant findings of our study.
First, we show how we setup a web crawler to collect information flows involving the Web Storage API.
Then, we analyze them along several axes to extract knowledge about the security and privacy of using web storage in real-world web applications.

Table 4.1: List of sources and sinks used for taint tracking

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

4.1 Methodology

We exhibit the list of sources and sinks wherewith we configure our taint tracking engine, which we use then with a web crawler to perform our analysis of the Web. Afterwards, we present a strategy for the automated classification of the tracked information flows.

4.1.1 Sources and sinks

In section 3.2.3, we define abstract values as sets of labels - combinations of type, location, and extra information - which annotate runtime values with information about data dependencies and, more generally, describe information flows.
Concretely, we represent a label with a JSON object which has the following fields:

- **type**: a string identifier for the kind of operation;
- **scriptUrl**: the URL of the script where the operation is located;
- **iid**: a number, assigned by Jalangi, that identifies the operation within a script (Instruction ID) - in pair with **scriptUrl**, they universally identify the operation;
- **extra**: an array of strings that provide further information about the operation, whose nature depends on the **type** field.

For the purposes of this project, we are interested in identifying information flows involving the Web Storage API. More formally, an information flow involves the Web Storage API if and only if: (*i*) it starts from a call to the `getItem` method and ends into a sink, or (*ii*) it starts from a source and ends into a call to the `setItem` method. We refer to the former as *confidentiality flows* and to the latter as *integrity flows*.
Table 4.1 reports the different sources and sinks considered in our analysis, largely inspired by previous web measurements based on information flow control [7, 30]. Note that the web storage was largely ignored as source or sink in previous work, to the best of our knowledge.
Below, we present five classes of sources and sinks, based on their functionality, and discuss the additional information for the labels of each class.

Web storage Web storage is the pivot of our analysis. This class comprises the call to the `getItem` method as a source, and the call to the `setItem` method as a sink. We also consider being able to access the content of web storage using the bracket notation: we model this case as a call to the corresponding method. Other than the specific instance of web storage (`localStorage` or `sessionStorage`), labels related to web storage maintain the accessed key and the stored value.

Network We monitor the most important client-side features that involve a network interaction, in order to investigate on the remote origin and destination of web storage data. The standard object that allows one to perform on-the-fly HTTP requests is `XMLHttpRequest` (XHR); we mainly focus on its methods and properties, which we consider both as sources and sinks. Additionally, we observe some less obvious ways to send network requests, that we treat as sinks of the analysis: the call to the `navigator.sendBeacon` function, which is meant to be used for sending analytics data, and setting the `src` attribute of HTML elements, that causes the browser to contact a server for retrieving a remote resource (e.g., `img` elements for images). Network-related labels include the so-called *network URL*, i.e., the remote URL from which data came or to which data was sent.

Cookies Despite web storage and cookies have the same underlying purpose, they are semantically different from each other. In particular, cookies can be set by the server and are transmitted together with requests to the server, while web storage is more convenient to access from JavaScript. Web applications may use these two technologies at the same time, in order to combine the advantages of both. For this reason, we identify reads and writes to the `document.cookie` property as sources and sinks, respectively. We also keep the value of such property as an extra information in this class's labels.

Current URL Other than cookies, there are some situations in which web applications may derive session information from the page's URL. The Browser Object Model (BOM) supports some special properties that enable to obtain the current URL of the page: `window.location`, or equivalently `location` or `document.location`, and `document.URL`. We observe how these properties are made persistent through web storage by considering them as sources and keeping additional information about their values in the corresponding labels. Conversely, we cannot detect when the `location` property is written, because such operation causes the page to navigate to the assigned URL, thus preventing the analysis code in the page from running. This is a limitation of our analysis, because the above observation would allow us to understand if data in web storage are sent to the server as a result of browsing.

Navigator Finally, we consider some read-only properties of the `navigator` object as sources. In particular, this object provides details about the browser's environment, which are often used for fingerprinting [30]: `geolocation` for the geographic location of the device, `language` for the user's language, `platform` for the operating system, and `userAgent` for the web browser. In this way, we observe the possible uses of web storage that are likely to be associated with fingerprinting purposes.

