Data-Confined HTML5 Applications

Devdatta Akhawe
Frank Li
Warren He
Prateek Saxena
Dawn Song

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-20
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-20.html

March 23, 2013

Data-confined HTML5 Applications

Devdatta Akhawe
Univ. of California, Berkeley
dev@cs.berkeley.edu

Frank Li
Massachusetts Inst. of Tech.
frankli@mit.edu

Warren He
Univ. of California, Berkeley
warrrenhe@cs.berkeley.edu

Prateek Saxena
Natl. Univ. of Singapore
prateeks@comp.nus.edu.sg

Dawn Song
Univ. of California, Berkeley
dawnsong@cs.berkeley.edu

Abstract

Rich client-side applications written in HTML5 proliferate diverse platforms such as mobile devices, commodity PCs, and the web platform. These client-side HTML5 applications are increasingly accessing sensitive data, including users' personal and social data, sensor data, and capability-bearing tokens. To fulfill their security and privacy guarantees, these applications need to maintain certain data-confinement invariants. *These invariants are not explicitly stated in today's HTML5 applications and are enforced using ad-hoc mechanisms. The complexity of web applications, coupled with hard-to-analyze client-side languages, leads to low-assurance data-confinement mechanisms in which the whole application needs to be in the TCB to ensure data-confinement invariants.*

We propose a new mechanism called a data-confined sandbox *or DCS. A DCS enables complete mediation of communication channels in a high-assurance, small-TCB manner. Our primitive extends currently standardized primitives and has negligible performance overhead and a modest compatibility cost to retrofit into existing applications. We re-implement four real-world HTML5 applications with our proposed design with a small amount of effort, achieving much stronger data-confinement guarantees. We also study over twenty HTML5 applications and find that data-confinement invariants are implicit in the vast majority of them and crucial to achieving their privacy expectations.*

1. Introduction

Rich client-side HTML5 applications—including browser extensions [15], packaged browser applications (Chrome Apps) [14], Windows 8 Metro applications [35], and applications in newer browser operating systems (B2G [4], Tizen [46])—are fast proliferating on diverse computing platforms. These applications run with access to sensitive user data, such as browsing history, personal and social data, and financial documents, as well as capability bearing tokens that grant access to these data. A recent study reveals that 58% of the 5,943 Google Chrome browser extensions studied require access to the user's browsing history, and 35% request permissions to the user's data on all websites [13]. In addition, the study found that 67% of 34,370 third-party Facebook applications analyzed have access to the user's personal data. HTML5 applications also form a significant chunk of mobile applications; a recent survey found that 75% of smartphone applications on Google Play and in the Apple App Store are HTML5-based applications [45]. These applications execute with access to the same sensor data available to native applications, including private data from GPS receivers, accelerometers, and cameras.

Applications handling sensitive data need the ability to *verifiably confine* data to specific principals and to prevent it from leaking to malicious actors. On one hand, the developers want an easy, high-assurance way to confine sensitive data; on the other, platform vendors and security auditors want to verify sensitive data confinement. For example, consider LastPass, a real-world HTML5-based password manager with close to a million users [33]. By design, LastPass only stores an encrypted version of the user's data in the cloud and decrypts it at the client side with the user's master password. It is critical that the decrypted user data (i.e., the clear-text password database) never leave the client. We term this requirement a *data-confinement invariant*. Data-confinement invariants are prevalent in a wide variety of applications: for instance, a client-side web application that manages user documents (such as the Dropbox [19] client-side code) needs to ensure that user documents are only sent to trusted (i.e., `dropbox.com`) servers. Data-confinement invariants are fundamental security specifications that limit the flow of sensitive data to a trusted set of security principals. These data-confinement invariants are not explicitly stated in today's HTML5 applications but are implicitly necessary to preserve their privacy and security guarantees.

Confining data in an HTML5 application is a challenging problem. One approach to enforcing confinement invariants on sensitive data is to encrypt the data and distribute its decryption keys only to the intended principals. However, this approach is often undesirable, because attackers can exploit security vulnerabilities and implementation errors in code handling sensitive data to violate data-confinement invariants. If an attacker compromises a principal handling sensitive data, the attacker gains access to its decryption key; the attacker can then decrypt and leak data to untrusted parties. As a real-world example, a cross-site scripting vulnerability was recently found in LastPass which allowed for the theft of the user's decrypted password database [11]. An alternative approach is to confine the communication of the code handling sensitive data to an explicitly allowed set of principals. This provides safety even when the application may be compromised due to software vulnerabilities. There are two key challenges in making this alternative approach practical: (1) to identify and isolate subcomponents of an application that access sensitive data, and (2) to provide complete mediation on the data communication channels of these isolated subcomponents. Prior work with this approach focuses on applications on commodity OS platforms. For example, mediation of data communication channels using system call sandboxing techniques has been shown to be relatively straightforward for current binary applications [32, 41]. Previous

work also developed techniques to automate identification of subcomponents that process sensitive data [7, 8, 32].

Data confinement in client-side HTML5 applications has not received much prior attention. We observe two hurdles that hinder practical data confinement in existing client-side HTML5 applications. First, mechanisms to specify and enforce data-confinement invariants are absent in HTML5 platforms; as a result, they remain hidden in application designs. Second, client-side HTML5 applications (including browser extensions, HTML5 web applications and Windows 8 Metro applications) have numerous channels to communicate with distrusting principals, and no unified monitoring interface like the OS system call interface exists. Previous proposals include numerous mechanisms that limit cross-origin communication channels like the `iframe sandbox` [3], Content Security Policy [43], HTTP Strict Transport Security [26, 29], web workers [27], code analysis [1, 34], or code rewriting [10].As we explain in §2.3, none of these offer comprehensive mediation, and most of them are not easy to retrofit into existing applications at a low compatibility cost [49].

Data-Confined Sandboxes. We introduce the data-confined sandbox (or DCS), a novel security primitive for client-side HTML5 applications. A data-confined sandbox is a unit of execution, such as code executing in an `iframe`, the creator of which explicitly controls all the data imported and exported by the DCS. Each DCS executes with no privileges in its own temporary origin, which is distinct from all other origins.[1] Simply isolating subcomponents, say into `iframes` hosted at temporary origins, is not sufficient—`iframes` do not have any data-confinement properties. Compromised code in an `iframe` can leak sensitive data to the network via image, script, style, frame, and anchor tags in addition to a number of client-side channels like `postMessage` and fragment ID messaging. Therefore, in our design, the creator of a DCS receives a clean security reference monitor interface to interpose on all communications, privileged API accesses and input/output data exchanges originating from the sandbox.

Data-confined sandboxes are a fundamental primitive to enabling a data-centric security architecture for emerging client-side HTML5 applications. By moving much of the application code handling sensitive data to data-confined sandboxes, we can enable applications that have better resilience to privacy violating attacks and that are easy to audit by security analysts.

