Toward Principled Browser Security

Edward Z. Yang[1] Deian Stefan[1] John Mitchell[1] David Mazières[1] Petr Marchenko[2] Brad Karp[2]
[1]Stanford University [2]University College London

ABSTRACT

To ensure the confidentiality and integrity of web content, modern web browsers enforce isolation between content and scripts from different domains with the same-origin policy (SOP). However, many web applications require cross-origin sharing of code and data. This conflict between isolation and sharing has led to an ad hoc implementation of the SOP that has proven vulnerable to such attacks as cross-site scripting, cross-site request forgery, and browser privacy leaks. In this paper, we argue that information flow control (IFC) not only subsumes the same-origin policy but is also more flexible and sound. IFC not only provides stronger confidentiality and integrity for today's web sites, but also better supports complex sites such as mashups, which are notoriously difficult to implement securely under the SOP.

1 INTRODUCTION

Web applications are deceptively simple to build: easy to get working, but challenging to secure. When written the "straightforward" way, they are at risk of being vulnerable to a wide range of attacks. Cross-site scripting (XSS) and cross-site request forgery (CSRF) are widely known problems, but there are far more subtle menaces: image resources may leak whether or not a user is logged in [9]; a third-party script may go rogue and steal users' credentials; or, given HTML5 support for smartphone sensors, a malicious script may surveil a user through their microphone or camera. The web browser is the common enabler of all these attacks.

The same-origin policy (SOP) was meant to provide confidentiality and integrity by isolating web pages of distinct origins [3]. A browser that perfectly enforced a strict SOP would be invulnerable to the aforementioned attacks: XSS would no longer be able to exfiltrate cookies to third-party websites, CSRF would no longer be able to initiate cross-site requests, and third-party images and style sheets would be disallowed. Alas, no one would want to use such a browser: cross-origin communication is vital to rich web applications. Indeed, ad hoc security policies helped the web succeed: they allowed a simple mechanism for serving marked-up static content to evolve into a full-blown distributed application platform, with downloadable code and dynamic content from multiple parties. The vulnerabilities that arose as browsers grew in functionality, however, have only been addressed piecemeal, e.g., with Cross-Origin Resource Sharing (CORS) [42] and Content Security Policy (CSP) [40]. It is time to re-examine browser security with the benefit of hindsight.

What would a more principled design for isolation between origins look like—one that precludes behavior that leads to vulnerabilities, but provides enough flexibility to allow *desirable* cross-origin behavior? We begin with the observation that *the SOP is an information flow control (IFC) policy.* An IFC mechanism labels data, tracks its flow through a system, and enforces policies that allow or deny flows on the basis of their labels. While its creators may not have thought of it in such terms, the SOP is merely a highly restrictive IFC policy in which flows between origins are denied. But IFC is far more general than the SOP, as it is capable of expressing policies that precisely constrain inter-origin flows. Such policies may be constructed to be more restrictive than today's ad hoc, permeable SOP; or, in some cases, they may be even more permissive—today's SOP may disallow flows that are safe in certain circumstances and stand in the way of useful web application functionality. Our thesis is that IFC is a good fit for *whole-browser* security: it captures the SOP nicely while supporting more perfect isolation than today's SOP. It not only helps eliminate existing vulnerabilities but also enables applications that today might be considered impossible to implement securely.

In this paper, we offer a brief overview of IFC and show how to use it to express today's browser security policies—any *deployable* browser security primitive must at least meet that bar. We then use concrete examples to highlight how IFC supports rich functionality with strong security guarantees impossible in today's browsers, including *thwarting a malicious browser extension* (§3.4), whose elevated privilege today permits it to leak all browser tabs' content to a third-party site; an *untrusted, extensible JavaScript image editor* (§4.2), which today risks exfiltrating an image edited on the client to a third-party site; and a *third-party mashup* (§4.3), which today risks leaking content from any site that contributes to the mashup to a third party.