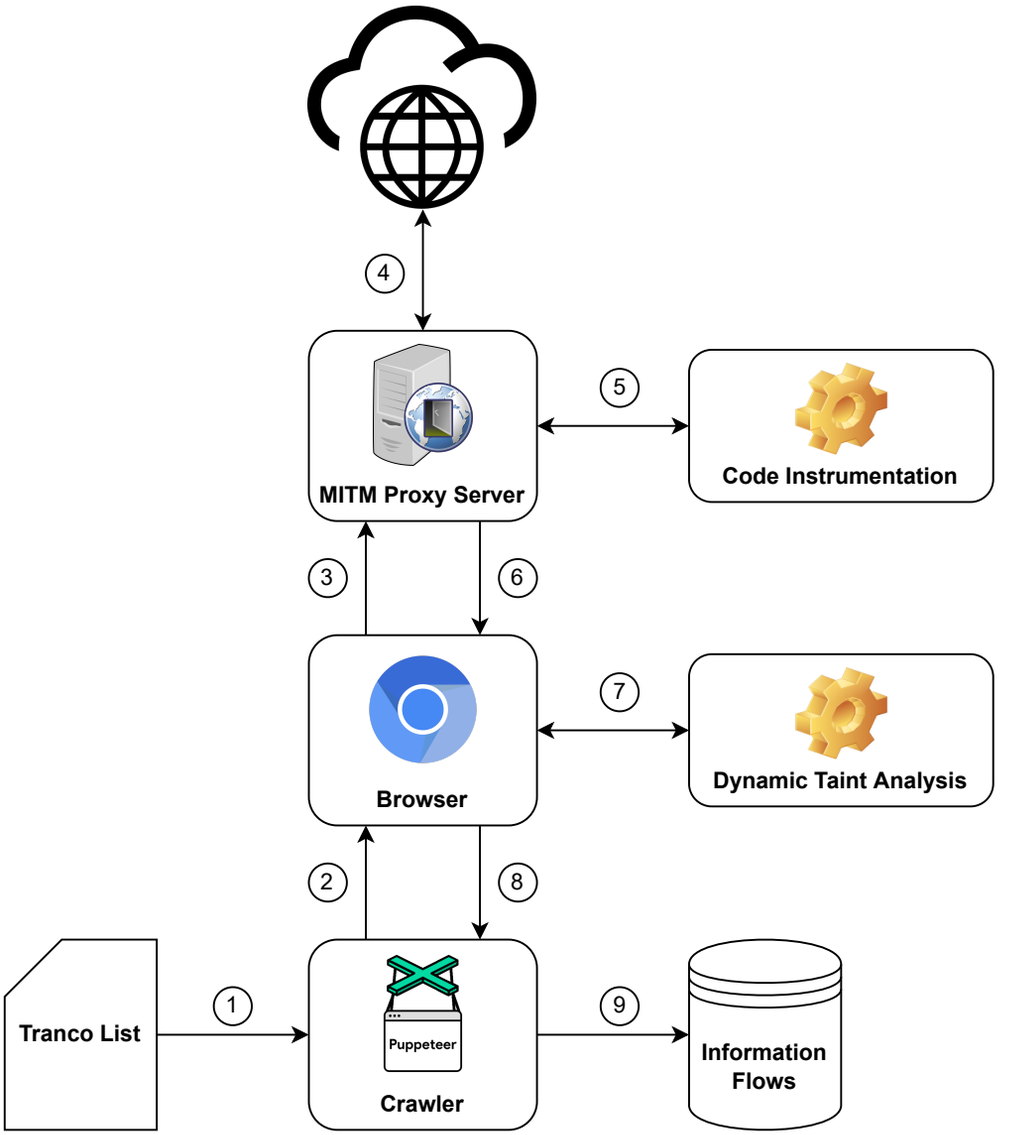

Figure 4.1: Overview of the crawling process

4.1.2 Web crawling

At this point, we use the developed dynamic taint tracking engine to automatically identify information flows involving the Web Storage API in the top 5k domains of the Tranco list [23] generated on December 14th, 2021[1].
Our taint tracking engine is implemented as a Jalangi2 analysis and also incorporates the code for the abstract machine. In particular, the instrumented scripts and the engine, both of which are written in JavaScript, are meant to run together in a browser: the former provide the web application logic and invoke the analysis callbacks of the latter, which generates the corresponding instructions that are executed by the abstract machine.
Therefore, to make the engine work in real-world sites, we need their scripts to be instrumented and inject the engine code into the browser page. For this purpose, we implement a Man In The Middle (MITM) proxy server as a Node application, that intercepts all the requests issued by the browser, fetches the required resources from the Internet, and sends back an instrumented version of the HTML

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

documents and scripts, while forwarding any other resources. The JavaScript instrumentation is performed by Jalangi, which also appends the engine code and the taint specification to the instrumented HTML documents. Furthermore, since Jalangi recognizes only ECMAScript 5.1 or below, we use Babel[2] to transpile ECMAScript 6+ code into equivalent ECMAScript 5.1 code before instrumentation.
Next, we build a crawler with a focus on automating the analysis process and making our results reproducible. The crawler, which is another Node application, uses Puppeteer[3] to drive a fresh instance of the Chromium browser to each domain in the Tranco list, leaving 60 seconds to render the HTML content after connecting. For each domain, we leverage taint tracking to collect all the information flows involving the Web Storage API.
Figure 4.1 shows a complete overview of the crawling process. The crawler reads from the Tranco list one domain at a time ① and drives the browser to navigate to such domain ②. Hence, the browser contacts the proxy server ③, first spontaneously to retrieve the index page of the site, then at each other HTTP request made by the page. After fetching the requested resource from the Internet ④, the proxy server instruments any HTML document or script ⑤ and then returns the instrumented resource to the browser ⑥. While executing the analysis ⑦, the crawler waits 60 seconds, after which it terminates the analysis, collects the information flows ⑧ and stores them in the disk for further analysis ⑨.