In making this mechanism practical, we make two additional contributions. First, we implement our data-confinement primitive in the Mozilla Firefox web browser. Our changes to the web browser are less than 350 lines of code. Our data-confinement primitive addresses explicit channels and not covert channels, side channels, or self exfiltration channels; however, we show that it is still useful for a large class of privacy conscious HTML5 applications. Addressing explicit channels is a critical first step in achieving complete confinement. Second, we analyze over twenty real-world client-side HTML5 applications and make their data-confinement invariants explicit. These applications include the top twenty Chrome extensions, a sample HTML5 password manager, an SSO implementation, an electronic medical record system, and a database administration interface. We enforce data-confinement invariants on four applications by re-implementing them using data-confined sandboxes. We show that the effort of such re-implementation is modest.

[1]Temporary origins are created on the fly by the web browser on each execution and destroyed thereafter. Modern different browsers already support mechanisms to create temporary origins in HTML5 applications [3].

Contributions. In summary, this paper makes the following main contributions:

- We introduce the concept of data confinement for client-side HTML5 applications which handle sensitive data (§2).
- We identify the limitations of current security primitives in the HTML5 platform that make them insufficient for implementing data-confinement invariants (§2.3).
- We design and implement a data-confined sandbox, a novel mechanism in web browsers that provides complete mediation on all explicit data communication channels (§3 & §4).
- To demonstrate the practicality of our approach, we modify four applications handling sensitive data to provide strong data confinement guarantees. (§5). All our code and case studies are publicly available online [18].

2. Problem & Approach Overview

In this section, we explain the concept of data-confinement invariants, taking several real-world examples. We state our goals in designing an ideal primitive for data confinement, explain the central challenges, and outline why existing primitives are insufficient.

2.1 Data-confinement Invariants

Data confinement is a data-centric property, which limits the flow of sensitive data to an explicitly allowed set of security principals. Data confinement is useful for any application handling sensitive user data. We discuss a number of real-world applications that handle sensitive user data and intuitively explain the concept of data-confinement invariants.

- *Example 1: Cloud-based Password Managers*. Password managers organize a user's credentials across the web in a centralized store. Consider LastPass, a popular password manager that stores encrypted credential data in the cloud. LastPass decrypts the password database only at the client side (in a 'vault') with a user provided master password. A number of data-confinement invariants are implicit in the design of LastPass.
  - First, the user's master password should never be sent to *any* web server (including LastPass servers).
  - Second, the password database should only be sent back to the LastPass servers after encryption.
  - Third, the decrypted password database on the client-side should not leak to *any* web site.
  - Finally, individual decrypted passwords should only be sent to their corresponding websites: e.g., the credentials for `facebook.com` should only be used on `facebook.com`.
- *Example 2: Client-side SSO Implementations*. Several single sign-on (SSO) mechanisms have emerged on the web to manage online identities of users, including purely client side ones. Consider Mozilla's recent SSO mechanism called BrowserID. It has the following data-confinement invariants implicit in its design:

  - It aims to share authorization tokens only with specific participants in one run of the protocol.
  - Similar to the 'vault' in LastPass, BrowserID provides an interface for managing credentials in a user 'home page.' This home page data should not leak to external websites.
  - The user's BrowserID credentials (master password) should never be leaked to a third party: only the authorization credentials should be shared with the intended web principals involved in the particular instance of the protocol flow.

  Other SSO mechanisms, like FBConnect, often process capability-bearing tokens (such as OAuth tokens). Implementation weaknesses and logic flaws can violate these invariants, as demonstrated recently for a number of SSO implementations [23, 44, 48].
- *Example 3: Electronic Medical Record Applications*. Electronic medical record (EMR) applications provide a central interface for patient data, scheduling, clinical decisions, and billing. Strict compliance regulations such as HIPAA require data confinement for these applications, with financial and reputational penalties for violations. OpenEMR is the most popular open-source EMR application [40], and has a strict confinement requirement:
  - OpenEMR should not leak user data to *any* principal other than hospital servers.

  Note the dual requirements in this application: first, OpenEMR's developers want to ensure data confinement to their application; second, hospitals need to verify that OpenEMR is not leaking patient data to any external servers. In the current design, it is difficult for hospitals to verify this: any vulnerability in the client-side software can allow data disclosure.
- *Example 4: Web Interfaces for Sensitive Databases*. Web-based database administration interfaces are popular today, because they are easy to use. PhpMyAdmin is one such popular interface with thousands of downloads each week [37]. The following data-confinement invariants are implicit in its design:
  - Data received from the database server is not sent to any website.
  - User inputs (new values to store) are only sent to the database server's data insertion endpoint.

  Currently, a code injection vulnerability in the client-side interface can enable attackers to steal the entire database, as the interface executes with the database user's privileges. Moreover, the application is large and not easily auditable to ensure data-confinement invariants.

These examples are only a sample; any application handling sensitive data typically has a confinement invariant that it needs to enforce to fulfill user privacy expectations. In §5, we present a more exhaustive list of such invariants for the top twenty Google Chrome extensions and four popular web applications.

2.2 Goals & Challenges

Our goal is to create a practical data-confinement primitive in the HTML5 platform to enforce such invariants. Ideally, a data-confinement mechanism should have the following properties:

Table 1: List of channels available for inadvertent data disclosure

Channel Name	Description	Examples
Client-side pointers	Cross-origin pointer access explicitly allowed by browsers	window.parent.location, window.parent.frames[0].hash
Client-side messaging	Client-side messaging channels	`postMessage`, `MessageChannel`
Network channels	Interfaces that allow a document to make network requests and thus leak data	scripts, images, stylesheets, objects, frames, `XMLHttpRequest`, fonts, audio, video, HTTP redirects, plugins

- *Minimal TCB*. Current implementations require the whole application to be in the trusted computing base (TCB) to check if a confinement invariant is enforced. Client-side languages such as JavaScript are notoriously hard to analyze—our aim is an application architecture with minimal TCB. The TCB should be amenable to static verification.
- *Flexible Granularity and Policies*. The application should be able to state confinement invariants with a customizable granularity, finer than the whole application, such as at the level of `iframes`. Additionally, the data invariant specification should be flexible, i.e., it should be possible to state simple access control policies or more complex stateful policies.
- *Complete Mediation*. The mechanism should interpose on all explicit data communication channels. We want to make the mediation infrastructure reliable, piggybacking on existing browser implementations as far as possible. We consider covert channels, side channels, and self exfiltration channels out of scope for our present work.
- *Minimal impact to backwards compatibility*. The mechanism should be easy to implement in the current HTML5 platform and in existing HTML5 applications.

Achieving these goals for data confinement is challenging in the existing web environment, as discussed next.

Large TCB Mechanisms. The present HTML5 platform lacks mechanisms to explicitly state data-confinement invariants—there is no clear separation between policy and enforcement mechanism. In existing HTML5 applications, enforcing these invariants puts the whole application in the TCB. Verifying data confinement requires verifying the whole application. Unfortunately, the JavaScript language and the DOM interface makes modular reasoning about individual components difficult. All code runs with ambient access to the DOM, cookie, `localStorage` and network privileges. Further, techniques like prototype hijacking can violate encapsulation assumptions and allow attackers to leak private variables in other modules. The DOM API makes confinement difficult to ensure even in the absence of code injection vulnerabilities [25, 52].

