Survey on JavaScript Security Policies and their Enforcement Mechanisms in a Web Browser

Nataliia Bielova

INRIA Rennes Bretagne Atlantique
Campus universitaire de Beaulieu
35042 Rennes Cedex, France
Email: nataliia.bielova@inria.fr

Abstract

We observe a rapid growth of web-based applications every day. These applications are executed in the web browser, where they interact with a variety of information belonging to the user. The dynamism of web applications is provided by the use of web scripts, and in particular JavaScript, that accesses this information through a browser-provided set of APIs. Unfortunately, some of the scripts use the given functionality in malicious ways. Over the last decade, a substantial number of web-based attacks that violate user's privacy and security have been detected.

For this reason, web script security has been an active area of research. Both computer security researchers and web developers have proposed a number of techniques to enforce different security and privacy policies in the web browser. Among all the works on web browser security, we survey dynamic techniques based on runtime monitoring as well as secure information flow techniques. We then combine and compare the security and privacy policies they enforce, and the way the enforcement is done.

We target two groups of readers: 1) for computer security researchers we propose an overview of security-relevant components of the web browser and the security policies based on these components, we also show how well-known enforcement techniques are applied in a web browser setting; 2) for web developers we propose a classification of security policies, comparison of existing enforcement mechanisms proposed in the literature and explanation of formal guarantees that they provide.

1. Introduction

We observe a rapid growth of web-based applications in all the aspects of the citizens life, ranging from banking and medical applications to social networks. The goal of these applications is to provide high-quality functionality and quick performance, and one of the main requirements to achieve this goal is to access the information belonging to its users. Depending on the application, this information varies a lot: financial (e.g., credit card numbers, or account information), health records, personal information (e.g., marital status, religious views, personal choices and preferences), etc. The web applications run in a web browser environment where the dynamism of an application is provided by the

scripts. In particular, *JavaScript* is one of the most used scripting languages on the today's web.

JavaScript can be inlined into a web page, or can be fetched from a remote server. When a remote JavaScript code is included into a web page, it gets the same privileges as any other code originally inlined in the page. These privileges give JavaScript code the power to perform malicious actions, violating the user's privacy and security. For example, malicious scripts can access the secret user data on the web page and send it to remote servers, or hijack the user's session and perform requests on behalf of the user.

In a recent survey on remote JavaScript inclusions, Nikoforakis et al. [46] found that out of Alexa [2] top 10,000 websites, 88.45% of sites had at least one remote JavaScript inclusion (different from the website domain), while the number of JavaScript inclusions per domain reaches 295 for some of the websites. Also, the number of new JavaScript inclusions per domain grows every year: the authors found that there were 24.48% new inclusions in 2001, and this number grew to 45.46% in 2010. These numbers show that more and more remote JavaScript programs get included into trusted web pages.

JavaScript security is a large field of research, hence it is not possible to cover the whole area in a single survey. In this survey we will talk about security enforcement rather than verification. The main difference is that verification techniques give a yes/no answer to the question "Does your program comply with the security policy?", while enforcement techniques are more powerful in a sense they not only answering this question, but can also *fix* the program on the fly such that the program execution does comply with the security policy. More discussion on the verification vs. enforcement is given in [22].

In this survey we describe two groups of security enforcement techniques. *Dynamic techniques based on runtime monitoring* observe the program execution and check whether this execution satisfies the security policies in question. These techniques are known to enforce a class of security policies that are based on a single program execution. *Secure information flow control techniques* propose program analysis (either static, dynamic or both) to find the flows of information inside the program. A particular definition of security policy enforced by these techniques is *non-interference*. It states that no secret inputs to the program can influence publicly observed outputs. Since it is not possible to detect such information flows by observing only one program execution, the definition of non-interference is based on two program executions.

We analyse and compare these two groups of techniques applied to JavaScript programs given the security policies, and map these techniques into the theory of runtime enforcement and theoretical results in information flow control. As far as we know, this is the first survey that covers both theoretical aspects of enforcement and practical considerations of the web browser architecture. The contributions of this survey are:

- identification of web browser security-relevant APIs, on which all the security policies in the literature are based,
- comparison of dynamic techniques based on runtime monitoring and secure information flow control techniques in the web browser setting in the following form:
  - description and capabilities of the mechanism,
  - a set of security policies it enforces,
  - formal guarantees that the enforcement mechanism provides,

  - implementation strategy,

- classification of useful security policies for JavaScript applications based on the set of APIs

There are a couple of related surveys on web application security, however these surveys either cover a broader area thus not comparing and analyzing enforcement mechanisms in details, or do not describe all the techniques we analyse in this survey because they were done several years ago. De Ryck et al. [16] published a survey on mashup security in 2010. The paper analyses the security requirements for separation and aggregation of mashup components, and mentions some, but not all, of monitoring and information flow control techniques that we analyse in this survey. De Groef et al. [15] briefly surveys the solutions for web script security including a subset of dynamic techniques and information flow techniques that we analyse in detail in this survey. Since the publication of De Groef et al.'s survey (2011), several new approaches for information flow control for JavaScript have been published. We cover all these works in this survey.

We start with the background on web browser architecture and security mechanisms already present in the current browsers, following by the description of JavaScript security problems in Section 2. Next, in Section 3 we identify a set of security-relevant APIs available to JavaScript programs that are used in the security policies of the mechanisms that we analyse in this survey. Then we compare the security enforcement techniques for JavaScript and the security policies. The dynamic techniques based on runtime monitoring are analysed in Section 4, while secure information flow techniques are compared in Section 5. Each of these two sections contains a collection of useful security policies for JavaScript found in the literature. Section 6 mentions other techniques for JavaScript security that are not analysed in this survey, such as static analysis of JavaScript libraries and widgets, and finally Section 7 concludes the paper.

2. Background on JavaScript Security

In this section, we give a brief overview on the web browser architecture and JavaScript[1], the access-control mechanisms present in the current browsers, and the security problems related to JavaScript programs.

Web Browser is a client-side application, and its basic function is to fetch the content from the web servers and display it in the browser's windows. Web browsers implement the *HyperText Transfer Protocol* (HTTP) and its secure version (HTTPS), that specifies the form of requests and responses between the browser and the web server. Every HTTP request to a server contains several HTTP headers: a domain and path to access the server, a content access method, and other kind of data. The web server responds with an HTTP response containing the state (such as "200 OK"), the content requested, and other headers.

[1] Interested readers can find more information about browser architecture and web browser security in Michael Zalewski's "Browser Security Handbook" [72] and the documentation about client-side JavaScript as it is implemented in the browsers in David Flanagan's "JavaScript - The Definitive Guide" [23].

Once the content is fetched from the server by means of this protocol, it is displayed by the browser. Originally web pages were simple HTML pages containing simple elements such as paragraphs, buttons, input boxes etc. With the evolution of the web, the new type of web applications appeared: *mashups*. These applications include the content from multiple sources, for example a housing rental website combines the information about the houses and maps them to Google maps. The inclusion of the remote content is usually implemented by the use of *iframes*, that separate this content from the main page. Internally, the browser implements the *Document Object Model* (DOM), that is a tree representation of the fetched web pages.

JavaScript is a widely used scripting language on the web today. Even though there exist other web scripting languages, whenever we write "script" we will mean JavaScript in this survey. JavaScript program is executed by the JavaScript interpreter of the web browser and can access a set of available APIs implemented by the browser. These APIs allow the script to communicate with other elements of the page (by DOM APIs), local browser data (such as cookies) or remote servers (by several communication APIs), and to manipulate the web page elements and other browser events. Except for the standard APIs, the new HTML5 standard [66] defines a set of additional APIs, and even though HTML5 is not yet officially accepted, most of the web browsers today have a partial support for HTML5 APIs. We will discuss security-relevant APIs in more details in the following sections.

Same-origin policy (SOP) is an access control policy which restricts the content of the web page that JavaScript code running on that page can interact with. SOP basically says that the script can read properties of windows and iframes that have the same origin as the document containing the script (and not the origin of the script itself).

An *origin* is a a *protocol, domain* and *port* of the URL. For example, in a URL `http://www.example.com:81/dir/page.html`, a protocol of this URL is "http", a domain is "www.example.com" and a port is 81. This definition is used in most of the popular browsers, except from Internet Explorer: there, an origin is just a protocol and domain. There are more inconsistencies in the SOP implementation in different browsers, for example, a script can access unrelated local files via DOM using the "file" protocol in Internet Explorer and Opera, but not in other browsers.

An *iframe* has an important property with respect to SOP: differently from remote scripts, the content and scripts of iframe is assigned an origin from which it is fetched. Hence, if the origin of an iframe is different from the one of the including page, the scripts of an iframe cannot access the DOM of the page. However, browser implements several communication APIs that allow components from different origins to communicate with each other. Notice that in addition to DOM access, SOP applies to other browser components. It was found that SOP implementations have a number of problems [56].

Malicious script inclusion and Content Security Policy. In addition to inclusion of remote scripts and iframes by a web developer, JavaScript code can be executed on a page as a result of an attack, such as Cross-site Scripting attack (XSS). A basic type of XSS is a vulnerability of the web server when it allows an attacker to store the malicious script on the server side, and then get executed in the user's browser. In this survey we are not going to discuss different ways of script injection on a page, but rather discuss

the security issues that occur when a JavaScript code is already running on the page.

In order to protect from the attacks like XSS, W3C proposed a *Content Security Policy* (CSP) [63]. This policy should be written by a web page developer and contains a set of sources from which the remote content can be loaded and executed on the page. The CSP applies to scripts, objects, style sheets, images, media, iframes, and fonts. Since CSP is oriented on disallowing to fetch the content from the forbidden sources, the enforcement of CSP described in the draft does not forbid to make an HTTP request, just the HTTP response should be seen by the browser as an empty response. It does not seem a completely secure solution since the HTTP requests (that may contain some information belonging to the user) can anyway be sent to arbitrary remote servers.

JavaScript security and privacy problems. Back in 2010, Singh et al. [56] were the first to point out that the implementation of SOP is incoherent for different browser objects and APIs, and revealed a number of vulnerabilities related to this fact. In the last couple of years more objects and APIs were specified and implemented, and hence in this survey we will describe a superset of such objects and APIs with respect to the one of Singh et al. [56]. It was also noticed by several authors that SOP does not prevent the leakage of sensitive user information: once malicious JavaScript accesses this information, it can easily transmit it to remote servers by a number of ways [9, 15, 33, 60].

