CDF: Predictably Secure Web Documents

Peter Snyder*, Laura Watiker†, Cynthia Taylor*, Chris Kanich*
*{psnyde2, cynthiat, ckanich}@uic.edu
Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607

†lwatiker@world.oberlin.edu
Department of Computer Science
Oberlin College
Oberlin, OH 44074

***Abstract*—Users wishing to protect their privacy and security on the modern web are asked do the impossible: to only send information to trusted hosts, to be careful with the functionality they expose to websites (in the form of plugins or advanced browser capabilities), and to generally only interact with "safe" websites. Such recommendations require non-expert web users to audit the origins a website will fetch resources from, and the browser features those sites will access. These recommendations are, in practice, impossible for non-expert web users.**

In this work we propose CDF, a novel document format for describing interactive websites. CDF provides client–enforced security and privacy guarantees by constraining the types of browser features web authors can access. In contrast to the current approach of using arbitrary JavaScript to deliver interactivity, CDF provides website authors a set of safe interactive primitives they can compose into rich, interactive websites. This approach allows for the types of interactive web experiences users expect, without the unpredictability or risk of arbitrary JavaScript or unconstrained HTML.

We evaluate the usability of this approach by implementing several different web applications, selected to be representative of the types of websites users regularly encounter. We found that we could replicate nearly all user-facing functionality in CDF.

I. Introduction

The web was initially designed to deliver mostly static documents. Assets requested from the server were mainly unchanging files that could be easily verified to be of a set structure and have a predictable effect when rendered by the client. Over time, the web has moved from being a text based document retrieval system to a robust application platform, with multiple programming languages, systems for delivering and executing byte code (Java, Flash, Unity, etc), and an ever increasing list of web-native APIs and features.

The current approach to creating interactive websites gives web authors a great deal of control and power over the environment of users' browsers. However, the web's flexibility and feature-set comes with a significant cost to web users' security and privacy. The web's dynamic nature and reliance on JavaScript makes it difficult, if not impossible, for users to predict the effect of visiting a website before the site has already been delivered to the client, even if they only wished to view some inert content like a blog post or photograph. No matter how simple a site's intended functionality might be, modern browsers expose one's machine to an incredibly broad collections of features and functionalities. While this design enables frictionless use of new features and new web experiences, each of these features increases the browser's attack surface and thus the potential for a malicious site to access a bug which causes a serious security breach.

JavaScript adds a large amount of risk to the web. Malicious websites use JavaScript to attack users through heap sprays, access Web API features that have known security and privacy vulnerabilities, perform fingerprinting of users, and generally send sensitive information to unknown and untrusted domains. In general, JavaScript is a significant part of the reason why it is difficult to evaluate the safety of a website before visiting it, and thus why it is difficult to give easil understood guidelines to non–expert users on how to understand and enact internet safety.

A security-concerned user could, in theory, browse the web without allowing JavaScript or plugins to run. However, much of the modern web would be severely broken for this user. If they did wish to gain access to any of the useful functionality of certain sites—to control a drop down menu or load newly posted comments from the server—they would be forced to allow these sites the permission to execute any of the esoteric, complicated, or unproven functionality that is made available to every website. Here we see a major issue with the modern web: permission to execute can not be granted on the basis of functionality, and at best can be granted on a per-site basis.

A more secure approach would reduce the amount of trust a user must give to use the web. Instead of receiving and executing arbitrary code delivered by the server, users could receive documents that by design cannot contain code. These safer documents could consist of only descriptions of safe and trusted functionality, from a finite list of safe functions. Clients who desire improved security when surfing the web could then limit the functionality exposed to untrusted sites to only those known-safe features. Such a scenario would allow page authors to describe safe and interactive web sites without the client needing to trust the web site to provide the page's implementation.