Numerous Communication Channels. The HTML5 platform has a large number of data disclosure channels, as it aims to ease cross-origin network resource loading and communication. We outline a number of these channels in Table 1. We categorize these channels as:

- *Network channels*. HTML5 applications can make network requests via HTML elements like `img`, `form`, `script`, and `video`, as well as JavaScript and DOM APIs like `XMLHttpR`

equest and window.open. Furthermore, CSS stylesheets can issue network requests by referencing images, fonts and, other stylesheets.

- *Client-side cross-origin channels.* Web browsers support a number of channels for client-side cross-origin communication. This includes legacy channels like fragment messaging (via the location.hash property of cross-origin windows) as well as newer channels like postMessage.

Given the wide number of channels available for inadvertent data disclosure, we observe that no unified interface exists for ensuring confinement of fine-grained code elements in the HTML5 platform. This is in contrast to system call interposition in commodity operating systems that provides complete mediation. In §2.3, we explain how existing mechanisms are insufficient to confine these aforementioned channels.

Covert Channels. We focus on *explicit* data communication channels in the HTML5 platform core, as defined above. Ensuring comprehensive mediation on explicit data channels is an important first step in achieving data-confined HTML5 applications. Our proposed primitive does not protect against covert and side channels (such as shared browser caches [30, 31] and timing channels [5]) or self exfiltration channels [12], which are a subject of ongoing research. These channels are important. However, we point out that popular isolation mechanisms on existing systems also don't protect against these [9, 50, 53]. We believe explicit channels cover a large space of attacks, and we plan to investigate extending our techniques to covert channels in the future.

Scope. In addition to focusing on explicit channels, our primitive only targets the core HTML5 platform; our ideas extend to add-ons/plugins, however we exclude them from our present implementation. We defend against the standard web attacker model in which the attacker cannot tamper with or observe network traffic for other web origins and cannot subvert the integrity of the HTML5 platform itself [2].

2.3 Insufficiency of Existing Mechanisms

Table 2: Comparison of current solutions for data confinement

System Name	Complete Mediation	Fine Grained	Compatibility Cost	Small TCB
CSP	No[w]	No[o]	High[e]	Yes
JS Static Analysis	No[d]	Possible	High[e]	No
JS IRMs (Cajole, Conscript)	No[d]	Yes	High[e]	Yes
Treehouse	Yes	Yes	High[c]	No
sandbox with Temporary Origins	No[n]	Yes	Low	Yes
Data-confined sandboxes	**Yes**	**Yes**	**Low**	**Yes**

[c]code change [d]no CSS & DOM [e]disables eval [n]all network channels
[o]origin whitelist [w]anchors and window.open

None of the primitives available in today's HTML5 platform achieve our stated goals (§2.2). Table 2 shows a comparison of the existing security primitives available to HTML5 applications today. Browser-supported primitives, such as Content Security Policy (CSP), block some network channels but not all. Current mechanisms are designed with the goal of providing integrity, not confidentiality. Even the most restrictive CSP policy cannot block data leaks through anchor tags and window.open.

Another approach to interpose on all data communication channels is to do static analysis of the application source code [1, 10, 22, 34]. However, static analysis methods have a high compatibility cost, because they cannot reason about code that uses dynamic constructs like eval, which are used pervasively in existing applications [38, 39] and modern JavaScript libraries [28]. When combined with rewriting techniques, such as cajoling [10], mediation on client-side cross-frame channels in JavaScript can be achieved. However, even these together do not provide complete mediation over DOM and CSS channels.

Finally, in recent work, Treehouse proposed using new primitives like web workers and EcmaScript5 sealed objects in the HTML5 platform to ensure better interposition [27]. Treehouse proposes to execute individual components in web workers at the client side. We argue that such mechanisms have two concerns. First, web workers also run with some ambient privileges: e.g., workers have access to XMLHttpRequest, synchronous file APIs, script imports, and spawning new workers, which attackers can use to leak data. Treehouse relies on the seal/unseal features of ES5 to prevent access to these APIs, but this mechanism requires intrusive changes to existing applications and has a high compatibility cost. The second concern is that web worker based approaches have a large TCB. Since web workers do not have direct access to the DOM, application code executes on a virtual DOM in the worker that the parent code needs to copy to the main web page. Ensuring correctness of this mechanism requires a trusted client-side monitor, which performs the difficult task of sanitizing the web application; this increases the TCB.

3. Design

We give an overview of our design in this section. First, we detail the existing primitives that we extend (§3.1); then, we give an overview of the typical application architecture in our design (§3.2); and finally, we discuss the details of our new primitive, the data-confined sandbox (§3.3).

3.1 Leveraging Existing Platform Capabilities

Our solution leverages the predominant isolation mechanism—iframes. Existing browsers allow iframes to run with a temporary origin [3]. Iframes with data: URI source in Webkit and Firefox. We use this to isolate the application into partitions: one privileged parent partition that has the unfettered privileges of a web origin, and an arbitrary number of child partitions with no privileges. Since access to privileged APIs is origin bound in web applications, only the parent has access to the *privileged API* interface, such as access to the XMLHttpRequests (for web applications), browser extension APIs (for browser extensions), and sensors (on mobile devices). Unprivileged children communicate with the parent through a tightly controlled postMessage channel (dotted arrows in Figure 1). The parent can enforce policies on the requests it receives over this postMessage channel from its unprivileged children [3]. The parent uses its privileged interfaces to fulfill approved requests, such as authenticated XMLHttpRequest calls (curved dotted arrow in Figure 1).

Though this privilege separation architecture provides integrity, it does *not* provide data confinement. Any compromised child can make arbitrary requests on the network through the numerous data disclosure channels outlined earlier. In this work, we extend this

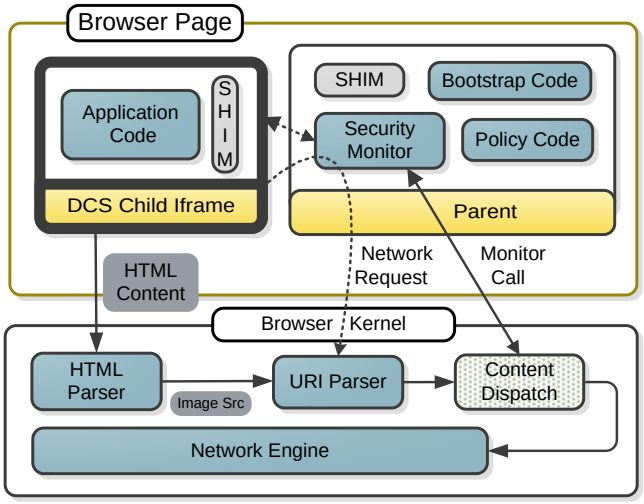

Figure 1: High-level design of our proposed architecture. The only component that runs privileged is the parent. The children run in data-confined sandboxes, with no ambient privileges and all communication channels monitored by the parent. Solid arrows illustrate monitoring of network requests by the parent. The child asks the browser to display some HTML content. On encountering say, an image, the browser asks the URI parser to parse the source URI, and then calls the content dispatch method. We modified the content dispatch code to call the parent's monitor code before requesting the image from the network.

design to enable easy data confinement in HTML5 applications.