Jang et al. [32] analysed several kinds of attacks from the literature and have collected them into several groups of vulnerabilities called *privacy-violating information flows*:

1. Cookie stealing. Since a remote script included in a web page can access all the browser objects that the original page can access, such script is able to access cookies, that may contain a session identifier or other sensitive information. The script afterwards can transmit the value of the cookies to an arbitrary remote website by a number of ways (we shall discuss the communication APIs in the following section).
2. Location hijacking. An untrusted JavaScript code can either influence the document's location directly or the string variables that are forming a URL. As a result, the script can navigate the page to a malicious website without the knowledge of the user.
3. History sniffing. In this attack a malicious script can check whether the user has ever visited a specific URL. JavaScript code can create an invisible link to the target URL and then use the browser's interface to check how this link is displayed. Since the browser displays visited and unvisited links in different colours, JavaScript program can deduce whether the URL has been visited by the user or not.
4. Behavior tracking. A script running on a web site can gather precise information about the user's mouse clicks and movements, scrolling behaviour, what parts of the page are highlighted, and clipboard content. Such capability of JavaScript can be useful for the user's interaction with the website, however a malicious script can also leak this information to the remote servers.

In the next section we are presenting a set of APIs that are used in these classes of privacy attacks and also in the security policies that we have found in the literature on JavaScript security.

3. Security-relevant browser APIs

Singh et al. [56] proposed a comparison between different browser objects APIs and the corresponding principals who owe the information stored in those objects. Inspired by their work, we present the summary of the standard and HTML5 resources and the corresponding access-control policy in Table 1. These APIs were collected from the literature that we later present in this survey. In this Table with "*" we mark the objects for which there is no access-control policy in place.

We divide the resources in two groups: *standard browser objects and APIs* that have been implemented several years ago and nowadays are present in all of the widely used browsers; and a large set of new *HTML5 APIs* [66] some of which are already available to scripts. On one hand, this set of HTML5 APIs allows the web developers to write much richer and interactive applications, but on the other hand, more security concerns come with the new functionality available to JavaScript programs.

Table 1: Objects and APIs and their corresponding access control policy

Resource	**Access control policy**
Standard objects and APIs	
Core DOM objects	SOP
Domain	SOP
Window	SOP
Cookies	domain/path
Referrer	SOP
History	not well-defined
Location	SOP
XmlHttpRequest	SOP (except for CORS)
XDomainRequest (Internet Explorer)	*
HTML5 APIs	
Web Storage	SOP
Cross-origin messaging	*
WebSocket	*
Server-Sent Events	SOP
IndexedDB	SOP
Geolocation	"first requested"
Web Workers	SOP

3.1. Standard security-relevant APIs

Core DOM objects are all the nodes of the HTML document, with the `document` node at the root of the tree. A malicious script can get an access to a `document` object by a number of other ways. An example used in the reviewed literature is an `iframe` creation using `document.createElement`. With this method a script can create an iframe element, and then access the window (and hence the document object) through the `contentWindow` property of an iframe.

Core DOM objects are accessible to scripts running within the origin of the page according to SOP.

Domain. A script may change the current SOP by changing the domain of the application: it is technically possible by changing `document.domain` to its valid domain suffix. For example, scripts running in the origins with domains `login.example.com` and `payments.example.com` can change its domain to `example.com`, but cannot change it to `ample.com` or top-level domain `com`. If the origins of these two scripts also have the same protocol and port (or just protocol in case of Internet Explorer), they will start running in the same origin and access each other's DOM. Notice that it does not mean that they can now also access the DOM of the page at `http://example.com` even if they have the same protocol and port. If a page at `http://example.com` needs to exchange some information with `payments.example.com` , it should perform an explicit assignment of `document.domain = "example.com"` [73]. Notice also, that it is possible to relax the `document.domain`, but it is not allowed to restrict it (set it to `login.example.com` on a page at `http://example.com`).

Singh et al. found out that changing the `document.domain` affects the SOP for DOM access, but not for other objects, such as cookies, XmlHttpRequest, postMessage and localStorage [56]. In addition, even though the well-known books on JavaScript and Web Browser security [23, 73] claim that it is not possible to set the `document.domain` to a top-level domain such as `com`, Nick Nikiforakis[2] recently found that in WebKit (a layout engine of Chrome and Safari) it is actually possible to make such assignment to a top-level domain.

Window. The `window` object represents an open window (or tab) in the browser. When a page contains iframes, a separate window object is created for each iframe. The window object is accessible to scripts running in the same origin.

Cookies. The HTTP(S) protocol is stateless by design, meaning that a a web server cannot define whether two HTTP requests are coming from the same client unless it adds a separate mechanism to track the clients. The most commonly used tracking mechanism is session identifiers. A session identifier is a unique random string that is generated by a web server and is sent to the client's browser usually by the means of cookies. A web server sends the cookies in the HTTP response header `Set-Cookie`. Once a web browser receives such a response, it sets the cookies in the browser, and now every HTTP request to that web server will contain the cookies in the HTTP request header `Cookie`[3].

Sessions are used in e-commerce applications, web-mails, and basically any other applications that need to provide an access control to the users. Session identifiers are a prime attack target since an attacker who accesses such identifier is able to have the same session as a user and hence access all sorts of information that belong to the user. The most common way to steal the session identifier is through the XSS.

Another common usage of cookies is in web tracking techniques [43]. In order to track it's users, the websites have to distinguish the users from each other. The simplest form

[2] `http://blog.securitee.org/?p=208`

[3] In the past, the HTTP standard also contained the headers `Cookie2` and `Set-Cookie2` that were supposed to be used

of tracking is made by storing a unique identifier in the user's browser cookies. Later, when the same user (using the same browser) comes back to this website, the cookies are automatically sent and hence the web server can recognise the same user (even if she does not log in this website). More complicated forms of tracking are classified in the recent paper by Roesner et al. [50].

Cookies are accessible to JavaScript programs through the `document.cookie` DOM API. In order to protect cookies from being stolen, Microsoft developers introduced the notion of `Http-Only` cookies and added support for them in Internet Explorer 6, SP1 [45]. `Http-Only` is an HTTP response header and it means that cookies will be included in every HTTP request and response to the web server, however they are not accessible to any script running in the browser.

Unfortunately, `Http-only` cookies are not widely used. Several groups of researchers showed that `Http-only` header is rarely set for the session identifiers. As such, Nikiforakis et al. [47] analysed Alexa top 500 websites and found out that only 22.3% of them are using `Http-only` cookies for session identifiers. In parallel with Nikiforakis et al., Tang et al. [58] proposed heuristics to detect cookies that contain session identifiers and add an `Http-Only` header to them.

Cookies' lifetime is not limited by a session: cookies can be stored in the user's browser for a much longer time, until they expire. That is why cookies are also widely used for different web tracking techniques [50].

The *access control policy* applied to cookies is different from SOP [56]. It allows scripts to access cookies only when the script is being executed in the context with the same *domain* and *path* as the cookies (without the protocol and port number as in SOP). Moreover, a cookie creator can set the cookie's `domain` to a postfix domain or the path name to a prefix path (similar to modifying `document.domain` in case of SOP for DOM access).

Referrer. Another HTTP header that is automatically added by the browser to all outgoing requests is a "Referer"[4] header. This header indicates the URL from which the current request originated. For example, imagine a user visiting a website located at `www.example.com/index.html?id=42` and is clicking on a link `www.shopping.com`. Then her browser would send an HTTP request containing a "Host" header set to `www.shopping.com` and referrer header set to `www.example.com/index.html?id=42` [48]. Notice that this header is sent not only when the user clicks on the link, but also in all the requests initiated by the browser while fetching the remote content (such as images, scripts, embedding objects, etc.). Hence, the remote servers learn about the user browsing history (and also user's id if it is specified in the URL) in this way. Referrer is accessible to JavaScript through `document.referrer` DOM API.

In HTML5 a web developer can add a special `noreferrer` attribute to the selected link tags, that will cause the browser not to add the referrer header to all the outcoming requests when this link is clicked. This technology is not implemented yet in most of the browsers and for this reason, is not widely used by web developers [48]. Similarly to other DOM objects, when referrer header is accessible to JavaScript, the SOP applies.

[4]The correct spelling is "referrer". The misspelling "referer" was introduced by mistake by Phillip Hallam-Baker [42] and later incorporated into the HTTP specification.

History. Currently the browser history is accessible to JavaScript only through the `window.history` object. For security reasons, JavaScript code is not allowed to access the array of all the URLs visited by the user through the `history` object. However, there are other history detection techniques [31]. One well-known technique is based on Cascading Style Sheets (CSS) `:visited` selector. It styles visited links differently from unvisited ones. Then, using the function `getComputedStyle(link, "").color`, it is possible to establish the colour of the given link `link`, and hence to conclude whether it was visited before. David Baron proposed a solution for this attack [7] based on rewriting the `:link` and `:visited` selectors so that they are no longer based on user's history, but return the style of unvisited link. It seems that the major browsers have implemented this solution. However, there are still other ways to (partially) access user's history: Jang et al. [32] describes other techniques based on user interation, while Roesner et al. [50] discusses how an attacker can learn about user's history through web tracking techniques that use the cookies or referrer header. The access control policy for browser history is not well defined.

Location bar is used to type, change and show the URL of the navigated page, but can also be used to run the JavaScript code. It is possible to access the `location` object from JavaScript program as a property of a `window` object or of a `document`: `document.location == window.location`. Moreover, a `location` object has a number of methods that allow JavaScript code to redirect the page.

Changing the location of the page that the user visits can be potentially dangerous: a malicious JavaScript code running on the page could redirect the user to an attacker's page. For this reason, some browsers today do not allow the redirection, or alert the user. The Same-Origin Policy applies to the location bar.

XMLHttpRequest object (known as XHR[5]) provides a way of communication between a client and a server [64] via the HTTP and HTTPS protocols. `XMLHttpRequest` objects provides a method `open()` to start a communication and a method `send()` to send the needed request.

Normally, SOP applies to the URLs of the requests made by the XHR objects. The scripts running in one origin are allowed to make requests only to the server with the same origin. However, differently from SOP for DOM objects, SOP for XHR does not change when the `document.domain` is changed. It was also found, that in Internet Explorer even though SOP for DOM does not contain the port, SOP for XHR contains it.