This paper presents CDF, or Contained Document Format, a

redesign of the client side language for describing responsive and secure interactive hypermedia documents. The design is intended to serve as a proof of concept that a hypermedia based communication medium with a generic design can provide the features necessary for a large subset of the popular uses of today's web, without the security and privacy risks posed by the web's current JavaScript-based design.

II. Related Work

Several attempts have been made to make JavaScript from untrusted sources safe to run in the browser. The most drastic of these, NoScript [1] disallows all JavaScript, Java or Flash on any sites that the user has not designated as trusted; of course, this drastically limits the functionality of untrusted sites. Static JavaScript analysis of third party scripts [2], [3], [4], [5], [6] has good performance, but is inherently limited by undecidability. Another technique is to transform untrusted JavaScript, adding run-time checks [7], [8], [9], [10]. These mechanisms restrict difficult-to-handle semantics, but can make debugging difficult and code execute slower. Other techniques seek a more flexible authorization control, which specifies what types of operations are allowed [11], [12], [13], [14]. We note that the above techniques either require fine-grain authorization or loss of functionality (i.e. breakage).

Static analyzers can be used to rewrite JavaScript; they are useful for developers who want to provide safe widgets. While not sufficiently flexible to support large, third party libraries [7], [5], [2], [9], they can be useful in constrained environments. AdSafe [15] inspects guest JavaScript code for patterns that allow unsafe access to the global "document" object or rely upon implied global variables. Untrusted code using such features is flagged as unsafe and not allowed to run on the page, trading off functionality for security.

JavaScript can also be constrained at runtime to prevent or mitigate security breaches. Google Caja [16] creates a virtual environment which can detect unsafe or privacy leaking behaviors. This approach is conceptually similar to automatically instrumenting C code with buffer overflow protections, and similarly trades off performance for predictability and security. Code injection defenses (like disabling inline scripts) can also be encapsulated in explicit policies [17] or inferred for legacy code with automated methods [18].

There has also been significant work adding user-controlled privacy features to the browser. The Tor onion router provides protection with respect to traffic analysis, but does not encrypt end to end, and does not anonymize client-to-server leakages [19], [20]. To provide a comprehensive solution, the Tor Browser Bundle is distributed with a version of Firefox customized to provide more privacy. The common inclusion of third party social plugins allows the plugin providers to track users throughout their browsing session, and researchers have built systems that mitigate this tracking capability [21]. Ghostery [22] both identifies and reports tracking packages to the user, and blocks cookies, in order to guarantee the user greater privacy.

III. Motivation and Design

CDF is an alternative method of creating modern, interactive websites, but with greater security and privacy guarantees than the current HTML-and-JavaScript system provides. The principal features in the design of CDF are as follows.

First, CDF prevents websites from running arbitrary code in the client browser. Instead, CDF authors create interactive websites by composing trusted, client-controlled implementations of interactive web functionality using an easily checked, declarative syntax. **Second**, CDF only uses a subset of browser features, allowing websites to access only the "core" or most popular and frequently used tools for creating interactive web sites. **Third**, CDF places stricter constraints on web documents than current HTML-and-JavaScript applications enforce, to better protect user privacy and security.

Table I provides a comparison of the capabilities and guarantees made by current HTML-and-JavaScript based applications, contrasted against CDF documents. The following subsections provide more detail about each aspect of the design of CDF.

A. Trusted Feature Implementation

CDF's main method for improving user security and privacy is by preventing websites from executing arbitrary JavaScript in the browser. The current JavaScript-based system for providing interactive websites is the cause of many web browser security problems. Web browsers must trust that code will carry out some non-malicious purpose when executing it, and that a given set of JavaScript instructions will benefit the user (by, for example, setting up a website's user interface elements) instead of harming the user (e.g. by fingerprinting the user, accessing a browser feature with a known security flaw, or sending a session token to a remote server).

