Run-time Monitoring and Formal Analysis of Information Flows in Chromium

Lujo Bauer Shaoying Cai* Limin Jia Timothy Passaro Michael Stroucken Yuan Tian
Carnegie Mellon University *Institute for Infocomm Research
{lbauer,liminjia,tpassaro,mxs,yt}@cmu.edu *cais@i2r.a-star.edu.sg

***Abstract*—Web browsers are a key enabler of a wide range of online services, from shopping and email to banking and health services. Because these services frequently involve handling sensitive data, a wide range of web browser security policies and mechanisms has been implemented or proposed to mitigate the dangers posed by malicious code and sites.**

This paper describes an approach for specifying and enforcing flexible information-flow policies on the Chromium web browser. Complementing efforts that focus on information-flow enforcement on JavaScript, our approach focuses on an existing browser and encompasses a broad range of browser features, from pages and scripts to DOM elements, events, persistent state, and extensions. In our approach, which is a coarse-grained, light-weight implementation of taint tracking, entities in the browser are annotated with information-flow labels that specify policy and track information flows. We develop a detailed formal model of our approach, for which we prove noninterference. We also develop a corresponding prototype system built on top of Chromium. We demonstrate, and experimentally confirm, that the system can enforce many existing browser policies, as well as practically useful policies beyond those enforceable in standard web browsers.

I. Introduction

Web browsers are used to access a wide range of services—from shopping and email to banking, health services, and document editing. While bringing about increased convenience and productivity, the continuing rise in popularity of web services also causes users to incur significant risks. Using these services causes users' confidential data—such as passwords, bank account numbers, and emails—to be exposed to the web browser, to scripts running in pages loaded in the browser, and to browser extensions and plugins. Scripts and extensions are particularly difficult to trust: even simple web pages commonly load multiple page scripts of often dubious provenance; similarly, tens of thousands of extensions are available to be installed on browsers like Firefox and Chrome, and popular extensions are downloaded by tens of millions of users.

To reduce the dangers posed by scripts and extensions, today's browsers use numerous security mechanisms, from the same origin policy (SOP), content security policies (CSPs), and permissions to process isolation and isolated worlds. Despite all these mechanisms, however, risks to users' data remain. Dangerous new attacks and vulnerabilities are regularly demonstrated. For example, page scripts and extensions can track user browsing behavior [14], [24], capture user input in web forms [36], steal cookies and confidential web page content [3], [12], and even hijack user sessions and make web service requests on behalf of the user [9].

A promising recent development is the emergence of information-flow control as an approach for preventing many such script misbehaviors [19], [27], [42], [48], [49]. Some such approaches are more heavy-weight; they enforce fine-grained policies, but require implementing a new JavaScript interpreter [19], [27]. In contrast, BFlow and COWL compartmentalize scripts and assign policies at the granularity of compartments that encapsulate content from a single origin [42], [49]. They enforce coarse-grained policies on communication across compartments and build on existing JavaScript engines.

Complementing existing research on information flow in web browsers, in this paper we pursue a light-weight, dynamic, taint-tracking-based approach to constrain information flows within a browser and to prevent secrets from leaving the browser. We provide a formal accounting of information flows among the many entities, both static and dynamic, which inhabit browsers. These include dynamic entities such as page scripts and extension scripts; ephemeral ones like browser tabs, loaded web pages, and events (e.g., a button click); and persistent entities like cookies, history, and bookmarks. We explore whether an information-flow enforcement system can encompass all the above entities and implement useful policies without unduly impacting regular browsing functionality and with reasonable run-time overhead.

More specifically, we develop an approach for specifying and enforcing flexible information-flow policies for Chromium. Compared to other successes in applying formal information-flow reasoning to web browsers, we believe our approach is novel in the combination of richness of policy specification it allows and breadth of the space it encompasses.

In our approach, all entities in a browser are described using information-flow labels. For web pages, extensions, and some APIs, these labels express the policy of the author or user, or built-in browser policies, e.g., a superset of a web page's CSP or variants of the SOP. For other entities in the browser, including ephemeral ones like button-click events and persistent ones like cookies and browser history, labels are created automatically and serve to track information flows and prevent those that violate specified policies.

NDSS '15, 8-11 February 2015, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2015.23295

Our information-flow labels are based on entity origin: domains in URLs, extension IDs, and the browser's user. Labels can express declassification and endorsement policies, permitting controlled flow of information to untrusted components when this is required to achieve specific functionality (e.g., a password stored by a password manager is allowed to be sent to a web site if the user clicks the page's login button).

Our approach is sufficiently expressive to encompass policies that are currently explicitly specified in the browser, such as specific CSP, domain relaxation, and cross-origin resource-sharing policies; and policies expressed via extension permissions. Our approach also explicitly exposes policy composition, e.g., between the CSP of a web page, the CSP of content loaded in iframes, and policies specified by developers of extensions whose content scripts are active in the page. This enables our approach to represent both the ad-hoc ways in which browsers currently handle policy composition (e.g., conflicts between policies of web pages and extension content scripts) and more principled and fine-grained ways of resolving policy conflicts.

We precisely define our approach by means of a formal model, which we show obeys trace-based noninterference. We concretely demonstrate through a prototype implementation on Chromium that our system can be used to enforce practically useful policies, including those that prevent malicious extensions from stealing user input or other extensions' secrets and colluding via explicit shared state. We also discuss inherent tradeoffs between security and functionality via case studies that expose the need for a semi-trusted script or extension to have dangerous declassification privileges in order to implement seemingly innocuous functionality.

In summary, this paper makes the following contributions:

- An approach for specifying and enforcing information-flow polices that encompasses the full range of components (entities) within the browser.
- A formalization of our approach in a model of an extended version of Chromium, for which we have proved a noninterference theorem.
- A functional prototype implementation on Chromium that demonstrates reasonable performance overhead while supporting fine-grained policies.
- Examples of enforcement of practically relevant policies enabled by our approach, and insights about tradeoffs implicitly made in everyday web browsing.

We do not contend that the design described in this paper is a complete, ready-to-use replacement for existing browser security mechanisms. Instead, we believe that it is an interesting point in the design space of possible solutions; as such, it is important to explore, and doing so helps provide a concrete basis for further discussion toward more satisfying practical solutions. Further, the browser entities that we model—shared state (e.g., cookies, history, bookmarks, DOM) and blocking and nonblocking event handlers—are common to many browser infrastructures and our model can potentially be reused in projects that require formal models of browser internals.

The rest of this paper proceeds as follows. We start with a motivating example, in Section I-A. Section II describes background and closely related work. Section III gives an overview of our approach, including describing how labels are specified and used to enforce policies. We describe the formal model and analysis in Section IV and the prototype implementation in Section V. Section VI discusses how our approach can be used to implement existing browser security mechanisms.

A. Motivating Example

We use a password manager extension as a motivating scenario to demonstrate the features of our approach. The password manager, $\text{ext}_{\text{pwdMgr}}$, collects usernames and passwords entered into login forms and saves them for future use. On subsequent visits to previously visited pages, the password manager automatically fills in usernames and passwords.

Suppose that a page for which $\text{ext}_{\text{pwdMgr}}$ has saved the user's password is cnn.com. Several other extensions, including ext_{eve}—a malicious extension masquerading as a web page translation extension—are installed in the browser and have access to cnn.com. cnn.com loads advertising content, which includes page scripts, from ad sites like ad.com.

Given currently deployed browser security mechanisms, ext_{eve} and scripts from ad.com may be able to interfere with the password manager in a number of ways, including:

- Once $\text{ext}_{\text{pwdMgr}}$ fills in a login form with previously saved passwords, these can be read by ext_{eve} and ad.com scripts and communicated to other sites.
- ext_{eve} can collect saved passwords even for sites that the user does not herself visit by opening new tabs that load arbitrary web pages; if $\text{ext}_{\text{pwdMgr}}$ has saved passwords for those web pages, it may automatically fill them in.
- ext_{eve} and page scripts can automatically cause the user to log in to sites for which $\text{ext}_{\text{pwdMgr}}$ stores a password by fabricating form-submit events.

As we describe in Section III, our approach makes it possible to define and enforce policies that prevent these attacks, as well similar attacks carried out by more constrained page scripts or extensions that collude to achieve similar malicious functionality.

II. Related Work

Improving the security and reliability of browsers has received much attention from both academia and industry [8], [10], [18], [25], [43], [45]. Most popular browsers, such as Chrome and Firefox, enforce forms of component isolation and privilege separation. Even with such architectures, new ways to exploit users by launching attacks within the browser or to compromise the browser are frequently reported [21], [31], [47]. Allowing browsers to be further extended by third-party extensions has brought a new set of security concerns [7], [12], [34], [35], [46]. Our information-flow policy enforcement mechanism enhances existing browser infrastructure, and can be used to enforce flexible information-flow policies, which can mitigate some of the above-mentioned attacks.

Ad-hoc browser security features (e.g., Content Security Policy (CSP) [44], origin header [9], X-Frame-Options [30])

have been adopted to prevent common attacks such as cross-site scripting (XSS) [26], cross-site request forgery (CSRF), and clickjacking [29]. Many of these security features constrain information flows. Our approach can enforce richer and more precise policies and can be used to approximate the information-flow policies enforced by those security features.

Enforcing information-flow policies has been an active area of research. Some develop novel information-flow type systems that enforce noninterference properties statically (cf. [41]); others use run-time monitoring, or hybrid techniques (e.g., [4], [6], [16], [28], [37], [40]). Much work has investigated preventing information leakage via JavaScript in browsers (e.g., [6], [13], [17], [19], [27], [28]). These works typically protect data at the variable level, which is much lower level than our system's entity-level granularity. This means that we do not need to track every instruction in the JavaScript engine, and instead only enforce policies when API calls are made and when event handlers and callbacks are invoked. This is not only more efficient, but also fits nicely into the browser's model of classifying information based on origin. Scripts from the same origin in our system have the same information-flow labels. Further, our work also encompasses other browser components, such as DOM elements and extensions.

In terms of policy specification and enforcement granularity, our system is similar to BFlow [49] and COWL [42]. There, each compartment, which includes scripts and DOM from the same origin, is associated with an information-flow label specifying its policies. Aside from the differences in the choice of policy language, we additionally examine policy composition, explore different ways to implement shared states (i.e., DOM, cookies, history, and bookmarks), and build a formal model of our design and analyze its security guarantees.

