Protecting Users by Confining JavaScript with COWL

Deian Stefan* Stanford
Edward Z. Yang Stanford
Petr Marchenko Google
Alejandro Russo† Chalmers
Dave Herman Mozilla
Brad Karp UCL
David Mazières Stanford

ABSTRACT

Modern web applications are conglomerations of JavaScript written by multiple authors: application developers routinely incorporate code from third-party libraries, and *mashup* applications synthesize data and code hosted at different sites. In current browsers, a web application's developer and user must trust third-party code in libraries not to leak the user's sensitive information from within applications. Even worse, in the status quo, the only way to implement some mashups is for the user to give her login credentials for one site to the operator of another site. Fundamentally, today's browser security model trades privacy for flexibility because it lacks a sufficient mechanism for *confining untrusted code.* We present COWL, a robust JavaScript confinement system for modern web browsers. COWL introduces label-based mandatory access control to browsing contexts in a way that is fully backward-compatible with legacy web content. We use a series of case-study applications to motivate COWL's design and demonstrate how COWL allows both the inclusion of untrusted scripts in applications and the building of mashups that combine sensitive information from multiple mutually distrusting origins, all while protecting users' privacy. Measurements of two COWL implementations, one in Firefox and one in Chromium, demonstrate a virtually imperceptible increase in page-load latency.

1 INTRODUCTION

Web applications have proliferated because it is so easy for developers to reuse components of existing ones. Such reuse is ubiquitous. jQuery, a widely used JavaScript library, is included in and used by over 77% of the Quantcast top-10,000 web sites, and 59% of the Quantcast top-million web sites [3]. While component reuse in the venerable desktop software model typically involves libraries, the reusable components in web applications are not limited to just JavaScript library code—they further include network-accessible content and services.

The resulting model is one in which web developers cobble together multiple JavaScript libraries, web-based content, and web-based services written and operated by various parties (who in turn may integrate more of these resources) and build the required application-specific functionality atop them. Unfortunately, some of the many contributors to the tangle of JavaScript comprising an application may not have the user's best interest at heart. The wealth of sensitive data processed in today's web applications (*e.g.,* email, bank statements, health records, passwords, *etc.*) is an attractive target. Miscreants may stealthily craft malicious JavaScript that, when incorporated into an application by an unwitting developer, violates the user's privacy by leaking sensitive information.

Two goals for web applications emerge from the prior discussion: *flexibility* for the application developer (*i.e.,* enabling the building of applications with rich functionality, composable from potentially disparate pieces hosted by different sites); and *privacy* for the user (*i.e.,* to ensure that the user's sensitive data cannot be leaked from applications to unauthorized parties). These two goals are hardly new: Wang *et al.* articulated similar ones, and proposed new browser primitives to improve isolation within *mashups,* including discretionary access control (DAC) for inter-frame communication [41]. Indeed, today's browsers incorporate similar mechanisms in the guises of HTML5's iframe sandbox and postMessage API [47]. And the *Same-Origin Policy* (SOP, reviewed in Section 2.1) prevents JavaScript hosted by one principal from reading content hosted by another.

Unfortunately, in the status-quo web browser security architecture, one must often sacrifice privacy to achieve flexibility, and vice-versa. The central reason that flexibility and privacy are at odds in the status quo is that the mechanisms today's browsers rely on for providing privacy—the SOP, Content Security Policy (CSP) [42], and Cross-Origin Resource Sharing (CORS) [45]—are all forms of discretionary access control. DAC has the brittle character of either denying or granting untrusted code (*e.g.,* a library written by a third party) access to data. In the former case, the untrusted JavaScript might *need* the sensitive data to implement the desired application functionality—hence, denying access prioritizes privacy over flexibility. In the latter, DAC exercises no control over what the untrusted code does with the sensitive data—and thus prioritizes flexibility over privacy. DAC is an essential tool in the privacy arsenal, but *does not fit cases where one runs untrusted code on sensitive input,* which are the norm for web applications, given their multi-contributor nature.

In practice, web developers turn their backs on privacy in favor of flexibility because the browser doesn't offer

*Work partly conducted while at Mozilla.
†Work partly conducted while at Stanford.

primitives that let them opt for both. For example, a developer may want to include untrusted JavaScript from another origin in his application. All-or-nothing DAC leads the developer to include the untrusted library with a `script` tag, which effectively bypasses the SOP, interpolating untrusted code into the enclosing page and granting it unfettered access to the enclosing page's origin's content.[1] And when a developer of a mashup that integrates content from *other* origins finds that the SOP forbids his application from retrieving data from them, he designs his mashup to require that the user provide the mashup her login credentials for the sites at the two other origins [2]—the epitome of "functionality over privacy."

In this paper, we present COWL (Confinement with Origin Web Labels), a mandatory access control (MAC) system that confines untrusted JavaScript in web browsers. COWL allows untrusted code to compute over sensitive data and display results to the user, but prohibits the untrusted code from exfiltrating sensitive data (*e.g.,* by sending it to an untrusted remote origin). It thus allows web developers to opt for *both* flexibility and privacy.

We consider four motivating example web applications—a password strength-checker, an application that imports the (untrusted) jQuery library, an encrypted cloud-based document editor, and a third-party mashup, none of which can be implemented in a way that preserves the user's privacy in the status-quo web security architecture. These examples drive the design requirements for COWL, particularly MAC with *symmetric and hierarchical confinement* that supports *delegation.* Symmetric confinement allows *mutually* distrusting principals each to pass sensitive data to the other, and confine the other's use of the passed sensitive data. Hierarchical confinement allows any developer to confine code she does not trust, and confinement to be nested to arbitrary depths. And delegation allows a developer explicitly to confer the privileges of one execution context on a separate execution context. No prior browser security architecture offers this combination of properties.

We demonstrate COWL's applicability by implementing secure versions of the four motivating applications with it. Our contributions include:

- We characterize the shared needs of four case-study web applications (Section 2.2) for which today's browser security architecture cannot provide privacy.
- We describe the design of the COWL label-based MAC system for web browsers (Section 3), which meets the requirements of the four case-study web applications.
- We describe designs of the four case-study web applications atop COWL (Section 4).
- We describe implementations of COWL (Section 5) for the Firefox and Chromium open-source browsers; our evaluation (Section 6) illustrates that COWL incurs minimal performance overhead over the respective baseline browsers.

[1] Indeed, jQuery *requires* such access to the enclosing page's content!

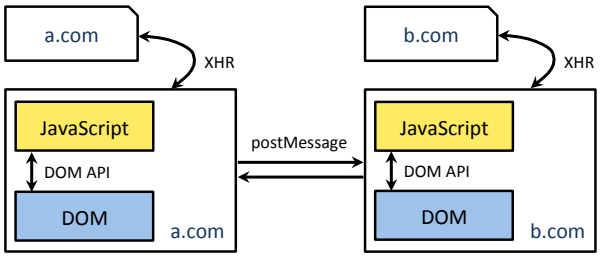

Figure 1: Simplified browser architecture.

2 Background, Examples, & Goals

A single top-level web page often incorporates multiple scripts written by different authors.[2] Ideally, the browser should protect the user's sensitive data from unauthorized disclosure, yet afford page developers the greatest possible flexibility to construct featureful applications that reuse functionality implemented in scripts provided by (potentially untrusted) third parties. To make concrete the diversity of potential trust relationships between scripts' authors and the many ways page developers structure amalgamations of scripts, we describe several example web applications, none of which can be implemented with strong privacy for the user in today's web browsers. These examples illustrate key requirements for the design of a flexible browser confinement mechanism. Before describing these examples, however, we offer a brief refresher on status-quo browser privacy polices.

2.1 Browser Privacy Policies

Browsing contexts Figure 1 depicts the basic building blocks of the current web security architecture. A *browsing context* (*e.g.,* a page or frame) encapsulates presentable content and a JavaScript execution environment (heap and code) that interacts with content through the *Document Object Model (DOM)* [47]. Browsing contexts may be nested (*e.g.,* by using iframes). They also may read and write persistent storage (*e.g.,* cookies), issue network requests (either implicitly in page content that references a URL retrieved over the network, or explicitly in JavaScript, using the `XMLHttpRequest` (XHR) constructor), and communicate with other contexts (IPC-style via `postMessage`, or, in certain cases, by sharing DOM objects). Some contexts such as Web Workers [44] run JavaScript but do not instantiate a DOM. We use the terms *context* and *compartment* interchangeably to refer to both browsing contexts and workers, except when the more precise meaning is relevant.