2 A SHORT INTRODUCTION TO IFC

IFC systems track and control the propagation of information by associating a *label* with every object. A label encodes a security policy as a pair of positive boolean formulas over *principals* (e.g., origins) specifying who may read or write data [37]. For example, an object labeled ⟨canRead: `a.com` ∨ `b.com`, canWrite: `a.com`⟩ can only be read by `a.com` or `b.com` and modified by

a.com.[1]

In general, labels are partially ordered according to a *can flow to* relation $\sqsubseteq$; for any labels L_A and L_B, if $L_A \sqsubseteq L_B$ then the policy encoded by L_A is *upheld* by that of L_B. For example, data labeled $L_A = \langle$canRead: a.com $\vee$ b.com, canWrite: a.com$\rangle$ can be written to an object $L_B = \langle$canRead: a.com, canWrite: a.com$\rangle$ since L_B preserves the secrecy of L_A. (In fact, L_B is *more* restrictive, as only a.com—not both a.com and b.com—can read.) Conversely, $L_B \not\sqsubseteq L_A$, and thus data labeled L_B cannot be written to an object labeled L_A (data secret to a.com cannot be leaked to b.com).

To keep track of communication restrictions, every execution context (e.g., window, frame, etc.) is associated with a label, called the *current label*. The current label reflects the sensitivity of all data read by the code (by accumulating taint when performing reads [29, 32, 39, 45]) and is used to restrict communication rights. Code can read an object only if the object's label can flow to the current label; code can write to an object only when the current label can flow to the object's label.

Within the confines of the browser, code can also act on behalf of principals by exercising *privileges*. A privilege is an unforgeable object which asserts authority over a principal and can be used to allow flows otherwise not permitted by the $\sqsubseteq$-relation. For example, code executing on behalf of a.com with a current label of $\langle$canRead: a.com$\rangle$ cannot inspect the result of an aggregate, mashup computation labeled $\langle$canRead: a.com $\wedge$ b.com$\rangle$ (note the conjunction!), since the object may contain information sensitive to b.com. However, a.com and b.com can independently exercise their privileges to *declassify* the data. For example, a.com can downgrade the object's label to $\langle$canRead: b.com$\rangle$, after which b.com can declassify it to $\langle$canRead: **anybody**$\rangle$, i.e., make it publicly readable. Code can further *endorse* data by exercising a privilege—e.g., b.com may copy an object and add itself to the copy's integrity label, thus indicating that it vouches for the content of the object.

3 Core Browser-Level IFC

We propose using IFC in the browser core as well as exposing an LIO-like [38] API in JavaScript.[2] In this section we argue for the applicability and benefits of LIO-style IFC to the browser core; in §4 we detail the benefits of exposing these IFC abstractions in JavaScript.

[1] For brevity we use hostnames in place of origin URIs (e.g., a.com in place of http://a.com:80). When only secrecy or integrity is of interest we simplify notation by excluding the other component, which is implicitly **anybody**. $\vee$ is logical disjunction; $\wedge$ indicates conjunction.

[2] Previous work either only implements IFC for JavaScript (omitting the C++ browser core) [12, 16] or implements IFC throughout the browser, but does not expose IFC to JavaScript code [7], and thus doesn't support fine-grained labeling or declassification—both vital to mashups.

For IFC to be applicable in the browser core, an important prerequisite is that existing websites operate without modification. Hence, current browser policies must be expressible in and enforceable with IFC. Below we consider how to express several representative policies.

3.1 Same-Origin Policy

The same-origin policy specifies that resources of an origin should only be readable by content from the same origin [3, 42, 46]; we refer to this definition as the *strict* SOP. However, many exceptions to the SOP allow cross-origin communication. For instance, the SOP does not restrict the embedding of images, scripts, styles and frames from arbitrary domains: an origin can read cross-origin data by inspecting image properties (e.g., size), executing scripts that call back with data (e.g., JSON-P [35]), etc. Furthermore, the SOP does not forbid performing cross-origin requests: thus, it is trivial to write data to a third-party website. In the browser, cross-origin windows and frames can communicate bidirectionally by setting a common document.domain, by modifying the window.name property, using fragment-id messaging [8], or using the postMessage API [19].

Expressing the Strict SOP with IFC As described in §2, IFC restricts the flow of information according to the labels associated with data and endpoints. Hence, the first step toward expressing the SOP with IFC is to properly label browser entities. Below we show the labeling of several representative components.