However, with the new *Cross-Origin Resource Sharing* (CORS)[6] technology, XHR is also used to make cross-origin requests when a target server explicitly allows such requests. For security reasons, it is not allowed to send cookies and other user credentials, however a malicious JavaScript can save the cookies as a parameter of a URL and then make a cross-origin request. In Internet Explorer this technology is implemented with `XDomainRequest` object.

[5] Historically, there were two specifications: XHR level 1 and XHR level 2. In the end of 2011 the two specifications were merged into a single XHR specification.

[6] `http://dvcs.w3.org/hg/cors/raw-file/tip/Overview.html`

3.2. HTML5 security-relevant APIs

Web Storage draft specification [69] defines two properties of the window object: `localStorage` and `sessionStorage`. Both of them allow to store some data in the browser that can be retrieved next time the user visits the page. The difference between the two objects is that the `localStorage` permanently stores the data in the browser while the `sessionStorage` has the same life-time as a top-level window or browser tab in which the script that created it is running. The data in storages is separated per-origin in a sense of the Same-Origin Policy.

Cross-origin messaging API allows asynchronous message passing between scripts running in different origins. A script running in one origin can invoke a `postMessage` method to send the message to another origin. A new DOM event `message` is fired when the new message is received by a target origin. The sender origin is stored in the `source` property of the `message` object and is recommended to be checked upon receival.

Differently from XHR, SOP does not apply to the requests made by `postMessage` because the purpose of this cross-document messaging method is to allow the documents from different origins to communicate with each other.

WebSocket API allows a bidirectional message exchange over socket-type connections. It uses `ws://` protocol or `wss://` for secure connection. Notice that WebSocket enables communication between parties on any domain. The server decides whether to make its service available to all clients or only those that reside on a set of well defined domains. This can be used by the web attackers: they can first inject their script into a page, and then make it communicate easily to the attacker's server using WebSockets. Notice that SOP does not apply to WebSockets.

Server-Sent Events API[7] provides a one-direction protocol that allows the server to communicate back to the client and is supposed to be used to update the client with some information from the server. Differently from WebSockets, it uses traditional HTTP. However, SOP applies to Server-Sent Events API: the script can only use this API to send messages to the server that has the same origin as the context in which the script is running.

IndexedDB APIs [65] provides an object database records holding simple values and hierarchical objects. Comparing to Web Storage APIs, IndexedDB is more powerful and efficient. However, IndexedDB is not widely implemented in the current browsers. As of the day of writing this paper, only FireFox and Chrome have implemented it.

Like Web Storage, the SOP is an access control policy for IndexedDB. Moreover, each origin can have a number of IndexedDB databases, while each database must have a unique name within the origin.

Geolocation API [67] allows JavaScript to access the user's current location via `navigator.geolocation` property. Browsers nowadays either ask permission before accessing it or deny it (either hardcoded in the browser or the user specified it in the

[7] `http://dev.w3.org/html5/eventsource/`

preferences). However, when a page contains several frames from different origins, and all of them are asking for a user's permission at the same time, the script that succeeds in invoking the permission window first wins, while all the other scripts are prevented from accessing the geolocation [56]. Once this permission is granted, JavaScript code is able to monitor the user's location every time it changes significantly.

Web Workers API[8] is a way for browsers to run JavaScript in the background. Originally, JavaScript was single-threaded, but with web workers this is no longer the case. The web workers are running in parallel threads, with no access to the DOM, the `window` and `document` object of the main thread. However, just like the third-party JavaScript application running on the page, web workers are able to access cookies, web storages, `navigator` object, `location` object (read-only). Web worker can communicate to the main thread via postMessage or XHR, or execute `setTimeout` and `setInterval` functions that can execute dynamically generated JavaScript. Moreover, web workers can import external scripts using the `importScripts` method, and spawn other web workers. The only argument to the Web Worker constructor is a URL that specifies the JavaScript code from which the worker is created. SOP applies to Web Workers: the web worker can be created only from the URL that has the same origin as the execution context where it is created.

3.3. Other analysis of security relevant APIs

In their recent ENISA report [17], De Ryck et al. have presented an abstract model of emerging web standards, and later grouped them into categories of security-sensitive APIs in the following paper including the same authors [60]. We show this model in Figure 1.

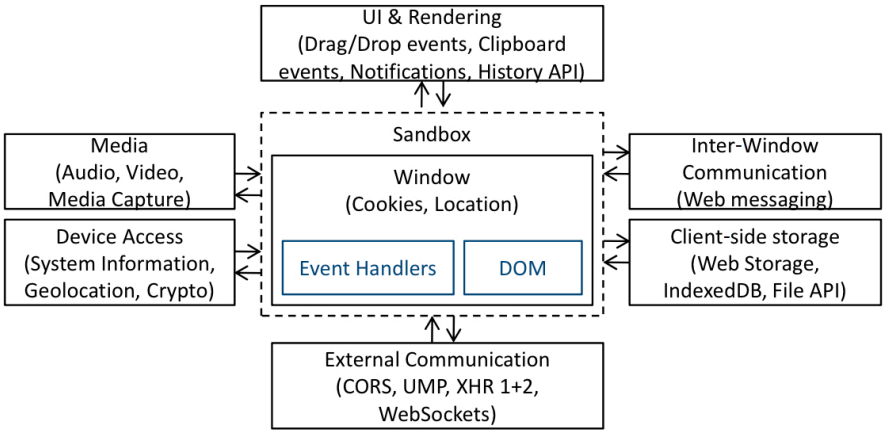

Figure 1: The categories of the emerging web standards [60]

[8] `http://dev.w3.org/html5/workers/`

4. Dynamic techniques based on runtime monitoring

In this section we present the dynamic techniques based on runtime monitoring for JavaScript security. We provide a collection of safety properties that were enforced in the literature for a better understanding of the problems of JavaScript security. These techniques implement runtime monitors that intercept the API calls made by JavaScript program and check whether the sequence of such calls complies with the given security policies. Some works in this area do not provide any formal guarantees, however, since the web browser is a very complex system and has a number of inconsistencies, formal proofs and methods are particularly important in this area of research.

This section is organised as follows: we start with the background on runtime monitoring theory, security policies and possible formal guarantees, and proceed with the analysis of monitoring techniques for JavaScript. Description of each technique starts with the main idea of the mechanism, then clearly states the security policies it enforces and finished with the provided formal guarantees. We finish this section with the collection of all the examples of security policies enforced by the dynamic techniques discussed in this section.

4.1. Background

Runtime monitoring is a common technique to observe the execution of the program and check whether it satisfies the desired security policies. The first formal model of runtime monitor was proposed by Schneider [55] and is called *security automaton*. It recognises legal (allowed by the policy) run of the program and halts it as soon as its behaviour violates the policy. These monitors are provably enforcing a restricting class of security policies called *safety properties*. The later works of Bauer, Ligatti and Walker [8] proposed another model of runtime monitor, called *edit automaton*. These monitors are able not only halt the execution of illegal program, but also to *fix* it (change it) at runtime. Edit automaton is able to enforce a bigger class of security policies, called *renewal properties* that are also based on a reasoning about a single program execution.

4.1.1. Security policies

Informally, safety property is defined as "nothing bad ever happens", meaning that if a sequence of actions is legal, then it does not have any illegal prefix. Safety properties have this nature: as soon as a sequence becomes invalid, no following continuations of this sequence can ever become valid. Therefore, the only way to enforce a safety property is to halt an execution of the program as soon as it violates the safety property.

An informal definition for renewal property is less straightforward. It is stated as a property in which "every valid infinite-length sequence has infinitely many valid prefixes" [36]. Notice that this definition differs from safety, because it allows a valid sequence to have some finite number of invalid prefixed. In this setting if a program execution does not satisfy the policy so far, it does not mean that this execution will not become legal later on. The authors also have shown that decidable renewal properties can be enforced by some edit automaton.

Both safety and renewal properties represent a property of one sequence of actions. In case of JavaScript, an action can be represented by an API call (possibly with some restrictions on arguments).

4.1.2. Formal guarantees

These works on runtime monitoring have the same goal: to enforce the security policies in question. However, the guarantees of enforcement that different techniques provide, sometimes varies. The main guarantee of enforcement is *soundness*. It states that the result of the enforcement is compliant with the security policies. Notice that soundness by itself can mean that a runtime monitor halts all the executions of the program and thus satisfies the security policy (that considers non-execution as a good behaviour). Hence, there is another important guarantee called *transparency*. Transparency means that if the program execution already satisfies the security policy, its behaviour should not be modified by the runtime monitor.

4.2. First monitoring technique for JavaScript

The first implementation of dynamic monitoring and logging of JavaScript code execution was done by Hallaraker and Vigna [28]. They were the first to describe in details the architecture of Mozilla FireFox and the way the JavaScript engine SpiderMonkey interacts with the DOM and other components of the browser. The FireFox browser is partially implemented in JavaScript, and hence the JavaScript engine executes both the "native" scripts and the scripts from the downloaded HTML page. The authors implemented an auditing mechanism that is able to distinguish the execution of these two classes of scripts and to audit only the scripts running on the page.

The auditing mechanism detects two kinds of behaviour: the deviation from high-level security policies and recognition of predefined attack scenarios ("signatures"). The high-level security policies are safety properties based on the state recorded by the mechanism. The signature for a simplified version of a cross-site scripting attack is hardcoded into the auditing mechanism (for example, do not redirect the page after cookie access).

Security policies (safety properties) presented in the paper are:

1. Do not open more than n windows,
2. Auditing a frequency of a `window.alert` method call,
3. After reading `document.cookie`, do not set a `document.location`

Formal guarantees No guarantees are provided.

4.3. JavaScript instrumentation

Yu et al. [71] proposed an instrumentation technique for a CoreScript language, which reflects the core features of JavaScript. This instrumentation is represented as a set of formal rewriting rules, that are based only on the syntax of the language. The idea of such rewriting is that the execution of instrumented code is compliant with the security policies.

The formal model behind the instrumentation is based on edit automata [8], and hence theoretically it should be able to enforce a class of renewal properties. However in the paper the examples of security policies are safety properties, while an edit automaton model helps to enforce these properties differently than just halting the execution. For example, a simple policy "Do not open more than n windows" is enforced in a following way: whenever an $n + 1$th window is about to be opened, an enforcement mechanism suppresses this action and waits for the opened windows to be closed first.

Security policies (safety properties) described in the paper:

1. Do not open more than n windows,
2. No foreign links after a cookie access

Formal guarantees The soundness and transparency of the rewritten programs are proved.