Instead of the difficult to secure JavaScript approach, CDF provides a set of trusted, client-side implemented interactive primitives, and allows websites to compose them using a declarative, easy to verify syntax. CDF authors can, for example, tie a *mouse click* event to a *document attribute change* event, not by writing code directly, but through the structure of the document. CDF clients include their own trusted libraries that handle generating code and executing the relevant functionality on the clients, without trusting code provided by the website.

The result is that CDF documents are composed from functionality implemented in trusted, client-controlled libraries. These libraries are designed to compose safely, and pages can only access them through a simple, declarative syntax. This is in contrast to the typical JavaScript based approach, where websites can execute arbitrary code, and web browsers must judge if the resulting behavior seems safe through heuristics like XSS filters and code origin reputation systems.

B. CDF Feature Selection

CDF also protects user security and privacy by reducing the browser's attack surface by preventing websites from accessing browser functionality that is either rarely used, or predominantly used for advertising and tracking.

Capability	HTML + JS	CDF
Load static media from remote and local domains	✓	✓
Load non-client controlled JavaScript	✓	-
Can express common web design idioms	✓	✓
Server control over HTTP `referer` and related privacy settings	✓	-
Client guarantees over HTTP `referer` and related privacy settings	-	✓
Read sensitive values from cookies, local storage, etc.	✓	-
Sub-page / AJAX requests and updates	✓	✓
Allow form submissions and AJAX updates to remote domains	✓	-
HTML5 multimedia (`<audio>`, `<video>`)	✓	✓
Supports common browser plugins (Flash, Java, Silverlight)	✓	-
Advanced JavaScript tools (WebGL, `<canvas>`, ASM.js)	✓	-
Client side storage (IndexDB, localStorage, etc)	✓	-
Offline Applications	✓	✓

TABLE I
FEATURE COMPARISON BETWEEN HTML AND CDF DOCUMENT FORMATS.

Modern web browsers implement a huge array of features, which site authors access through JavaScript APIs (either the DOM or the Web API). Some features are closely related to the web's core purpose of interactive documents. Examples of such features include the *DOM Level 2: Core* standard [23], used to manipulate and inspect an HTML document with JavaScript, or the *XMLHttpRequest* standard [24], which allows websites to make sub-document updates to the origin domain without refreshing the page.

Other browser features are more esoteric and only loosely related to the goal of providing interactive documents. For example, modern web browsers implement the *Web Audio* standard [25], which allows websites to perform full audio synthesis, the *Ambient Light* standard [26], which allow websites to access any light sensors on the device, and the *WebRTC* standard [27], which allows web browsers to create peer-to-peer networks.

Research done by Snyder et al [28] found that a small number of browser features, primary related to document-manipulation and updating, were frequently used on the web, while the majority of JavaScript-accessed features were never used by websites. The same work found that even more browser features became rarely used when filtering out advertising and tracking related usage.

CDF improves user security and privacy by only allowing websites access to the frequently used, document-manipulation related features in the browser. By preventing websites from accessing features that do not generally provide user serving interests (either because those features are primary used for advertising and tracking, or because the features are rarely used at all), CDF brings web browsers into closer alignment with the security principal of least privilege. The attack surface exposed to websites is dramatically reduced, with minimal impact to the user experience.

C. Document Constraints

HTML-based applications include several other design aspects that make them difficult to secure. HTML and JavaScript based applications allow scripts to be loaded from remote locations from any part of the HTML document, enabling many XSS attacks. HTML documents can contain full sub-documents through the use of `<iframe>` elements, enabling drive by downloads and related attacks. And HTML applications generally include a "referer" header when requesting remote resources, enabling some forms of user tracking.

CDF improves user security and privacy by tightly-controlling what kinds of resources documents can fetch, and what information is sent during the fetch request. CDF documents cannot include arbitrary code (either inline or hosted remotely), include sub-documents, or send information generated in the client directly to remote domains.