- **User-credentials** stored by the browser, such as cookies, HTML5 local storage, and HTTP authentication information, are labeled according to their origins and attributes. For instance, a cookie set by a.com with attributes Domain=a.com; HttpOnly is labeled $\langle$canRead: a.com $\wedge$ core://, canWrite: a.com $\wedge$ core://$\rangle$ to indicate that it can be read and modified by the browser core running on behalf of a.com. The core:// term ensures that only the core, which runs with the core:// privilege (and not JavaScript), can read and modify the cookie.
- **Contexts**, e.g., windows and frames, have a current label and privileges corresponding to their content's origin. For example, a frame loaded from a.com is labeled $\langle$canRead: a.com$\rangle$ and owns the privilege a.com. As mentioned previously, the browser additionally owns the special core:// privilege, giving it access to certain objects "hidden" from JavaScript.
- **Messages** sent using postMessage from a.com's context to b.com's context are labeled $\langle$canRead: a.com $\vee$ b.com, canWrite: a.com$\rangle$. This allows both origins to inspect the message and verify that a.com sent the message. Importantly, the message label *always* indicates the *intended recipient* and thus

follows the safe `postMessage` API [6].

- **Content**, including images and scripts, is labeled using the content's origin and type (e.g., images from `a.com` have the label ⟨canRead: `a.com` ∨ `#img`⟩).[3]

Recall that information is permitted to flow from an entity labeled L_A to an entity labeled L_B only if $L_A \sqsubseteq L_B$. Under our labeling scheme, code typically runs with a current label of the form ⟨canRead: *A*-origin⟩ while a resource's label has the form ⟨canRead: *R*-origin⟩. Hence, to read the resource it must be that ⟨canRead: *R*-origin⟩ $\sqsubseteq$ ⟨canRead: *A*-origin⟩, which holds only if $R = A$. In other words, our labeling scheme restricts code to reading and writing to the same origin, as per the strict SOP.

Extending the Strict SOP with Declassification Although the above scheme captures the strict SOP, the full SOP allows reading resources from different origins, sending messages (with `postMessage`) without intended recipients, etc. Such cross-origin communication violates *non-interference* [33]: information from an origin influences the execution of code in some other origin. Hence, to encode the full SOP we must explicitly encode such exceptions using *declassification*.

Consider an `img` element on a page in `a.com` with the `src` property set to `http://b.com/lena.png`. Without declassification, our system would not allow this image to be fetched and loaded, since the label of the resource does not flow to the label of the context (⟨canRead: `b.com`⟩ $\not\sqsubseteq$ ⟨canRead: `a.com`⟩). Because browsers' implementations of the SOP allow this flow, we must declassify the resource by downgrading the label of the image from ⟨canRead: `b.com`⟩ to ⟨canRead: **anybody**⟩, which now flows to the context label. The full SOP combines labels and declassifiers, which specify circumstances when labels can be downgraded without exercising a privilege.

Discussion As detailed above, IFC can be used to specify and enforce real-world security policies. Policies like the strict SOP that enforce non-interference map naturally to IFC, while policies that may leak information must employ declassifiers. New browser features explicitly opt out of the strict same-origin policy with a declassifier—and these declassifiers force one to analyze a feature's implications for users' security and privacy.

As an example, consider the design of the `postMessage` API. When using the labeling scheme described above, our IFC system enforces both integrity and confidentiality for messages exchanged between components. Recall that when `a.com` sends `b.com` a message, its label will be ⟨canRead: `a.com` ∨ `b.com`, canWrite: `a.com`⟩. The label indicates the intended recipients and message creator. Hence, even if a message is sent to the wrong origin, such as `evil.com` in Figure 1, the origin will not be able to read the message because ⟨canRead: `a.com` ∨ `b.com`, canWrite: `a.com`⟩ $\not\sqsubseteq$ ⟨canRead: `evil.com`⟩.

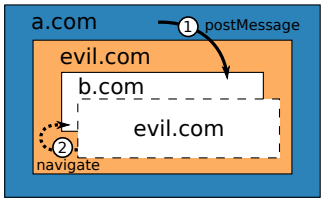

Figure 1: Message hijacking, where `a.com` sends a message to `b.com`, but `evil.com` hijacks the frame and replaces `b.com`.