4.4. Lightweight Self-protecting JavaScript

Phung et al. [12] propose an inlining technique for JavaScript based on aspect-oriented programming. The security policy and the enforcement mechanism are implemented as a remote JavaScript script and hence can be added to any HTML page. Like in a previous approach [28], the security policies are stateful, and this mechanism is able to modify some of the actions at runtime, hence it is potentially able to enforce renewal properties. The authors, however, prove their results for safety properties and show the connection between their model and security automaton.

Security policies (safety properties) given in this paper:

1. Only up to n pop-up windows can be opened, URL must be from a whitelist
2. Disallow `iframe` creation via `document.createElement`
3. If the sensitive data (such as `cookie`, `history` object, and the values of the values of `window.location` , `document.URL` and `document.referer`) have been read, disallow setting of `document.location`, `windows.location`, and `src` property of the instances of frames, images, forms.
4. Disallow setting of `document.location`, `windows.location`, and `src` property of the instances of frames, images, forms except if the new URL is in a whitelist.
5. Disallow `XMLHttpRequest` to open and send data except for the URIs that are in the whitelist
6. Disallow pop-up windows without location and status bar
7. Disallow `window.alert` and `window.prompt` methods (can case resource abuse)

Formal guarantees The authors prove the *soundness* and argue that transparency does not hold.

The problem is that JavaScript can inspect the source of the page in which it is embedded. Hence, a script can detect that it is being monitored, because in this technique a monitor is implemented as a remote script. The authors discuss a possibility that a script might behave well when is not watched, and behave badly when monitored.

4.5. ConScript: Applying client-side deep advice for security

Meyerovich and Livshits introduced the ConScript [44] framework, a client side implementation for fine-grained security policies based on *aspects* [21]. The authors make modifications to the JavaScript engine of Internet Explorer 8, changing the original implementations of the security-relevant functions. Later this implementation was repeated by De Groef [13] on top of Mozilla Firefox browser.

ConScript policies are either written manually or can be generated through static analysis of server-side code or runtime analysis of client-side code. The authors also present a type system to ensure correctness of the ConScript policies, but do not provide formal guarantees of the enforcement mechanism.

Security policies (safety properties) The authors present 17 security policies grouped in 4 classes, some of which are taken from [34, 12]. Here we do not present all

the policies, but only flexible policies that are comparable to the security policies in the other works on dynamic techniques. For example, we do not include very strict policies, such as "disable dynamic scripts", and omit policies specific for the behaviour of jQuery library. The chosen security policies are:

1. Disallow string arguments to `setInterval`, `setTimeout` (because these functions run callbacks in response to passing of time),
2. `XMLHttpRequest` is restricted only to HTTPS connections,
3. HTTP-only cookies: script cannot access cookies,
4. `postMessage` function may transmit primitive values only to origins in a whitelist of URIs,
5. `XDomainRequest` for communicating to foreign servers is checked against a whitelist of URIs.
6. No foreign links after a cookie access,
7. Limit the number of popup windows opened,
8. Disable dynamic `<iframe>` creation,
9. Whitelist URL redirections,
10. Prevent abuse of resources like modal dialogs.

Formal guarantees No guarantees are provided.

4.6. WebJail: Security architecture for mashups

Van Acker et al. [60] proposed WebJail, a new client-side security architecture that enables least-privilege integration of components into a web mashup, based on *aspect weaving* [21] (similar to ConScript [44]) while the security policies are specified for every iframe of the page.

The language of the security policy is relatively simple and is similar to the Content Security Policy (CSP) [63]. The authors first defined categories of security-sensitive APIs that we presented in Figure 1. Then, the security policy specifies a self-defined whitelist for every category of APIs. For example, "extcomm : [google .com, youtube .com]" means that external communication are only allowed to the given domains.

Security policies The policy is a new attribute of an iframe in a mashup, which means that a mashup integrator can impose restrictions on the behaviour of untrusted third-party components. In the iframe policy, particular security-sensitive events can be fully enabled, fully disabled or enabled only for a self-defined whitelist.

Formal guarantees No formal guarantees are provided.

This approach is different from the other dynamic techniques presented in this section because it enforces specific policies for mashup integration, however the policies are still safety properties: the JavaScript programs are not allowed to invoke the APIs that contradict the security policy.

4.7. Summary of security policies

In Table 2 we present an extensive summary security policies enforced by dynamic mechanisms presented in in this section (while only omitting some strict policies or very specific policies, such as for jQuery library). These policies are stateful policies that reason about one execution of the program. Some of the techniques [12, 71] are able to

Table 2: Safety properties enforced in the literature

Corresponding APIs	**Related properties and methods**	**Security Policy**	Hallaraker and Vigna [28]	Yu et al. [71]	Phung et al. [12]	Meyerovich and Livshits [44]
Window	`window.open`	Limited number of popup windows opened	✓	✓	✓	✓
Core DOM objects, Cookies, Location,	`document.cookie`, `window.cookie`; `document.location`; `src` property of nodes	No foreign links after a cookie access	✓	✓	✓	✓
Window	`window.alert`, `window.confirm`, `window.prompt`	Prevent abuse of resources like modal dialogs	✓		✓	✓
Window	`window.open`	No popup windows without location and status bar			✓	
Core DOM objects	`document.createElement`	No dynamic `iframe` creation			✓	✓
Core DOM objects	`src` property of nodes	No setting of `src` property of frames, images, forms			✓	
Location	`document.location`, `window.location`	No setting of `location` property			✓	✓
Location	`document.location`, `window.location`	Redirections are allowed only for a whitelist of URLs			✓	✓
XmlHttpRequest	`XmlHttpRequest.open`, `XmlHttpRequest.send`	No `open` and `send` methods of XHR object			✓	✓
XmlHttpRequest	`XmlHttpRequest.open`	XHR is restricted to HTTPS connections				✓
XDomainRequest	`XDomainRequest.open`	`XDomainRequest` URL is checked against a whitelist				✓
Cross-origin messaging	`window.postMessage`	postMessage can only send to the origins in a whitelist				✓
JavaScript specific	`setInterval`, `setTimeout`	No string arguments to `setInterval`, `setTimeout` functions				✓

Table 3: Comparison of dynamic mechanisms based on runtime monitoring for JavaScript

	Hallaraker and Vigna [28]	Yu et al. [71]	Phung et al. [12]	Meyerovich and Livshits [44]
Security properties				
safety	✓	✓	✓	✓
renewal		✓		
Formal guarantees				
soundness	✗	✓	✓	✗
transparency	✗	✓	✗	✗
Implementation strategy	auditing interactions	code rewriting	code rewriting	JS engine modification

enforce renewal properties, however the examples of the policies we found in the paper are safety properties.

We list the papers in the columns in a chronological order showing how the set of security policies in the papers grew with the evolution of the research field. When we mark a particular policy and a paper by a tick "✓", we mean that the authors have presented such policy in their paper explaining how their mechanism would enforce it. Whenever there is no tick in the table, it does not mean that the give technique is not able to enforce the given policy, it just was not presented in the paper. Overall, those mechanisms that were formally proven to be able to enforce safety properties (such as [71, 12]), are also able to enforce all the policies from Table 2.

Notice that some of the policies, like "No foreign links after a cookie access" were presented in all the papers in one form or another. However, the authors proposed to monitor different properties or methods when they defined "foreign links". Also notice that all of the listed enforcement mechanisms are able to enforce policies with whitelisted exceptions. We do not add the examples of WebJail policies [60] because they put restriction on a category of APIs, and thus cannot be compared to more specific policies proposed in previous papers.

4.8. Comparison of enforcement

In Table 3 we compare the above mentioned techniques by the following parameters: the security properties they potentially enforce (safety or renewal properties), the formal guarantees they provide (soundness and transparency) and the implementation strategy.

When we specify security properties, we mean that a mechanism marked by "✓" is in theory capable of enforcing a given class of properties. However, as mentioned earlier, all the proposed approaches show to enforce policies that are safety properties.

When it comes to formal guarantees, we marked an approach by "✓" when the guarantee was proven in the corresponding paper. Notice that our marking by "✗" does not mean that the formal security guarantee does not hold for a given mechanism, but it just shows the fact that in the original paper no proofs of such guarantees were given. The only mechanism, for which soundness and transparency is proven is a JavaScript instrumentation by Yu et al. [71]. The authors considered these formal guarantees because

this mechanism is modelled as an edit automaton. The other proposed mechanisms were not considered to be modelled as runtime monitors, but the novel technology for security enforcement in the web browsers was proposed.

The implementation strategy differs in these mechanisms. Hallaraker and Vigna [28] were the first to introduce a mechanism that logs all the interactions between the JavaScript engine and the other components of the web browser. This log was then audited against some malicious behaviors (in Table 3 we denote this strategy by "*auditing interactions*"). Yu et al. [71] proposed a rewriting technique for a subset of JavaScript, instrumenting it with security checks. Phung et al. [12] adapted an inlining reference monitor approach for JavaScript by implementing it using aspects. Hence, both of these latter mechanisms use *code rewriting* techniques. The ConScript [44] framework is radically different from the other approaches. It proposes to modify only the JavaScript engine of the web browser, thus implementing security checks. In Table 3 we denote it as *JS engine modification*.

5. Information flow security analysis for JavaScript

Information flow security for programming languages have been studied for decades and now it is a large field of research. We start this section with the definitions of information flow security policies, and give a brief introduction to information flow analysis. We then proceed with the main achievements in applying information flow analysis to JavaScript, and conclude by a comparison of these techniques with respect to the security policies they enforce, the formal guarantees they provide and the implementation techniques applied.

5.1. Background

There are two main security properties that can be enforced by information flow control: confidentiality and integrity. *Confidentiality* defines that the private information should not be revealed to the public observer, while *integrity* means that the public entities should not influence the private information. Both properties can be enforced by information flow analysis. For this reason, researchers build their analysis around confidentiality policies, claiming that the same approach can be applied for integrity.

For programming languages, the notion of confidentiality has been redefined into the notion of *non-interference*. It specifies that no public outputs of the program depend on secret inputs (which means no public observer can deduce anything about the private information). In the language-based security community, information flow analysis is implemented for a given programming language and it insures that there is no leakage of private information into the public outputs of the program.

The first extensive survey on language-based information flow was given by Sabelfeld and Myers [53]. After their publication there has been a number of important results in the information flow analysis, and currently it is often classified according to the following parameters.