Origins and the Same-Origin Policy Since different authors may contribute components within a page, today's

[2] Throughout we use "web page" and "web application" interchangeably, and "JavaScript code" and "script" interchangeably.

status quo browsers impose a security policy on interactions among components. Policies are expressed in terms of *origins*. An origin is a source of authority encoded by the protocol (*e.g.,* `https`), domain name (*e.g.,* `fb.com`), and port (*e.g.,* `443`) of a resource URL. For brevity, we elide the protocol and port from URLs throughout.

The same-origin policy specifies that an origin's resources should be readable only by content from the same origin [7, 38, 52]. Browsers ensure that code executing in an `a.com` context can only inspect the DOM and cookies of another context if they share the same origin, *i.e.,* `a.com`. Similarly, such code can only inspect the response to a network request (performed with XHR) if the remote host's origin is `a.com`.

The SOP does not, however, prevent code from *disclosing* data to foreign origins. For example, code executing in an `a.com` context can trivially disclose data to `b.com` by using XHR to perform a network request; the SOP prevents the code from inspecting responses to such cross-origin XHR requests, but does not impose any restrictions on sending such requests. Similarly, code can exfiltrate data by encoding it in the path of a URL whose origin is `b.com`, and setting the `src` property of an `img` element to this URL.

Content Security Policy (CSP) Modern browsers allow the developer to protect a user's privacy by specifying a CSP that limits the communication of a page—*i.e.,* that disallows certain communication ordinarily permitted by the SOP. Developers may set individual CSP directives to restrict the origins to which a context may issue requests of specific types (for images or scripts, XHR destinations, *etc.*) [42]. However, CSP policies suffer from two limitations. They are *static:* they cannot change during a page's lifetime (*e.g.,* a page may not drop the privilege to communicate with untrusted origins before reading potentially sensitive data). And they are *inaccessible:* JavaScript code cannot inspect the CSP of its enclosing context or some other context, *e.g.,* when determining whether to share sensitive data with that other context.

postMessage and Cross-Origin Resource Sharing (CORS) As illustrated in Figure 1, the HTML5 `postMessage` API [43] enables cross-origin communication in IPC-like fashion within the browser. To prevent unintended leaks [8], a sender always specifies the origin of the intended recipient; only a context with that origin may read the message.

CORS [45] goes a step further and allows controlled cross-origin communication between a browsing context of one origin and a remote server with a different origin. Under CORS, a server may include a header on returned content that explicitly whitelists other origin(s) allowed to read the response.

Note that both `postMessage`'s target origin and CORS are purely discretionary in nature: they allow static selection of which cross-origin communication is allowed and which denied, but enforce no confinement on a receiving compartment of differing origin. Thus, in the status-quo web security architecture, a privacy-conscious developer should only send sensitive data to a compartment of differing origin if she completely trusts that origin.

2.2 Motivating Examples

Having reviewed the building blocks of security policies in status-quo web browsers, we now turn to examples of web applications for which strong privacy is not achievable today. These examples illuminate key design requirements for the COWL confinement system.

Password Strength Checker Given users' propensity for choosing poor (*i.e.,* easily guessable) passwords, many web sites today incorporate functionality to check the strength of a password selected by a user and offer the user feedback (*e.g.,* "too weak; choose another," "strong," *etc.*). Suppose a developer at Facebook (origin `fb.com`) wishes to re-use password-checking functionality provided in a JavaScript library by a third party, say, from origin `sketchy.ru`. If the developer at `fb.com` simply includes the third party's code in a `script` tag referencing a resource at `sketchy.ru`, then the referenced script will have unfettered access to both the user's password (provided by the Facebook page, which the library *must* see to do its job) and to write to the network via XHR. This simple state of affairs is emblematic of the ease with which naïve web developers can introduce leaks of sensitive data in applications.

A more skilled web developer could today host the checker script on her *own* server and have that server specify a CSP policy for the page. Unfortunately, a CSP policy that disallows scripts within the page from initiating XHRs to any other origins is *too inflexible,* in that it precludes useful operations by the checker script, *e.g.,* retrieving an updated set of regular expressions describing weak passwords from a remote server (essentially, "updating" the checker's functionality). Doing so requires communicating with a remote origin. Yet a CSP policy that permits such communication, even with the top-level page's same origin, is *too permissive:* a malicious script could potentially carry out a *self-exfiltration attack* and write the password to a public part of the trusted server [11, 50].

This trade-off between flexibility and privacy, while inherent to CSP, need not be fundamental to the web model. The key insight is that it is entirely safe and useful for an untrusted script to communicate with remote origins *before* it reads sensitive data. We note, then, the requirement of a confinement mechanism that allows code in a compartment to communicate with the network *until it has been exposed to sensitive data.* MAC-based confinement meets this requirement.

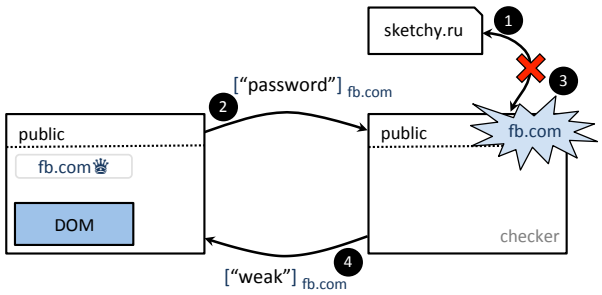

Figure 2: Third-party password checker architecture under COWL.

Figure 2 shows how such a design might look. In this and subsequent examples, rectangular frames denote compartments, arrows denote communication (either between a compartment and the network, or IPC-style between compartments), and events during execution are numbered sequentially in time. As we have proposed previously [49], compartments may be *labeled* (Section 3.1) with the origins to whose sensitive data they have been exposed. A compartment that has not yet observed sensitive data is denoted `public`; however, when it wishes to incorporate sensitive data, the compartment *raises* its label (at the cost of being more restricted in where it can write). We illustrate the raising of a label with a "flash" connoting the sensitivity of data being integrated. A compartment's *privilege* (Section 3.3), which specifies the origins for which a script executing in that compartment is trusted, is indicated by a crown. Here, a top-level page at `fb.com` encapsulates a password-checker script from a third-party origin in a new compartment. The label of the new compartment is initially `public`. First, in step (1), the checker script is free to download updated regular expressions from an arbitrary remote origin. In step (2), the top-level page sends the user's password to the checker script's worker using `postMessage`; the password is *labeled* `fb.com` to indicate that the data is sensitive to this origin (Section 3.2). In step (3) the checker raises its label to reflect that the context is about to be exposed to sensitive data from `fb.com` and inspects the password. When the label is raised, COWL atomically denies the context further access to the network in step (3).[3] However, the checker script is free to compute the result, which it then returns via `postMessage` to the top-level page in step (4); the result carries the label `fb.com` to reflect that the sender may be sending data derived from sensitive data owned by `fb.com`. Since the top-level page has the `fb.com` privilege, it can simply read the data (without raising its label).

[3] For clarity, we use `fb.com` as the label on the data. This label still allows the checker to send XHR requests to `fb.com`; to ensure that the checker cannot communicate with *any* origin, COWL provides fresh origins (see Section 3.3).

Encrypted Document Editor Today's web applications, such as in-browser document editors backed by cloud-based storage (*e.g.,* Google Docs), typically require the user to trust the app developer/cloud-based storage provider (often the same principal under the SOP) with the data in her documents. That is, the provider's server observes the user's data in cleartext. Suppose an organization wished to use an in-browser document editor but did *not* want to reveal its users' document data to the editor provider's server. How might the provider offer a privacy-preserving editor app that would satisfy the needs of such a privacy-conscious organization? One promising approach might be for the "customer" privacy-sensitive organization to implement a trusted document encryption service hosted at its own origin, distinct from that which hosts the editor app. The editor app could allow the user to specify a JavaScript "plugin" library she trusts to perform cryptography correctly. In this design, one origin serves the JavaScript code for the editor app (say, `gdocs.com`) and a different origin serves the JavaScript code for the cryptography library (say, `eff.org`). Note that these two origins may be *mutually distrusting*. `gdocs.com`'s script must pass the document's cleartext to a script from `eff.org` for encryption, but would like to confine the execution of the encryption script so that it cannot exfiltrate the document to any origin *other* than `gdocs.com`. Similarly, `eff.org`'s cryptography library may not trust `gdocs.com` with the cleartext document—it would like to confine `gdocs.com`'s editor to prevent exfiltration of the cleartext document to `gdocs.com` (or to any other origin). This simple use case highlights the need for *symmetric confinement:* when two mutually distrusting scripts from different origins communicate, *each must be able to confine the other's further use of data it provides.*