Unfortunately, `postMessage` allows messages to be sent without specifying the intended recipient. In order to implement this behavior, we must downgrade the label of every message with a declassifier before sending it to another frame. In the above example, the message label ⟨canRead: `a.com` ∨ `b.com`, canWrite: `a.com`⟩ is downgraded to ⟨canRead: **anybody**, canWrite: `a.com`⟩. Since ⟨canRead: **anybody**, canWrite: `a.com`⟩ $\sqsubseteq$ ⟨canRead: `evil.com`⟩, a hijacked frame can now read the message—an unintended leak. Discovering such leaks is typically non-trivial [6], however, with IFC, leaks are accompanied by explicit declassification.

Declassification can also serve as a warning signal in more subtle cases. Consider the previous scenario, where `a.com` loads `http://b.com/lena.png`. When requesting the image, the browser sends cookies to `b.com`'s server. Suppose that the image is private, and depending on the cookies, the server may respond with the image or redirect to the login page. Since the SOP does not restrict a script from determining if an image load succeeds, the label of this response is downgraded to ⟨canRead: **anybody**, canWrite: **anybody**⟩. However, this flow leaks information about the user's credentials (labeled ⟨canRead: `b.com`, canWrite: `b.com`⟩) to `a.com`: the website can tell whether or not the user is logged in!

In general, declassification of non-opaque objects is unsafe. Sensitive information can be retrieved by inspecting object properties (e.g., `height` and `width` in the case of images) or attaching event handlers [9]. Importantly, declassifiers are added as explicit exceptions to the strict SOP—this places the burden of validating safety on the declassifier author. In the case of existing features, focusing attention on declassifiers may reveal yet unknown leaks. For new features, IFC forces developers to reason about and justify violations of the strict SOP.

3.2 Cross-Origin Resource Sharing

CORS [42] defines a protocol on top of HTTP intended to enable client-side cross-origin requests that the SOP prohibits. Specifically, an origin `b.com` can permit a cross-origin request (e.g., from `a.com`) by including the allowed origins in a dedicated header.

Expressing CORS with IFC is straightforward. To en-

[3] For brevity, we henceforth omit the element types from labels.

force the SOP we label a resource solely with the *single* principal corresponding to the origin of the server it lives on (e.g., ⟨canRead: `b.com`⟩); for CORS, we simply include all the allowed origins in the label. For example, a resource on `b.com` with the header `Access-Control-Allow-Origin: a.com` is labeled ⟨canRead: `a.com` ∨ `b.com`⟩, allowing both origins to access the resource.

3.3 Content Security Policy

CSP [40] addresses some of the shortcomings of the SOP that have led to attacks such as XSS by allowing web apps to specify where different resources (scripts, images, etc.) may be loaded from. For example, if a page from `a.com` supplies the CSP header with the directives `default-src: 'self'; img-src: b.com`, the page is restricted to loading images from `a.com` and `b.com`.

Interestingly, most policies in CSP are information flow policies—i.e., they specify the origins from where the page may fetch resources or load scripts (and thus to which it may potentially leak information). We can implement CSP atop our IFC system by modifying the implementation of the declassification method of §3.1. Specifically, instead of fully downgrading the label of a resource before retrieving it, CSP directives can be used to control the level of declassification. For instance, if content from `a.com` wished to fetch `http://b.com/lena.png`, the SOP would fully downgrade the label of the resource from ⟨canRead: `b.com`⟩ to ⟨canRead: **anybody**⟩. Conversely, with CSP, the declassifier must take into consideration the `img-src` directive. If the directive contains the origin `b.com`, the label will again be downgraded to ⟨canRead: **anybody**⟩ and the request to `b.com` will be performed. On the other hand, if the directive contains only origin `c.com`, the downgrade is a no-op and the image will not be fetched—this ensures that `a.com`'s data cannot be leaked to `b.com` through the request parameters.

3.4 New Policies

Browser-based IFC enables novel security policies, as well, three of which we now briefly consider.