5.1.1. Explicit and implicit information flows

There are two basic kinds of information flows: explicit and implicit. Information is passed from the right-hand side of an assignment to the left-hand side forms an *explicit*

flow. For example, a statement `public:=secret` represents an explicit flow from a variable `secret` to a variable `public`. *Taint analysis* is a well-known technique that handles only explicit information flow. *Implicit flows* occur when the information is passed through the control flow structure. For example, `if secret then public:=1` represents an explicit flow from a boolean variable `secret` to `public`.

The later works by Austin and Flanagan have proposed a dynamic mechanism for dealing with implicit flows with *no-sensitive upgrade* (NSU) semantics [4], and later they presented *permissive-upgrade* (PU) semantics [5].

5.1.2. Formal guarantees

An information flow analysis should be *sound*, meaning that all the programs accepted by the analysis are noninterferent. There are several notions of noninterference proposed in the literature. *Termination-insensitive noninterference* [53] only gives a guarantee about terminating programs, ignoring that non-termination may leak some confidential information. *Time-sensitive noninterference* reasons about programs that terminate in a given number of steps, and hence it is a stronger notion that termination-insensitive noninterference.

Proving the soundness of information flow analysis is not enough: an analysis can reject all the programs, and still be sound. The notion of *transparency* (or *precision*) was introduced to specify how many secure programs got rejected by the analysis. Notice that this definition differs from paper to paper, and it is based on the number of secure programs that do not get rejected by the analysis. The size of this set of good programs is usually evaluated by the authors, and often it is a set of programs accepted by some security type system, such as the one by Volpano et al. [62].

5.1.3. Flow-sensitivity

Another important property of the information flow analysis is *flow-sensitivity*. The analysis is *flow-insensitive* if it does not take into account the order of execution, so the analysis results for $C_1; C_2$ are the same as for $C_2; C_1$. In this respect, the security type system of Volpano et al. [62] is flow-insensitive. If the analysis takes into account the execution order, then it is *flow-sensitive*. As a result, in flow-sensitive information flow analysis the variables can store information of different sensitivity (secret and public).

Consider a program `secret := 0; if secret then public:=1`. Flow-insensitive analysis rejects this program because `public` depends on the secret value of `secret`. However, flow-sensitive analysis accepts this program since the security level of `secret` is changed to low after the first assignment.

5.1.4. Static vs. dynamic vs. hybrid analysis

Initially, the enforcement techniques for secure information flow were based on purely *static analysis*, like Denning-style enforcement [18] and it was proven to be a sound technique for explicit and implicit flows. We are not going to discuss these techniques in details, but rather point interested readers to a survey of Sabelfeld and Myers [53]. Another way is to *dynamically monitor* the execution of the program, this approach was reopened in Le Guernic's PhD thesis [26], where he gave an extensive survey on the field.

5.1.5. Declassification

It has been repeatedly shown in the literature, that non-interference is too strong for real life systems. A well-known example of password checking shows it: the password is a secret information and hence, it cannot influence public outputs of the program, but on the other hand, it should be sent to the remote server (which is usually considered public). *Declassification* is an exception mechanism, that allows secret inputs to influence public outputs under certain conditions. We will show which of the techniques for secure information flow, that we survey here, support declassification.

5.1.6. Relations between the analysis

Hunt and Sands [30] have proven that static flow-senstive analysis generalises static flow-insensitive analysis, and accepts more secure programs.

Sabelfeld and Russo [54] have proven that sound purely dynamic information-flow enforcement is more permissive than static analysis in *flow-insensitive* case (dynamic enforcement rejects less good programs than static analysis). The authors have also shown that both dynamic and static analysis guarantee the same security property: *termination-insensitive noninterference*.

In case of *flow-sensitive* analysis, Russo and Sabelfeld [51] have proven that one has to choose between soundness and permissiveness guarantees for purely dynamic monitors: the authors claimed that having both is impossible. However, the later work by Devriese and Piessens on secure multi-execution [19] demonstrated that there exists another purely dynamic technique, that is not based on monitoring, and has both soundness and permissiveness guarantees. Russo and Sabelfeld [51] have also shown though that a hybrid analysis (combining static and dynamic analysis) can be both sound and permissive. Several hybrid flow sensitive analysis were later introduced in the literature [61, 11, 19, 6] and we are going to discuss and compare them in this section.

5.2. Dynamic Data Tainting and Static Analysis

Vogt et al. [61] were the first to introduce a hybrid information flow analysis in the web browser setting. The dynamic component is the taint analysis that deals only with explicit flows. The static analysis component is invoked for implicit flows and it analyses the scope of every branch in the control flow that depends on tainted values. This analysis ensures that all the assigned variables in both executed and not executed branches are tainted accordingly.

Security policies enforced The sources of sensitive information are the sources containing "information that could be abused by an adversary to launch attacks or to learn information about the user", for example `document.cookie`, properties of `history` object, location information and others. To ensure that the tainted data does not leave the page it belongs to, authors monitor a set of data transmission operations that might send the data to a third party:

- changing the location by setting `document.location`;
- changing the source of the image on the page;
- submitting a form in the page;
- using special objects, such as `XmlHttpRequest` object.

The security policy is hence a type of information flow policy: no tainted data influences the data being sent by the operations specified above.

Formal guarantees No formal guarantees provided.

5.3. Staged Information Flow

Chugh et al. [11] proposed a staged technique to enforce information flow properties of JavaScript. The idea is to perform heavyweight static analysis of information flow of the available JavaScript code on the server side, and then use the results of this analysis in the succinct *residual checks* that are done dynamically on the client side.

This technique has one important assumption: the inlined scripts are fully trusted and are used to compute the residual policy, while the external scripts are potentially malicious. The residual check is then just a syntactic check whether the external scripts comply with the residual policy.

Security policies enforced A security policy is a confidentiality or integrity policy represented by a set of pairs for which the flow is *disallowed*. For example,

- *Integrity policy*: The first parameter in the function `post` must not be affected by any variable declared within an untrusted part of the webpage. For example, variables declared within the external script must not flow into the value of the location bar : `(*, document.location)`.
- *Confidentiality policy*: The sensitive data must not affect variables within the external script. For example, cookies must not flow into any variable within untrusted code: `(document.cookie, *)`

Notice that nowadays more and more scripts are added to the web application as external script, and even W3C suggests to move the inlined scripts and style out-of-line in its Content Security Policy specification [63]. Consider that there is no inlined script on a page, then no static information flow analysis will be done on the server side, and hence the residual policy will be identical to the original policy. Then, the residual checks that will be done on the external scripts are simple safety policies, like "JavaScript code cannot read `document.cookies`" or "JavaScript code cannot write into `document.location`".

Formal guarantees No formal guarantees provided.

5.4. Information flow control for Mashups

Mash-IF [35] is a hybrid technique for information flow control within mashups. Mash-IF consists of a labelling tool (to mark sensitive data), a reference monitor (to prevent leakage of sensitive information within the mashup) and a declassification tool based on a static analysis of the mashup components.

The static data flow analysis is done for a subset of JavaScript language named JavaScript$_{SA}$, that was used in the GATEKEEPER framework [25]. Whenever a script reads a sensitive data, a static analysis tool identifies all the execution paths that could propagate the sensitive information to the other parts of a mashup or to remote servers. These paths are recorded by their corresponding function calls. The reference monitor then compares these sequences with the monitored script's call sequences to find the potential information leakage.

Mash-IF is hence implemented as an add-on for the Mozilla Firefox and evaluated using 10 real mashups. The performance overhead for static analysis and monitor ranges from 50% to 500%, while the monitor only (assuming that the static analysis for the script has been done in the past) ranges from 7% to around 40%.

Security policies enforced The information that should not be leaked to unauthorised third parties is separated into sensitive and highly sensitive. The information flow security policy in its usual sense is applied to sensitive information, while highly sensitive information should not be accessed from other domains even locally. For example, to protect from the cross-site request forgery, cookies in Mash-IF are considered highly sensitive information since they contain user's session identifier.

Formal guarantees No formal guarantees provided.

5.5. Secure multi-execution (SME) and FlowFox

Devriese and Piessens proposed a novel technique called *Secure multi-execution* (SME) [19]. SME runs the original program multiple times, once per each security level, using special rules for I/O operations and synchronisation of communication between the runs. The authors originally proposed SME for a simple programming language, but since SME does not depend on the language semantics, it can be applied to any programming language .

The first advantage of this approach is that it has two formal guarantees: soundness guarantee in a sense of time-sensitive noninterference (which is stronger than termination-insensitive noninterference) and precision. The second advantage is that it automatically *fixes* the execution of the program, silently substituting secret values with dummy values in illegal flows.

The technique was tested on the benchmarks of the JavaScript engine. Similar approach was taken by Louw et al. [59] applied to the web advertisements. Bielova et al. [9] implemented the SME for the Featherweight Firefox model [10].

De Groef et al. [14] present an implementation of SME based on the Mozilla Firefox browser, called FlowFox. An interesting part of this implementation is the security policy language: it is able to model such policies as "`XmlHttpRequest.send` method has a level H in case the origin to where the request is sent is the same as the origin of the document the script is part of". De Groef et al. have shown the feasibility of FlowFox on Alexa top 500 websites: the behaviour of FlowFox was indistinguishable from the behaviour of FireFox with around 20% or performance cost and 88% of memory overhead (due to a double execution of the same JavaScript program).

Security policies enforced A generic information flow security policy: the security lattice can be an arbitrary lattice of security levels, and the DOM APIs are marked with security levels.

Formal guarantees This approach guarantees *soundness* in a sense of time-sensitive noninterference and a good *precision*, meaning that the technique does not change the original I/O relation for terminating runs of time-insensitive noninterferent programs except for some re-ordering.

5.6. Multiple Facets for Dynamic Information Flow

Inspired by the secure multi-execution approach, Austin and Flanagan [6] proposed a novel technique that combines the benefits of multi-execution with the efficiency of

a single execution. They introduced *faceted value* that contains raw values for each security level, and by manipulating these faceted values, a single process can *simulate* several processes of multi-execution approach. In case the raw values at different levels are identical (e.g., after exiting the scope that depends on a secret value), the two executions collapse into one, thus reducing the overhead.

Faceted evaluation is implemented in a Narcissus [20] JavaScript engine that is installed in the browser using Zaphod Firefox plugin [3]. The original formalisation of faceted evaluation presented in the paper is given for a simple programming language with references and without while loops. The ZaphodFacets implementation extends the faceted semantics to handle additional complexities of JavaScript.