Bohannon et al. formally modeled the core of Firefox [15]. Their model contains all key components of a browser, including the DOM, cookies, and bookmarks. In particular, they model the browser as a reactive system, where entities in the browser are modeled as consumers and producers that communicate by sending and receiving messages through channels. In contrast, our model of the browser uses different abstractions, supports policy specification, and is compatible with a real browser. Recent work by Fett et al. also models a browser as a transition system [22]. Some of their transition rules overlap with ours. Our model additionally includes our information-flow tracking mechanism and encompasses browser extensions. The differences in the models are driven by different analysis goals: Fett et al. aim to analyze browser single sign-on systems, while we aim to prove noninterference.

III. System Overview

In this section we first define the threat model that our approach addresses (Section III-A). We then describe how policies are specified (Section III-B), outline the enforcement approach (Section III-C), and illustrate the approach on an example (Section III-D).

A. Threat Model

Attackers can be scripts in the DOM, extension cores, and extension (content) scripts. (A Chromium extension is composed of an extension *core*, of which only one instance is maintained in the browser, and one or more *content scripts* that are injected by the browser into loaded pages.) We assume an attacker is associated with a label (policy) that describes its capabilities, and that this label accurately describes the desired restrictions on the attacker's behavior. An attack is successful if the attacker is able to violate this policy, e.g., learn secrets that belong to the user or other browser extensions or influence access to sensitive resources. This notion of attackers is standard in information-flow systems. Specifying such policies correctly, e.g., without unwittingly giving entities the capability to access sensitive information, is often a challenging user-interface issue [39], but is largely orthogonal to the design of the information-flow enforcement mechanism.

We assume that the attacker can interact with other browser components only via standard browser and JavaScript APIs, and does not have out-of-band access to low-level system behaviors such as power consumption or the scheduler. We do not consider leaks through timing-based covert channels.

B. Policy Specification

Following prior work on OS-level DIFC [32], [33], in our approach each browser entity is associated with an information-flow label, specifying allowed flows of information between entities. (We use *entity* instead of *component*, as the latter is overloaded in this context.) A novelty of our approach is the breadth of entities it encompasses (see Figure 1). These fall into two categories: (1) web pages and their DOM trees, page scripts, and extensions (and their instances)—entities for which authors and the user can already specify policies (e.g., via CSPs); (2) APIs (e.g., for network requests), shared browser state (e.g., history), and events (e.g., mouse clicks)—entities to which the browser controls access via simple, fixed policies.

Our approach attaches to each of these entities a policy label. The label describes the privileges the user, author, or browser confers on that entity, as well as potential reductions in those privileges that result from interactions with other entities. Reference monitors examine labels when requests (including messages or calls) cross entity boundaries. They prevent requests that are not consistent with the caller's and callee's labels, and, when appropriate, augment the labels of callees to track the flow of sensitive information.

1) Information-flow Labels: We next describe the structure of and features supported by the labels in the system.

Basic labels An information-flow label, written (S, I, D), is composed of a secrecy label S, an integrity label I, and a declassification label D. The basic secrecy label is a set of secrecy tags $\{s_1, \ldots, s_n\}$. Each secrecy tag represents an origin of a secret. We treat the hostname parts of URLs, extension IDs, and the user operating the browser (notated as the tag user) as origins. The integrity label is a set of integrity tags $\{i_1, \ldots, i_n\}$. Each integrity tag represents the privilege to access a sensitive resource (namely, APIs). Even though these tags are reminiscent of permissions, our enforcement mechanism treats them in such a way that it can prevent the privilege escalation that commonly occurs in permission-based systems. The declassification label is a set of capabilities for endorsement $(+i)$, declassification $(-s)$, and reclassification

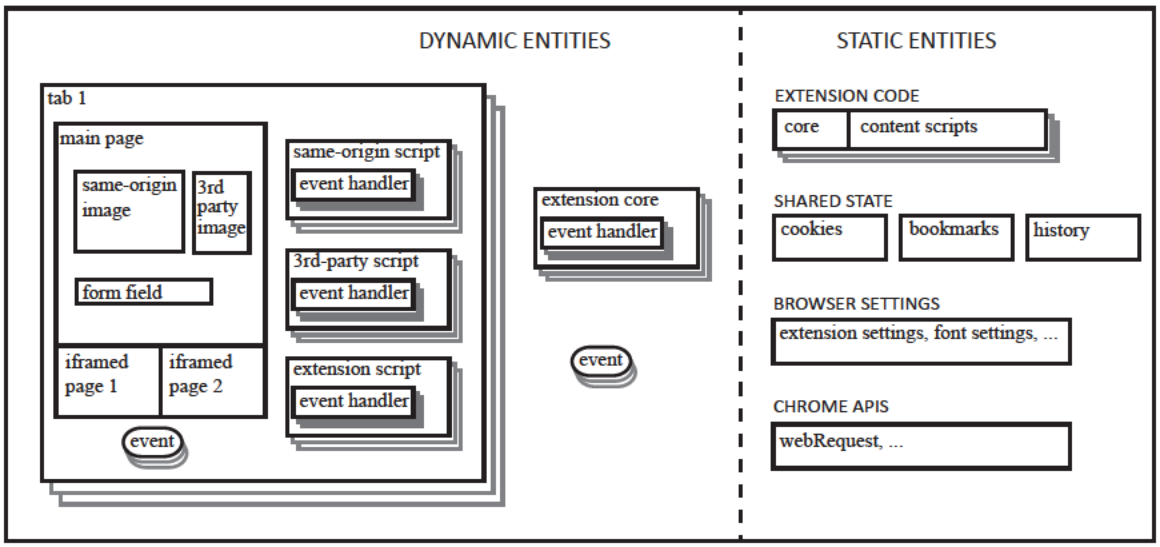

Fig. 1: Informal illustration of the entities within Chromium that we reason about.

$(s_1 \rightarrow s_2)$; we explain these later in this subsection. Certain entities, for example, events, are passive and have no declassification capabilities. We define simple labels, denoted κ, to represent policy labels for these entities. A simple label is a pair of a set of secrecy tags and a set of integrity tags (S, I).

Ignoring declassification, an entity labeled $(S_1, I_1, \{\})$ can send data to an entity labeled $(S_2, I_2, \{\})$ if $S_1 \subseteq S_2$ (the destination is authorized for at least as many secrets as the source) and $I_1 \supseteq I_2$ (the source has at least as many "permissions" as the destination). For example, a DOM subtree that represents content loaded from ad.com would have a label that includes the secrecy tag ad (for brevity, we write ad instead of ad.com throughout); APIs that allow extensions to access the browser's local storage have labels that include a localStorage integrity tag. Data from a script labeled $(\{\mathsf{cnn}, \mathsf{ad}\}, \{\}, \{\})$ wouldn't be allowed to flow to a DOM node labeled $(\{\mathsf{cnn}\}, \{\}, \{\})$ because the latter is permitted fewer secrets (e.g., isn't permitted secrets labeled ad).

Floating labels The basic secrecy label is too rigid to allow entities to adapt to the browser's dynamic environment. E.g., the label of a DOM node on a cnn.com page might initially contain only a cnn secrecy tag to reflect that it contains information from cnn.com; after a password manager fills in a form field on the page, however, the label of the form field's DOM node needs to change to reflect that it also contains information from another source.

We express the policy that allows dynamic tainting of an entity as a floating secrecy label. This is similar to JIF's parametric labels [38]. A floating secrecy label is written as $F(\sigma_1, \sigma_2)$: σ_1 is the set of secrecy tags that the entity has at initialization time; σ_2 is a ceiling (upper limit) of the secret that this entity can be tainted with. In other words, a secrecy label $F(\sigma_1, \sigma_2)$ can float to $F(\sigma_1', \sigma_2)$ as long as $\sigma_1' \subseteq \sigma_2$.

This is also useful for preventing information from reaching an entity. For example, a floating secrecy label of the form $F(\{\mathsf{siteA}\}, \{\mathsf{siteA}, \mathsf{siteB}\})$ indicates that the labeled entity possesses siteA secrets and is willing to receive siteB secrets. Its label will then change to $F(\{\mathsf{siteA}, \mathsf{siteB}\}, \{\mathsf{siteA}, \mathsf{siteB}\})$, continuing to protect the siteB secret. To clearly distinguish floating labels from ordinary ones, we henceforth write non-floating secrecy labels as, e.g., $C(\{\mathsf{siteA}\})$.

Compound labels Going back to the password example, the password field is owned by cnn.com, but can be written to by the user or an extension. We would want both labels, as both secrets are involved, but we may want to maintain the notion of a *primary* owner, for purposes that we will shortly show. To this end, we introduce dot-separated compound tags: $s_1.s_2$ indicates that s_1 is the primary owner of the data.

Returning to our example, the secrecy tag cnn.user would be part of the label of a DOM node originally loaded from cnn.com (hence cnn) at the user's behest (hence user), e.g., if the tab was opened and the URL typed in by the user. More concretely, nodes in the DOM of the cnn.com page would initially be labeled with the secrecy tag $F(\{\mathsf{cnn.user}\}, \{\mathsf{cnn}.*\})$; the $\{\mathsf{cnn}.*\}$ ceiling indicates that it is OK for the node to be tainted (repeatedly) with secrets of all entities whose secrecy label has the form $\{\mathsf{cnn}.*\}$. In contrast, $F(\{\}, \{\dagger.\mathsf{user}\})$ means that an entity with that label can be tainted exactly once, e.g., to $F(\{\mathsf{cnn.user}\}, \{\mathsf{cnn.user}\})$. This label is suitable for content scripts, which are injected into multiple pages, but any script instance is injected into exactly one page.

One purpose for compound labels is to allow labels to reflect which entities influenced the content, while retaining the ability to leave a specific entity in control of the content. For example, we may choose to allow requests to send requests to cnn.com only if they are compatible with the destination label $(C(\{\mathsf{cnn}.*\}), \{\mathsf{network}\}, \{\})$, which concisely expresses the policy that only cnn.com pages are allowed to make requests to cnn.com, and that they can do so even if their content has absorbed input from the user or other entities (e.g., they include a secrecy tag like cnn.user). Similar policies can be expressed with declassification, which we discuss next.