Third-Party Mashup Some of the most useful web applications are *mashups*; these applications integrate and compute over data hosted by multiple origins. For example, consider an application that reconciles a user's Amazon purchases (the data for which are hosted by `amazon.com`) against a user's bank statement (the data for which are hosted by `chase.com`). The user may well deem both these categories of data sensitive and will furthermore not want data from Amazon to be exposed to her bank or vice-versa, nor to any other remote party. Today, if one of the two providers implements the mashup, its application code must bypass the SOP to allow sharing of data across origin boundaries, *e.g.,* by communicating between iframes with `postMessage` or setting a permissive CORS policy. This approach forfeits privacy: one origin sends sensitive data to the other, after which the receiving origin may exfiltrate that sensitive data at will. Alternatively, a third-party developer may wish to implement and offer this mashup application. Users of such a *third-party mashup* give up their privacy, usually by simply

handing off credentials, as again today's browser enforces no policy that confines the sensitive data the mashup's code observes within the browser. To enable third-party mashups that do not sacrifice the user's privacy, we note again the need for an untrusted script to be able to issue requests to multiple remote origins (*e.g.,* `amazon.com` and `chase.com`), but to lose the privilege to communicate over the network once it has read the responses from those origins. Here, too, MAC-based confinement addresses the shortcomings of DAC.

Untrusted Third-Party Library Web application developers today make extensive use of third-party libraries like jQuery. Simply importing a library into a page provides no isolation whatsoever between the untrusted third-party code and any sensitive data within the page. Developers of applications that process sensitive data want the convenience of reusing popular libraries. But such reuse risks exfiltration of sensitive data by these untrusted libraries. Note that because jQuery requires access to the content of the entire page that uses it, we cannot isolate jQuery in a separate compartment from the parent's, as we did for the password-checker example. Instead, we observe that jQuery demands a design that is a mirror image of that for confining the password checker: we place the *trusted* code for a page in a separate compartment and deem the rest of the page (including the untrusted jQuery code) as untrusted. The trusted code can then communicate with remote origins and inject sensitive data into the untrusted page, but the untrusted page (including jQuery) cannot communicate with remote origins (and thus cannot exfiltrate sensitive data within the untrusted page). This refactoring highlights the need for a confinement system that supports *delegation* and *dropping privilege*: a page should be able to create a compartment, confer its privileges to communicate with remote origins on that compartment, and then give these privileges up.

We note further that any *library* author may wish to reuse functionality from another untrusted library. Accordingly, to allow the broadest reuse of code, the browser should support *hierarchical confinement*—the primitives for confining untrusted code should allow not only a single level of confinement (one trusted context confining one untrusted context), but arbitrarily many levels of confinement (one trusted context confining an untrusted one, that in turn confines a further untrusted one, *etc.*).

2.3 Design Goals

We have briefly introduced four motivating web applications that achieve rich functionality by combining code from one or more untrusted parties. The privacy challenges that arise in such applications are unfortunately unaddressed by status-quo browser security policies, such as the SOP. These applications clearly illustrate the need for robust yet flexible confinement for untrusted code in browsers. To summarize, these applications would appear to be well served by a system that:

- Applies mandatory access control (MAC);
- Is *symmetric, i.e.,* it permits two principals to *mutually* distrust one another, and each prevent the other from exfiltrating its data;
- Is *hierarchical, i.e.,* it permits principal *A* to confine code from principal *B* that processes *A*'s data, while principal *B* can independently confine code from principal *C* that processes *B*'s data, *etc.*
- Supports *delegation* and *dropping privilege, i.e.,* it permits a script running in a compartment with the privilege to communicate with some set of origins to confer those privileges on another compartment, then relinquish those privileges itself.

In the next section, we describe COWL, a new confinement system that satisfies these design goals.

3 The COWL Confinement System

The COWL confinement system extends the browser security model while leaving the browser fully compatible with today's "legacy" web applications.[4] Under COWL, the browser treats a page exactly like a legacy browser does unless the page executes a COWL API operation, at which point the browser records that page as running in *confinement mode,* and all further operations by that page are subject to confinement by COWL. COWL augments today's web browser with three primitives, all of which appear in the simple password-checker application example in Figure 2.

Labeled browsing contexts enforce MAC-based confinement of JavaScript at the granularity of a context (*e.g.,* a worker or iframe). The rectangular frames in Figure 2 are labeled contexts. As contexts may be nested, labeled browsing contexts allow hierarchical confinement, whose importance for supporting nesting of untrusted libraries we discussed in Section 2.2.

When one browsing context sends sensitive information to another, a sending context can use *labeled communication* to confine the potentially untrusted code receiving the information. This enables symmetric confinement, whose importance in building applications that compose mutually distrusting scripts we articulated in Section 2.2. In Figure 2, the arrows between compartments indicate labeled communication, where a subscript on the communicated data denotes the data's label.

COWL may grant a labeled browsing context one or more *privileges,* each with respect to an origin, and each of which reflects trust that the scripts executing within

[4] In prior work, we described how confinement can subsume today's browser security primitives, and advocated replacing them entirely with a clean-slate, confinement-based model [49]. In this paper, we instead prioritize incremental deployability, which requires coexistence alongside the status quo model.

that context will not violate the secrecy and integrity of that origin's data, *e.g.,* because the browser retrieved them from that origin. A privilege authorizes scripts within a context to execute certain operations, such as *declassification* and *delegation*, whose abuse would permit the release of sensitive information to unauthorized parties. In COWL, we express privilege in terms of origins. The crown icon in the left compartment in Figure 2 denotes that this compartment may execute privileged operations on data labeled with the origin `fb.com`—more succinctly, that the compartment holds the privilege for `fb.com`. The compartment uses that privilege to remain unconfined by declassifying the checker response labeled `fb.com`.

We now describe these three constructs in greater detail.

3.1 Labeled Browsing Contexts

A COWL application consists of multiple labeled contexts. Labeled contexts extend today's browser contexts, used to isolate iframes, pages, *etc.*, with MAC *labels*. A context's label specifies the security policy for all data within the context, which COWL enforces by restricting the flow of information to and from other contexts and servers.

As we have proposed previously [33, 49], a label is a pair of boolean formulas over origins: a *secrecy* formula specifying which origins may read a context's data, and an *integrity* formula specifying which origins may write it. For example, only Amazon or Chase may read data labeled ⟨`amazon.com` ∨ `chase.com`, `amazon.com`⟩, and only Amazon may modify it.[5] Amazon could assign this label to its order history page to allow a Chase-hosted mashup to read the user's purchases. On the other hand, after a third-party mashup hosted by `mint.com` (as described in Section 2.2) reads *both* the user's Chase bank statement data *and* Amazon purchase data, the label on data produced by the third-party mashup will be ⟨`amazon.com` ∧ `chase.com`, `mint.com`⟩. This secrecy label component specifies that the data may be sensitive to both parties, and without both their consent (see Section 3.3), it should only be read by the user; the integrity label component, on the other hand, permits only code hosted by Mint to modify the resulting data.

COWL enforces label policies in a MAC fashion by only allowing a context to communicate with other contexts or servers whose labels are at least as restricting. (A server's "label" is simply its origin.) Intuitively, when a context wishes to send a message, the target must not allow additional origins to read the data (preserving secrecy). Dually, the source context must not be writable by origins not otherwise trusted by the target. That is, the source must be at least as trustworthy as the target. We say that such a target label "subsumes" the source label. For example, a context labeled ⟨`amazon.com`, `mint.com`⟩ can send messages to one labeled ⟨`amazon.com` ∧ `chase.com`, `mint.com`⟩, since the latter is trusted to preserve the privacy of `amazon.com` (and `chase.com`). However, communication in the reverse direction is not possible since it may violate the privacy of `chase.com`. In the rest of this paper, we limit our discussion to secrecy and only comment on integrity where relevant; we refer the interested reader to [33] for a full description of the label model.

A context can freely *raise* its label, *i.e.,* change its label to any label that is more restricting, in order to receive a message from an otherwise prohibited context. Of course, in raising its label to read more sensitive data from another context, the context also becomes more restricted in where it can write. For example, a Mint context labeled ⟨`amazon.com`⟩ can raise its label to ⟨`amazon.com` ∧ `chase.com`⟩ to read bank statements, but only at the cost of giving up its ability to communicate with Amazon (or, for that matter, any other) servers. When creating a new context, code can impose an upper bound on the context's label to ensure that untrusted code cannot raise its label and read data above this *clearance*. This notion of clearance is well established [14, 17, 34, 35, 51]; we discuss its relevance to covert channels in Section 7.