Security policies enforced Similar to secure multi-execution, faceted evaluation enforces generic information flow policies.

The authors introduce a *declassification* mechanism based on robust declassification [74], which guarantees that an active attacker, who is able to introduce the code, is not more powerful than a passive attacker. In order to provide such declassification, every principal P has two associated labels: secret to P and untrusted by P. The idea is that the corresponding execution (if you think about an execution corresponding to one view) labeled as untrusted by P cannot downgrade the data that is secret to P.

Formal guarantees *Soundness* for termintation-insensitive non-interference, however no precision or transparency guarantees are given.

5.7. Dynamic type system for Information Flow Security

Hedin and Sabelfeld [29] introduce a dynamic type system that guarantees information flow security for JavaScript. The authors first propose a core of JavaScript that is interesting for the information flow perspective: objects, higher-order functions, exceptions, `eval`. A dynamic type system supports the security label upgrades similar to Austin and Flanagan's [4] *no sensitive upgrade* (NSU) discipline.

The technique also handles a number of JavaScript features, that were not covered in other language-based information flow analyses. The dynamic type system halts the program execution whenever an illegal information flow is found.

This approach is being implemented in a JavaScript interpreter written in JavaScript. However, as of the day of writing this paper, the implementation results have not been published yet.

Security policies enforced Differently from previous works on information flow security in the web browser settings, Hedin and Sabelfeld proposed to extend the notions of information that can be marked as a secret (in the other works only variables or APIs are marked by security labels):

- the information about the structure of the objects in JavaScript or about existence of particular fields;
- an existence of a variable in a particular scope.

Formal guarantees *Soundness* for termintation-insensitive non-interference and *transparency* are proven.

5.8. Comparison of enforcement

In Table 4 we compare the techniques by the following parameters: the security policy they enforce (explicit vs. implicit information flow), the formal guarantees they provide and the implementation strategy. In the first row we shown how different techniques deal with implicit and explicit flows: statically (stat) or dynamically (dyn).

Table 4: Comparison of information flow security analysis for JavaScript

	Vogt et al. [61]	Chugh et al. [11]	Li et al. [35]	Devriese and Piessens [19], De Groef et al. [14]	Austin and Flanagan [6]	Hedin and Sabelfeld [29]
Information flow						
explicit flow	dyn	stat	stat	dyn	dyn	dyn
implicit flow	stat	stat	✗	dyn	dyn	dyn
Formal guarantees						
termination-insensitive noninterference	✗	✗	✗	✓	✓	✓
time-sensitive noninterference	✗	✗	✗	✓	✗	✗
precision or transparency	✗	✗	✗	✓	✗	✓
Implementation strategy						
browser extension		✓	✓			
JavaScript engine	✓			✓	✓	✓

Notations:

stat = static analys
den = dynamic analysis

The Vogt et al. [61] approach was the first technique for JavaScript information flow, and did not provide any formal guarantees. Later on, Russo et al. [52] found unsound aspects in this work related to the structure and navigation on DOM trees. Staged information flow (SIF) [11] was a novel approach to JavaScript security, however, since most of the analysis is done statically, it is not very precise and moreover, no formal guarantees are provided. Notice that both Mash-IF and SIF are using static analysis, and both techniques are implemented as a separate tool, while a browser extension only monitors all the requests for an external script in case of SIF and all the function calls in case of Mash-IF.

The most recent and promising enforcement mechanisms for JavaScript information flow security are secure multi-execution (SME) [19, 14], faceted evaluation [6] and dynamic type system for information flow [29]. Let us compare these techniques in more details.

Implicit flows In order to deal with implicit flows dynamically, Hedin and Sabelfeld [29] use an approach similar to no-sensitive upgrade (NSU) semantics [4], while faceted evaluation was proven to generalize NSU and permissive upgrade [5] semantics. SME is

incomparable since executing the same program multiple times while filtering I/O does not impose any restrictions on the semantics that should be used.

Formal guarantees All techniques provide termination-insensitive noninterference soundness guarantee, but moreover, SME is proven to have a stronger guarantee: time-sensitive noninterference. SME also has a precision guarantee in a sense that if the program is time-insensitively noninterferent, then SME does not change its I/O relations. Dynamic type system [29] approach proves transparency: if the program is able to run (be noninterferent) in the instrumented semantics, then the run is consistent with the run of the un-instrumented semantics. Faceted evaluation approach does not have a proof of transparency or precision.

Language features SME does not have to deal with the specific features of JavaScript language because it only handles the I/O of the program. Multiple facets were modelled for a simple programming language with references and without while loops and it is not clear how many JavaScript language features can be supported. For example, it seems that exceptions cannot be handled in this approach. Dynamic type system works for a core of JavaScript that is carefully chosen: it contains higher-order functions, exceptions, eval, with and other specific JavaScript features.

Type of enforcement SME proposed a novel approach where the program execution is not halted because an illegal information flow is found. Instead, this approach *fixes* the illegal flows at runtime. Faceted evaluation technique was inspired by SME, and hence performs a similar kind of enforcement. Hedin and Sabelfeld [29] technique, being a dynamic approach, stops the execution of the program when an illegal flow is found.

Declassification Faceted evaluation introduced robust declassification mechanism claiming that other kinds of declassification can be easily integrated into their semantics. SME and Dynamic type system do not discuss declassification, however both groups of authors described how useful policies can be enforced by their techniques without declassification. As such, Hedin and Sabelfeld discuss scenarios of malicious online advertisement and user password tracking and explain how their approach would work. De Groef et al. (SME) have implemented FlowFox browser and have practically shown that their technique is able to enforce such useful policies as non-leaking of session cookies, history sniffing and tracking libraries almost without effect on the behaviour of the Alexa top 500 [2] websites.

Performance overhead While SME and Faceted evaluation approaches are currently implemented and available for download, Dynamic type system implementation is an unpublished work as of the day of writing this paper. Faceted evaluation was compared with the original SME implementation in case of different number of security levels. As expected, the performance overhead of SME grows exponentially with respect to the number of levels, while Faceted evaluation does not. However, before making any comparison of the performance overhead, it would be interesting to know: how many security levels are needed for the security of the current web applications?

5.9. Summary of information flow policies

In Section 5 we have presented and compare the main works on information flow security mechanisms for the web browser applications and JavaScript. In this section we gather the information flow policies that were enforced in these works. Most of the works are based on confidentiality policies and only Chugh et al. [11] discusses some integrity policies.

In Table 5[9] we present objects and APIs that we split into two groups: APIs that can be/are used to store sensitive information and APIs that provide communication. The idea of *confidentiality policy* is that the data that has been read from sensitive information APIs should not be sent by the communication APIs to the remote servers that do not owe this information. For example, we can use SOP to define a remote server: the sensitive information that was read in one origin cannot be sent to another origin by the communication APIs.

Table 5: Sensitive information APIs and Communication APIs for Confidentiality Policy.

APIs	Related properties
Sensitive information APIs (high input)	
Core DOM objects	
Domain	`document.domain`
Window	`window.status, window.getSelection, window.clipboardData`
Cookies	`window.cookie, document.cookie`
Referrer	`document.referrer`
History	`document.title, history.current, history.next, history.previous, object.getProperyValue`
Location	`document.location, window.location, document.URL`
Web Storage	`localStorage, sessionStorage`
IndexedDB	
Geolocation	
Mouse data	`MouseEvent.clientX, MouseEvent.clientY`
Communication APIs (low output)	
Core DOM objects	assign `src` property of a node
XmlHttpRequest	`XmlHttpRequest.open`, `XmlHttpRequest.send`
	submit() a form
Cross-origin messaging	`window.postMessage`
WebSocket	
Server-Sent Events	

Integrity policy was shown to be dual to a confidentiality policy, and hence many authors of the surveyed papers did not try to discuss or implement it in the web browser settings. Chugh et al. [11] gives some examples of integrity with respect to the location property of the window. High Integrity objects contain information that should not be modified by the JavaScript program. Integrity policy claims that low integrity values

[9] object.getProperyValue is used for history sniffing via CSS, and window.clipboardData is used to set and get the clipboard data in Internet Explorer starting from version 5.0

should not flow into high integrity objects. Low integrity values are all the variables of the JavaScript program, while high integrity objects are presented in Table 6.

Table 6: High Integrity objects for Integrity Policy.

APIs	Related properties
High Integrity objects (high output)	
Domain	`document.domain`
Location	`document.location, window.location`, `assign` and `replace` methods of the `location` object

6. Other related work on JavaScript Security

There is a number of other works done in the area of JavaScript security, and applying formal analysis to web applications and JavaScript programs. We will discuss here some of such works.

6.1. Static analysis of widgets

One of the first proposals on static analysis for JavaScript was done by Guarnieri and Livshits [25] in their framework called GATEKEEPER. They modelled the safe subset of JavaScript, the inference rules and the security policies in Datalog, and then applied bddbddb solver [68] to produce policy violations. The authors propose a flow- and context-insensitive points-to analysis for a subset of JavaScript, and several security policies that they consider important in JavaScript widgets. In this framework the security policies reflect the safety properties enforced by dynamic techniques.

As a follow-up of their work, Guarnieri and Livshits propose a staged static analysis framework GULFSTREAM [24]. The motivation for the staging approach is that the JavaScript application is never available in its entirety: as the user interacts with the application, more code is sent to the browser. With the small updates of the code, GULFSTREAM updates the results of the static analysis made for the previous version of the code. This framework proposes an implementation of a hand-coded point-to analysis using graph-based representation of the program.

6.2. Static analysis of sandboxing libraries

In order to protect its users from malicious JavaScript code, companies propose the developers to use their sandboxing libraries that would restrict the capabilities of JavaScript. As an example, FBJS was a sandboxing for Facebook, AdSafe for Yahoo!, Google Caja and Microsoft Web Sandbox for their companies. Researchers have tried to analyse the code of these libraries to see whether there implementation corresponds to the guarantees these libraries should provide.

First of all, sandboxed code should not be able to access security-critical resources. To enforce this policy Maffeis and Taly propose a language-based isolation of untrusted

JavaScript [41] and together with Mitchell they proposed isolating JavaScript with filtering, rewriting and wrapping in [39]. This work was done for JavaScript operational semantics defined in their earlier publication [38].