Preventing Exfiltration by Browser Extensions Chrome and Firefox browser extensions are JavaScript code that executes with elevated privilege—e.g., they typically enjoy read access to tabs of all origins [1, 14, 29]. Many need network access as well. AdBlock, for example, which prevents the display of ads in pages, requests updated ad identification information from a remote server. A maliciously written extension can thus exfiltrate information from a user's tab to a remote server [29]. The Hails system [15] applies IFC to prevent untrusted code executing on a web server from disclosing sensitive information to unauthorized parties. We expect IFC to be similarly useful to block such exfiltration by extensions, whether malicious or vulnerable to script injection by malicious pages [24, 25, 34].

Protecting the User's Browser History Numerous attacks exfiltrate a user's browsing history [2, 9, 21]. To address them, most of today's browsers disallow scripts from reading the CSS `:visited` selector value with `getComputedStyle`. In our IFC framework, we need not take so ad hoc a measure—we can simply label history data with a distinguishable label (e.g.,`user://`). Under this IFC-based policy, the renderer must declassify the history data (labeled ⟨canRead: `user://`⟩) before incorporating it into layout. The declassifier highlights possible leaks and suggests that browser history data must only cause *local* changes to layout, as is also implemented in current browsers.

Preventing Self-Exfiltration Attacks In a self-exfiltration attack [10], an attacker leaks data to the same "trusted" origin where the user is logged in, for example, by posting a public comment. As in Hails [15], our IFC approach allows server-side code to label a response—and thus set the current label—using a dedicated header, `X-LIO-Label`, to indicate that it contains user-sensitive data. Since subsequent client-side requests will be at least as sensitive (the current label cannot be arbitrarily lowered), the server can inspect the request label and disallow publicly observable modifications.

4 Exposing IFC in JavaScript

The benefits of IFC go beyond the browser core; exposing the strong underlying IFC protection mechanism in JavaScript allows developers to build more secure applications, libraries and mashups. Developers can take advantage of this functionality using the LIO API [38, 39].

At its core, LIO exposes three functions: (i) `label`, used to explicitly wrap objects with labels; (ii) `unlabel`, used to unwrap labeled objects and taint the execution context; and, (iii) `toLabeled`, which is used to execute arbitrary code in a disjoint ("sandboxed") compartment without tainting the current context. This "IFC as a library" approach differs from prior work on JavaScript IFC (e.g., [16, 18, 36, 44]) in several ways. First, the approach is coarse-grained in the common case (e.g., we only associate a label with each global), while still allowing developers to protect individual objects (with `label`). Second, it does not impose IFC on pages that do not use the API, ensuring that existing websites do not break. Third, it allows code to exercise privileges (e.g., to declassify data): this is useful when building practical IFC-based applications [22, 31, 45]. Below we consider several use cases that illustrate the benefits an LIO-style IFC system would offer JavaScript.

4.1 Preventing CSRF Attacks

The SOP does not prevent websites from making POST requests to other websites; thus, CSRF attacks arise when a website fails to check the authenticity of received POST requests. The usual way to protect against CSRF attacks is to include a special token in every form and rely on the SOP to prevent other origins from forging a request with it [4]. Unfortunately, the SOP cannot prevent such a token from being leaked by accident (as in [43]) or malice (if the website is vulnerable to XSS attacks). IFC can be used both to protect the token's secrecy (by labeling it with the form's origin) and to prevent forged requests by associating a "high-integrity" label with each form. Browser-core IFC can guarantee that that the POST request will propagate the form label (e.g., using a special header), which, on the server side, can be verified by the victim website.

4.2 Executing Untrusted Code on User Data

Most web applications rely on third-party JavaScript libraries. Developers typically place complete trust in this code, since it is not generally possible to guarantee that arbitrary code will not exfiltrate private data. Systems such as Caja [30], ADSafe [11], and FBJS [13] offer sandboxing by defining safe subsets of JavaScript; these subsets tend not to support JavaScript's full functionality, and as retrofits atop existing browser interfaces, they are also vulnerable to various attacks [26, 27, 41].