As noted, COWL allows a labeled context to create additional labeled contexts, much as today's browsing contexts can create sub-compartments in the form of iframes, workers, *etc.* This functionality is crucial for compartmentalizing a system hierarchically, where the developer places code of different degrees of trustworthiness in separate contexts. For example, in the password checker example in Section 2.2, we create a child context in which we execute the untrusted checker script. Importantly, however, code should not be able to leak information by laundering data through a newly created context. Hence, a newly created context implicitly inherits the current label of its parent. Alternatively, when creating a child, the parent may specify an initial current label for the child that is *more* restrictive than the parent's, to confine the child further. Top-level contexts (*i.e.,* pages) are assigned a default label of `public`, to ensure compatibility with pages written for the legacy SOP. Such browsing contexts can be restricted by setting a `COWL-label` HTTP response header, which dictates the minimal document label the browser must enforce on the associated content.

COWL applications can create two types of context. First, an application can create standard (but labeled) contexts in the form of pages, iframes, workers, *etc.* Indeed, it may do so because a COWL application is merely a regular web application that additionally uses the COWL API. It thus is confined by MAC, in addition to today's web security policies. Note that to enforce MAC, COWL must mediate all pre-existing communication channels—even

[5] ∨ and ∧ denote disjunction and conjunction. A comma separates the secrecy and integrity formulas.

subtle and implicit channels, such as content loading—according to contexts' labels. We describe how COWL does so in Section 5.

Second, a COWL application can create labeled contexts in the form of *lightweight labeled workers (LWorkers)*. Like normal workers [44], the API exposed to LWorkers is minimal; it consists only of constructs for communicating with the parent, the XHR constructor, and the COWL API. Unlike normal workers, which execute in separate threads, an LWorker executes in the same thread as its parent, sharing its event loop. This sharing has the added benefit of allowing the parent to give the child (labeled) access to its DOM, any access to which is treated as both a read and a write, *i.e.,* bidirectional communication. Our third-party library example uses such a *DOM worker* to isolate the trusted application code, which requires access to the DOM, from the untrusted jQuery library. In general, LWorkers—especially when given DOM access—simplify the isolation and confinement of scripts (*e.g.,* the password strength checker) that would otherwise run in a shared context, as when loaded with `script` tags.

3.2 Labeled Communication

Since COWL enforces a label check whenever a context sends a message, the design described thus far is already symmetric: a source context can confine a target context by raising its label (or a child context's label) and thereafter send the desired message. To read this message, the target context must confine itself by raising its label accordingly. These semantics can make interactions between contexts cumbersome, however. For example, a sending context may wish to communicate with multiple contexts, and need to confine those target contexts with different labels, or even confine the same target context with different labels for different messages. And a receiving context may need unfettered communication with one or more origins for a time before confining itself by raising its label to receive a message. In the password-checker example application, the untrusted checker script at the right of Figure 2 exhibits exactly this latter behavior: it needs to communicate with untrusted remote origin `sketchy.ru` before reading the password labeled `fb.com`.

Labeled Blob Messages (Intra-Browser) To simplify communication with confinement, we introduce the *labeled Blob,* which binds together the payload of an individual inter-context message with the label protecting it. The payload takes the form of a serialized immutable object of type Blob [47]. Encapsulating the label with the message avoids the cumbersome label raises heretofore necessary in both sending and receiving contexts before a message may even be sent or received. Instead, COWL allows the developer sending a message from a context to specify the label to be attached to a labeled Blob; any label as or more restrictive than the sending context's current label may be specified (modulo its clearance). While the receiving context may receive a labeled Blob with no immediate effect on the origins with which it can communicate, it may only inspect the label, not the payload.[6] Only after raising its label as needed may the receiving context read the payload.

Labeled Blobs simplify building applications that incorporate distrust among contexts. Not only can a sender impose confinement on a receiver simply by labeling a message; a receiver can delay inspecting a sensitive message until it has completed communication with untrusted origins (as does the checker script in Figure 2). They also ease the implementation of integrity in applications, as they allow a context that is not trusted to modify content in some other context to serve as a passive conduit for a message from a third context that *is* so trusted.

Labeled XHR Messages (Browser–Server) Thus far we have focused on confinement as it arises when two browser contexts communicate. Confinement is of use in browser-server communication, too. As noted in Section 3.1, COWL only allows a context to communicate with a server (whether with XHR, retrieving an image, or otherwise) when the server's origin subsumes the context's label. Upon receiving a request, a COWL-aware web server may also wish to know the current label of the context that initiated it. For this reason, COWL attaches the current label to every request the browser sends to a server.[7] As also noted in Section 3.1, a COWL-aware web server may elect to label a response it sends the client by including a `COWL-label` header on it. In such cases, the COWL-aware browser will only allow the receiving context to read the XHR response if its current label subsumes that on the response.

Here, again, a context that receives labeled data—in this case from a server—may wish to defer raising its label until it has completed communication with other remote origins. To give a context this freedom, COWL supports *labeled XHR* communication. When a script invokes COWL's labeled XHR constructor, COWL delivers the response to the initiating script as a labeled Blob. Just as with labeled Blob intra-browser IPC, the script is then free to delay raising its label to read the payload of the response—and delay being confined—until after it has completed its other remote communication. For example, in the third-party mashup example, Mint only confines itself once it has received all necessary (labeled) responses from both Amazon and Chase. At this point it processes the data and displays results to the user, but it can no longer send requests since doing so may leak

[6] The label itself cannot leak information—COWL still ensures that the target context's label is at least as restricting as that of the source.

[7] COWL also attaches the current privilege; see Section 3.3.

information.[8]

3.3 Privileges

While confinement handily enforces secrecy, there are occasions when an application must eschew confinement in order to achieve its goals, and yet can uphold secrecy while doing so. For example, a context may be confined with respect to some origin (say, `a.com`) as a result of having received data from that origin, but may need to send an encrypted version of that data to a third-party origin. Doing so does not disclose sensitive data, but COWL would normally prohibit such an operation. In such situations, how can a context *declassify* data, and thus be permitted to send to an arbitrary recipient, or avoid the recipient's being confined?

COWL's *privilege* primitive enables safe declassification. A context may hold one or more privileges, each with respect to some origin. Possession of a privilege for an origin by a context denotes trust that the scripts that execute within that context will not compromise the secrecy of data from that origin. Where might such trust come from (and hence how are privileges granted)? Under the SOP, when a browser retrieves a page from `a.com`, any script within the context for the page is trusted not to violate the secrecy of `a.com`'s data, as these scripts are deemed to be executing on behalf of `a.com`. COWL makes the analogous assumption by granting the privilege for `a.com` to the context that retrieves a page from `a.com`: scripts executing in that context are similarly deemed to be executing on behalf of `a.com`, and thus are trusted not to leak `a.com`'s data to unauthorized parties—even though they can declassify data. Only the COWL runtime can create a new privilege for a valid remote origin upon retrieval of a page from that origin; a script cannot synthesize a privilege for a valid remote origin.

To illustrate the role of privileges in declassification, consider the encrypted Google Docs example application. In the implementation of this application atop COWL, code executing on behalf of `eff.org` (*i.e.,* in a compartment holding the `eff.org` privilege) with a current label $\langle$`eff.org` $\wedge$ `gdoc.com`$\rangle$ is permitted to send messages to a context labeled $\langle$`gdoc.com`$\rangle$. Without the `eff.org` privilege, this flow would not be allowed, as it may leak the EFF's information to Google.

Similarly, code can declassify information when unlabeling messages. Consider now the password checker example application. The left context in Figure 2 leverages its `fb.com` privilege to declassify the password strength result, which is labeled with its origin, to avoid (unecessarily) raising its label to `fb.com`.

COWL generally exercises privileges *implicitly:* if a context holds a privilege, code executing in that context will, with the exception of sending a message, always attempt to use it.[9] COWL, however, lets code control the use of privileges by allowing code to get and set the underlying context's privileges. Code can drop privileges by setting its context's privileges to **`null`**. Dropping privileges is of practical use in confining closely coupled untrusted libraries like jQuery. Setting privileges, on the other hand, increases the trust placed in a context by authorizing it act on behalf of origins. This is especially useful since COWL allows one context to *delegate* its privileges (or a subset of them) to another; this functionality is also instrumental in confining untrusted libraries like jQuery. Finally, COWL also allows a context to create privileges for *fresh* origins, *i.e.,* unique origins that do not have a real protocol (and thus do not map to real servers). These fresh origins are primarily used to *completely* confine a context: the sender can label messages with such an origin, which upon inspection will raise the receiver's label to this "fake" origin, thereby ensuring that it cannot communicate except with the parent (which holds the fresh origin's privilege).