Declassification, reclassification, and endorsement Declassification and endorsement capabilities allow an entity (e.g., an extension core) to circumvent constraints that it would otherwise incur because of its secrecy and integrity tags. Declassification is a powerful (and dangerous) operation, and declassification capabilities should be granted to entities only judiciously. At the same time, declassification is necessary, since some extensions, like the password manager, collect many secrets, yet their functionality requires that they (selectively) copy those secrets into arbitrary web pages.

In our example, the $\text{ext}_{\text{pwdMgr}}$ core has the $-*.\text{ext}_{\text{pwd}}$ capability. This is to ensure that no matter how many secrecy tags like $\text{someSite.ext}_{\text{pwd}}$ it accumulates in its secrecy label as a result of saving passwords, it is still able to send data (passwords) to individual web pages (e.g., cnn.com). Without declassification, those secrecy tags in $\text{ext}_{\text{pwdMgr}}$'s label would cause the label check to fail, since those tags are not in cnn.com's secrecy tag, including its ceiling. Declassification (and reclassification and endorsement) are used only when a label check would otherwise fail; they don't affect an entity's secrecy and integrity tags beyond the label check.

Reclassification is a weaker form of declassification: the $s_1 \to s_2$ reclassification tag indicates that a secrecy tag s_1 can be converted (for the purpose of a label check) to tag s_2.

Endorsement tags are similar to declassification tags. To protect the local storage API, we give the API the integrity label $\{\text{localStorage}\}$; only entities that have localStorage in their integrity label, or can add it via endorsement, can use it. Hence, we give the $\text{ext}_{\text{pwdMgr}}$ core the $+\text{localStorage}$ capability, allowing it to elevate its privileges sufficiently to use the local storage API. As with de- and reclassification, endorsement only enables a label check to succeed, and has no persistent effect on the integrity tags in a label.

2) Policy Composition: An interesting problem that arises in the browser setting is composing policies created by different principals. When an entity A (e.g., a web page) loads another entity B (e.g., an embedded iframe) from a source other than where A comes from, B may come with its own policy as to what kind of secret B can receive, while at the same time A may wish to constrain B's capability to protect itself from B. Browsers today have fixed schemes for handling such (potential) conflicts; e.g., if a page A loads a page B in an iframe, B's capabilities to load external resources are not restricted by A's CSP.

Rather than advancing a single "correct" option for how to combine conflicting policies, we allow the enclosing entity A to specify whether the policy (label) applied to B should allow (1) the union of what A's and B's labels allow, (2) the intersection; or simply (3) A's policy or (4) B's desired policy. We call this a *generalized* CSP.

C. Enforcement

Enforcing policies in our approach has two aspects: assigning labels to entities when they are created and allowing communication between two entities only when supported by labels. We discuss them here in the abstract; more detail about implementation aspects is given in Section V.

Assigning labels Labels are assigned in several ways. For entities like web pages and extensions, the labels are derived from the policy (e.g., CSP, permissions) with which these are already annotated, but also reflect who created the entity. For example, if a user caused the browser to load cnn.com, and cnn.com's CSP permits only the extension $\text{ext}_{\text{pwdMgr}}$ to run and permits third-party content only from ad.com, then cnn.com's label might be

$$S = F(\{\text{cnn.user}\}, \{\text{cnn.user}, \text{cnn.ext}_{\text{pwd}}, \text{ad.}*\}),$$
$$I = \{\}, \quad D = \{+\text{network}, \text{cnn.}* \to \text{ad.}*\}$$

Generating a label for a new entity may involve policy composition, e.g., for the content of an iframe, whose label may reflect (based on the browser's or user's policies) some combination of the policies expressed in the iframed page's CSP and the policies of the host page (see Section III-B2).

Entities like APIs are given labels whose integrity tags correspond to the permissions with which they are protected. To implement built-in browser policies like the same-origin policy, labels on APIs (e.g., for accessing the network) can depend on the parameters of the API call (e.g., the hostname of the site to be accessed).

Other entities in the browser, like button-click events, are assigned labels whose purpose is simply to record which secrets (expressed as secrecy tags) were in possession of the entity that created them. This serves as a way of tracking the flow of information through entities that themselves have no specific policy, but that could otherwise be used as channels between entities whose policy forbids them from communicating. For example, a button-click event would be given the union of the secrecy tags of the DOM element that housed the button and the secrecy tags of the entity that caused the event to be created (e.g., user if the user clicked on the button or the secrecy tags of the script that manufactured the click event); later, this label might prevent the button-click even from being received by an extension content script that doesn't have permission to receive, e.g., user input.

In our approach, a user inspects and approves labels for entities such as extensions, and, therefore, prevents these entities from being assigned inappropriate policy; e.g., allowing the entity to access and declassify all data. An entity can only influence the label of another entity by sending data to it; it is not allowed to directly set the labels of other entities.

Checking labels Checking labels is conceptually straightforward: a reference monitor must mediate every path between entities and compare a sender's label (S_1, I_1, D_1) to a receiver's label (S_2, I_2, D_2). Putting together the aspects of labels discussed in Section III-B1, communication should be allowed only if: (1) there exists a S_1' obtained by applying through declassifying or reclassifying S_1 as permitted by D_1 and a S_2' obtained by raising S_2 (if S_2 is a floating label) up to the point permitted by its ceiling, such that $S_1' \subseteq S_2'$; (2) there exists a I_1' obtained by applying endorsement capabilities in D_1 to I_1, and $I_1' \supseteq I_2$. If communication was allowed as a result of raising the receiver's secrecy label S_2 to S_2', then the receiver's secrecy label after the call will remain S_2'.

D. Example Walkthrough

We next show the initial labels for our example from Section I-A, and walk through the checks and label updates as

ℓ_{user} $S = C(\{\mathsf{user}\})$ $I = \{\mathsf{network}\}$ $D = \{\mathsf{user} \to *.\mathsf{user}\}$

ℓ_{Pwd_0} $S = F(\{\}, \{*.\mathsf{extPwd}, *.\mathsf{user}\})$
$I = \{\}$ $D = \{+\mathsf{localStorage}, -*.\mathsf{extPwd}, -*.\mathsf{user}\}$

ℓ_{PwdCS_0} $S = F(\{\}, \{\dagger.\mathsf{extPwd}, \dagger.\mathsf{user}\})$ $I = \{\}$ $D = \{\}$

ℓ_{PwdLS_0} $S = F(\{\}, \{*.\mathsf{user}, *.\mathsf{extPwd}\})$
$I = \{\mathsf{localStorage}\}$ $D = \{\}$

ℓ_{Eve_0} $S = F(\{\}, \{\mathsf{cnn.extEve}, \mathsf{evil.extEve}, \mathsf{cnn.user}, \mathsf{evil.user}\})$
$I = \{\}$ $D = \{+\mathsf{network}\}$

ℓ_{EveCS_0} $S = F(\{\}, \{\mathsf{cnn.extEve}, \mathsf{cnn.user}, \mathsf{evil.extEve}, \mathsf{evil.user}\})$
$I = \{\}$ $D = \{+\mathsf{network}\}$

Fig. 2: Initial labels for the user (ℓ_{user}), the password manager extension's core (ℓ_{Pwd_0}), content scripts (ℓ_{PwdCS_0}), and local storage (ℓ_{PwdLS_0}), and ext_{eve}'s core and content scripts (ℓ_{Eve_0}, ℓ_{EveCS_0}). For brevity, we omit additional labels that give the user access to other APIs and cnn.com the ability to load content from partner sites.

a user loads a web page.

Initial labels The initial labels of the built-in user principal and $\text{ext}_{\text{pwdMgr}}$'s and ext_{eve}'s cores and content scripts are shown in Figure 2. The user label ℓ_{user} cannot change, and denotes that user input contains secrets (user) that must be protected, but the user can choose to weaken that protection ($\mathsf{user} \to *.\mathsf{user}$), e.g., by inputting them to web pages. The extension cores (ℓ_{Pwd_0}, ℓ_{Eve_0}) and content scripts (ℓ_{PwdCS_0}, ℓ_{EveCS_0}) start with floating secrecy labels that convey that the extensions as yet possess no secrets, but are allowed to be tainted with secrets from any URL (ℓ_{PwdCS_0}) or from select URLs (ℓ_{EveCS_0}). As previously explained, the † wildcard in ℓ_{PwdCS_0} will allow each of $\text{ext}_{\text{pwdMgr}}$'s content script instances to float exactly once, after which its secrecy label will become fixed. The local storage of $\text{ext}_{\text{pwdMgr}}$'s initial label is ℓ_{PwdLS_0}, indicating that it has no secrets yet, but can receive any secrets that either the user or $\text{ext}_{\text{pwdMgr}}$ enters to a page.

To make ext_{eve} potentially more dangerous, we allow it to access the network; we will show that this is not enough for ext_{eve} to steal passwords and send them to her server.

Loading a page Suppose the user opens a new tab, enters a cnn.com URL, and a page with a login form is loaded. The tab's label will be automatically assigned as

$$\ell_{Tab} \quad S = F(\{\mathsf{user}\}, \{*.*\}),\ I = \{\},\ D = \{\}$$

to indicate that the tab was created by the user and could become tainted by other secrets. The network request for the cnn.com URL will be protected (by the browser) by the label $\big(C(\{\mathsf{cnn}.*\}), \{\mathsf{network}\}, \{\}\big)$; since the URL is provided by the user and the user's label permits reclassification of the user secrecy tag to $\mathsf{cnn.user}$, the request will be allowed.

If cnn.com's CSP allows content from an ad network and Google, the secrecy label of cnn.com's top-level DOM node, denoted ℓ_{doc_1}, will reflect that it contains content from cnn.com that was accessed as a result of user input and that the page can receive information from the ad network and Google. Its declassification label will similarly include sufficient reclassification capabilities to allow cnn.com's data to be sent to these sites (e.g., via HTTP GET requests).

ℓ_{doc_1} $S = F(\{\mathsf{cnn.user}\}, \{\mathsf{cnn}.*, \mathsf{ad}.*, \mathsf{google}.*\}),$
$I = \{\},$ $D = \{+\mathsf{network}, \mathsf{cnn}.* \to \mathsf{ad}.*, \mathsf{cnn}.* \to \mathsf{google}.*\}$

Other DOM nodes at this point have the same label as ℓ_{doc_1}.

Next, the browser injects content scripts into the DOM; their secrecy labels float to allow them to access page secrets.