There are some cases in which the analysis may not detect any information flow in a given domain with the above approach. Let us analyze the possible reasons, assuming that the site is loaded properly: (*i*) the site does not actually use Web Storage; (*ii*) the crawler does not give the site enough time to observe the use of web storage, also considering that the instrumented code is slower than the original one due to the analysis overhead; (*iii*) the use of web storage requires some complex interactions within the site, which go beyond the simple loading of the index page (e.g., authentication). We try to discriminate the first case from the last two by looking in the original script for the keywords "`getItem`" and "`setItem`", which are very likely to be related to web storage. While this solution works for most of the cases, it is ineffective in presence of obfuscated code.

4.1.3 Flow classification

To better understand the use of web storage, we then perform an automated classification of the collected information flows. This is a challenging 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, described below.

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 4.1), unless network communication only involves the same origin where the flow is detected. The other flows, which we call *external*, are more interesting from a security and privacy perspective, because they involve third parties. For example, a page at `https://www.foo.com` may include a script

[2] `https://babeljs.io/`
[3] `https://pptr.dev/`

from `https://www.bar.com`, which reads the content of the local storage and sends it to `https://www.bar.com`, thus potentially leaking sensitive information from `https://www.foo.com`.

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 [12], we can detect the involvement of known web trackers in web storage accesses.

Persistence A last relevant aspect is the *persistence* of the information involved in the flow. Though both local storage and session storage can store arbitrary information, the content of local storage may persist indefinitely. Persistence may have important implications on both security and privacy. For example, the local storage may become a source of persistent XSS [32] and may enable perpetual tracking of web users. For each flow, we thus track the type of the involved web storage. Note that the same flow may involve both the local storage and the session storage, e.g., because local storage and session storage information is combined before network communication.