4 APPLICATIONS

In Section 2.2, we characterized four applications and explained why the status-quo web architecture cannot accommodate them satisfactorily. We then described the COWL system's new browser primitives. We now close the loop by demonstrating how to build the aforementioned applications with the COWL primitives.

Encrypted Document Editor The key feature needed by an encrypted document editor is symmetric confinement, where two mutually distrusting scripts can each confine the other's use of data they send one another. Asymmetrically conferring COWL privileges on the distrusting components is the key to realizing this application.

Figure 3 depicts the architecture for an encrypted document editor. The editor has three components: a component which has the user's Google Docs credentials and communicates with the server (`gdoc.com`), the editor proper (also `gdoc.com`), and the component that performs encryption (`eff.org`). COWL provides privacy as follows: if `eff.org` is honest, then COWL ensures that the cleartext of the user's document is not leaked to any origin. If only `gdoc.com` is honest, then `gdoc.com` may be able to recover cleartext (*e.g.,* the encryptor may have used the null "cipher"), but the encryptor should not be able to exfiltrate the cleartext to anyone else.

How does execution of the encrypted document editor proceed? Initially, `gdoc.com` downloads (1) the en-

[8] To continuously process data in "streaming" fashion, one may partition the application into contexts that poll Amazon and Chase's servers for new data and pass labeled responses to the confined context that processes the payloads of the responses.

[9] While the alternative approach of explicit exercise of privileges (*e.g.,* when registering an `onmessage` handler) may be safer [23, 34, 51], we find it a poor fit with existing asynchronous web APIs.

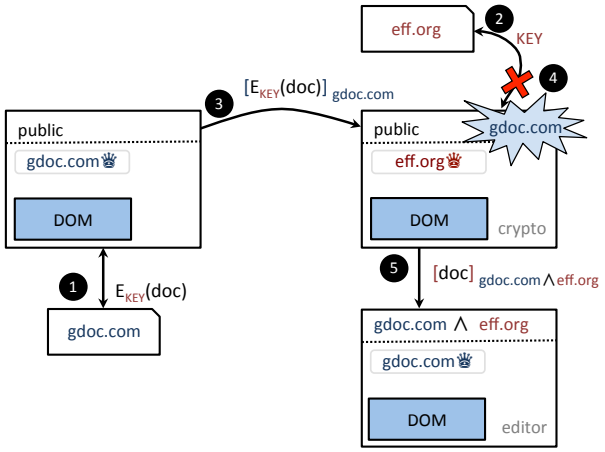

Figure 3: Encrypted document editor architecture.

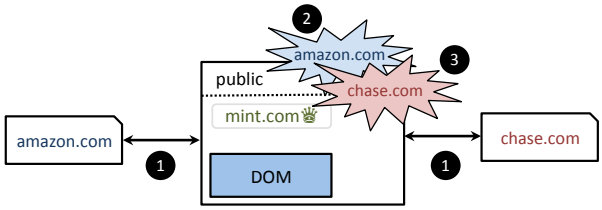

Figure 4: Third-party mashup under COWL.

crypted document from Google's servers. As the document is encrypted, it opens an iframe to `eff.org`, with initial label `public` so it can communicate with the `eff.org` server and download the private key (2) which will be used to decrypt the document. Next, it sends the encrypted document as a labeled Blob, with the label $\langle$`gdoc.com`$\rangle$ (3); the iframe unlabels the Blob and raises its label (4) so it can decrypt the document. Finally, the iframe passes the decrypted document (labeled as $\langle$`gdoc.com` $\wedge$ `eff.org`$\rangle$) to the iframe (5) implementing the editor proper.

To save the document, these steps proceed in reverse: the editor sends a decrypted document to the encryptor (5), which encrypts it with the private key. Next, the critical step occurs: the encryptor exercises its privileges to send a labeled blob of the encrypted document which is *only* labeled $\langle$`gdoc.com`$\rangle$ (3). Since the encryptor is the only compartment with the `eff.org` privilege, all documents must pass through it for encryption before being sent elsewhere; conversely, it itself cannot exfiltrate any data, as it is confined by `gdoc.com` in its label.

We have implemented a password manager atop COWL that lets users safely store passwords on third-party web-accessible storage. We elide its detailed design in the interest of brevity, and note only that it operates similarly to the encrypted document editor.

Third-Party Mashup Labeled XHR as composed with CORS is central to COWL's support for third-party mashups. Today's CORS policies are DAC-only, such that a server must either allow another origin to read its data and fully trust that origin not to disclose the data, or deny the other origin access to the data altogether. Under COWL, however, a server could CORS-whitelist a foreign origin to permit that origin to read its data, and by setting a label on its response, be safe in the knowledge that COWL would appropriately confine the foreign origin's scripts in the browser.

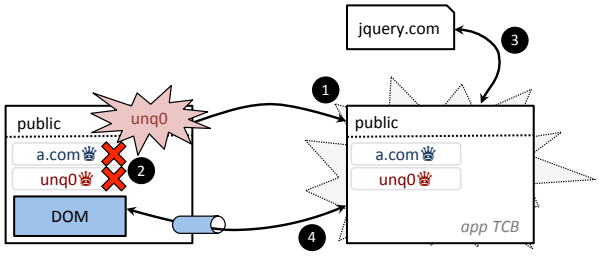

Figure 5: Privilege separation and library confinement.

Figure 4 depicts an application that reconciles a user's Amazon purchases and bank statement. Here, Chase and Amazon respectively expose authenticated read-only APIs for bank statements and purchase histories that whitelist known applications' origins, such as `mint.com`, but set MAC labels on responses.[10] As discussed in Section 7, with MAC in place, COWL allows users to otherwise augment CORS by whitelisting foreign origins on a per-origin basis. The mashup makes requests to both web sites using labeled XHR (1) to receive the bank statement and purchase history as labeled Blobs. Once all of the information is received, the mashup unlabels the responses and raises its context's label accordingly (2–3); doing so restricts communication to the web at large.

Note that in contrast to when solely using CORS, by setting MAC labels on responses, Chase and Amazon need not trust Mint to write bug-free code—COWL confines the Mint code to ensure that it cannot arbitrarily leak sensitive data. As we discuss in Section 7, however, a malicious Mint application could potentially leak data through covert channels. We emphasize that COWL nevertheless offers a significant improvement over the status quo, in which, *e.g.,* users give their login credentials to Mint, and thus not only trust Mint to keep their bank statements confidential, but also not to steal their funds!

Untrusted Third-Party Library COWL can confine tightly coupled untrusted third-party libraries like jQuery by delegating privileges to a trusted context and subsequently dropping them from the main page. In doing so, COWL completely confines the main page, and ensures that it can only communicate with the trusted and unconfined context. Here, the main page may start out with sensitive data in context, or alternatively, receive it from the trusted compartment.

[10] On authentication: note that when the browser sends any XHR (labeled or not) from a foreign origin to origin `chase.com`, it still includes any cookies cached for `chase.com` in the request.

```
interface Label :
  Label Label(String)
  Label and(String or Label)
  Label or(String or Label)
  bool subsumes(Label [,Privilege])
```

```
interface Privilege :
  Privilege FreshPrivilege()
  Privilege combine(Privilege)
  readonly attribute Label asLabel
```

(a) Labels and privileges.

```
interface LabeledBlob :
  readonly attribute Label label
  readonly attribute Blob blob
```

(b) Labeled Blobs.

```
interface COWL :
  static void enable()
  static attribute Label label
  static attribute Label clearance
  static attribute Privilege privilege
```

```
interface LWorker :
  LWorker LWorker(String, Label
                  [, Privilege, object])
  postMessage(object)
  attribute EventHandler onmessage
```

(c) Labeled compartments.

Figure 6: COWL programming interface in simplified WebIDL.

Figure 5 shows how to use COWL to confine the untrusted jQuery library referenced by a web page. The goal is to establish a separate DOM worker with the `a.com` privilege, while the main browsing context runs jQuery in confined fashion—without privileges or the ability to talk to the network. Initially the main browsing context holds the `a.com` privilege. The page generates a fresh origin `unq0` and spawns a DOM worker (1), delegating it both privileges. The main context then drops its privileges and raises its label to $\langle$`unq0`$\rangle$ (2). Finally, the trusted worker downloads jQuery (3) and injects the script content into the main context's DOM (4). When the library is loaded, the main context becomes untrusted, but also fully confined. As the trusted DOM worker holds both privileges, it can freely modify the DOM of the main context, as well as communicate with the wider web. One may view this DOM worker as a *firewall* between the page proper (with the untrusted library) and the rest of the world.