ℓ_{PwdCS_1} $S = F(\{\mathsf{cnn.extPwd}\}, \{\mathsf{cnn.extPwd}, \mathsf{cnn.user}\})$
$I = \{\},\ D = \{\}$

ℓ_{EveCS_1} $S = F(\{\mathsf{cnn.extEve}\},$
$\{\mathsf{cnn.extEve}, \mathsf{evil.extEve}, \mathsf{cnn.user}, \mathsf{evil.user}\})$
$I = \{\},\ D = \{+\mathsf{network}\}$

Entering a password Suppose the user inputs a username and password into the login form. This incurs a check that data may flow from ℓ_{user} to ℓ_{domNd_1}, the label of the login form's DOM node, which is initially the same as ℓ_{doc_1}. The reclassification tag ($\mathsf{user} \to *.\mathsf{user}$) in ℓ_{user} allows this check to succeed; the destination label (ℓ_{domNd_1}) remains unchanged.

The password manager content script reads the contents of those nodes. The content script's label floats to reflect that the content script has been tainted with secrets read from the DOM nodes. ext_{eve}'s content script is similarly allowed to read the DOM nodes; its label similarly floats.

ℓ_{PwdCS_2} $S = F(\{\mathsf{cnn.extPwd}, \mathsf{cnn.user}\}, \{\mathsf{cnn.extPwd}, \mathsf{cnn.user}\})$
$I = \{\},\ D = \{\}$

ℓ_{EveCS_2} $S = F(\{\mathsf{cnn.extEve}, \mathsf{cnn.user}\},$
$\{\mathsf{cnn.extEve}, \mathsf{evil.extEve}, \mathsf{cnn.user}, \mathsf{evil.user}\})$
$I = \{\},\ D = \{+\mathsf{network}\}$

Suppose ext_{eve} tries to send the password to evil.com. The network interface's label will be dynamically constructed as

$$\ell_{network_1} = \big(C(\{\mathsf{evil}.*\}), \{\mathsf{network}\}, \{\}\big)$$

to capture the browser's policy that the source must both have the network integrity tag and be able to de- or reclassify to $\mathsf{evil.entity}$ to be allowed to send data to evil.com (similarly, to other sites). The script's source label will be checked against the destination label $\ell_{network_1}$. Although ℓ_{EveCS_2} permits ext_{eve} access to the network (via $+\mathsf{network}$), it does not enable ext_{eve}'s content script to de- or reclassify its $\mathsf{cnn}...$ secrecy tags. Hence, the request will fail.

The password manager content script sends the password to its extension core, which saves it, tainting both the core and its local storage with secrets from cnn.com.

ℓ_{Pwd_1} $S = F(\{\mathsf{cnn.extPwd}, \mathsf{cnn.user}\}, \{*.\mathsf{extPwd}, *.\mathsf{user}\})$
$I = \{\mathsf{localStorage}\}$
$D = \{+\mathsf{localStorage}, -*.\mathsf{extPwd}, -*.\mathsf{user}\}$

ℓ_{PwdLS_1} $S = F(\{\mathsf{cnn.user}, \mathsf{cnn.extPwd}\}, \{*.\mathsf{user}, *.\mathsf{extPwd}\})$
$I = \{\mathsf{localStorage}\},$ $D = \{\}$

Subsequent visits The next time the user visits cnn.com's login page, the page and injected content scripts will have labels ℓ_{doc_1} and ℓ_{PwdCS_1}. After $\text{ext}_{\text{pwdMgr}}$'s content script is injected, $\text{ext}_{\text{pwdMgr}}$'s core will fetch the stored password from its local storage and send it to its content script. Reading the password from local storage will cause the core's secrecy label to be raised to include $\mathsf{cnn.user}$ and $\mathsf{cnn.extPwd}$, if it did not already. If $\text{ext}_{\text{pwdMgr}}$'s core stored passwords from multiple sites, then sending the appropriate password to its content script running in cnn.com's page will be possible only because of its ability to declassify passwords from other sites (using its $-*.\mathsf{extPwd}$) declassification capability.

When $\text{ext}_{\text{pwdMgr}}$'s content script writes the password into the page, the appropriate DOM node's label floats (after the

check to make sure the write is allowed succeeds) to reflect that it received information from $\text{ext}_{\text{pwdMgr}}$'s content script. The node would then have the following label.

ℓ_{domNd_2} $S = F(\{\mathsf{cnn.extPwd}, \mathsf{cnn.user}\}, \{\mathsf{cnn.}*, \mathsf{ad.}*, \mathsf{google.}*\})$
$I = \{\}, D = \{+\mathsf{network}, \mathsf{cnn.}* \rightarrow \mathsf{ad.}*, \mathsf{cnn.}* \rightarrow \mathsf{google.}*\}$

If ext_{eve}'s content script now tries to read the password, the label check will fail, because the target node contains the secrecy tag cnn.extPwd, which ext_{eve}'s content script doesn't have and cannot float to. This prevents, for instance, attacks in which ext_{eve} might open a new tab and load a page for the purpose of harvesting passwords automatically filled in by a password manager.

IV. Formal Model and Verification

We develop a formal model to serve as the design specification of our information-flow control enforcement mechanism and its integration with Chromium. We use the model to confirm that the design obeys noninterference, a key desirable property for any information-flow enforcement system. The formal model of our enhanced browser includes: (1) the policy language and policy enforcement mechanisms, (2) key entities in the browser environment and their interactions, and (3) formal definitions of security properties and proof that the design of our enforcement mechanism has the desired security properties. Due to space constraints, we omit formal definitions of the policy language, and focus on the specification and proof of the noninterference property and its implications. The full syntax and transition rules, as well as the full definition and proof of noninterference, can be found in the companion technical report [11].

A. Modeling the Browser

The Chromium architecture is complex (see Figures 1 and 4); the purpose of the model is to identify key components relevant for reasoning about how information flows in the browser, and to define the rules that are to govern the interactions between these components if our approach is to successfully enforce specified information-flow policies.

We model the behavior of the browser using a labeled transition system. The state modeled by the transition system encompasses the user, servers, as well as entities in the browser such as the DOM, extensions, browser APIs, browser events, and persistent browser state such as cookies and history. Using the model, we later state and prove the security properties of our approach.

System state Each browser entity is a tuple consisting of elements in that entity. We summarize key entities in Figure 3. Typically, each entity in the browser is associated with an information-flow label ℓ or κ, when that entity has no declassification capabilities.

The top-level system state Σ contains the following. Ψ is a list of internal browser states; for instance, $\mathsf{ws.beforeRequest}(\kappa, \cdots)$ denotes the state of a web request. $Tabs$ are the open browser tabs. $ExtCoreRs$ are run-time instances of extension cores. $proInjCSs$ are the programmatically injected content scripts. $Exts$ are installed extensions. Browser state also includes shared state objects: cookies, bookmarks, and browser history. Finally, UI captures user actions, e.g., when a user presses "Ctrl + T".

An event is a tuple consisting of a unique event ID (id_e), an event type, whether actions are needed after the event is processed ($return$), additional arguments of the event ($info$), and the information-flow label for the event (κ). An event handler has its own unique ID, the type of event that it processes, and the code for processing events ($x.cmd$). An event handler can only process one event at a time; events waiting to be processed are stored in an event queue $\mathcal{E}$. The $BlockingFlag$ indicates whether a handler is a blocking event handler. The last three fields in the event handler are the script processing the current event and the ID and the return information of the event being processed.

We model the main page and iframed subpages in a browser tab as a list of documents $Docs$. A document Doc is defined as $(id_d, url, nodes, DocCSs, CSP, \ell)$. id_d is the document ID. url is the page URL. $nodes$ denotes the page elements. $DocCSs$ are the content scripts injected by extensions. CSP denotes the content security policies of the page. Each document is associated with a policy label. The elements in a page are modeled as tree nodes in a document. A node is defined as $(id, attributes, nodes, content, \ell)$. id denotes the node ID. $attributes$ contains general information about the node, e.g., the content type, the URL (if the node loads an external object), and the parent node ID. $nodes$ are the child nodes. $content$ is a piece of data with a specific format, e.g., an image file. ℓ is the policy label attached to the node. $DocCSs$ is the list of active content scripts. id_r is the unique identifier for that run-time instance. A browser tab is defined as $(id_t, Docs, url, EventHandlers, \ell)$, where id_t is the unique ID of the tab, $Docs$ denotes the documents in the tab, url is the URL of the displayed page, and $EventHandlers$ the JavaScript event handlers that come from the page scripts. Finally, ℓ is the information-flow policy associated with the tab.

An extension is a tuple consisting of: a unique ID, one extension core, several content scripts, local storage, an active flag, and a policy label. The active flag $aFlag$ indicates whether an extension is active. A static extension core contains programs, which are modeled as a variable environment Γ, commands cmd, and a list of event handlers. A content script $ExtCS$ contains three identifiers: the ID of the extension it belongs to, its own unique ID, and the ID of the tab in which it runs. Similar to the extension core, Γ denotes the variables, cmd is the main program of the script, and $EventHandlers$ is a set of event handlers in the content script. The index $runat$ indicates when to inject the script to a tab.

We define a multi-level bookmark $MBookmarks$ data structure, consisting of a set of pairs of a bookmark $bookmark$ and a simple label κ. The label indicates the secrecy and integrity level of the bookmark. A bookmark is a tree: each leaf node is a bookmark entry and each non-leaf node represents a directory. (Multi-level bookmarks are discussed in more detail in Section IV-C.) Both cookie entries and history entries are labeled with simple labels κ.