Table 4.2: Sources and sinks involved in confidentiality and integrity flows

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

Table 4.3: Classification of the detected information flows

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

4.2 Measurement Results

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

Table 4.2 reports a first breakdown of the detected flows in terms of the involved sources and sinks.[4] As we can see, a significant number of flows involves network sources or sinks: this happened for 329 confidentiality flows (62%) and 913 integrity flows (34%).

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

To provide further insights on the use of web storage in the wild, we also investigate potential correlations between the different axes considered in our classification. The results of our analysis are shown in Table 4.3. The table supports the following selected observations:

- Confidentiality flows are roughly equally split between internal and external flows, while integrity flows are mostly internal (70%). This shows that it is

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

Table 4.4: Additional breakdown of the external information flows

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

much more common to send web storage information to third parties, rather than having third parties write information in the web storage.

- The majority of the confidentiality flows can be attributed to trackers (65%). Remarkably, however, even a higher percentage of integrity flows can be attributed to trackers (72%). Indeed, the table also shows that the majority of tracking flows are integrity flows (85%). This suggests that trackers routinely both read and write web storage information in the wild.

- External flows are more likely to be confidentiality flows than internal flows (25% vs. 12%) and the very large majority of the external flows can be attributed to trackers (78%). Moreover, tracking flows are more likely to be external than non-tracking flows (36% vs. 25%). This suggests that trackers normally send web storage information to third parties.

- Tracking flows operate on local storage more frequently than non-tracking flows (81% vs. 69%). Moreover, flows involving the session storage are more likely to be internal than flows involving the local storage (83% vs. 63%). This suggests that local storage is the prime target of trackers and session storage is largely dedicated to internal use within a single origin.

- Finally, we observe that confidentiality flows are more likely to operate on local storage than integrity flows (88% vs. 76%), just like external flows involve local storage more frequently than internal flows (88% vs. 73%). This shows that the persistent information saved in the local storage is often the target of information leaks, likely towards third parties.

To further shed light on the security and privacy implications of web storage in the wild, we also perform an additional classification of the detected *external* information flows, i.e., information flows involving two different origins. In particular, we analyze how many such flows are still within the same site[5] and how many are cross site. This is an interesting information, because different domains under the same site normally belong to the same owner, i.e., the entity who performed the domain registration, hence same-site external flows are less significant from a security and privacy perspective. The results of our analysis are shown in Table 4.4. They highlight that the very large majority of the external flows are cross-site and this is uniform across all classes of external flows. This further confirms the relevance of our findings.

[5]A site is defined as an effective top-level domain (eTLD) + 1.

Table 4.5: Most popular libraries introducing information flows

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

The last analysis we carry out estimates how many information flows are introduced by *libraries*. These flows are particularly interesting, because libraries are normally used by multiple pages, hence the analysis of a single library may shed light on the behavior of multiple pages. To identify libraries, we look for duplicate flows within different domains and we aggregate them based on the script URL information provided by Jalangi. Specifically, we use the script URL of the source for the integrity flows and the script URL of the sink for the confidentiality flows. Table 4.5 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 large majority of these libraries (8 out of 10) is related to web tracking and the most popular library is used for tracking on 66 domains, i.e., roughly 10% of the domains where we identified an information flow involving the Web Storage API.