5 IMPLEMENTATION

We implemented COWL in Firefox 31.0a1 and Chromium 31.0.1612.0. Because COWL operates at a context granularity, it admits an implementation as a new DOM-level API for the Gecko and Blink layout engines, without any changes to the browsers' JavaScript engines. Figure 6 shows the core parts of this API. We focus on the Firefox implementation and only describe the Chromium one where the two diverge non-trivially.

Channel	Mechanism
`postMessage`	Cross-compartment wrappers[11]
DOM window properties	Cross-compartment wrappers
Content loading	CSP
XHR	CSP + DOM interposition
Browser storage	SOP + CSP (sandbox)
Other (*e.g.,* iframe height)	DOM interposition

Table 1: Confining code from exfiltrating data using existing browser mechanisms.

5.1 Labeled Browsing Contexts

Gecko's existing isolation model relies on JavaScript compartments, *i.e.,* disjoint JavaScript heaps, both for efficient garbage collection and security isolation [40]. To achieve isolation, Gecko performs all cross-compartment communication (*e.g.,* `postMessage` between iframes) through *wrappers* that implement the object-capability *membrane* pattern [21, 22]; membranes enable sound reasoning about "border crossing" between compartments. Wrappers ensure that an object in one compartment can never directly reference another object in a different compartment. Wrappers also include a security policy, which enforces all inter-compartment access control checks specified by the SOP. Security decisions are made with respect to a compartment's security principal, which contains the origin and CSP of the compartment.

Since COWL's security model is very similar to this existing model, we can leverage these wrappers to introduce COWL's new security policies. We associate a label, clearance, and privilege with each compartment alongside the security principal. Wrappers consider all of these properties together when making security decisions.

Intra-Browser Confinement As shown in Table 1, we rely on wrappers to confine cross-compartment communication. Once confinement mode is enabled, we "recompute" all cross-compartment wrappers to use our MAC wrapper policy and thereby ensure that all subsequent cross-compartment access is mediated not only by the SOP, but also by confinement. For `postMessage`, our policy ensures that the receiver's label subsumes that of the sender (taking the receiver's privileges into consideration); otherwise the message is silently dropped. For a cross-compartment DOM property access, we additionally check that the sender's label subsumes that of the receiver—*i.e.,* that the labels of the compartments are equivalent after considering the sender's privileges (in addition to the same-origin check performed by the SOP).

Blink's execution contexts (the dual to Gecko's compartments) do not rely on wrappers to enforce cross-context access control. Instead, Blink implements the

[11] Since the Chromium architecture does not have cross-compartment wrappers, we modify the DOM binding code to insert label checks.

SOP security checks in the DOM binding code for a limited subset of DOM elements that may allow cross-origin access. Since COWL policies are more fine-grained, we modified the binding code to extend the security checks to all DOM objects and also perform label checks when confinement mode is enabled. Unfortunately, without wrappers, shared references cannot efficiently be revoked (*i.e.,* without walking the heap). Hence, before enabling confinement mode, a page can create a same-origin iframe with which it shares references, and the iframe can thereafter leak any data from the parent even if the latter's label is raised. To prevent this eventuality, our current Chromium API allows senders to disallow unlabeling Blobs if the target created any children before entering confinement mode.

Our implementations of LWorkers, whose API appears in Figure 6c, reuse labeled contexts straightforwardly. In fact, the `LWorker` constructor simply creates a new compartment with a fresh origin that contains a fresh JavaScript global object to which we attach the XHR constructor, COWL API, and primitives for communicating with the parent (*e.g.,* `postMessage`). Since LWorkers may have access to their parents' DOM, however, our wrappers distinguish them from other contexts to bypass SOP checks and only restrict DOM access according to MAC. This implementation is very similar to the content scripts used by Chrome and Firefox extensions [10, 26].

Browser-Server Confinement As shown in Table 1, we confine external communication (including XHR, content loading, and navigation) using CSP. While CSP alone is insufficient for providing flexible confinement,[12] it sufficiently addresses our external communication concern by precisely controlling from where a page loads content, performs XHR requests to, *etc.* To this end, we set a custom CSP policy whenever the compartment label changes, *e.g.,* with `COWL.label`. For instance, if the effective compartment label is `Label("https://bank.ch").and("https://amazon.com")`, all the underlying CSP directives are set to `'none'` (*e.g.,* `default-src 'none'`), disallowing all network communication. We also disable navigation with the `'sandbox'` directive [46–48].

Browser Storage Confinement As shown in Table 1, we use the `sandbox` directive to restrict access to storage (*e.g.,* cookies and HTML5 local storage [47]), as have other systems [5]. We leave the implementation of labeled storage as future work.

6 Evaluation

Performance largely determines acceptance of new browser features in practice. We evaluate the performance of COWL by measuring the cost of our new primitives as well as their impact on legacy web sites that do not use COWL's features. Our experiments consist of micro-benchmarks of API functions and end-to-end benchmarks of our example applications. We conducted all measurements on a 4-core i7-2620M machine with 16GB of RAM running GNU/Linux 3.13. The browser retrieved applications from the Node.js web server over the loopback interface. We note that these measurements are harsh for COWL, in that they omit network latency and the complex intra-context computation and DOM rendering of real-world applications, all of which would mask COWL's overhead further. Our key findings include:

- COWL's latency impact on legacy sites is negligible.
- Confining code with LWorkers is inexpensive, especially when compared to iframes/Workers. Indeed, the performance of our end-to-end confined password checker is only 5 ms slower than that of an inlined `script` version.
- COWL's incurs low overhead when enforcing confinement on mashups. The greatest overhead observed is 16% (for the encrypted document editor). Again, the absolute slowdown of 16 ms is imperceptible by users.

	Firefox			Chromium		
	vanilla	unlabeled	labeled	vanilla	unlabeled	labeled
New iframe	14.4	14.5	14.4	50.6	48.7	51.8
New worker	15.9	15.4	0.9†	18.9	18.9	3.3†
Iframe comm.	0.11	0.11	0.12	0.04	0.04	0.04
XHR comm	3.5	3.6	3.7	7.0	7.4	7.2
Worker comm.	0.20	0.24	0.03‡	0.07	0.07	0.03‡

Table 2: Micro-benchmarks, in milliseconds.

6.1 Micro-Benchmarks

Context Creation Table 2 shows micro-benchmarks for the stock browsers (vanilla), the COWL browsers with confinement mode turned off (unlabeled), and with confinement mode enabled (labeled). COWL adds negligible latency to compartment creation; indeed, except for LWorkers (†), the differences in creation times are of the order of measurement variability. We omit measurements of labeled "normal" Workers since they do not differ from those of unlabeled Workers. We attribute COWL's iframe-creation speedup in Chromium to measurement variability. We note that the cost of creating LWorkers is considerably less than that for "normal" Workers, which run in separate OS threads (†).

Communication The iframe, worker, and XHR communication measurements evaluate the round-trip latencies across iframes, workers, and the network. For the XHR benchmark, we report the cost of using the labeled XHR constructor averaged over 10,000 requests. Our

[12] There are two primary reasons. First, JavaScript code cannot (yet) modify a page's CSP. And, second, CSP does not (yet) provide a directive for restricting in-browser communication, *e.g.,* with `postMessage`.

Chromium implementation uses an LWorker to wrap the unmodified XHR constructor, so the cost of labeled XHR incorporates an additional cross-context call. As with creation, communicating with LWorkers (‡) is considerably faster than with "normal" Workers. This speedup arises because a lightweight LWorker shares an OS thread and event loop with their parent.

Labels We measured the cost of setting/getting the current label and the average cost of a label check in Firefox. For a randomly generated label with a handful of origins, these operations take on the order of one microsecond. The primary cost is recomputing cross-compartment wrappers and the underlying CSP policy, which ends up costing up to 13ms (*e.g.,* when the label is raised from public to a third-party origin). For many real applications, we expect raising the current label to be a rare occurrence. Moreover, there is much room for optimization (*e.g.,* porting COWL to the newest CSP implementation, which sets policies 15× faster [19]).

DOM We also executed the Dromaeo benchmark suite [29], which evaluates the performance of core functionality such as querying, traversing, and manipulating the DOM, in Firefox and Chromium. We found the performance of the vanilla and unlabeled browsers to be on par: the greatest slowdown was under 4%.