3.2 Data-Confined Application Architecture

Figure 1 provides an overview of HTML5 applications in our design. In this architecture, HTML5 applications have one privileged parent component that spawns a number of unprivileged children with null authority. The parent loads the components processing sensitive data into these children, and the parent interposes on all communication channels using our new primitive: a *data-confined sandbox*. A data-confined sandbox (or DCS) is like a `sandboxed iframe`, but it provides confidentiality in addition to integrity. The parent defines a *security reference monitor*, which interposes on all explicit communication channels available to the DCS, easing data confinement in a high-assurance manner. The security monitor in the parent is transparent to the child.

Parent. The parent runs with the ambient privileges of the HTML5 application's origin, and is minimal in size. When the user navigates to the HTML5 application, a specific part of the parent code called the *bootstrap code* executes. This bootstrap code downloads the application code and initializes one or more unprivileged children, each running in its own temporary origin. The parent also defines a security monitor function in its global namespace. Our modified browser invokes this monitor function each time the child attempts to communicate over the network. Our design also enforces several invariants on the parent code to minimize the possibility of security vulnerabilities in the small, trusted parent code [3]. In particular, we disable dynamic code evaluation in the parent, allow only a text interface with the children, and set appropriate MIME types for static code downloaded by the bootstrap code [3].

Data-Confined Child. Nearly all application logic executes in unprivileged DCS children with null authority. Application code running in the data-confined child is subject to the following invariants:

- Application code executes in a unique temporary origin.
- Except for a blessed `postMessage` channel to the parent, the browser disables all client-side communication channels in a DCS child.
- The parent monitors all network channels.

Note that the `postMessage` channel is the *only* client-side cross-origin channel available to the data-confined child, and the browser guarantees that the channel only connects to the parent. The `postMessage` channel allows the parent to proxy privileged APIs for the child. Further, the `postMessage` channel also allows the parent to provide a channel to proxy `postMessages` to other client-side `iframes`—our design only enforce complete mediation by the parent.

Security Monitor. The key novelty of our architecture is the ability of the parent to monitor all network channels available to the child. Such monitoring, similar to system call interposition, enables high-assurance confinement. In our design, this monitoring is transparent to the child. Any action in the child that causes a network request results in the browser calling a 'monitor' function defined in the parent (solid arrows in Figure 1). The browser passes the URL of the network request, the type of the network request (e.g., image, style sheet, script) and the unique id of the child `iframe` that caused this request to the monitor.

Example. Consider the 'vault' for the LastPass web application. In our redesign, when the user navigates to the LastPass application, the server returns bootstrap code (the parent) that downloads the original application code and executes it in a data-confined sandbox (the child). The code in the DCS starts executing and makes network requests to include all the complex UI, DOM, and encryption libraries. Finally, the LastPass child code in the DCS makes a request for the encrypted password database and decrypts it with the user provided password. It then proceeds to show the decrypted vault to the user. Any vulnerability, such as XSS, only compromises the DCS child, and not the parent. If the compromised child attempts to leak sensitive data from the DCS, our design (in particular, complete mediation) ensures that the browser invokes the parent's monitor function before allowing the request.

The ability to monitor all explicit communication channels allows the parent to enforce a number of interesting policies. A simple policy is to allow network requests to `https://lastpass.com` only. However, since the parent's monitor code is in JavaScript, the parent can also enforce stateful policies: e.g., the monitor function only allows resource loads (i.e., scripts, images, styles) until the DCS child loads the encrypted password database. After loading the encrypted database, the security monitor disallows all future network requests. In general, since the security monitor function can store state, it can define a flexible policy based on a finite state machine. We leave the design of expressive policy languages for the future; our focus is on mechanisms.

3.3 Data-confined Sandbox: A New Primitive

We introduce a new primitive: the data-confined sandbox. A data-confined sandbox provides stronger confinement guarantees than `sandboxed iframes`, by enforcing mediation on all network requests and restricting client-side channels. It provides the parent with the requisite APIs to monitor all the explicit channels avail-

able to the child. To ensure correctness, the parent must block or monitor all communication channels outlined in Table 1.

Client-side Channels. Because children execute in temporary origins, the browser blocks access to any origin bound persistent channels. For example, temporary origins have no persistent data storage channels like `cookie`, `Storage`, and `FileSystem` APIs. The same-origin policy restricts JavaScript access across an iframe boundary: code can freely access the JavaScript object of another same-origin window/iframe. Since temporary origins are cross origin to *all* other origins, only explicit exceptions to the same-origin policy can cross the iframe boundary. These include cross-origin communication channels like `postMessage` and cross-origin window properties (like `location.hash`). These exceptions can allow disclosure of confidential data. A cross-origin *write*, e.g., a write to `window.location`, allows a data-confined child to leak data to another origin. In our design, the browser blocks *all* cross-origin writes from a DCS to another window, without calling the security monitor. Cross-origin reads allow an origin to read sensitive data from a data-confined child; thus, our design also block cross-origin reads.

Our design restricts the child to a blessed `postMessage` channel that can only communicate with the parent. Note that the child can still communicate with other origins via the parent, but cannot use arbitrary cross-origin client-side channels. Therefore, our design just enforces complete mediation without hindering functionality.

Network Requests. HTML5 applications can request network resources via markup like scripts, images, links, anchors, and forms and JavaScript APIs like `XMLHttpRequest`. In our design, the children can continue to make these network requests; the DCS transparently interposes on all these network channels. The parent defines a 'monitor' function that the browser executes before dispatching a network request; if the function returns false, the browser will not make the network request.

We rely on an external monitor (i.e., one running in the parent) over an inline one because the same-origin policy isolates the monitor from the child. We avoid an inline monitor that shares state with the unprivileged child in our design because it is harder to reason about its runtime integrity and correctness. As we discuss in the implementation section, the security monitor is not hard to implement—most browsers already have an internal API for controlling network access. We only expose this API to the security monitor function.

Compatibility Considerations. Our design for network request mediation is discretionary, as compared to client-side channels that we block outright. An alternative design is to disallow all network requests too, and only permit network access via the `postMessage` channel between the parent and child. Such a design has a significantly higher compatibility cost. Network channels are commonly utilized in HTML5 applications. In contrast, the use of client-side channels is rare—for example, Wang et al. report that cross-origin `window.location` read and writes occur in less than 0.1% of pages [42]. Therefore, we find that it is acceptable to disable cross-origin client-side channels completely, and force the child to use the blessed `postMessage` channel to the parent to access these.

Requests made by the DCS have an empty `Referer` and `Origin` header. Resource requests that require the application's origin in these headers will fail. This design is intentional: the ability to make requests with the application's URI in the `Referer` and `Origin` headers is an authority not available to the unprivileged children.