Consider an in-browser photo editor that allows users to integrate third-party code (e.g., as bookmarklets) that extends its core functionality. Systems like Caja can prohibit these third-party libraries from communicating over the network and guarantee that potentially sensitive user data (photo) are not exfiltrated. But this approach limits its functionality: it also prevents libraries from fetching clip-art, stock photographs, fonts, etc. from remote websites. By contrast, IFC is a natural fit for securing such applications: we can prevent the leaking of a user's sensitive data while still allowing code the flexibility to fetch resources from remote hosts. Specifically, the core editor, which we assume to be trusted, can label the photo (with `label`) such that its label does not flow to arbitrary origins. Third-party library functions can then safely be called with the labeled photo (e.g., by using `toLabeled`). These library functions do not have the privilege corresponding to the label the core editor allocated and thus must respect it: while they may initiate network requests to fetch resources prior to inspecting the photo, once they have `unlabel`ed the photo, their communication rights will be revoked [38].

4.3 Safe Mashups

Mashups compose content from different websites. In many cases, a third-party developer independent of the content provider offers a mashup. Privacy is a concern when running such services on sensitive data, and despite much effort to make mashups safe, there remains a tradeoff between privacy and functionality [5, 20, 28].

Consider, for instance, a mashup that matches Amazon purchases with bank statements.When one party (the bank or Amazon) provides the mashup, the solution is generally to bypass the SOP, whether by iframe and cross-origin communication with `postMessage`, or by using CORS to grant one provider access to the user's data on the other. Unfortunately, these approaches require the user to trust the mashup provider not to exfiltrate the user's data from the other site. Moreover, users must give up their privacy in order to use mashups written by third-party developers. Even authorization services such as OAuth 2.0 [17] do not prevent applications from arbitrarily leaking data post-authorization [15, 23].

IFC, however, supports client-side mashups well. Exposing an LIO-like IFC system in JavaScript allows us to extend the `XMLHttpRequest` object so that code can safely perform arbitrary cross-domain requests, labeling responses by their origins (as described in §3.2). Untrusted third-party code can thus fetch data from Amazon, which the browser labels ⟨canRead: `amazon.com`⟩, and if it does not unlabel the result, such as to inspect purchases, it can subsequently perform a request to the bank website. Once the responses are unlabeled, however, the current label will be ⟨canRead: `amazon.com` ∧ `bank.ch`⟩. At this point, the script will not be able to initiate further requests (to any origin), but can still match the purchases with bank statements.

5 Conclusion

We believe that information flow control is a promising replacement for existing browser security mechanisms. It is not only expressive enough to support existing policies and reveal latent vulnerabilities, but also enables rich applications with strong security properties that are unachievable today. IFC is both a mechanism that supports novel applications and a conceptual tool for understanding core browser security. The rapid expansion of browser functionality has begotten a wide range of threats to users' privacy and a patchwork of browser security mechanisms that do not robustly ensure privacy. Drawing upon these lessons, we have made the case for IFC as a principled and flexible browser security primitive.

Acknowledgments We thank Amit Levy, Patrick Mutchler, and the anonymous reviewers for their helpful comments. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by the EPSRC under grant EP/K032542/1, and by multiple gifts from Google (to Stanford and UCL). Deian Stefan is supported through the NDSEG Fellowship Program.

REFERENCES

[1] S. Bandhakavi, S. King, P. Madhusudan, and M. Winslett. VEX: Vetting browser extensions for security vulnerabilities. In *Usenix Security*, 2010.

[2] L. D. Baron. Preventing attacks on a user's history through CSS :visited selectors. `http://dbaron.org/mozilla/visited-privacy`, 2010. Visited January 7, 2012.

[3] A. Barth. The web origin concept. Technical report, IETF, 2011. URL `https://tools.ietf.org/html/rfc6454`.

[4] A. Barth, C. Jackson, and J. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88, 2008.

[5] A. Barth, C. Jackson, and W. Li. Attacks on javascript mashup communication. In *Proceedings of the Web*, volume 2, 2009.

[6] A. Barth, C. Jackson, and J. Mitchell. Securing frame communication in browsers. *Communications of the ACM*, 52(6):83–91, 2009.

[7] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 97–104. IEEE, 2011.

[8] J. Burke. Browser and Dojo updates on fragment ID messaging. `http://tagneto.blogspot.com/2008/01/browser-and-dojo-updates-on-fragment-id.html`, 2008. Visited January 31, 2008.