Chapter 5

Related work

The findings and approaches from previous studies are summarized in this section, which have significantly influenced our analysis approach.
Sjosten et al. proposed EssentialFP [30], a principled approach to the dynamic detection of browser fingerprinting. EssentialFP is a fingerprinting detection approach which is based on observable tracking, set out sinks and sources and formulate a metric which specify fingerprinting patterns via aggregated labels. 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). EssentialFP is based on dynamic analysis and in particular on an extension of JSFlow [16]. JSFlow is a security-enhanced JavaScript interpreter that allows fine-grained information flow tracking. The authors demonstrate how to overcome practical obstacles in implementing information-flow throughout the entire JavaScript language, as well as tracking data in the presence of libraries using browser APIs. The interpreter is written in JavaScript, allowing it to be installed as a browser extension.
Chen and Kapravelos presented a taint tracking engine called Mystique [7] and used it to track information leakage from browser extensions. Mystique, a novel taint analysis tool that incorporates both dynamic and static taint tracking. Based on methodologies that leverage information acquired from static data flow and control-flow dependency analysis, the authors provide the first full implementation of hybrid taint tracking for the V8 JavaScript engine. 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 [8]. In particular, the authors estimated that around 57% of the sites in the Alexa Top 10k include at least one cookie containing a unique user identifier which is exchanged with multiple third parties.
A novel dynamic taint tracking system TaintSNIFFER [21] has been built by Miller and Sandhu for Microsoft's research homogeneous C3 web browser. The authors extended the C3 browser's Javascript engine [5] and its DOM subsystem to dynamically taint and track user-provided data and sensitive internal data of certain DOM nodes. While it does specify a reasonable policy for propagating taint, as well as providing a decent test suite, there is no policy or mechanism given or implemented for using taint to do anything. One notable missing feature

is that their implementation requires the use of C3's JavaScript interpreter, missing out on its just-in-time (JIT) compiler's performance benefits.
To enforce confidentiality and integrity policies, Jan et al. proposed a rewriting based approach, called TSET [17], that define what information can flow into and from untrusted third party JavaScript code in the Chrome browser. To invoke on any JavaScript code, a rewriting function as a C++ method is implemented and then sent into the V8 execution engine. The resource loader of Chrome is modified in order to put the TSET library code into JavaScript program it retrieves.
Karim et al. implemented a platform-independent dynamic taint analysis tool for JavaScript, called Ichnaea [19]. This approach can be used with nay JavaScript engine and can track taint on primitive values without boxing. To associate a taint value with each value stored in memory, the technique uses a shadow memory model. They encoded the taint propagation logic as instructions for an abstract machine, so as to leverage an existing JavaScript instrumentation framework called Jalangi. 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. In particular, the instrumented JavaScript preserves the semantics of the original JavaScript, while running an abstract machine to track information flows in parallel. The abstract machine manipulates a stack of abstract values that reflect the taints of values on the runtime. To evaluate Ichnaea, the authors applied it to a Tizen web application to detect privacy leaks and identified flows of tainted input data to sensitive sinks in Node.js modules, thus detecting both known and unknown vulnerabilities. Our implementation follows the approach proposed in Ichnaea with few modifications, yet it is targeted to a different application scenario.

Chapter 6

Conclusion

In this thesis, we performed a first empirical analysis of the use of web storage in the wild, based on dynamic taint tracking and an automated classification of the detected information flows. Our analysis showed that web storage is routinely accessed by third parties, including known web trackers, who are particularly eager to have both read and write access to persistent web storage information. This motivates the need for further research on the security and privacy implications of web storage content, that we plan to pursue as a follow-up work of this preliminary investigation.
In particular, we plan to reuse known heuristics from the literature [8, 12] to detect personally identifiable information and better investigate the real-world privacy implications of the detected tracking flows. We also want to take a more in-depth look into the most popular libraries introducing information flows involving the Web Storage API, given the impact that libraries may have. Finally, we would like to further refine our classification of information flows to account for common use cases that we anticipate, e.g., web authentication and browser fingerprinting. Digging into selected use cases may be helpful to provide additional insights of the uses and abuses of web storage in the wild.

Bibliography

[1] Zubair Ahmad, Samuele Casarin, and Stefano Calzavara. What storage? An empirical analysis of web storage in the wild. 2022.

[2] Abdullah Mujawib Alashjee, Salahaldeen Duraibi, and Jia Song. Dynamic taint analysis tools: A review. *International Journal of Computer Science and Security (IJCSS)*, 13(6): 231–244, 2019.

[3] Musard Balliu, Daniel Schoepe, and Andrei Sabelfeld. We are family: Relating information-flow trackers. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, volume 10492 of *Lecture Notes in Computer Science*, pages 124–145. Springer, 2017. doi: 10.1007/978-3-319-66402-6_9. URL https://doi.org/10.1007/978-3-319-66402-6_9.