IV. IMPLEMENTATION

We implemented CDF in two parts, first as a document specification, and second as several additions to the browser's trusted base: a *parser* that converts CDF documents into trusted HTML and JavaScript, a *HTTP proxy* that converts CDF documents for use in web browsers, and a set of *trusted JavaScript libraries* that run in the browser to implement the interactive aspects of CDF documents.

The described system was implemented to allow CDF documents to be run in web browsers today, with no additions or modifications needed to any recently released browser. The same design could be implemented by modifying a browser to be able parse and understand CDF documents "natively", though at the cost of a much greater engineering task.

We also adopted cascading style sheets, or CSS, to handle the presentation of CDF applications. We did so to minimize the engineering effort needed to implemented the CDF concept, and because of the relative lack of security issues associated with CSS compared to JavaScript.

This section gives a high level explanation of how our implementation of the CDF design works. Documentation for creating CDF documents, including type specifications, nesting rules, and the interactivity primitives included in CDF can be found in an open source implementation and accompanying documentation[1].

[1]https://github.com/bitslab/cdf.

A. Document Format

CDF uses JSON strings to represent documents. CDF documents are trees of typed objects. Types in CDF fall into one of four categories.

- **Elements.** The structure and text of the document.
- **Events.** New input from the network or the user.
- **Behaviors.** Descriptions of what should happen when an Event has triggered.
- **Deltas.** Changes to be applied to the document.

Each type defines the configuration it can receive (e.g. the URL that a `image` object can refer to), and the types it accepts as children in the tree. For example `text` objects can be children of `button` objects (to create labels on buttons), but `button` objects cannot be children of `text` objects. Since the types in CDF are all well defined, they can be strictly checked to ensure they will have predictable effects when rendered in the client.

Some types accept configuration parameters (e.g. the class names to add to the element when rendered in HTML, or the local URL to post a form's information to). These configuration parameters are also strictly typed, and so can be checked for safety and correctness before being rendered in the client.

Types are designed to emphasize predictable information flow and user privacy. For example, in CDF `form` elements are only allow to send information to the origin domain, while in HTML applications, `<form>` elements can be configured to send information to any domain.

B. Trusted Base Additions

We implemented the CDF design through three additions to the current trusted web browser trusted base. These additions, in tandem, enforce the security and privacy properties discussed in Section III.

1) Parser: The first addition CDF makes to the browser's trusted base is a CDF parser. The role of the CDF parser is to take strings and either identify them as invalid CDF documents, or to render an equivalent and safe HTML and JavaScript string that can be rendered in the browser. The parser also provides debugging information as a convenience to CDF authors.

If the parser is given a valid CDF document, it converts it into a combination of HTML tags, escaped text, `<script>` tags referencing JavaScript libraries that are part of the CDF trusted base, and `<script>` tags containing parameters to be passed to those trusted libraries. Invalid documents "fail closed", and return an error code and no output.

2) HTTP Proxy: The second addition CDF makes to the browser's trusted base is an HTTP proxy that sits between the browser and the internet. The proxy passes requests from the browser to the destination server unchanged. Once the server responds, the proxy examines the response. If the response appears to be a CDF document, the HTTP proxy extracts the body of the request and provides it to the parser. If the parser accepts the response as a valid CDF document, the proxy passes the parser-generated HTML and JavaScript back to the client. If the parser rejects the server's response as invalid CDF, the proxy instead passes back an error message to the client, informing the user that the server provided an invalid document.

3) Client JavaScript Libraries: The third addition to the browser's trusted base is in a small number of JavaScript libraries (14) that implement the interactive elements of each page. These libraries handle all the client-side logic and functionality needed for all of the event, behavior and delta types used in the system, plus some plumbing code needed to route the parameters extracted by the parser to the correct library implementations.