The lack of the correct `Referer` and `Origin` headers did not affect any of our case studies. Currently, only browsers based on WebKit send the `Origin` header, and web applications do not rely on these headers, as privacy conscious users often turn them off. To maintain compatibility with servers that rely on these features, the DCS can send a message to the parent, requesting it to make the appropriate request. An alternate design, which we have not investigated, would be to insert the parent's URI in the `Referer` and its origin in the `Origin` headers automatically.

Security Considerations. Our design of the DCS primitive is careful not to introduce new security vulnerabilities in the browser. In particular, we do not want to allow an arbitrary website to learn information or execute actions that it could not already learn or execute. The security policy of the current web platform is the same-origin policy. Our invariants are designed so that we do not violate any of the existing same-origin policy invariants baked into the platform. We enforce this goal with the following two invariants:

- *Invariant 1*: The parent should *only* be able to monitor application code that it could already monitor on the current web platform (albeit, through more fragile mechanisms).
- *Invariant 2*: The parent should not be able to infer anything about a resouce requested by a DCS that is not already possible on the current web platform.

We explain how our design enforces the above invariants. First, in our design, a data-confined sandbox can only apply to `iframe`s with a `data:` URI source, not to arbitrary URIs. Therefore, a malicious site cannot monitor arbitrary web pages. In an iframe with a `data:` URI source, the source code that executes is specified inline in the `src` attribute of the `iframe`. This code is under complete control of the parent. The parent can parse the `data:` URI source for static requests and redefine the DOM APIs to monitor dynamic requests [24]. Thus, even in the absence of our primitive, the parent can already monitor any requests a `data:` URI `iframe` makes. Note that the browser prevents one web origin from reading arbitrary content (including HTML code) from other web origins [6].

To ensure Invariant 2, we only call the security monitor for the *first* request made for a particular resource. As we noted above, this request can already be monitored by the parent. Future requests, notably redirects, are not in the control of the parent, and we do not call the security monitor for them.

For example, consider again the page at `http://socialnetwork.com/home` that redirects to `http://socialnetwork.com/username`. Consider a DCS child created by `attacker.com` parent. If this child creates an iframe with source `http://socialnetwork.com/home`, our modified browser calls the security monitor with this URI before dispatching the request. But, to ensure Invariant 2, the browser does *not* call the security monitor with the redirect URI (i.e., `http://socialnetwork.com/username`). Further, since the iframe is now executing in the security context of `http://socialnetwork.com/`, Invariant 1 ensures that any image or script loads made by the `socialnetwork.com` iframe do not call the security monitor.

Finally, we point out that due to the semantics of network requests in HTML5, the monitor function runs synchronously: a long running monitor function could freeze the child. The abil-

ity to cause stability problems via long running synchronous tasks is already a problem in browsers, and is not an artifact stemming from our design.

4. Implementation

We implemented our prototype for the Firefox browser, and re-architected four real-world web applications. In this section, we discuss the general application design and our Firefox implementation with one of our case studies: the Clipperz password management system. We discuss full details of all of our case studies in §5.

4.1 Data-Confined Application Architecture

As we noted earlier, our data-confined web application architecture depends on a small, high-assurance parent that executes the application code in data-confined sandboxes. We focus on the web application architecture in this section, and discuss our implementation of the data-confined sandbox in the Firefox Browser in §4.2.

Parent. In our architecture, the 'parent' is the code that executes first when the user loads the web application. Similar to Akhawe et al.'s design [3], the parent downloads and executes the application code in new data-confined sandbox children with `data:` URI sources of the original web application code. We also maintain a number of invariants for achieving high-assurance. In particular, we rely on Content Security Policy to restrict script execution to an origin whitelist, and disable all mechanisms to convert strings to code (including inline scripts and `eval`). The CSP policy for the parent is `script-src'self';`.

Child. Application code executes in data-confined sandboxes, with temporary origins, monitored by the parent. The application code can access two types of resources: privileged and unprivileged.

Unprivileged resources are resources available to all origins on the web platform. For example, the Clipperz web application includes a number of images using a `data:` URI source. When executing in a DCS, access to unprivileged resources is transparent to the application code. The monitor code (discussed below) can deny access to any of these resources, causing the browser not to load them.

Privileged resources are resources only available to the original application. Since application code in the DCS executes in a temporary origin, it does not have direct access to these resources. We rely on existing shim code [3] to proxy privileged APIs, such as `XMLHttpRequest` to the parent.

Security Monitor Code. The key new primitive of our implementation is the monitor providing complete mediation. Our monitor code provides complete mediation on all communication channels available to the DCS child. In particular, this includes all the network and client-side channels available to the application code executing in a DCS.

The parent needs to define a monitor function in the global namespace (i.e., assign a function to `window.monitor`) and the browser calls this function before dispatching any requests. Our prototype (discussed below) invokes the monitor function with one argument, the `params` object. The `params` object consists of three fields: the id of the DCS frame that made the request, the type of request, and depending on the request, the URL of the request, or the message body. The browser refuses to dispatch a request if the monitor function returns false.

```
1 window.monitor = function(p){
2   if (p.id !== 'mainframe') return false;
3   switch(p.type) {
4     case "IMAGE" : return img_whitelist(p.url);
5     case "SCRIPT": return script_whitelist(p.url);
6     case "postMessage": return postMsg_policy(p.msg);
7   }
8   return false;
9 }
```

Listing 1: Security monitor code for Clipperz

Listing 1 lists code for a simple monitor for the Clipperz web application. Depending on the type of the request (Lines 3-7), the parent calls the appropriate function that checks the URL against a whitelist of allowed loads. It is immediately apparent that this security monitor only permits image and script loads, disabling fonts, objects, `iframes`, and videos. A request of type `postMessage` indicates a `postMessage` from the child. The security monitor function checks the message against the parent's policy (Line 6) for privileged API calls, and forwards them to the usual `postMessage` event listeners, or drops the message if it violates policy.

4.2 Implementation of Data-Confined Sandbox

We implemented support for data-confined sandboxes in the Firefox browser. Our implementation has three distinct components: first, we implement support for the `dcfsandbox` attribute for the `iframe` tag. Second, we block client-side channels. Finally, we implement the security reference monitor for network requests. Our modifications are open source and available online [18].

dcfsandbox support. We modified the HTML parser to implement support for the `dcfsandbox` attribute for the `iframe` tag. An `iframe` that has this attribute only supports a `data:` URIs for its `src` attribute. Such an `iframe` implements all the restrictions that a `sandboxed iframe` supports, but adds further restrictions on client side and network channels, as we explain below.

Blocking Client-Side Channels. Recall that since the child executes in a temporary origin, it is cross origin to *all* other origins, including other sandboxes. Thus, all access outside of the `iframe` is cross origin. The same-origin policy restricts cross-origin JavaScript access to a restrictive whitelist of properties. In Firefox, this whitelist is present in `js/xpconnect/wrappers/AccessCheck.cpp`. We modified the `IsPermitted` function to block all cross-origin accesses, except for the blessed `postMessage` channel.