[4] Adam Barth. HTTP State Management Mechanism. RFC 6265, April 2011. URL https://rfc-editor.org/rfc/rfc6265.txt.

[5] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. Spur: a trace-based jit compiler for cil. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 708–725, 2010.

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

[7] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1687–1700. ACM, 2018. doi: 10.1145/3243734.3243823. URL https://doi.org/10.1145/3243734.3243823.

[8] Quan Chen, Panagiotis Ilia, Michalis Polychronakis, and Alexandros Kapravelos. Cookie swap party: Abusing first-party cookies for web tracking. In Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia, editors, *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, pages 2117–2129. ACM / IW3C2, 2021. doi: 10.1145/3442381.3449837. URL https://doi.org/10.1145/3442381.3449837.

[9] Andrey Chudnov and David A. Naumann. Inlined information flow monitoring for javascript. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 629–643. ACM, 2015. doi: 10.1145/2810103.2813684. URL https://doi.org/10.1145/2810103.2813684.

[10] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19 (5):236–243, 1976. doi: 10.1145/360051.360056. URL https://doi.org/10.1145/360051.360056.

[11] ECMAScript 5.1, Jul 2011. URL https://262.ecma-international.org/5.1/.

[12] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1388–1401. ACM, 2016. doi: 10.1145/2976749.2978313. URL https://doi.org/10.1145/2976749.2978313.

[13] Joint Task Force. Security and privacy controls for information systems and organizations. Technical report, National Institute of Standards and Technology, 2017.

[14] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982. doi: 10.1109/SP.1982.10014. URL https://doi.org/10.1109/SP.1982.10014.

[15] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In Matthew B. Dwyer and Frank Tip, editors, *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 177–187. ACM, 2011. doi: 10.1145/2001420.2001442. URL https://doi.org/10.1145/2001420.2001442.

[16] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. Jsflow: tracking information flow in javascript and its apis. In Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong, editors, *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pages 1663–1671. ACM, 2014. doi: 10.1145/2554850.2554909. URL https://doi.org/10.1145/2554850.2554909.

[17] Dongseok Jang, Ranjit Jhala, and Sorin Lerner. Rewriting-based dynamic information flow for javascript. 2012.

[18] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009. doi: 10.1007/978-3-642-03237-0_17. URL https://doi.org/10.1007/978-3-642-03237-0_17.

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

[20] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973. doi: 10.1145/362375.362389. URL https://doi.org/10.1145/362375.362389.

[21] Aaron Miller and Paramjit Singh Sandhu. Taintsniffer : A robust dynamic taint tracking system for a homogenous web browsing environment. 2010.

[22] Henrik Nielsen, Jeffrey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999. URL https://rfc-editor.org/rfc/rfc2616.txt.

[23] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. URL https://www.ndss-symposium.org/ndss-paper/tranco-a-research-oriented-top-sites-ranking-hardened-against-manipulation/.

[24] Eric Rescorla. HTTP Over TLS. RFC 2818, May 2000. URL https://rfc-editor.org/rfc/rfc2818.txt.

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

[26] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2003. doi: 10.1007/978-3-540-37621-7_9. URL https://doi.org/10.1007/978-3-540-37621-7_9.

[27] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. Explicit secrecy: A policy for taint tracking. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 15–30. IEEE, 2016. doi: 10.1109/EuroSP.2016.14. URL https://doi.org/10.1109/EuroSP.2016.14.

[28] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 317–331. IEEE Computer Society, 2010. doi: 10.1109/SP.2010.26. URL https://doi.org/10.1109/SP.2010.26.

[29] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 488–498. ACM, 2013. doi: 10.1145/2491411.2491447. URL https://doi.org/10.1145/2491411.2491447.

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

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

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

[33] Web Storage. Web storage, Jan 2022. URL https://html.spec.whatwg.org/multipage/webstorage.html.