Different sanboxed codes should be isolated from each other, or more precisely "one third-party component must not write to a heap location that the other third-party component can read from. This policy is weaker than non-interference. It only prevents the communication via the heap." To enforce inter-component isolation Maffeis et al. proposed an object capabilities model [40].

The access to some particular resources should be given, and the hosting page can create trusted APIs for such resources. The sandboxed code should access these resources *only* through trusted APIs. This policy also leads to *API Confinement problem*: verification that sandboxed code cannot obtain a direct access to the security critical resources. Taly et al. propose an API analysis for a version of JavaScript strict mode by context-insensitive and flow-insensitive points-to analysis [57]. They were able to catch several bugs in the ADsafe library.

Independently, the group of Krishnamurthi proposed another technique to analyse JavaScript sandboxing libraries [49] and specifically address the security of ADsafe. Their analysis is based on the type system for JavaScript where the authors encode and verify sandboxing properties. This type system is based on the semantics of JavaScript called λ_{JS} defined in Guha et al. earlier work [27]. The definition of security in this work is related to ADSafe sandbox. The safety definition says: if all the embedded widgets pass the ADsafe's static checked JSLint, then 1) widgets cannot load new code at runtime, or cause ADsafe to load new code on their behalf; 2) widgets cannot affect the DOM outside of their designated subtree; 3) widgets cannot obtain direct references to DOM nodes; and 4) multiple widgets on the same page cannot communicate. The authors prove that if a widget is well-typed in their type system, then properties 1 and 3 are preserved. However, it is not possible to prove property 2 with their tool, and while verifying property 4 a bug in ADsafe library was found.

6.3. Other approaches to Mashup security

The security of web mashups is an active field of research that is strongly related to JavaScript security. We refer interested readers to the survey on web mashups by De Ryck et al. [16]. The recent sandbowing libraries techniques described above contribute to this area, however, they are sometimes not practical due to the fact that static analysis only covers a subset of JavaScript. A recent contribution to this field is due to Luo and Rezk [37], that presented *Mashic Compiler*.

The mashup consists of an integrator code and the gadgets to be added. Luo and Rezk consider the gadgets that are added by a `<script>` tag. In this case, the gadget and the integrator would get assigned the same origin (according to the Same Origin Policy) and hence, if the gadget is non-benign, it can break the security of the mashup (for example, a gadget can redefine a function and execute arbitrary malicious code).

Given the gadgets code and and the integrating code, *Mashic* compiles the integrating code in such a way that each gadget and the integrator run in their own iframe. A small library is also added to each gadget, which are otherwise unmodified. This novel approach allows the integrators to write secure mashups where security is achieved via the Same Origin Policy.

The authors prove that the compiled code is equivalent to the original code, when the gadgets are benign. The definition of a benign gadget is a novel notion that is defined through a decorated semantics. It is proven that 1) the gadgets only learn what is being sent to them by the integrator, 2) the gadgets may only interact with the integrator by replying to its messages, therefore they cannot directly modify the heap of the integrator.

6.4. Formal models of the web browsers

Bohannon and Pierce proposed *Featherweight Firefox* (FF) browser model [10] that includes many browser features such as multiple browser windows; cookies; sending HTTP requests and receiving HTTP responses; essential HTML elements; building document node trees, and also the basic features of JavaScript. It is is implemented as an executable model in OCaml, and in Coq. FF is a reactive system, with a detailed definition of the input and output events, and the internal state of the browser. Input events can either come from the user (loading a URL in a new window `load_in_new_window`, entering text in a text box `input_text`, etc.), or from the network (receiving an HTTP response `receive`). Output events can be sent to the user (web page is updated `page_updated`, window is opened `window_opened`) or to the network (sending HTTP request `send`). The FF browser model defines precisely how the browser will react to these inputs by emitting outputs. The FF model is surprisingly rich. It can represent the execution of event handlers implemented as scripts in an html page.

Akhawe et al. [1] proposed a novel way to formally analyze security on the web platform. It is implemented a subset in Alloy and supports the features like browsers with script contexts per origin, DNS, HTTP requests and responses, redirects, etc. Alloy implementation allows to translate the declarative object-modeling syntax into propositional input to a SAT solver. The SAT solver then finds the counterexample (interactions between the site and the browser) that violates the specified security goal. Using this implementation Akhawe et al. were able to find two known vulnerabilities for Origin header and Cross-Origin Resource Sharing and three new vulnerabilities for Referer Validation, HTML5 forms and WebAuth.

7. Conclusions

JavaScript security is a new area of research that grows rapidly. The first paper on dynamic techniques for JavaScript security appeared in 2005 [28], and the first information flow control for JavaScript was proposed in 2007 [61]. Because of the youth of this field, the number of contributions is relatively small, however many impressive solutions have been proposed so far and their number increased in the last couple of years. We foresee even more techniques for JavaScript security coming in the future, especially with the acceptance of HTML5 standard and new standard of JavaScript language EcmaScript-6.

In this survey we proposed a detailed comparison of the existing techniques for JavaScript security, that are based on runtime monitoring or information flow control. Both techniques first appeared as a reaction to the Cross-Site-Scripting (XSS) attacks, trying to monitor the behaviour of JavaScript programs and reveal the malicious ones. However, since for the moment of writing this paper, researchers did not come up with the definition of XSS [70], these solutions do not solve completely the XSS problem, however they still provide very good theoretical and practical results.

The security enforcement mechanisms we describe in this survey were proposed both by researchers and industry. Therefore, not all of the approaches were easily fit into the theoretical framework of runtime monitors and information flow control. Moreover, we have been rigorously analysing formal properties of the mechanisms and implementation strategies. Tables 3 and 4 and corresponding sections present our findings. Since these mechanisms were implemented for different versions of the web browsers and in different ways (often as a browser extension or as a modification of a JavaScript engine), it is very hard to compare their effectiveness and practicality. For more practical discussion on JavaScript security mechanisms, we invite the future authors to accurately compare their novel enforcement mechanisms with the existing ones by 1) following our schema: security policies enforced, formal guarantees provided, implementation strategy; 2) comparing implementation characteristics, such as runtime overhead, and, importantly, evaluation on how well the proposed techniques will work with the current websites.

Acknowledgements

We thank Tamara Rezk, Alan Shmitt, Frédéric Besson and Thomas Jensen for their valuable suggestions and feedback. We would also like to thank the anonymous referees for comments that have helped improve the paper.

References

[1] Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D., 2010. Towards a formal foundation of web security, in: Proceedings of the 23rd Computer Security Foundations Symposium (CSF'10), pp. 290–304.

[2] Alexa.com, . Alexa top websites. Available at `http://www.alexa.com/topsites`.

[3] Austin, T., . Zaphod add-on for the firefox browser. Available at `https://addons.mozilla.org/en-us/firefox/addon/zaphod/`.

[4] Austin, T.H., Flanagan, C., 2009. Efficient purely-dynamic information flow analysis, in: Proceedings of the 2009 workshop on Programming Language and analysis for security, pp. 113–124.

[5] Austin, T.H., Flanagan, C., 2010. Permissive dynamic information flow analysis, in: Proceedings of the 2010 workshop on Programming Language and analysis for security, ACM Press. pp. 3:1–3:12.

[6] Austin, T.H., Flanagan, C., 2012. Multiple facets for dynamic information flow, in: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press. pp. 165–178.

[7] Baron, D., April 2010. Preventing attacks on a user's history through css :visited selectors. Online: `http://dbaron.org/mozilla/visited-privacy`.

[8] Bauer, L., Ligatti, J., Walker, D., 2005. Edit automata: Enforcement mechanisms for run-time security policies. International Journal of Information Security 4, 2–16.

[9] Bielova, N., Devriese, D., Massacci, F., Piessens, F., 2011. Reactive non-interference for a browser model, in: Proceedings of the 5th International Conference on Network and System Security (NSS 2011), IEEE Computer Society Press. pp. 97–104.

[10] Bohannon, A., Pierce, B.C., 2010. Featherweight firefox: Formalizing the core of a web browser, in: Proceedings of the USENIX Conference on Web Application Development 2010.

[11] Chugh, R., Meister, J., Jhala, R., Lerner, S., 2009. Staged information flow for Javascript, in: Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation, ACM Press. pp. 50–62.

[12] ¡D-d¿Phung, P., Sands, D., Chudnov, A., 2009. Lightweight self-protecting javascript, in: Proceedings of ACM Symposium on Information, Computer and Communications Security (ASIACCS'09), ACM Press. pp. 47–60.

[13] De Groef, W., 2010. Conscript for firefox - developer report. Abvailable at `http://www.cqrit.be/conscript/report/conscript.pdf`.