V. Evaluation

We tested the usability and expressiveness of CDF by implementing several popular types of web applications in the the system. We selected these applications (a blog modeled on http://www.vogue.com/, an online-banking site based on https://www.bankofamerica.com/, a social media site modeled on https://twitter.com/, and a collaborative web application similar to HotCRP [29]) to represent the range of sites that web users commonly interact with. In each case we were able to replicate the user-facing functionality of each page.

This section evaluates the security benefits of CDF's approach for describing interactive websites. For each issue, we briefly describe a vulnerability common in current web applications, and then describe how CDF improves the situation.

A. Cross-Site Scripting

Cross-Site scripting (XSS) refers to when attackers are able to inject JavaScript code into an HTML document, so that the code is executed by all site visitors, trusted as if the code came from the site author. The technique is used for many malicious purposes, including extracting session tokens from the client or redirecting the user to a domain the attacker controls.

CDF protects the client from XSS attacks. First and most significantly it removes the ability for a document to describe any kind of JavaScript code directly. Instead of arbitrary code, CDF documents can only describe a composition of trusted, safe types. While a malicious attacker could possibly corrupt a target server to present visitors a different composition of types than the application author intended, CDF's types constrain the functionality that can be described to only safe activities. CDF does not include, for example, anyway to access cookie values or redirect the client in JavaScript to another location, common goals of XSS attacks.

B. Page Alteration / Defacement

When application authors do not adequately sanitize or validate the inputs users provide to their site, they risk giving users the ability to deface, or otherwise unexpectedly alter, the presentation of their website. This can lead to a blurring of the line between a message provided by the page author (which may be trusted by site visitors) and other web site visitors (which may be untrusted). This may happen when a naive application author concatenates the user's input, represented as a string, into a larger string the author is using for the returned content.

CDF's type system makes this kind of error more difficult to make. The CDF author must construct pages as trees of instances of types. Overall page structure and styling cannot be modified from within an individual child node in the document tree. In cases where page authors are taking inputs from users, and anticipate that input to be in the form of an unstructured piece of text (such as a comment on an article), the page author would do so by setting the user's input string as the content of a `text` element. When CDF then renders the document to send to visitors of the site, the CDF parser escapes all content in `text` instances to ensure that the content cannot change the structure of the page (such as by including JavaScript code or altering the balance of tags on the page).

While CDF does not make this kind of attack impossible (it is possible to conceive of ways that a sufficiently naive page author would construct a vulnerable document), it makes the attack much more difficult to pull off. Instead of becoming relatively easy for page authors to be affected by this kind of attack, CDF instead makes it difficult and less likely.

C. Limited Trusted Base

A further source of vulnerability in HTML documents is that they allow attackers to take advantage of a greatly expanded trusted base, in the form of browser plugins like Java and Flash, and in the form of infrequently used Web API features. As the frequent rate of browser updates shows, securing just the browser is an extremely difficult task. Needing to trust the browser *in addition to* closed source, third party plugins with long histories of exploitability makes the problem of securing the web dramatically more difficult.

CDF further reduces the attack surface by removing the ability of CDF documents to include or refer to plugins. As previously discussed, CDF does not include any way to represent an `<object>`, `<embed>` or `<iframe>` tag on the page, nor does it have a `<script>` type that could be used to include the same client side. Earlier in the web's evolution, popular features like audio and video could only be provided by these third party plugins. Now that the web has matured and all popular browsers support standards for audio and video with hardware-accelerated playback, the absolute necessity of these extensions is limited. The CDF specification makes it impossible for CDF page authors to reference or interact with any plugins that might be on the system.

D. Client Side Fingerprinting

Web users who have not authenticated or intentionally identified themselves to a website expect to be semi-anonymous. Once a user discards any identifying tokens they've received from a website (e.g. deleting their browser cookies) they have a reasonable expectation that they are no longer known to the site. Such assumptions are even built into the state-less nature of HTTP, and required the addition of cookies to add state into the web.