Transition rules The top-level transition rules are of the form $\Xi; \Sigma; \mathcal{E} \xrightarrow{\beta} \Xi'; \Sigma'; \mathcal{E}'$. Here, Ξ denotes remote servers, which are active entities that exchange information with the

System state	Σ	::=	$\Psi, Tabs, ExtCoreRs, proInjCSs, Exts, Cookies, MBookmarks, histories, UI$
Browser internal state	Ψ	::=	$\cdot \mid \Psi, \mathsf{ws.beforeRequest}(\kappa, \cdots) \mid \cdots$
Event	e	::=	$id_e, eventType, return, info, \kappa$
Event Queue	$\mathcal{E}$	::=	$\cdot \mid \mathcal{E} :: e$
Event handlers	$EventHandlers$	::=	$\cdot \mid EventHandlers, (id, eventType, x.cmd, \mathcal{E}, BlockingFlag, cmd, id_e, return)$
Documents	$Docs$	::=	$\cdot \mid Docs, (id_d, url, nodes, DocCSs, CSP, \ell)$
Nodes	$nodes$	::=	$\cdot \mid nodes, (id, attributes, nodes, content, id_{parent}, \ell)$
Doc content scripts	$DocCSs$	::=	$\cdot \mid DocCSs, (id_{ext}, id_{cs}, id_r, \Gamma, cmd, EventHandlers, \ell)$
Tabs	$Tabs$	::=	$\cdot \mid Tabs, (id_t, Docs, url, EventHandlers, \ell)$
Installed extension	Ext	::=	$id_{ext}, ExtCore, ExtCSs, Storage, activeFlag, \ell$
Content scripts	$ExtCSs$	::=	$\cdot \mid ExtCSs, ExtCS$
Content script	$ExtCS$	::=	$id_{ext}, id_{cs}, id_t, \Gamma, cmd, EventHandlers, runat, \ell$
Injection time tag	$runat$	::=	$\mathsf{DocBegin} \mid \mathsf{DocEnd} \mid \mathsf{DocIdle}$
Extension core	$ExtCore$	::=	$\Gamma, cmd, EventHandlers$
Installed extensions	$Exts$	::=	$\cdot \mid Ext :: Exts$
Extension cores	$ExtCoreRs$	::=	$\cdot \mid ExtCoreRs, (id_{ext}, ExtCore, \ell)$
Injected content scripts	$proInjCSs$	::=	$\cdot \mid proInjCSs, ExtCS$
Multi-level bookmarks	$MBookmarks$	::=	$\cdot \mid MBookmarks, (Bookmarks, \kappa)$
Bookmarks	$Bookmarks$	::=	$\cdot \mid Bookmarks, bookmark$
Bookmark	$bookmark$	::=	$id, title, Bookmarks \mid id, title, url$
Cookies	$Cookies$	::=	$\cdot \mid Cookies, (name, value, url, \kappa)$
Histories	$histories$	::=	$\cdot \mid histories, (id, url, name, visitTime, visitType, \kappa)$
UI	UI	::=	user, cmd, ℓ

Fig. 3: Definitions of browser state

browser. Σ is the browser state (see Figure 1), which includes tabs, extensions, bookmarks, cookies, history, etc. $\mathcal{E}$ denotes events waiting to be processed. Events can be user inputs, API requests, and other internal browser events. Each transition is labeled with an action β, representing the observable effects of that transition. Observable actions include API calls, invocations of callbacks, processed events, etc. The browser makes internal transitions, which do not produce observable effects. We use τ to label such transitions, and call them silent transitions. We define an execution trace ρ as the sequence of non-silent actions in a transition sequence.

For space reasons, we omit detailed rules. We show one example rule for processing an event below. When e is not a blocking event, we enqueue e in the event queues of relevant event handlers. The enqueue operation is defined using the $\lhd_Q$ operator. This rule does not perform label checks. We check the event handler's label against e's label to make sure that the handler is allowed to learn secrets contained in e when the event handler is ready to process e. The action of this transition is silent (τ), as there is no observable behavior.

$$\frac{\begin{array}{l} e \text{ is not a blocking event} \\ \Sigma = (\Psi, Tabs, ExtCoreRs, \cdots) \\ \Sigma' = \Sigma[ExtCoreRs \Leftarrow ExtCoreRs \lhd_Q e][Tabs \Leftarrow Tabs \lhd_Q e] \end{array}}{\Xi; \Sigma; \mathcal{E} :: e \xrightarrow{\tau} \Xi; \Sigma'; \mathcal{E}}$$

We define transition rules for processing scripts, processing events, browser internal state transitions, API calls to access shared state, and other API calls (e.g., tab accessing and programmatic content script injection). Transition rules for events include event firing and dequeuing rules and special rules about events for sending/receiving data to/from web sites. Browser internal state transition rules include rules for deciding whether to send a web request, web-request-state transitions, content loading and DOM creation, and content script injection. Rules for accessing shared browser state include accesses to page contents (accessed through the DOM API), bookmarks, cookies, and history.

B. Noninterference

We analyze the security guarantees of our model by proving a noninterference theorem. We would like to show that the attacker cannot learn secrets beyond what's allowed by the policy. Before presenting the final theorem, we introduce several necessary supporting concepts.

Attacker model revisited A policy label specifies an attacker's capabilities. We denote such a label κ_A. Given κ_A, the system's components are partitioned into two sets: one containing components whose labels are lower than or equal to κ_A ($\kappa \sqsubseteq \kappa_A$); the other containing components whose labels are either higher or not compatible with κ_A ($\kappa \not\sqsubseteq \kappa_A$). Our enforcement mechanism enforces the policy that information can flow to an attacker's component from the first set, but not easily from the second. We discuss possible implicit flows from the second set to the attacker in Section IV-D.

Not all actions are observable to the attacker. Formally, any action whose label is not lower than or equal to κ_A is not visible to the attacker. The observable behavior of the attacker is a projection of the trace that contains only actions whose labels are lower than or equal to κ_A. The attacker's goal is to guess secrets based on the trace it has observed so far.

Declassification steps A declassification step is one where a component that contains a secret reveals information about the secret by generating an element, which can be an event or a node or a content script, that is observable to the attacker (i.e., has a label lower than or equal to the attacker's label). For instance, when a cnn.com page loads a script from ad.com, the request is observable to ad.com. In the worst case, the URL could contain secrets that belong to cnn.com. If cnn.com allows loading of scripts from ad.com (by specifying a reclassification policy cnn.com $\rightarrow$ad.com), this is a declassification step explicitly allowed by the policy.

Declassification steps are necessary for the browser to carry out certain functionality. The first kind of declassification directly uses an entity's declassification capabilities. Such declassification happens when an entity generates an event, makes an API call, and sends an event. Other declassification steps allowed in our model are at places where labels are computed for new entities based on policy composition rules: (1) requesting external sources, (2) generating a new DOM node based on data received from external sources, (3) injecting content scripts from extensions, and (4) embedding a page in an iframe.

Declassification is only allowed according to policy. Separating declassification from other parts of the policy helps to enforce least privilege. In practice, one would want to validate the legitimacy of components that request declassification labels carefully to ensure that their declassification capabilities are justified.

Equivalent states We define system configurations that are equivalent from the attacker's perspective based on a set of projection rules. The projection operation ($\downarrow_{\kappa_A}$) is defined for each entity in a configuration. It removes components not visible to the attacker (i.e., components with labels not lower than or equal to κ_A). For instance, the projection rules for bookmarks are defined as follows: If the bookmark's label is not lower or equal to κ_A, then the projection removes that bookmark, as it is not observable to an attacker with label κ_A.

$$\frac{\kappa_B \sqsubseteq \kappa_A}{(Bookmarks, \kappa_B)\downarrow_{\kappa_A} = (Bookmarks, \kappa_B)}\ \text{MBOOKMARK1}$$

$$\frac{\kappa_B \not\sqsubseteq \kappa_A}{(Bookmarks, \kappa_B)\downarrow_{\kappa_A} = \cdot}\ \text{MBOOKMARK2}$$

We say that two configurations $(\Xi_1, \Sigma_1, \mathcal{E}_1)$ and $(\Xi_2, \Sigma_2, \mathcal{E}_2)$ are equivalent at label κ_A if $\Xi_1\downarrow_{\kappa_A} = \Xi_2\downarrow_{\kappa_A}$, $\Sigma_1\downarrow_{\kappa_A} = \Sigma_2\downarrow_{\kappa_A}$, and $\mathcal{E}_1\downarrow_{\kappa_A} = \mathcal{E}_2\downarrow_{\kappa_A}$.

Noninterference theorem We prove the following theorem to demonstrate the correctness of our design. We write $C \stackrel{\rho}{\Longrightarrow}$ to mean that ρ is the trace generated by executing from the configuration C.

Theorem 1 (Noninterference): If two configurations $(\Xi_1, \Sigma_1, \mathcal{E}_1)$ and $(\Xi_2, \Sigma_2, \mathcal{E}_2)$ are equivalent at label κ then

- forall ρ s.t. $\Xi_1; \Sigma_1; \mathcal{E}_1 \stackrel{\rho}{\Longrightarrow}$ and the transitions do not have declassification steps there exists ρ' such that $\rho \equiv_\kappa \rho'$ and $\Xi_2; \Sigma_2; \mathcal{E}_2 \stackrel{\rho'}{\Longrightarrow}$
- forall ρ s.t. $\Xi_2; \Sigma_2; \mathcal{E}_2 \stackrel{\rho}{\Longrightarrow}$ and the transitions do not have declassification steps there exists ρ' such that $\rho \equiv_\kappa \rho'$ and $\Xi_1; \Sigma_1; \mathcal{E}_1 \stackrel{\rho'}{\Longrightarrow}$

The theorem states that if $\Xi_1; \Sigma_1; \mathcal{E}_1$ and $\Xi_2; \Sigma_2; \mathcal{E}_2$ differ only in the secrets they contain, then they behave the same from the attacker's perspective, provided that the attacker can only observe portions of the traces. To put it another way: the attacker cannot guess the secrets in the initial configuration, because the trace that the attacker sees does not depend on those secrets. Even though the theorem does not allow declassification during transitions, the initial states can be ones produced as a result of using declassification. This means that after initial declassification, if there is no further declassification, then the attacker cannot learn more secrets than what was revealed earlier.

C. Implications for Browser Design

Our analysis exposed several implication for browser design.

Blocking event handlers Chromium has events for which an extension core can register a blocking event handler; the browser waits until all blocking handlers for an event finish before proceeding based on the values returned by the handlers.

Ordinarily we would allow an event e to be processed by an event handler if e's label is lower than or equal to the handler's label. This follows the principle that less-secret information can flow to components that are allowed to learn more-secret information. However, for blocking event handlers, tainting the handler in this way leaks information. Consider the following scenario: An extension A knows a secret (0 or 1). If the secret is 0, A registers a blocking handler to redirect the web request from cnn.com to nytimes.com; otherwise, A does not register that handler. Here, the request to load cnn.com is an event observable to the attacker. After A handles the blocking event, the request to nytimes.com will not be observable to the attacker. When the attacker observes that cnn.com is loaded, it know that the secret is 1. This leak will be prevented if blocking handlers accept only those events that have the same secrecy label as the handler. Enforcing this would prevent extension A from registering the handler and subsequently using it to leak a secret to the attacker.