[14] De Groef, W., Devriese, D., Nikiforakis, N., Piessens, F., 2012. Flowfox: a web browser with flexible and precise information flow control, in: Proceedings of the 19th ACM Conference on Communications and Computer Security (CCS'12), ACM Press. pp. 748–759.
[15] De Groef, W., Devriese, D., Piessens, F., 2012. Better security and privacy for web browsers: A survey of techniques, and a new implementation, in: Proceedings of the 8th International Workshop on Formal Aspects in Security and Trust, Springer-Verlag. pp. 21–38.
[16] De Ryck, P., Decat, M., Desmet, L., Piessens, F., Joosen, W., 2010. Security of web mashups: a survey, in: Proceedings of The 15th Nordic Conference in Secure IT Systems (NordSec'10), Springer-Verlag. pp. 223–238.
[17] De Ryck, P., Desmet, L., Philippaerts, P., Piessens, F., 2011. A security analysis of next generation web standards. Technical Report. European Network and Information Security Agency (ENISA).
[18] Denning, D.E., Denning, P.J., 1977. Certification of programs for secure information flow. Communications of the ACM 20, 504–513.
[19] Devriese, D., Piessens, F., 2010. Non-interference through secure multi-execution, in: Proceedings of the 2010 IEEE Symposium on Security and Privacy, IEEE Computer Society Press. pp. 109–124.
[20] Eich, B., . Narcissus-js implemented in js. Available on `https://github.com/mozilla/narcissus`.
[21] Elrad, T., Filman, R.E., Bader, A., 2001. Aspect-oriented programming - introduction. Communications of the ACM 44, 29–32.
[22] Falcone, Y., 2010. You should better enforce than verify, in: Proceedings of the 9th International Workshop on Runtime Verification (RV'10), Springer-Verlag Heidelberg. pp. 89–105.
[23] Flanagan, D., 2011. JavaScript: The Definitive Guide. O'Reilly Media. 6 edition. Print ISBN:978-0-596-80552-4, ISBN 10: 0-596-80552-7, Ebook ISBN:978-1-4493-0212-2, ISBN 10: 1-4493-0212-2.
[24] Guarnieri, S., Livshits, B., 2010. Gulfstream: staged static analysis for streaming javascript applications, in: Proceedings of the USENIX Conference on Web Application Development 2010, Usenix Association. pp. 6–6.
[25] Guarnieri, S., Livshits, V.B., 2009. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code, in: Proceedings of the 18th USENIX Security Symposium (USENIX Security'09), Usenix Association. pp. 151–168.
[26] Guernic, G.L., 2007. Confidentiality Enforcement Using Dynamic Information Flow Analyses. Ph.D. thesis. Kansas State University.
[27] Guha, A., Saftoiu, C., Krishnamurthi, S., 2010. The essence of javascript, in: Proceedings of the 24rd European Conference on Object-Oriented Programming, pp. 126–150.
[28] Hallaraker, O., Vigna, G., 2005. Detecting malicious javascript code in mozilla, in: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05), pp. 85 – 94.
[29] Hedin, D., Sabelfeld, A., 2012. Information-flow security for a core of javascript, in: Proceedings of the 25rd Computer Security Foundations Symposium (CSF'12), IEEE Press. pp. 3–18.
[30] Hunt, S., Sands, D., 2006. On flow-sensitive security types, in: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA. pp. 79–90.
[31] Janc, A., Olejnik, L., 2010. Feasibility and real-world implications of web browser history detection, in: Proceedings of WEB 2.0 Security and Privacy 2010 Workshop.
[32] Jang, D., Jhala, R., Lerner, S., Shacham, H., 2010. An empirical study of privacy-violating information flows in javascript web applications, in: Proceedings of the 17th ACM Conference on Communications and Computer Security (CCS'10), ACM Press. pp. 270–283.
[33] Johns, M., 2008. On javascript malware and related threats. Journal in Computer Virology 4, 161–178.
[34] Kikuchi, H., Yu, D., Chander, A., Inamura, H., Serikov, I., 2008. Javascript instrumentation in practice, in: The Sixth ASIAN Symposium on Programming Languages and Systems, Springer-Verlag. pp. 326–341.
[35] Li, Z., Zhang, K., Wang, X., 2010. Mash-IF : Practical Information-Flow Control within Client-side Mashups, in: Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010), IEEE Press. pp. 251–260.
[36] Ligatti, J., Bauer, L., Walker, D., 2009. Run-time enforcement of nonsafety policies. ACM Transactions on Information and System Security 12, 1–41.
[37] Luo, Z., Rezk, T., 2012. Mashic compiler: Mashup sandboxing based on inter-frame communication, in: IEEE (Ed.), Proceedings of 25th IEEE Computer Security Foundations Symposium, Cambridge, MA, USA. pp. 157–170.
[38] Maffeis, S., Mitchell, J., Taly, A., 2008. An operational semantics for JavaScript, in: The Sixth

ASIAN Symposium on Programming Languages and Systems, Springer-Verlag. pp. 307–325. See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.

[39] Maffeis, S., Mitchell, J., Taly, A., 2009. Isolating javascript with filters, rewriting, and wrappers, in: Proceedings of the 14th European Symposium on Research in Computer Security, Lecture Notes in Computer Science. pp. 505–522.

[40] Maffeis, S., Mitchell, J.C., Taly, A., 2010. Object capabilities and isolation of untrusted web applications, in: Proceedings of the 2010 IEEE Symposium on Security and Privacy, IEEE Computer Society Press. pp. 125–140.

[41] Maffeis, S., Taly, A., 2009. Language-based isolation of untrusted javascript., in: Proceedings of the 2009 IEEE Computer Security Foundations Symposium, IEEE Computer Society Press. pp. 77–91.

[42] ietf-http-wg mailinglist, 1995-03-09. Re: Referer: (sic). In reply to Roy Fielding, `http://lists.w3.org/Archives/Public/ietf-http-wg-old/1995JanApr/0109.html`.

[43] Mayer, J.R., Mitchell, J.C., 2012. Third-party web tracking: Policy and technology, in: Proceedings of the 2012 IEEE Symposium on Security and Privacy, IEEE Computer Society Press. pp. 413–427.

[44] Meyerovich, L., Livshits, B., 2010. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser, in: Proceedings of the 2010 IEEE Symposium on Security and Privacy, IEEE Computer Society Press. pp. 481–496.

[45] Microsoft, . Mitigating cross-site scripting with HTTP-only cookies.

[46] Nikiforakis, N., Invernizzi, L., Kapravelos, A., Acker, S.V., Joosen, W., Kruegel, C., Piessens, F., Vigna, G., 2012a. You are what you include: Large-scale evaluation of remote javascript inclusions, in: Proceedings of the 19th ACM Conference on Communications and Computer Security (CCS'12), ACM Press. pp. 736–747.

[47] Nikiforakis, N., Meert, W., Younan, Y., Johns, M., Joosen, W., 2011. Sessionshield: Lightweight protection against session hijacking., in: Proceedings of the International Symposium on Engineering Secure Software and Systems 2011, Springer-Verlag. pp. 87–100.

[48] Nikiforakis, N., Van Acker, S., Piessens, F., Joosen, W., 2012b. Exploring the ecosystem of referrer-anonymizing services, in: Proceedings of the 2012 Privacy Enhancing Technologies Symposium, Springer-Verlag. pp. 259–278.

[49] Politz, J.G., Eliopoulos, S.A., Guha, A., Krishnamurthi, S., 2011. Adsafety: Type-based verification of javascript sandboxing, in: Proceedings of the 20th USENIX Security Symposium (USENIX Security'11), Usenix Association.

[50] Roesner, F., Kohno, T., Wetherall, D., 2012. Detecting and defending against third-party tracking on the web, in: Proceedings of The 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12).

[51] Russo, A., Sabelfeld, A., 2010. Dynamic vs. static flow-sensitive security analysis, in: Proceedings of the 23rd Computer Security Foundations Symposium (CSF'10), IEEE Computer Society Press, Washington, DC, USA. pp. 186–199.

[52] Russo, A., Sabelfeld, A., Chudnov, A., 2009. Tracking information flow in dynamic tree structures, in: Proceedings of the 14th European Symposium on Research in Computer Security, Springer-Verlag. pp. 86–103.

[53] Sabelfeld, A., Myers, A.C., 2003. Language-based information-flow security. IEEE Journal on Selected Areas in Communication 21, 5–19.

[54] Sabelfeld, A., Russo, A., 2009. From dynamic to static and back: riding the roller coaster of information-flow control research, in: Proceedings of the 7th international Andrei Ershov Memorial conference on Perspectives of Systems Informatics (PSI'09), Springer-Verlag, Berlin, Heidelberg. pp. 352–365.

[55] Schneider, F., 2000. Enforceable security policies. ACM Transactions on Information and System Security 3, 30–50.

[56] Singh, K., Moshchuk, A., Wang, H.J., Lee, W., 2010. On the incoherencies in web browser access control policies, in: Proceedings of the 2010 IEEE Symposium on Security and Privacy, IEEE Computer Society Press. pp. 463–478.

[57] Taly, A., Erlingsson, U., Mitchell, J.C., Miller, M.S., Nagra, J., 2011. Automated analysis of security-critical javascript apis, in: Proceedings of the 2011 IEEE Symposium on Security and Privacy, IEEE Computer Society Press. pp. 363–378.

[58] Tang, S., Dautenhahn, N., King, S.T., 2011. Fortifying web-based applications automatically, in: Proceedings of the 18th ACM Conference on Communications and Computer Security (CCS'11), ACM Press. pp. 615–626.

[59] Ter Louw, M., Ganesh, K., Venkatakrishnan, V., 2010. AdJail : Practical enforcement of confidentiality and integrity policies on web advertisements, in: Proceedings of the 19th USENIX Security

Symposium (USENIX Security'10), Usenix Association. pp. 371–388.
[60] Van Acker, S., De Ryck, P., Desmet, L., Piessens, F., Joosen, W., 2011. Webjail: Least-privilege integration of third-party components in web mashups, in: Proceedings of 27th Annual Computer Security Applications Conference, ACM New York. pp. 307–316.
[61] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G., 2007. Cross-site scripting prevention with dynamic data tainting and static analysis, in: Proceedings of the Symposium on Network and Distributed System Security (NDSS'07), The Internet Sociery.
[62] Volpano, D., Smith, G., Irvine, C., 1996. A sound type system for secure flow analysis. Journal of Computer Security 4, 167–187.
[63] W3C, a. Content security policy 1.1. W3C Editor's Draft 02 August 2012. Online: `http://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`.
[64] W3C, b. XMLHttpRequest Level 2. W3C Working Draft. Retreived on 2012-02-20 at `http://www.w3.org/TR/XMLHttpRequest/`.
[65] W3C, 24 May 2012. Indexed database API. W3C working draft. `http://www.w3.org/TR/IndexedDB/`.
[66] W3C, 29 March 2012. A vocabulary and associated apis for html and xhtml. Online: `http://dev.w3.org/html5/spec/single-page.html#history`.
[67] W3C, May 2012. Geolocation api specification. Available at `http://dev.w3.org/geo/api/spec-source.html`.
[68] Whaley, J., Avots, D., Carbin, M., Lam, M.S., 2005. Using datalog with binary decision diagrams for program analysis, in: The ASIAN Symposium on Programming Languages and Systems (APLAS'05), Springer. pp. 97–118.
[69] WHATWG, . HTML Living Standard. Web Storage. Last Updated 29 June 2012. `http://www.whatwg.org/specs/web-apps/current-work/multipage/webstorage.html\#webstorage`.
[70] Wilander, J., 2012. Is XSS solved? Available at `http://appsandsecurity.blogspot.fr/2012/11/is-xss-solved.html`.
[71] Yu, D., Chander, A., Islam, N., Serikov, I., 2007. Javascript instrumentation for browser security, in: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press. pp. 237–249.
[72] Zalewski, M., 2011a. Browser security handbook part 1-3. Online: `http://code.google.com/p/browsersec/wiki/Main`.
[73] Zalewski, M., 2011b. The Tangled Web: A Guide to Securing Modern Web Applications. ISBN 9781593273880.
[74] Zdancewic, S., 2003. A type system for robust declassification. Electronic Notes in Theoretical Computer Science 83.