Implement Network Monitor. The `NSIContentPolicy` interface is a standard Firefox API used to monitor network requests. Popular security and privacy extensions, such as NoScript, AdBlock, and RequestPolicy, rely on this API. We register one of these listener to forward requests for monitored children to the parent's security monitor function. To identify the `iframe`, we use its `id` attribute, which the parent specifies at creation time. For ease of development, we have implemented this as a Firefox extension written in JavaScript. In total, our implementation is fewer than 214 lines of code, with only 60 lines being the core functionality of our extension.

Correctness Argument. We make significant changes to the internals of Firefox, and we rely on previously existing invariants to maintain correctness.

Client-Side Channels The list of cross-origin objects (including messaging constructs like `postMessage`) is a strict whitelist, explicitly specified in all browsers. Our implementation adds a flag that disables all such accesses for a data-confined sandbox. Any cross-origin access other than this list is a same-origin policy bypass, a bug with the highest severity rating on all browsers.

Network Channels We *do not* add our own implementation of a new monitor: we just hook into an existing monitor. In particular, we rely on the `NSIContentPolicy` interface for implementing our monitor. Popular extensions like NoScript, RequestPolicy, as well as internal browser services like the Content Security Policy and the HTML5 sandbox implementation, rely on this interface for enforcing mediation. A network request that goes through without calling the interface would be a critical bug, allowing for same-origin policy bypass.

We do not protect against data disclosure by browser extensions, as we consider extensions part of the browser TCB.

Performance. Our modifications synchronously block all network requests, until the parent's monitor returns. To measure the overhead of calling the parent's monitor code, we measured the increase in latency caused by a simple monitor that allows all requests. We measured the time required for script loads from a web server running on the local machine, and found that the load time increased from 16.73ms to 16.74ms. This increase is statistically insignificant, and pales in comparison to the typical latencies of 100ms observed on the web.

5. Case Studies

We retrofit our application architecture to four web applications to demonstrate the practicality of our approach. All our case studies, like our browser modifications, are open-source and freely available [18].

Table 3 lists our case studies and summarizes our results. We find that our redesigns are minimally intrusive (fewer than 184 lines changed in each of our case studies) and achieve significant TCB reduction. We evaluate our design on these case studies by measuring (a) the TCB reduction, (b) the lines of code changed to implement our redesign, and (c) the invariants we are able to enforce on the redesigned applications.

5.1 Clipperz

Clipperz is an open-source HTML5 password manager that allows a user to store a variety of sensitive data, such as website logins, bank account credentials, and credit card information [17]. Sensitive data is stored encrypted in the cloud and is decrypted at the client side with the user provided password. Users access their data in a single 'vault' page. Users can also click on 'direct login' links that load a site's login page, fill in the user name/password, and submit the login form.

The application relies on open-source components including the MochiKit library suite [36] and the YUI library [51]. In sum, Clipperz consists of 1.4MB of JavaScript code, all of which runs in a single security principal, with access to all sensitive data. The Clipperz application uses inline scripts and `data:` URIs extensively. We found that enforcing strong CSP restrictions to protect against XSS breaks several subcomponents of the Clipperz application.

Privilege Separation. We modified Clipperz to execute its application code in an unprivileged DCS. We reused existing shim code [3] to achieve seamless privilege separation. The one key change was to proxy handling links (such as Clipperz help page) and 'direct logins,' to the parent, since our design does not grant a DCS the privilege to open pop-up windows. Privilege separating the Clipperz application required changing 67 lines of code. Note that privilege separation in and of itself does not ensure data confinement: if an attacker compromises the code in the child, it can send data to an attacker website, for example, by loading an image.

Data-Confinement Invariants. Our privilege-separated design executes the Clipperz application code in a data-confined child, which allows the parent to enforce a confinement policy. One implemented policy was simple, like the security monitor in Listing 1, which allows the DCS child access only to `postMessage` and a whitelist of images and JavaScript libraries. However, the flexibility of our primitive allows for a powerful policy that can be temporal in nature. Our more expressive monitor function allows the Clipperz application to make network requests only until it downloads the password database; once the DCS child downloads the password database, the monitor function disallows further network access.[2] Note that relying on a whitelist of network resources means that we can guarantee the secrecy of the user entered master password, an invariant impossible to ensure in the current HTML5 platform without DCS.

Although our redesign makes data theft significantly harder, a compromised instance of Clipperz still has one (self) exfiltration channel. Clipperz's 'direct login' functionality navigates to a saved webpage and auto-fills the login credentials. A malicious script, executing in the compromised DCS, can request the parent to 'direct login' to an attacker controlled webpage, and provide the username and password for (say) facebook.com. This would allow the attacker controlled webpage to learn the user credentials for facebook.com.

In our redesign, we mitigate the above attack by creating two children: the UI component and the non-UI component. The UI component does not have direct access to the 'direct login' feature. Instead, a direct login requires sending a message to the non-UI component. The new component retrieves the associated credentials and completes the direct login process. In contrast to the vault page, this component does not need complex UI code and other supporting JavaScript libraries. In our implementation, this component executes with a strong CSP, providing higher assurance.

The DCS approach affords us the flexibility of enforcing a different policy on each child. The security monitor allows images and scripts to be loaded in the UI component from a set of whitelisted URLs. The direct login component has no UI and the security monitor disallows image loads in that component. Both components are always allowed access to the parent via `postMessage`. Again, the policy is temporal in nature, where upon database access, the security monitor blocks all communication in

[2] Except for navigation to pages like the help page.

Table 3: List of our case studies, as well as the individual components and policies in our redesign.

Application	Initial TCB	New TCB	Lines Changed	Component	Confinement Policy	Other Policies
Clipperz	1.4MB	6.3KB	67	Vault UI	Only to Clipperz server & Direct Login Child	None
				Direct Login	Open arbitrary websites	CSP Policy disabling dynamic code
BrowserID	206.9KB	5.7KB	184	Management	Only to BrowserID server	None
				Dialog	Only to BrowserID server, secure password input	API requests must match state machine
OpenEMR	149.1KB	6.1KB	51	Patient Information	Whitelist of necessary request signatures	None
SQL Buddy	100KB	2.97KB	11	Admin UI	Only to MySQL server	User confirmation for database writes

both components except to the parent.

5.2 BrowserID

BrowserID is a new authentication service by Mozilla. Similar to other single sign-on mechanisms like Facebook Connect and OpenID, BrowserID enables websites (termed Relying Parties) to authenticate a user using the BrowserID centralized service. Users create a single username/password to log in to the trusted BrowserID service and can register any number of email addresses as identities. Other single sign-on mechanisms share similar designs, and our results are more generally applicable to other single sign-on systems.

The implementation has the following components, typically hosted on the `https://login.persona.org` origin:

- A dialog window that is opened by the Relying Party when the user chooses to login using BrowserID. This window prompts asking the user to sign in using pre-registered email ids. We call this the *dialog* page.
- Other pages that contain public information materials and account management options for the authenticated user. We call these pages the *management* component.

The production BrowserID front end includes 101.1KB and 105.8KB of minified JavaScript in the management and dialog components respectively. The actual TCB is larger, since BrowserID uses the EJS templating system [20]. Similar to a number of modern JavaScript templating languages [28], EJS loads template files from the server and converts them to code at runtime using `eval`. Note that all modern templating languages rely on `eval`, which limits the applicability of CSP and static analysis techniques.