Malicious websites violate this assumption through client side fingerprinting, or by including JavaScript code in their pages to take a large number of quasi-identifiying measurements, and combine them in such a way that site visitors can be uniquely identified. These quasi-identifiers are not sensitive to users deleting their cookies, modifying their user agent string, or taking other similar steps, making it difficult for users to regain their privacy.

While not all of these techniques rely on client executed JavaScript code, many do, such as canvas based fingerprinting [30], [31], identifying the JavaScript engine being used[32] or font and plugin enumeration [33]. CDF prevents these client-side fingerprinting techniques by removing the ability of page authors to include code that takes the relevant measurements. For example, there is no way for a CDF author to construct a CDF document that will query the versions of what plugins are installed on the system, or to use the `<canvas>` tag to take semi-uniquely-identifying measures of the visitors browser. By removing the ability of document authors to include arbitrary JavaScript in their pages, and by making it impossible to create documents that take the same identifying measures, CDF prevents client-side fingerprinting and increases the amount of anonymity users can expect.

E. Predictable Information Flow

A final threat to the privacy and security of web users is that it is difficult, if not impossible, for the average user to predict what information they are sharing when they visit a website, and where that information is being sent. A user may visit a website on a domain they trust—and not intending to trust any other domains in doing so—only to later learn that the site (maliciously or unknowingly) notified a third party that they visited the site.

CDF addresses this issue in three ways. First, the most popular and intrusive tracking systems used today rely, at least in part, on JavaScript run on the client. Inclusion of third party tracking libraries is inexpressible in CDF, and thus the user automatically gains a great deal of privacy-preservation.

Second, the CDF parser sets the `Content-Security-Policy` of all documents to `referrer never` though an included `<meta>` tag element, instructing browsers to not send a referrer header when requesting remote resources, further protecting the privacy of the user.

Finally, it is extremely difficult for web users to be sure where their content will be sent when they submit a form on a webpage, regardless of whether that form is to perform a search or login in a sensitive website. Even inspecting the source HTML of the page being viewed is no guarantee, since JavaScript could have manipulated where the form values will be sent. CDF removes these uncertainties by only allowing forms and sub-page requests to send to the current domain.

Tracking pixels which load from third party domains with unique per-user IDs in their URL are still usable in CDF. While this allows some level of tracking to persist in CDF, a third party providing Google Analytics style functionality would need to synchronize the user IDs with every colluding site, rather than rely on JavaScript, cookies, and referer headers to reconstruct user browsing history.

VI. Conclusion and Future Work

CDF was designed to meet the needs of today's web users, with the intention that it can live alongside the traditional web as a more secure alternative. Our implementation is not meant as a formal or complete specification of CDF, but rather a proof of concept that a minimal collection of features can enable a sufficient amount of expressiveness without sacrificing the flexibility necessary to implement many sites that modern users visit every day. Our full implementation of the translating proxy and authorship tools are available as unencumbered open source software,[2] and it is our desire that the community continues to explore the full breadth of the design space for the next iteration of the web, so that the best balance can be found between security and expressiveness.

VII. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1405886. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] G. Maone, "Noscript - javascript/java/flash blocker for a safer firefox experience!" https://noscript.net/, 2015, [Online; accessed 08-February-2015].

[2] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, "ADsafety: type-based verification of JavaScript sandboxing," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 12–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028079

[3] S. Maffeis and A. Taly, "Language-based isolation of untrusted JavaScript," in *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, ser. CSF '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 77–91. [Online]. Available: http://dx.doi.org/10.1109/CSF.2009.11

[4] S. Maffeis, J. C. Mitchell, and A. Taly, "Run-time enforcement of secure JavaScript subsets," in *In Proc of W2SP09. IEEE*, 2009.

[5] S. Guarnieri and B. Livshits, "Gatekeeper: mostly static enforcement of security and reliability policies for javascript code," in *Proceedings of the 18th conference on USENIX security symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 151–168. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855778

[6] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for JavaScript," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 50–62. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542483

[7] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Caja: Safe active content in sanitized JavaScript." Google white paper. http://google-caja.googlecode.com, 2007.