6.2 End-to-End Benchmarks

To focus on measuring COWL's overhead, we compare our apps against similarly compartmentalized but non-secure apps—*i.e.,* apps that perform no security checks.

Password-Strength Checker We measure the average duration of creating a new LWorker, fetching an 8 KB checker script based on [24], and checking a password sixteen characters in length. The checker takes an average of 18 ms (averaged over ten runs) on Firefox (labeled), 4 ms less than using a Worker on vanilla Firefox. Similarly, the checker running on labeled Chromium is 5 ms faster than the vanilla counterpart (measured at 54 ms). In both cases COWL achieves a speedup because its LWorkers are cheaper than normal Workers. However, these measurements are roughly 5 ms slower than simply loading the checker using an unsafe `script` tag.

Encrypted Document Editor We measure the end-to-end time taken to load the application and encrypt a 4 KB document using the SJCL AES-128 library [32]. The total run time includes the time taken to load the document editor page, which in turn loads the encryption-layer iframe, which further loads the editor proper. On Firefox (labeled) the workload completes in 116 ms; on vanilla Firefox, a simplified and unconfined version completes in 100ms. On Chromium, the performance measurements were comparable; the completion time was within 1ms of 244ms. The most expensive operation in the COWL-enabled Firefox app is raising the current label, since it requires changing the underlying document origin and recomputing the cross-compartment wrappers and CSP.

Third-Party Mashup We implemented a very simple third-party mashup application that makes a labeled XHR request to two unaffiliated origins, each of which produces a response containing a 27-byte JSON object with a numerical property, and sums the responses together. The corresponding vanilla app is identical, but uses the normal XHR object. In both cases we use CORS to permit cross-origin access. The Firefox (labeled) workload completes in 41 ms, which is 6 ms slower than the vanilla version. As in the document editor the slowdown derives from raising the current label, though in this case only for a single iframe. On Chromium (labeled) the workload completes in 55 ms, 2 ms slower than the vanilla one; the main slowdown here derives from our implementing labeled XHR with a wrapping LWorker.

Untrusted Third-Party Library We measured the load time of a banking application that incorporates jQuery and a library that traverses the DOM to replace phone numbers with links. The latter library uses XHR in attempt to leak the page's content. We compartmentalize the main page into a public outer component and a sensitive iframe containing the bank statement. In both compartments, we place the bank's trusted code (which loads the libraries) in a trusted labeled DOM worker with access to the page's DOM. We treat the rest of the code as untrusted. As our current Chromium implementation does not yet support DOM access for LWorkers, we only report measurements for Firefox. The measured latency on Firefox (labeled) is 165 ms, a 5 ms slowdown when compared to the unconfined version running on vanilla Firefox. Again, COWL prevents sensitive content from being exfiltrated and incurs negligible slowdown.

7 Discussion and Limitations

We now discuss the implications of certain facets of COWL's design, and limitations of the system.

User-Configured Confinement Recall that in the status-quo web security architecture, to allow cross-origin sharing, a server must grant individual foreign origins access to its data with CORS in an all-or-nothing, DAC fashion. COWL improves this state of affairs by allowing a COWL-aware server to more finely restrict how its shared data is disseminated—*i.e.,* when the server grants a foreign origin access to its data, it can confine the foreign origin's script(s) by setting a label on responses it sends the client.

Unfortunately, absent a permissive CORS header that whitelists the origins of applications that a user wishes to use, the SOP prohibits foreign origins from reading responses from the server, even in a COWL-enabled browser. Since a server's operator may not be aware of all applications its users may wish to use, the result is

usually the same status-quo unpalatable choice between functionality and privacy—*e.g.,* give one's bank login credentials to Mint, or one cannot use the Mint application. For this reason, our COWL implementation lets browser users augment CORS by configuring for an origin (*e.g.,* `chase.com`) any foreign origins (*e.g.,* `mint.com`, `benjamins.biz`) they wish to additionally whitelist. In turn, COWL will confine these client-whitelisted origins (*e.g.,* `mint.com`) by labeling every response from the configured origin (`chase.com`). COWL obeys the server-supplied label when available and server whitelisting is *not* provided. Otherwise, COWL conservatively labels the response with a *fresh* origin (as described in Section 3.3). The latter ensures that once the response has been inspected, the code cannot communicate with *any* server, including at the *same* origin, since such requests carry the risks of self-exfiltration [11] and cross-site request forgery [39].

Covert Channels In an ideal confinement system, it would always be safe to let untrusted code compute on sensitive data. Unfortunately, real-world systems such as browsers typically exhibit *covert* channels that malicious code may exploit to exfiltrate sensitive data. Since COWL extends existing browsers, we do not protect against covert channel attacks. Indeed, malicious code can leverage covert channels already present in today's browsers to leak sensitive information. For instance, a malicious script within a confined context may be able to modulate sensitive data by varying rendering durations. A less confined context may then in turn exfiltrate the data to a remote host [20]. It is important to note, however, that COWL does not introduce new covert channels—our implementations re-purpose existing (software-based) browser isolation mechanisms (V8 contexts and SpiderMonkey compartments) to enforce MAC policies. Moreover, these MAC policies are generally more restricting than existing browser policies: they prevent unauthorized data exfiltration through *overt* channels and, in effect, force malicious code to resort to using covert channels.

The only fashion in which COWL relaxes status-quo browser policies is by allowing users to override CORS to permit cross-origin (labeled) sharing. Does this functionality introduce new risks? Whitelisting is user controlled (*e.g.,* the user must explicitly allow `mint.com` to read `amazon.com` and `chase.com` data), and code reading cross-origin data is subject to MAC (*e.g.,* `mint.com` cannot arbitrarily exfiltrate the `amazon.com` or `chase.com` data after reading it). In contrast, today's mashups like `mint.com` ask users for their passwords. COWL is strictly an improvement: under COWL, when a user decides to trust a mashup integrator such as `mint.com`, she *only* trusts the app to not leak her data through covert channels. Nevertheless, users can make poor security choices. Whitelisting malicious origins would be no exception; we recognize this as a limitation of COWL that must be communicated to the end-user.

A trustworthy developer can leverage COWL's support for *clearance* when compartmentalizing his application to ensure that only code that actually relies on cross-origin data has access to it. Clearance is a label that serves as an upper bound on a context's current label. Since COWL ensures that the current label is adjusted according to the sensitivity of the data being read, code cannot read (and thus leak) data labeled above the clearance. Thus, Mint can assign a "low" clearance to untrusted third-party libraries, *e.g.,* to keep `chase.com`'s data confidential. These libraries will then not be able to leak such data through covert channels, even if they are malicious.

Expressivity of Label Model COWL uses DC labels [33] to enforce confinement according to an information flow control discipline. Although this approach captures a wide set of confinement policies, it is not expressive enough to handle policies with a circular flow of information [6] or some policies expressible in more powerful logics (*e.g.,* first order logic, as used by Nexus [30]). DC labels are, however, as expressive as other popular label models [25], including Myers and Liskov's Decentralized Label Model [27]. Our experience implementing security policies with them thus far suggests they are expressive enough to support featureful web applications.

We adopted DC labels largely because their fit with web origins pays practical dividends. First, as developers already typically express policies by whitelisting origins, we believe they will find DC labels intuitive to use. Second, because both DC labels and today's web policies are defined in terms of origins, the implementation of COWL can straightforwardly reuse the implementation of existing security mechanisms, such as CSP.

8 Related Work

Existing browser confinement systems based on information flow control can be classified either as *fine-grained* or *coarse-grained*. The former associate IFC policies with individual objects, while the latter associate policies with entire browsing contexts. We compare COWL to previously proposed systems in both categories, then contrast the two categories' overall characteristics.

Coarse-grained IFC COWL shares many features with existing coarse-grained systems. BFlow [50], for example, allows web sites to enforce confinement policies stricter than the SOP via *protection zones*—groups of iframes sharing a common label. However, BFlow cannot mediate between mutually distrustful principals—*e.g.,* the encrypted document editor is not directly implementable with BFlow. This is because only asymmetric confinement is supported—a sub-frame cannot impose any restrictions on its parent. For the same reasons, BFlow cannot support applications that require security policies more flexible

than the SOP, such as our third-party mashup example. These differences reflect different goals for the two systems. BFlow's authors set out to confine untrusted third-party scripts, while we also seek to support applications that incorporate code from mutually distrusting parties.