[9] M. Cardwell. Abusing HTTP status codes to expose private information. `https://grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information`, 2011. Visited January 10, 2013.

[10] E. Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson. Self-exfiltration: The dangers of browser-enforced information flow control. *Web 2.0 Security and Privacy*, 2012.

[11] D. Crockford. ADsafe. `http://www.adsafe.org/`, 2012. Visited January 8, 2013.

[12] V. Djeric and A. Goel. Securing script-based extensibility in web browsers. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[13] Facebook. FBJS: Facebook JavaScript. `https://developers.facebook.com/docs/fbjs/`, 2012. Visited January 8, 2012.

[14] A. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proc. of the USENIX Conference on Web Application Development*, 2011.

[15] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–60. USENIX, 2012.

[16] W. D. Groef and D. Devriese. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[17] D. Hardt. The OAuth 2.0 Authorization Framework. Technical report, IETF, 2012. URL `https://tools.ietf.org/html/rfc6749`.

[18] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 3–18. IEEE, 2012.

[19] I. Hickson. HTML5 web messaging. `http://www.w3.org/TR/webmessaging/`, 2012. Visited January 5, 2012.

[20] J. Howell, C. Jackson, H. Wang, and X. Fan. Mashupos: Operating system abstractions for client mashups. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 32–42, 2007.

[21] A. Janc and L. Olejnik. Feasibility and real-world implications of Web browser history detection. *Proceedings of W2SP*, 2010.

[22] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st Symp. on Operating Systems Principles*, October 2007.

[23] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A world wide web without walls. In *6th ACM Workshop on Hot Topics in Networking (HotNets)*, Atlanta, GA, November 2007.

[24] Lostmon. Gmail checker plus Chrome extension XSS. `http://lostmon.blogspot.co.uk/2010/06/gmail-checker-plus-chrome-extension-xss.html`, 2010. Visited January 10, 2013.

[25] Lostmon. Notifier for Google Wave Chrome extension XSS/CSRF. http://lostmon.blogspot.co.uk/2010/06/notifier-for-google-wave-chrome.html, 2010. Visited January 10, 2013.

[26] S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*, pages 77–91, 2009.

[27] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 125–140. IEEE, 2010.

[28] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proceedings of the 5th ACM symposium on information, computer and communications security*, pages 15–23. ACM, 2010.

[29] P. Marchenko, U. Erlingsson, and B. Karp. Keeping sensitive data in browsers safe with ScriptPolice. Technical Report RN/13/02, UCL, January 2013.

[30] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript, 2008. Technical report, Google, 2009.

[31] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Computer Systems*, 9(4):410–442, October 2000.

[32] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security (NDSS) Symp.*, 2005.

[33] A. Sabelfeld and A. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[34] Secunia Advisory. Dokodemo Rikunabi 2013 unspecified cross-site scripting vulnerability. https://secunia.com/advisories/48813, 2010. Visited January 10, 2013.

[35] K. Simpson. Defining safer JSON-P. http://json-p.org/, 2011. Visited November 16, 2011.

[36] K. Singh, S. Bhola, and W. Lee. xbook: Redesigning privacy control in social networking platforms. In *USENIX Security*, 2009.

[37] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *16th Nordic Conference on Security IT Systems, NordSec*, volume 7161 of *LNCS*, pages 223–239. Springer, October 2011.

[38] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, pages 95–106. ACM SIGPLAN, September 2011.

[39] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2012.

[40] B. Sterne and A. Barth. Content Security Policy 1.0. http://www.w3.org/TR/CSP/, 2012. Visited January 5, 2013.

[41] A. Taly, J. C. Mitchell, M. S. Miller, J. Nagra, et al. Automated analysis of security-critical javascript apis. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 363–378. IEEE, 2011.

[42] A. Van Kesteren. Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/, 2012.

[43] B. Vibber. CSRF token-stealing attack (user.tokens). https://bugzilla.wikimedia.org/show_bug.cgi?id=34907, 2012. Visited January 9, 2012.

[44] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with bflow. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 233–246. ACM, 2009.

[45] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

[46] M. Zelwski. Browser security handbook, part 2. HTtp://code.google.com/p/browsersec/wiki/Part2, 2011. Visited March 30, 2011.