Privilege Separation. We moved all the application code to an unprivileged DCS. Minor changes were required for compatibility. In particular, we modified code that reads the window location, and added a base tag to ensure that links navigate the parent window. The EJS library uses synchronous `XMLHttpRequests` to download the templates. Since the same-origin policy restricts `XMLHttpRequests` to the same origin, the shim code proxies requests in the parent via the asynchronous `postMessage` channel. We modified the EJS templating code to download the templates asynchronously. The authentication component uses `postMessage` to communicate with the Relying Party: shim code enforces parent mediation on this exchange. In total, 184 lines of code were modified.

Data-Confinement Invariants. Executing in a data-confined sandbox, we are able to provide two key guarantees as part of our implementation:

- The login and credential managers (management component) do not communicate with any servers other than the BrowserID servers. This allows us to enforce secrecy on the main BrowserID username/password.
- In one instance of the BrowserID protocol, only 3 specific web principals interact. Our design guarantees sensitive tokens are never leaked to parties outside these three participants. In particular, the parent ensures that the child executes the whole protocol with the same principal and same Relying Party window. In the past, single sign-on mechanisms have had implementation bugs that allowed a MITM of an authentication flow [44, 48]; our design prevents such bugs.

For further hardening, we modified the dialog's login process to move the password entry to the trusted parent. The parent prompts the user for her password and sends it to the server. This way, a compromised dialog will never see the user's password. We also implemented a state machine in the security monitor policy based on the intended dialog behavior. In particular, this state machine ensures that the dialog component performs a series of requests consistent with transitions possible in the state machine. This prevents a compromised dialog from making arbitrary requests in the user's session.

5.3 OpenEMR

OpenEMR is the most popular open-source electronic medical record system [40]. With support for a variety of records like patients, billing, prescriptions, medical reports amongst others, OpenEMR is a comprehensive and complex web application. Patient records, prescriptions and medical reports are highly sensitive data, with most jurisdictions having laws regulating their access and distribution, possibly with penalties for inadvertent disclosure.

We focus on the patient information component of the OpenEMR application. OpenEMR accesses the patient details by setting a session variable, namely the *patient id*. Once the patient id is set, all future requests, such as 'demographic data,' 'notes,' and so on, are returned for the particular patient. If the user wants to navigate to another patient, the user has to use the search interface to reset the patient id.

Setting the patient id for a particular session just requires a GET request with a `set_pid` parameter. This can be achieved by any content injection. If an attacker successfully injects any content (e.g., an image tag) that causes a user to make a network request, a properly crafted request can set the patient id to that of Alice. As a result, the OpenEMR server will return Alice's medical records for the attacker. Note that this is not an XSS attack, but a content injection attack.

Privilege Separation. We focus on `demographics.php` which presents patient data. It loads with a `set_pid` parameter in the URL and the server sets the patient id accordingly. Scripts on the page then use `XMLHttpRequest` to download patient details, such as history and notes. We modified this page to serve its content as plain text, and a loader page requests the code and runs it in a DCS. The loader page proxies the `XMLHttpRequest` and cross-frame procedure calls through the parent using `postMessage`.

Data-Confinement Invariants. First, the DCS verifiably ensure that sensitive medical data does not leak to untrusted principals. The DCS can also prevent the page from making arbitrary calls to the large, feature-rich application. In our case, we programmed the security monitor to allow only a short whitelist of (method, URL) pairs necessary for the page to function. For example, the monitor denies any request with a `set_pid` parameter. This protects against the content injection attack discussed above. This would not be possible with an origin-based whitelist.

5.4 SQL Buddy

SQL Buddy is an open-source web-based application to handle the administration of MySQL databases. Written in PHP, it allows browsing of possibly sensitive data stored in a MySQL DBMS and supports standard database operations, including SQL queries and the creation, modification, and deletion of databases, tables, fields, and rows. It also allows for management of MySQL users.

Privilege Separation. We re-architected SQL Buddy to execute all code in a DCS. We re-used most of the privilege separation open sourced in [3], only adding a monitor function in the parent. The key change was in the script that logged a user into MySQL. The original implementation returned a new login page upon a failed login attempt, an action disallowed within a DCS. In our redesign, we return an error code over `XMLHttpRequest`. The client-side code utilizes this code to display the new login page. This modification required changes to only 11 lines of code. The SQL Buddy code does not use any of the client-side communication channels we blocked in a DCS: as a result, modifying it to run in a DCS is essentially the same as privilege separating it.

Data-Confinement Invariants. By executing the SQL Buddy application code in a DCS, the parent can enforce strong confidentiality policies. The application runs in two logical stages; the flexibility of the DCS monitor allows us to enforce a policy for each stage.

- Initially, the monitor function restricts communication to only SQL Buddy resources. The monitor allows the application to load a number of whitelisted JavaScript libraries and stylesheets.
- After loading the code and stylesheets, the application no longer requires network access except for loading SQL Buddy resource images and making `XMLHttpRequests` to SQL Buddy PHP code, which are proxied at the parent via `postMessage`. Our monitor code now locks down communication to these two channels.

Our monitoring function restricts all explicit communication channels: if the SQL Buddy code gets compromised, it still cannot send data to arbitrary servers. Separating out a small, trusted parent allows us to enforce finer grained policies. For example, our implementation also limits writes to the database. Any writes to the database require the user to explicitly confirm the write with a simple confirmation prompt created by the parent. Compromised code can not modify the database in the background; the user needs to confirm that she wants to modify the database.

5.5 Chrome Extensions

To demonstrate the prevalence of data-confinement needs, we also studied the top 20 most popular extensions for the Google Chrome platform and identified their data-confinement invariants. Our analysis indicates that data confinement is a widely prevalent requirement; with 16 of the 20 extensions we studied maintaining an invariant implicitly. The TCB size for the extensions varies from 7.5KB to 1.24MB. Sensitive data available to the extensions vary from access to the user's browsing history to the user's social media information. The remaining four extensions without an invariant dealt with the UI appearance of websites, and did not access sensitive data and made no network communications. Full details of our study are available online [47].

6. Related Work

A number of previous works share our goals of improving assurance in web applications. We gave a detailed comparison to closely related works in §2.3. We discuss other approaches targeting data isolation in HTML5 applications.

Khatiwala et al. propose a technique for data confinement for binary applications [32]. IceShield demonstrated the efficacy of modern ES5 features to create a tamper-resistant mediation layer for JavaScript in modern browsers [24]. Data-confinement invariants can be violated by non-scripting attacks, and thus a more general primitive is needed [25, 52].

Recent work on information flow and non-interference show promise for ensuring fine-grained data-confinement in JavaScript; unfortunately, these techniques currently have high overhead for modern applications [16]. IBEX proposed writing extensions in a high-level language (FINE) that can later by analyzed to ensure conformance with specific policies [21]. In contrast, our work does not require significant changes to web applications.