More recently, Akhawe *et al.* propose the data-confined sandbox (DCS) system [5], which allows pages to intercept and monitor the network, storage, and cross-origin channels of `data:` URI iframes. The limitation to `data:` URI iframes means DCS cannot confine the common case of a service provided in an iframe [31]. Like BFlow, DCS does not offer symmetric confinement, and does not incorporate functionality to let developers build applications like third-party mashups.

Fine-grained IFC Per-object-granularity IFC makes it easier to confine untrusted libraries that are closely coupled with trusted code on a page (*e.g.,* jQuery) and avoid the problem of *over-tainting,* where a single context accumulates taint as it inspects more data.

JSFlow [15] is one such fine-grained JavaScript IFC system, which enforces policies by executing JavaScript in an interpreter written in JavaScript. This approach incurs a two order of magnitude slowdown. JSFlow's authors suggest that this cost makes JSFlow a better fit for use as a development tool than as an "always-on" privacy system for users' browsers. Additionally, JSFlow does not support applications that rely on policies more flexible than the SOP, such as our third-party mashup example.

The FlowFox fine-grained IFC system [12] enforces policies with secure-multi execution (SME) [13]. SME ensures that no leaks from a sensitive context can leak into a less sensitive context by executing a program multiple times. Unlike JSFlow and COWL, SME is not amenable to scenarios where declassification plays a key role (*e.g.,* the encrypted editor or the password manager). FlowFox's labeling of user interactions and metadata (history, screen size, *etc.*) do allow it to mitigate history sniffing and behavior tracking; COWL does not address these attacks.

While fine-grained IFC systems may be more convenient for developers, they impose new language semantics for developers to learn, require invasive modifications to the JavaScript engine, and incur greater performance overhead. In contrast, because COWL repurposes familiar isolation constructs and does not require JavaScript engine modifications, it is relatively straightforward to add to legacy browsers. It also only adds overhead to cross-compartment operations, rather than to all JavaScript execution. The typically short lifetime of a browsing context helps avoid excessive accumulation of taint. We conjecture that coarse-grained and fine-grained IFC are equally expressive, provided one may use arbitrarily many compartments—a cost in programmer convenience. Finally, coarse- and fine-grained mechanisms are not mutually exclusive. For instance, to confine legacy (non-compartmentalized) JavaScript code, one could deploy JSFlow within a COWL context.

Sandboxing The literature on sandboxing and secure subsets of JavaScript is rich, and includes Caja [1], BrowserShield [28], WebJail [37], TreeHouse [18], JSand [4], SafeScript [36], Defensive JavaScript [9], and Embassies [16]). While our design has been inspired by some of these systems (*e.g.,* TreeHouse), the usual goals of these systems are to mediate security-critical operations, restrict access to the DOM, and restrict communication APIs. In contrast to the mandatory nature of confinement, however, these systems impose most restrictions in discretionary fashion, and are thus not suitable for building some of the applications we consider (in particular, the encrypted editor). Nevertheless, we believe that access control and language subsets are crucial complements to confinement for building robustly secure applications.

9 Conclusion

Web applications routinely pull together JavaScript contributed by parties untrusted by the user, as well as by mutually distrusting parties. The lack of confinement for untrusted code in the status-quo browser security architecture puts users' privacy at risk. In this paper, we have presented COWL, a label-based MAC system for web browsers that preserves users' privacy in the common case where untrusted code computes over sensitive data. COWL affords developers flexibility in synthesizing web applications out of untrusted code and services while preserving users' privacy. Our positive experience building four web applications atop COWL for which privacy had previously been unattainable in status-quo web browsers suggests that COWL holds promise as a practical platform for preserving privacy in today's pastiche-like web applications. And our measurements of COWL's performance overhead in the Firefox and Chromium browsers suggest that COWL's privacy benefits come at negligible end-to-end cost in performance.

Acknowledgements

We thank Bobby Holley, Blake Kaplan, Ian Melven, Garret Robinson, Brian Smith, and Boris Zbarsky for helpful discussions of the design and implementation of COWL. We thank Stefan Heule and John Mitchell for useful comments on formal aspects of the design. And we thank our shepherd Mihai Budiu, the anonymous reviewers, and the UPenn CRASH team for their helpful comments. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, the EPSRC under grant EP/K032542/1, the Swedish research agencies VR and STINT, the Barbro Osher Pro Suecia foundation, and by multiple gifts from Google (to Stanford and UCL). Deian Stefan and Edward Z. Yang are supported through the NDSEG Fellowship Program.

REFERENCES

[1] Google Caja. A source-to-source translator for securing JavaScript-based web content. http://code.google.com/p/google-caja/, 2013.

[2] Mint. http://www.mint.com/, 2013.

[3] jQuery Usage Statistics: Websites using jQuery. http://trends.builtwith.com/javascript/jQuery, 2014.

[4] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC*, 2012.

[5] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song. Data-confined HTML5 applications. In *ESORICS*, 2013.

[6] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *Security and Privacy*, 1995.

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

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

[9] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security*, 2013.

[10] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the Google Chrome extension security architecture. In *USENIX Security*, 2012.

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

[12] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *CCS*, 2012.

[13] D. Devriese and F. Piessens. Noninterference through Secure Multi-Execution. In *Security and Privacy*, 2010.

[14] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *OSDI*, 2005.

[15] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC*, 2014.

[16] J. Howell, B. Parno, and J. R. Douceur. Embassies: Radically refactoring the Web. In *NSDI*, 2013.

[17] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your ifcexception are belong to us. In *Security and Privacy*, 2013.

[18] L. Ingram and M. Walfish. Treehouse: JavaScript sandboxes to help web developers help themselves. In *USENIX ATC*, 2012.

[19] C. Kerschbaumer. Faster Content Security Policy (CSP). https://blog.mozilla.org/security/2014/09/10/faster-csp/, 2014.

[20] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson. Cross-origin pixel stealing: timing attacks using CSS filters. In *CCS*, 2013.

[21] M. S. Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006.

[22] M. S. Miller and J. S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *ASIAN*, 2003.

[23] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://zesty.ca/capmyths/usenix.pdf.

[24] S. Moitozo. http://www.geekwisdom.com/js/passwordmeter.js, 2006.

[25] B. Montagu, B. C. Pierce, and R. Pollack. A theory of information-flow labels. In *CSF*, June 2013.

[26] Mozilla. Add-on builder and SDK. https://addons.mozilla.org/en-US/developers/docs/sdk/, 2013.

[27] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *TOSEM*, 9(4), 2000.

[28] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic HTML. *TWEB*, 1(3), Sept. 2007.

[29] J. Reisg. Dromaeo: JavaScript performance testing. http://dromaeo.com/, 2014.

[30] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *SOSP*, 2011.

[31] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postMessage in HTML5 websites. In *NDSS*, 2013.

[32] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in JavaScript. In *ACSAC*, 2009.

[33] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *NordSec*, 2011.

[34] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, 2011.

[35] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *ICFP*, 2012.

[36] M. Ter Louw, P. H. Phung, R. Krishnamurti, and V. N. Venkatakrishnan. SafeScript: JavaScript transformation for policy enforcement. In *Secure IT Systems*, 2013.

[37] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: least-privilege integration of third-party components in web mashups. In *ACSAC*, 2011.

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

[39] B. Vibber. CSRF token-stealing attack (user.tokens). `https://bugzilla.wikimedia.org/show_bug.cgi?id=34907`, 2014.

[40] G. Wagner, A. Gal, C. Wimmer, B. Eich, and M. Franz. Compartmental memory management in a modern web browser. *SIGPLAN Notices*, 46 (11), 2011.

[41] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. *ACM SIGOPS Operating Systems Review*, 41(6), 2007.

[42] WC3. Content Security Policy 1.0. `http://www.w3.org/TR/CSP/`, 2012.

[43] WC3. HTML5 web messaging. `http://www.w3.org/TR/webmessaging/`, 2012.

[44] WC3. Web Workers. `http://www.w3.org/TR/workers/`, 2012.

[45] WC3. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`, 2013.

[46] WC3. Content Security Policy 1.1. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`, 2013.

[47] WC3. HTML5. `http://www.w3.org/TR/html5/`, 2013.

[48] WHATWG. HTML living standard. `http://developers.whatwg.org/`, 2013.

[49] E. Yang, D. Stefan, J. Mitchell, D. Mazières, P. Marchenko, and B. Karp. Toward principled browser security. In *HotOS*, 2013.

[50] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *EuroSys*, 2009.

[51] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

[52] M. Zelwski. Browser security handbook, part 2. `HTtp://code.google.com/p/browsersec/wiki/Part2`, 2011.