Synchronous DOM reads Chrome's DOM APIs are synchronous, so when a script calls the read API, the API always returns a result, even when the target doesn't exist. The synchronous read can be used to leak information. Suppose a content script A has a secret (0 or 1). If the secret is 1, A taints a specific DOM node. An attacker B attempts to read the DOM node; if the return value is an error code, it knows A's secret is 1. The only way to fix this problem (while retaining floating labels) is to implement synchronous DOM access APIs that do not return in case of a violation.

Multi-level bookmarks Like the DOM, bookmarks have a tree structure. Operations on bookmarks include insertion, deletion, and mutation of nodes and subtrees. As with the DOM, we could allow each node to be tainted with the label of the entity that updates the data structure. The drawback is that to prevent information leakage, many simple operations are

prohibited. For instance, if a script with many secrets wrote to the root of the bookmark tree, then no entities that are allowed fewer secrets can read any bookmark. Since bookmarks have a long life cycle, this is too prohibitive. Instead, we borrow ideas from multi-level secure execution [20], and keep multiple copies of the bookmarks, one at each active security level. Given an update to a bookmark at label κ, if there is no bookmark with label κ, we select one from the set of bookmarks such that its label $\kappa' \sqsubseteq \kappa$, copy this bookmark to label κ, and apply the update to this bookmark. This approach provides noninterference and allows more flexible operations on bookmarks.

Cookies and history Cookies and history are similar to bookmarks in that they are long lived; tainting them would interfere with normal functionality. Hence, we label each with a simple label κ. For cookies, the label corresponds to the cookie's domain, so web sites can set and retrieve their cookies, which is the main functionality of the cookies. For history, it is the secrecy and integrity label of the entity that caused the history entry to be created.

To operate on cookies, an entity needs to reclassify to the secrecy label of a cookie's domain, which is consistent with having the ability to access content from that domain.

When querying history, an entity with label ℓ is given results composed of entries whose label is lower than or equal to ℓ. When deleting history entries, only entries with label equal to or higher than ℓ are removed.

D. Limitations (Implicit Flows)

Our enforcement mechanism is a form of policy-based dynamic taint tracking; and, therefore, provides similar security guarantees as those methods. In particular, trace-based noninterference assumes that an attacker can only observe sequences of actions. Our enforcement mechanism cannot prevent attacks where the attacker has more knowledge than just traces; e.g., behavior of the scheduler, timing channels.

More concretely, our implemented enforcement mechanism allows subtle implicit flows resulting from scripts branching on secrets, and then floating the label of an entity only in one of the branches. To explain this apparent conflicts between the noninterference theorem and implicit flows, we use the following example. Let x, y, and z represent DOM nodes. eh, eh_1, and eh_2 are event handlers processing different types of events. x is public, y can float, and z contains the value of a secret, which is either true ($\top$), or false ($\bot$).

$$
\begin{array}{l}
x = \bot^{L}, y = \bot^{F(L,H)}, z =?^{H} \\
eh() = \{\mathsf{trigger}\ eh_1(); \mathsf{trigger}\ eh_2(); \mathsf{read}\ x\ \} \\
eh_1() = \{\mathsf{read}\ z; \mathsf{if}\ (\neg z)\ y := \top\} \\
eh_2() = \{\mathsf{read}\ y; \mathsf{if}\ (\neg y)\ x := \top\}
\end{array}
$$

Initially, an event triggers eh. When the secret value z is $\top$, the following trace is observed by the attacker if the programs are run sequentially: $\mathsf{read}\ y, \mathsf{write}\ x\ \top$, with $x = \top$ in the end. If the attacker observes this trace, he cannot, without knowledge of the scheduler, be certain that $z = \top$. This is because, if eh_2 is scheduled before eh_1, the attacker would observes the same trace regardless of z's value. This scheduling is reasonable because these event handlers are called asynchronously.

When $z = \bot$, y is tainted to secret, and the trace of sequential execution observable to an attacker is $\mathsf{read}\ x$, with $x = \bot$ in the end. Similarly, in this case the attacker cannot be certain that $z = \bot$: when $z = \top$, x could be read before eh_1 and eh_2 execute, which would generate the same trace. In other words, the attacker cannot know whether $z = \bot$, or $z = \top$ and eh_2 has not been triggered.

In both cases, security relies on the attacker's lack of knowledge of the scheduler. The first relies on the fact that event handlers can be scheduled out of order. The second relies on the fact that the scheduler does not guarantee that asynchronous calls will execute in a timely manner. However, neither assumption is true for most implementations of JavaScript schedulers. In the first case, it is almost certain that eh_1 executes before eh_2, thus invalidating the possibility that $z = \bot$. For the second case, the scheduler will schedule calls soon after they are made, and, hence, the attacker that waits longer will see the action of reading y and will be able to eliminate the possibility that $z = \top$.

There are several ways to mitigate this problem. One is to modify the scheduler to make it less predictable, and hence make it harder or impossible for the attacker to guess the secret. Another approach is to modify the JavaScript interpreter and implement finer-grained enforcement mechanisms that implement the no-sensitive-update strategy [2], [5], as is done in several projects [13], [27], [28]. However, this approach may be too restrictive, and still only partially solves the problem. These mechanisms rely on halting the entire program to prevent attackers from getting more information if an entity is about to be floated in a branch conditioned on secrets. However, this only works for a stand-alone program. In the browser setting, attackers can collaborate with remote servers, which the reference monitor implemented in the browser cannot stop. The attacker can thus observe that the browser is halted (e.g., no more requests are sent to the server). We believe adding non-determinism or probabilistic execution to the JavaScript interpreter, or using Secure Multi-Execution (SME) are the only ways to achieve stronger formal security guarantees. Our approach prevents direct transfer of secrets to the attacker, and the methods for circumventing it require attackers to engage in behaviors (e.g., polling on shared variables) that are likely to be detectable through run-time behavioral profiling.

Stronger notions of noninterference exist to rule out some of the subtle leaks. For instance, by using bi-simulation as the notion of behavioral equivalence, reasoning about noninterference will encompass certain timing-based attacks and implicitly leaks. Such an adversary model allows the adversary to distinguish between two states that are reached from the same initial state through two different execution paths, even when those paths have equivalent traces. Several browser features, mainly synchronous calls and floating labels, prevent us from achieving this stronger notion of noninterference. Similarly, when an attacker can measure other system state (e.g., power consumption), even stronger notions of noninterference are needed. However, the majority and most damaging web attacks are not that sophisticated, and our proposed mechanism (provably) can defend against them.

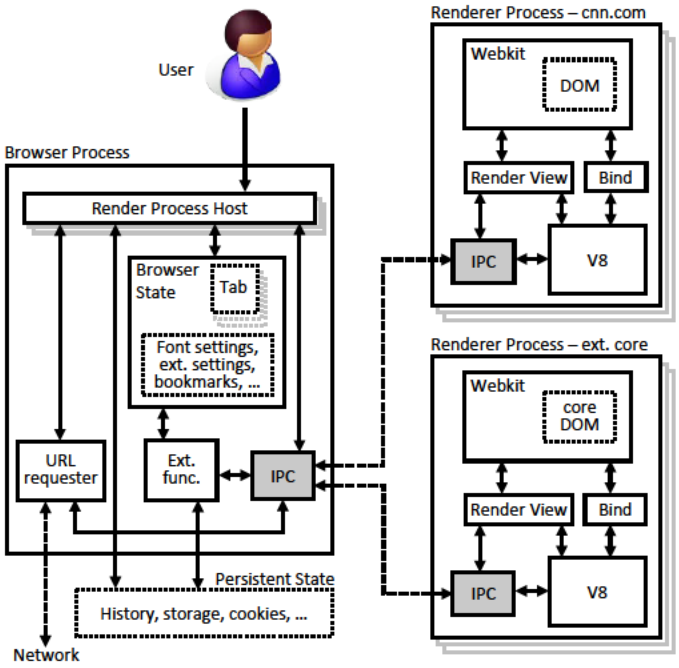

Fig. 4: An abstract view of the parts of Chromium's architecture relevant for implementing our enforcement mechanism.

Browser data structure	*Label operation*	*Modified component (object)*
Extension core	init	Binding (ScriptController)
	check	Extension Function (ExtensionHost, APIs), IPC (ExtensionHelper)
Content script	init	Binding (ScriptController)
	check	IPC (Dispatcher), Binding (V8*)
DOM	init	WebKit (DocumentInit)
	check	Binding (V8*), WebKit (Document, Element)
Event	init	WebKit (Document)
	check	Binding (V8*), WebKit (Document)
Browser state	init	Chrome Tabs (WebContents), ...
	check	Ext Func (TabsCreateFunction)
Persistent state	init	ValueStore (storage), ...
	check	Ext Func (StorageStorageAreaGetFunction)

TABLE I: Browser components (and objects within them) modified for adding or checking labels. V8* stands for the automatically generated code that connect V8 and WebKit, e.g., V8Document, V8Element.

V. Implementation

To gain practical experience with mapping browser policies to our model, we instantiated our model in a proof-of-concept prototype built on Chromium version 32.0.1660.0 (initial import was from Google rev. r197479). In this section we briefly overview Chromium's architecture, discuss some implementation challenges, and give examples of browser flows that we instrumented. We also report on performance, which is promising even with our prototype implementation.

A. Chromium Architecture Overview

A high-level overview Chromium's architecture as it pertains to implementing our approach is shown in Figure 4. The Chromium browser is comprised of two main parts: the main browser process and the renderer processes. Each renderer, which could be servicing a web page or an extension core, is isolated in a separate process. The main browser process handles the UI and manages the renderers. The renderers use the WebKit open-source layout engine to interpret the HTML.[1] The browser process communicates with the renderer processes via inter-process communication (IPC). The browser maintains a RenderProcessHost object for each renderer process. The renderer has one or more RenderView objects, each of which represents the contents of a tab or pop-up window. The RenderProcessHost in the browser maintains a RenderViewHost (inside RenderProcessHost) corresponding to each view in the renderer and handles the input and painting. The RenderView objects communicate with the WebKit engine and the RenderViewHost to display web pages and handle user input. Content scripts and page scripts are scheduled for execution by WebKit and interpreted by the V8 JavaScript engine; a binding layer consisting of automatically generated code joins the two.