7. Conclusion

Modern HTML5 applicaitons handle increasingly sensitive personal data, and require strong data-confinement guarantees. However, current approaches to ensure confinement are ad-hoc and do not provide high assurance. We presented a new design for achieving data-confinement that guarantees complete mediation with a small TCB. Our design is practical, has negligible performance overhead and does not require intrusive changes to the HTML5 platform. We empirically show that our new design can enable data-confinement in a number of applications handling sensitive data, and achieve a drastic reduction in TCB. Future work includes investigating and mitigating covert channels.

8. References

[1] AdSafe. http://www.adsafe.org/.

[2] AKHAWE, D., BARTH, A., LAM, P., MITCHELL, J., AND SONG, D. Towards a Formal Foundation of Web Security. *CSF* (2010).

[3] AKHAWE, D., SAXENA, P., AND SONG, D. Privilege Separation in HTML5 Applications. *USENIX Security* (2012).

[4] Mozilla Boot2gecko. https://wiki.mozilla.org/B2G.

[5] BARTH, A. Timing Attacks on CSS Shaders. http://www.schemehostport.com/2011/12/timing-attacks-on-css-shaders.html, 2011.

[6] BARTH, A. X-Script-Origin, We Hardly Knew Ye. http://www.schemehostport.com/2011/10/x-script-origin-we-hardly-knew-ye.html, 2011.

[7] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. *NSDI* (2008).

[8] BRUMLEY, D., AND SONG, D. Privtrans: Automatically Partitioning Programs for Privilege Separation. *USENIX Security* (2004).

[9] CABUK, S., BRODLEY, C. E., AND SHIELDS, C. Ip covert timing channels: design and detection. In *Proceedings of the 11th ACM conference on Computer and communications security* (New York, NY, USA, 2004), CCS '04, ACM, pp. 178–187.

[10] Google Caja. http://developers.google.com/caja/.

[11] CARDWELL, M. Lastpass vulnerability exposes account details. http://bit.ly/eQR9q0, 2011.

[12] CHEN, E., GORBATY, S., SINGHAL, A., AND JACKSON, C. Self-exfiltration: The dangers of browser-enforced information flow control. *W2SP* (2012).

[13] CHIA, P. H., YAMAMOTO, Y., AND ASOKAN, N. Is this app safe?: a large scale study on application permissions and risk signals. *WWW* (2012).

[14] Chrome web store. https://chrome.google.com/webstore.

[15] Chrome extensions. https://chrome.google.com/webstore/category/extensions.

[16] CHUGH, R., MEISTER, J. A., JHALA, R., AND LERNER, S. Staged information flow for JavaScript. *PLDI* (2009).

[17] Clipperz. http://www.clipperz.com/.

[18] Code Release. https://sites.google.com/site/dataconfinedhtml5applications/home.

[19] Dropbox Developer Reference. http://www.dropbox.com/developers/reference.

[20] Ejs javascript templates. http://embeddedjs.com/.

[21] GUHA, A., FREDRIKSON, M., LIVSHITS, B., AND SWAMY, N. Verified security for browser extensions. *IEEE S&P* (2011).

[22] GUHA, A., KRISHNAMURTHI, S., AND JIM, T. Using static analysis for ajax intrusion detection. *WWW* (2009).

[23] HANNA, S., SHIN, E., AKHAWE, D., BOEHM, A., SAXENA, P., AND SONG, D. The emperor's new apis: On the (in) secure usage of new client-side primitives. *W2SP* (2010).

[24] HEIDERICH, M., FROSCH, T., AND HOLZ, T. Iceshield: detection and mitigation of malicious websites with a frozen dom. *RAID* (2011).

[25] HEIDERICH, M., NIEMIETZ, M., SCHUSTER, F., HOLZ, T., AND SCHWENK, J. Scriptless attacks: stealing the pie without touching the sill. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 760–771.

[26] HODGES, J., JACKSON, C., AND BARTH, A. Http strict transport security (hsts). http://bit.ly/MnbLag.

[27] INGRAM, L., AND WALFISH, M. Treehouse: Javascript sandboxes to help web developers help themselves. *USENIX ATC* (2012).

[28] Issue 107538. http://code.google.com/p/chromium/issues/detail?id=107538#c35.

[29] JACKSON, C., AND BARTH, A. Forcehttps: protecting high-security web sites from network attacks. *WWW* (2008).

[30] JACKSON, C., BORTZ, A., BONEH, D., AND MITCHELL, J. C. Protecting browser state from web privacy attacks. *WWW* (2006).

[31] JAKOBSSON, M., AND STAMM, S. Invasive browser sniffing and countermeasures. *WWW* (2006).

[32] KHATIWALA, T., SWAMINATHAN, R., AND VENKATAKRISHNAN, V. Data Sandboxing: A Technique for Enforcing Confidentiality Policies. *ACSAC* (2006).

[33] Lastpass. http://lastpass.com/.

[34] MAFFEIS, S., MITCHELL, J. C., AND TALY, A. Object capabilities and isolation of untrusted web applications. *IEEE S&P* (2010).

[35] MICROSOFT. Metro style app development. http://msdn.microsoft.com/en-us/windows/apps/.

[36] Mochikit. http://mochi.github.com/mochikit/.

[37] phpmyadmin. http://www.phpmyadmin.net/.

[38] RICHARDS, G., HAMMER, C., BURG, B., AND VITEK, J. The eval that men do. *ECOOP* (2011).

[39] RICHARDS, G., LEBRESNE, S., BURG, B., AND VITEK, J. An analysis of the dynamic behavior of javascript programs. *ACM SIGPLAN Notices* (2010).

[40] S. RILEY. 5 OpenSource EMRs worth reviewing, 2011. http://bit.ly/hUa6l1.

[41] Google seccomp sandbox for linux. http://code.google.com/p/seccompsandbox/.

[42] SINGH, K., MOSHCHUK, A., WANG, H., AND LEE, W. On the incoherencies in web browser access control policies. *IEEE S&P* (2010).

[43] STERNE, B., AND BARTH, A. Content security policy: W3c editor's draft, 2012. http://bit.ly/foq8vf.

[44] SUN, S., HAWKEY, K., AND BEZNOSOV, K. Systematically breaking and fixing openid security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures. *Computers & Security* (2012).

[45] Move over android? telefonica prepare to release firefox os, 2012. http://bit.ly/LIHyDK.

[46] Tizen. https://www.tizen.org/.

[47] Chrome Extension Study. https://sites.google.com/site/dataconfinedhtml5applications/chrome-extensions-study.

[48] WANG, R., CHEN, S., AND WANG, X. Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web

services. *IEEE S&P* (2012).

[49] WEINBERGER, J., BARTH, A., AND SONG, D. Towards Client-side HTML Security Policies. *HotSec* (2011).

[50] XU, Y., BAILEY, M., JAHANIAN, F., JOSHI, K., HILTUNEN, M., AND SCHLICHTING, R. An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop* (2011), ACM, pp. 29–40.

[51] Yui library. `http://yuilibrary.com/`.

[52] ZALEWSKI, M. Postcards from the post-xss world. `http://lcamtuf.coredump.cx/postxss/`.

[53] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 305–316.