[8] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer, "Talking to strangers without taking their candy: isolating proxied content," in *Proceedings of the 1st Workshop on Social Network Systems*, ser. SocialNets '08. New York, NY, USA: ACM, 2008, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/1435497.1435502

[9] Microsoft Corporation, "Microsoft Web Sandbox," http://www.websandbox.org/, 2010.

[10] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "Browsershield: vulnerability-driven filtering of dynamic HTML," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 61–74. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298462

[11] J. Terrace, S. R. Beard, and N. P. K. Katta, "Javascript in javascript (js.js): sandboxing third-party scripts," in *Proceedings of the 3rd USENIX conference on Web Application Development*, ser. WebApps'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342863.2342872

[12] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan, "Adjail: practical enforcement of confidentiality and integrity policies on web advertisements," in *Proceedings of the 19th USENIX conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 24–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929820.1929852

[13] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 601–610. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242654

[14] L. A. Meyerovich and B. Livshits, "Conscript: Specifying and enforcing fine-grained security policies for JavaScript in the browser," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 481–496. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.36

[15] D. Crockford, "Adsafe," http://www.adsafe.org/, 2011.

[16] M. S. Miller, "Google caja," https://developers.google.com/caja/, 2013.

[17] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International Conference on World Wide Web*. ACM, 2010, pp. 921–930.

[18] P. Saxena, D. Molnar, and B. Livshits, "Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, 2011, pp. 601–614.

[19] R. Dingledine, N. Mathewson, and P. F. Syverson, "Tor: The second-generation onion router," in *Proceedings of the 13th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, Aug. 2004, pp. 303–320. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec04/tech/dingledine.html

[20] ——, "Deploying low-latency anonymity: Design challenges and social factors," *IEEE Security & Privacy*, vol. 5, pp. 83–87, 2007.

[21] G. Kontaxis, M. Polychronakis, A. D. Keromytis, and E. P. Markatos, "Privacy-preserving social plugins," in *Proceedings of the 21st USENIX Security Symposium (August 2012)*, 2012.

[22] D. Cancel, "Ghostery, inc," https://www.ghostery.com/, 2015, [Online; accessed 16-May-2015].

[23] A. L. Hors, P. L. Hgaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne, "Document object model (dom) level 2 core specification," https://www.w3.org/TR/DOM-Level-2-Core/, 2000.

[24] A. van Kesteren, "Xmlhttprequest," https://xhr.spec.whatwg.org/, 2016, [Online; accessed 10-May-2016].

[25] P. Adenot, C. Wilson, and C. Rogers, "Web audio api," https://www.w3.org/TR/webaudio/, 2013.

[26] D. Turner and A. Kostiainen, "Ambient light events," https://www.w3.org/TR/ambient-light/, 2105.

[27] A. Bergkvist, D. C. Burnett, C. Jennings, A. Narayanan, and B. Aboba, "Webrtc 1.0: Real-time communication between browsers," https://www.w3.org/TR/webrtc/, 2016, [Online; accessed 11-August-2016].

[28] P. Snyder, L. Ansari, C. Taylor, and C. Kanich, "Browser feature usage on the modern web," in *Proceedings of the 2016 Internet Measurement Conference*, 2016.

[29] E. Kohler, "Hotcrp conference management software," http://www.read.seas.harvard.edu/~kohler/hotcrp/, 2014, [Online; accessed 16-May-2015].

[30] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," *Proceedings of W2SP*, 2012.

[31] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 674–689.

[32] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting information in javascript implementations," *Proceedings of W2SP*, 2011.

[33] P. Eckersley, "How unique is your web browser?" in *Privacy Enhancing Technologies*. Springer, 2010, pp. 1–18.

[2] https://github.com/bitslab/cdf