To display a web page and run extensions, the above components interact in the following ways. First, extensions are loaded from disk and each extension is given a rendering process in which to run. Next, the user opens a tab and inputs a URL. The browser process handles the input and creates a renderer process to load and render the specified page. The RenderViewHost in the renderer uses WebKit to interpret the HTML and communicates to the RenderViewHost in the browser process to cause it to paint the page on the screen. Finally, injecting extension content scripts into the page involves the ScriptController in the Binding scheduling and causing V8 to execute the scripts.

B. Implementation Overview and Challenges

Consistently with the discussion of enforcement in Section III-C, implementing our approach requires modifying Chromium to support: (1) adding labels to extensions, scripts, events, and other browser data structures that correspond to entities of interest; (2) adding label checks to guard access to sensitive APIs, communication between scripts or extensions, etc.; and (3) propagating enough information about labels through the browser to enable these checks.

The main data structures we modify, and the browser components that we needed to modify in order to assign labels to the entities represented in those data structures, are shown in Figure III-C and detailed in Table I.

Tracking information flow across processes Chrome is a multi-threaded, event-driven application. A key challenge was to analyze its control flows. The call stack of an action, such as a resource request, will originate from an IPC call or from within the V8 engine, and a traditional backtrace would end at

[1]Newer versions of Chromium instead use the Blink rendering engine, which is an offshoot of WebKit.

that point. To obtain an end-to-end trace of control flow, from mouse click to outgoing network activity, required multiple backtraces. If the IPC message was specific to the action under analysis, the process became easier by indicating the origin of an IPC within the code. Similarly, a significant implementation challenge was locating the places where labels needed to be checked, which span the modules and processes that comprise Chromium. E.g., Chromium's APIs are executed in the browser process but are accessed by extensions running in a renderer process. Hence, labels that would naturally be attached to data structures that live only within specific processes or components need to be marshalled across process boundaries. To achieve this, Chromium's IPC layer was modified to carry labels between processes.

JavaScript calls The V8 JavaScript engine controls its own internal control flow, and does not follow calling conventions such as `cdecl`, which would support traversal with a debugger. Therefore, the V8 subsystem is a black box to our analysis, and we only regain control once control transfers to the binding layer between WebKit and V8. Fortunately, we had the ability to attach the label to the script execution context, and we are able to then check against it inside the binding code.

C. Examples of Implementation Details

We next show several examples of how our implementation initializes, propagates, and checks labels. The examples are representative of information-flow paths through browsers.

An extension core calls a Chrome API Chrome APIs provide rich features for extensions to access sensitive information, including bookmarks, history, and cookies. Each such Chrome API call will be subject to label checks. The label assignment and checking involves both the extension process and the browser main process.

API calls originate from the V8 instance within the renderer in which an extension core resides. An extension's label is initialized when it is loaded into Chrome. When the extension, running within V8, makes an API call, a copy of the label of the extension is propagated from V8 to the IPC layer. The IPC layer then relays the extension's request to use an API function to the browser process. The browser checks whether the caller's label allows the specific API call and hands off the request to the appropriate API function. The response is returned over IPC, and before the V8 runtime receives any information, the label of the data is checked against the label of the target script. If this check passes, the result is released.

When the API call adds to an API's information store (e.g., history), the same process occurs, except the check before executing the call occurs within the API implementation instead of in the IPC layer. Specifically, before information is released to an information store, we check that the store's label permits it to receive the information.

JavaScript reads or modifies a DOM node Scripts operating in the context of a web page can normally modify any piece of the page's DOM. Our approach introduces more stringent, per-DOM-node checks. To implement them we modify WebKit, the renderer and the binding between V8 and WebKit.

Labels are assigned to scripts when they are loaded into the V8 runtime environment. The run-time context of the script is labeled with respect to the manifest provided by the extension. The DOM is labeled at the time of conception, with the each child in the DOM inheriting its label from its parent.

When a call is made from a script to interact with a DOM node (e.g., `getElementById`), V8 dispatches a call to WebKit over the binding layer. In order to track information flow, requests to WebKit contain a reference to the context of the requesting script. On such a call, a check is performed in the binding layer to determine whether the call should be allowed. The checks take into account whether the call is for getting or setting DOM data. If the check fails, the call never returns.

D. Implementation Status, Experience, and Performance

Our prototype implementation is fully functional and instruments all flows between the pairs of entities that we consider. Several data structures are not currently instrumented: Cascading Style Sheets, Scalable Vector Graphics, and Media. For event handling APIs, we only instrumented those that are used by our case studies. We also have not yet implemented multi-level bookmarks (see Section IV-C). To obtain broad coverage of Web IDL functions, we modified the generator that emits the binding code to include appropriate checks in autogenerated code.

Our implementation comprises $\sim 10,200$ lines of code (mostly additions). The browser component with the most additions and modifications is WebKit, with $\sim 3,100$ lines added or changed. The additions to WebKit included changing the Web IDL binding code generator, causing it to add $\sim 26,700$ lines of instrumentation to auto-generated binding code.

Our prototype assigns labels to web pages based on their CSPs. Many pages don't have CSPs; in most cases, this causes them to fail to fully load in our browser. This is because in the absence of a CSP a web page is given a label that does not let it de- or reclassify sufficiently to load content from any site but its own; this prevents the common practice of using third-party content on a page. To allow for more interesting exploration, we manually generated CSPs for Alexa's global top-100 sites to allow them to fully load. Automatic generation of these enhanced CSPs is outside the scope of this work; in many cases, it is not feasible, since a goal of our effort was to allow more flexible policies than are currently used or specified.

Similarly, extension labels are based on the permissions in extensions' manifests. However, as our approach allows interesting, richer policies, we manually specified labels of interest for several popular extensions (including Google mail checker [23] and Facebook for Chrome [1]), as well as our own password manager and password-stealing extensions. Expressing desired policies directly via our labels rather than instantiating labels based on legacy or implicit policies enables a range of policies that are more powerful or more precise than those normally implemented by browsers.

This allowed us to confirm with a small set of experiments, including the example from Section III-D, that our approach was generally able to enforce interesting policies that would prevent some misbehaviors allowed by current browsers, while allowing pages and extensions to function normally. Next,

	base load time (ms)			*load time (ms)*			ovhd	num.	num.
website	average	std. dev.	median	average	std. dev.	median	%	checks	requests
Google.com	411	51	398	547	89	522	31	29	6
Facebook.com	655	64	622	845	95	813	31	29	21
Youtube.com	3801	205	3876	6995	528	7160	85	344	267
Yahoo.com	2117	122	2091	3784	576	3679	76	518	180
Baidu.com	1452	276	1384	1236	168	1185	-14	252	10
Amazon.com	2530	996	2219	3736	1023	3665	65	386	181
Wikipedia.org	561	52	552	1098	110	1118	103	190	72
Taobao.com	2748	290	2683	4999	749	5017	87	681	213
Twitter.com	807	34	796	1027	31	1025	29	15	10
Qq.com	6942	2210	5798	9465	1955	9190	59	2639	271

TABLE II: Performance for page loads of the main pages of the top 10 Alexa global top-500 web sites, with GCSP's configured to allow all content to load. The "base" columns report load time for unmodified Chromium; the next three columns report load time for our instrumented browser. The remaining columns report the overhead of our mechanism (computed by comparing the median load times achieved by the instrumented and uninstrumented browsers), the number of label checks incurred during page load, and the number of network requests made by the browser while loading the page. Reported load times are over 40 runs, with outliers due to network errors removed. Performance was measured on a Dell Optiplex 9020 with a Core i7–4770 CPU and 32 GB memory, running Linux 3.14.12. Web sites often varied the content they served from run to run, introducing significant variability and imprecision into the measurements.

in Section VI, we discuss how common high-level browser policies mapped onto our approach, and the current limitations of this mapping.

Our focus was on experimenting with functionality rather than on attempting to minimize overhead.

With our unoptimized implementation, the overhead added by our system to page load time averaged 55%.[2] Performance for a sampling of Alexa top-20 sites is shown in Table II. The page load time is measured in similar fashion as Chromium's PageLoadHistograms :: LogPageLoadTime method. We start the timer when the document loading begins and stop the timer when the document finishes loading. Surprising is the relatively large number of label checks that takes place as a page is loaded. A large portion of the checks occurs during page loading. These checks are caused by DOM tree construction when contents arrive at the browser and deciding which cookies to send to the server. The rest of the checks are due to page scripts trying to either read DOM elements or changing their attributes. We believe that an optimized implementation could easily be very efficient.

VI. Approximating Existing Browser Security Mechanisms

As discussed throughout, browsers currently implement many security policies. Some of these policies are clearly about information flow and map cleanly to our labels; for others the mapping is less clear. We next revisit several such policies, examining to what extent they map into a policy language like ours, as well as whether the policy language's expressiveness allows richer or more powerful variants of the policies to be stated and enforced.

Same-origin policy Browsers use the same-origin policy (SOP) to manage access to different origins. Origins are usually defined as the tuple (scheme,host,port). Scripts from one origin cannot read content from another origin (e.g., via `XMLHttpRequest`), nor can they locally read data from tabs from other origins. The precise implementation of the SOP is slightly more nuanced: outgoing requests to other origins are allowed, but data that they return to the browser is not forwarded to the entity that initiated the request.

As mentioned in Section III-D, this policy can be easily implemented using our labels. When an entity makes a network request, the label for the network controller is instantiated using the outgoing (scheme,host,port) tuple.[3] For an attempted access to cnn.com, this results in the label $\ell_{network} = \big(C(\{\mathsf{cnn}.*\}), \{\mathsf{network}\}, \{\}\big)$. For an entity with label ℓ_e to be allowed to send data on the network, label checks would have to permit a flow from ℓ_e to $\ell_{network}$. To return data from the network, label checks would have to allow the flow from $\ell_{network}$ to ℓ_e. This will succeed only if the secrecy label of ℓ_e contains $\mathsf{cnn}.*$.

In the absence of additional restrictions, the calling page or script could have a sufficiently flexible label ℓ_e to enable either the outgoing or the incoming path. Hence, to enforce the SOP on an entity, the browser needs only to prohibit that entity from having a label that allows it gather secrecy tags other than those conveying its origin. If we wished to also disallow outgoing cross-origin requests, the browser would need to prevent the entity's label from being able to declassify the tags that describe its origin.

In practice, a strict SOP prevents many commonly used web idioms, and our prototype does not attempt to enforce it.

Domain relaxation A page can set its `document.domain` value to a suffix of its current domain, allowing pages with different prefixes of the same hostname to communicate. E.g., a page from login.a.com and a page from profile.a.com can both set their domain to a.com, at which point their origins will be

[2]Note that web sites regularly varied the content they served during the sets of runs on which we computed performance, leading to our implementation sometimes appearing to outperform stock Chromium.

[3]All our hostname-based tags include the scheme and port, though we generally elide this for clarity.

considered the same, and the pages will be allowed to access each other's DOM.

Domain relaxation can be implemented in our system in several ways. One is for profile.a.com to have the secrecy tag $F(\{\mathsf{profile.a.com}\}, \{\mathsf{profile.a.com}, \mathsf{login.a.com}\})$, which allows it to receive secrets from login.a.com; and for login.a.com to have a corresponding secrecy label.

Another option is to give each page the *name*.a.com→a.com reclassification capability. This would allow such pages to talk to a.com, but not yet to each other (because we currently apply reclassification only if necessary to complete a request, and only on the source entity). To accomplish that, their respective secrecy tags *name*.a.com would additionally need to be replaced with a.com, which could be accomplished by the browser crawling over the page's DOM and changing the secrecy tags of any nodes with the appropriate labels from *name*.a.com to a.com.

CSP A CSP allows a page to specify from where page resources (e.g., third-party scripts) can be loaded. The policy applies to images, scripts, etc. CSPs can be broadly interpreted as policies that a host page sets to constraint the information flow between the host page and remote servers from which external resources originate. When the request (e.g., HTTP GET) is sent to a remote server, information flows from the browser to the remote server. The host page can send arbitrary information to the remote server in this way by, e.g., embedding it in the URL string of the HTTP GET request. Once loaded, external resources such as scripts can interact with the rest of the page as well as with remote servers. Our generalized CSP (GCSP) (Section III-B2) can be used to specify the above-mentioned information-flow constraints present in CSPs.

There are two main differences between our GCSP and the existing CSP. First, the existing CSP takes effect only at resource-loading time and does not constrain transitive information flows. E.g., if url's CSP forbids scripts from ad.com, it doesn't mean that an extension's content script running in the same page is prevented from sending to or receiving information from ad.com. GCSP enforces a stricter policy: Any information tagged with a url secrecy tag cannot be sent to components that do not have that tag. Second, CSPs also enforce policies other than information flow. For instance, not loading resources from an external resource also prevents the external resource from using local resources such as the screen or CPU. This will effectively protect the user from seeing offensive ads, prevent scripts from draining the laptop battery, etc.

In modern browsers, web pages are allowed to embed third-party content with little restriction. Our modified browser has stricter constraints. To allow web pages to load third-party content, we explicitly enable two-way communication between the page and the external resources. For example, the DOM label from Section III-D explicitly allows the cnn.com page to receive secrets from ad.com and to reclassify its own secrets to allow requests to ad.com.

ℓ_{doc_1} $S = F(\{\mathsf{cnn.user}\}, \{\mathsf{cnn.}*, \mathsf{ad.}*, \mathsf{google.}*\})$,
$I = \{\}$, $D = \{+\mathsf{network}, \mathsf{cnn.}* \rightarrow \mathsf{ad.}*, \mathsf{cnn.}* \rightarrow \mathsf{google.}*\}$

If we wish all label checks to succeed, we can assign the permissive label $(F(\{\}, \mathsf{all}_S), \mathsf{all}_I, \{-\mathsf{all}_S, +\mathsf{all}_I\})$ to extensions and DOM elements. Here, all_S and all_I denote all the secrecy and integrity tags. This label allows an entity to access and declassify all private data.

postMessage postMessage is a JavaScript API which allows web pages to communicate across domains on the client side. postMessage works in two conditions: A parent page embeds another page in an iframe or a parent page opens another page in a new tab. In both cases, the API allows two-way communication. The postMessage send needs to specify the destination, and the receiver can check the source.

To allow communications using postMessage APIs in our system, the sender and receiver's labels need to be adjusted. If a host page were to send data directly to an iframe from a different origin, the request would be denied by our browser. To allow postMessages to work, labels are assigned to the host and iframed page in similar ways as discussed for the SOP and CSP.

iframe policies iframes were introduced as an isolation mechanism for a parent page to confine untrusted pages. However, iframes have been abused to embed trusted pages within malicious pages, which then mount phishing and clickjacking attacks.

To prevent such attacks, a server can specify, using the `X-Frame-Options` header, that the page should not be rendered inside a iframe at all, or should only be rendered inside an iframe of a page from a specified origin.

In a pure information-flow approach, disallowing a page from loading in an iframe cannot easily be done. We can, however, prevent the parent from gaining information from a loaded iframe. For example, if a.com tries to place victim.com in an iframe on its page and receive information from the iframe, it would have to have a secrecy label that can float to include victim.com's secrets. To prevent a.com from having a label that allows this, the browser would have to generate a.com's label from something other than a.com's (self-supplied) CSP. Such restrictions could be expressed cleanly using our composition operators (Section III-B2).

VII. Conclusion

We develop an approach for reasoning about the information flows in a fully fledged web browser. Our approach can support common browser policies, such as the same-origin policy, as well as more flexible, practically useful policies that current browsers cannot. We apply our approach to Chromium, developing a formal model and corresponding, functional prototype system. A proof of noninterference provides assurance of the model's correctness. At the same time, a formal analysis reveals some limits to the provided protections. Using the prototype, we experimentally validate the ability of our design to offer additional protection while continuing to support standard browsing functionality.

We believe our approach and model show one way to strike a balance between practicality and formal guarantees. As such, the approach we explored serves as a step towards developing rich information-flow enforcement models that acknowledge practical constraints.

ACKNOWLEDGMENT

This research was supported in part by US Navy grant N000141310156; NSF grants 0917047, 1018211, and 1320470; and the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office.

REFERENCES

[1] 64px.com. Facebook for Chrome, 2014. https://chrome.google.com/webstore/detail/facebook-for-chrome/gdalhedleemkkdjddjgfjmcnbpejpapp.

[2] S. A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.

[3] S. Agarwal. Session hijacking or cookie stealing using PHP and JavaScript, 2011. http://www.martani.net/2009/08/xss-steal-passwords-using-javascript.html.

[4] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *Proc. IEEE S&P*, 2012.

[5] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, 2009.

[6] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. POPL*, 2012.

[7] S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, and M. Winslett. Vetting browser extensions for security vulnerabilities with VEX. *Commun ACM*, 54(9):91–99, Sept. 2011.

[8] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proc. NDSS*, 2010.

[9] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proc. CCS*, 2008.

[10] A. Barth, C. Jackson, C. Reis, and T. G. C. Team. The security architecture of the Chromium browser. Technical report, 2008.

[11] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time monitoring and formal analysis of information flows in Chromium. Technical Report CMU-CyLab-14-015, CyLab, Carnegie Mellon University, 2014.

[12] L. Bauer, S. Cai, L. Jia, T. Passaro, and Y. Tian. Analyzing the dangers posed by Chrome extensions. In *Proc. IEEE CNS*, 2014.

[13] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *POST*, 2014.

[14] K. Boda, Á. M. Földes, G. G. Gulyás, and S. Imre. User tracking on the web via cross-browser fingerprinting. In *Information Security Technology for Applications*. 2012.

[15] A. Bohannon and B. C. Pierce. Featherweight Firefox: Formalizing the core of a web browser. In *Proc. USENIX WebApps*, 2010.

[16] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proc. IEEE CSF*, 2010.

[17] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proc. PLDI*, 2009.

[18] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *Proc. IEEE S&P*, 2006.

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

[20] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. IEEE S&P*, 2010.

[21] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *Proc. ACM CCS*, 2000.

[22] D. Fett, R. Küsters, and G. Schmitz. An expressive model for the Web infrastructure: Definition and application to the Browser ID SSO system. In *Proc. IEEE S&P*, 2014.

[23] Google.com. Google mail checker, 2014. https://chrome.google.com/webstore/detail/google-mail-checker/mihcahmgecmbnbcchbopgniflfhgnkff.

[24] W. Gordon. Many browser extensions have become adware or malware. Check yours now, 2014. http://lifehacker.com/many-browser-extensions-have-become-adware-or-malware-1505117457.

[25] C. Grier, S. Tang, and S. T. King. Designing and implementing the OP and OP2 web browsers. *ACM Trans Web*, 5(2):11:1–11:35, May 2011.

[26] J. Grossman. *XSS Attacks: Cross-site scripting exploits and defense*. Syngress, 2007.

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

[28] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proc. IEEE CSF*, 2012.

[29] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: attacks and defenses. In *Proc. USENIX Security*, 2012.

[30] IETF. HTTP header field X-Frame-Options, 2013. http://tools.ietf.org/html/rfc7034.

[31] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proc. ACM CCS*, 2010.

[32] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-time enforcement of information-flow properties on Android. In *Proc. ESORICS*, 2013.

[33] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. SOSP*, 2007.

[34] L. Liu, X. Zhang, and S. Chen. Botnet with browser extensions. In *Proc. IEEE SocialCom*, 2011.

[35] R. S. Liverani and N. Freeman. Abusing Firefox extensions, 2009. http://www.defcon.org/images/defcon-17.

[36] F. Martani. XSS, Passwords theft using JavaScript, 2009. http://www.martani.net/2009/08/xss-steal-passwords-using-javascript.html.

[37] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *Proc. IEEE CSF*, 2011.

[38] A. C. Myers. Practical mostly-static information flow control. In *Proc. POPL*, 1999.

[39] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proc. IEEE S&P*, 2012.

[40] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE CSF*, 2010.

[41] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J Sel Area Comm*, 21(1):5–19, 2003.

[42] D. Stefan, E. Z. Yang, B. Karp, P. Marchenko, A. Russo, and D. Mazières. Protecting users by confining JavaScript with COWL. In *Proc. OSDI*, 2014.

[43] S. Tang, H. Mai, and S. T. King. Trust and protection in the Illinois browser operating system. In *Proc. OSDI*, 2010.

[44] W3C. Content Security Policy 1.1, 2014. http://www.w3.org/TR/CSP11/.

[45] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proc. USENIX Security*, 2009.

[46] J. Wang, X. Li, X. Liu, X. Dong, J. Wang, Z. Liang, and Z. Feng. An empirical study of dangerous behaviors in Firefox extensions. In *ISC*, 2012.

[47] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys. In *Proc. NDSS*, 2006.

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

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