

Verified Security for Browser Extensions

Nikhil Swamy Benjamin Livshits Arjun Guha Matthew Fredrikson
Microsoft Research Microsoft Research Brown University U. Wisconsin, Madison

Microsoft Research Technical Report

MSR-TR-2010-157

Microsoft[®]
Research

Abstract

Popup blocking, form filling, and many other features of modern web browsers were first introduced as third-party extensions. New extensions continue to enrich browsers in unanticipated ways. However, powerful extensions require capabilities, such as cross-domain network access and local storage, which, if used improperly, pose a security risk. Several browsers try to limit extension capabilities, but an empirical survey we conducted shows that many extensions are over-privileged under existing mechanisms.

This paper presents a comprehensive new model for extension security that aims to redress the shortcomings of existing extension mechanisms. Our model includes various components. First, we develop a logic-based specification language for describing fine-grained access control and data flow policies that govern an extension's privilege over web content. We show how to understand security policies by providing visualization tools that highlight the impact of a policy on particular web pages. We formalize the semantics of policies in terms of a safety property on the execution of extensions and we develop a verification methodology that allows us to statically check extensions for safety. Static verification eliminates the need for costly runtime monitoring, and increases robustness since verified extensions cannot raise security exceptions. We also provide compiler tools to translate extension code authored in ML to either .NET or JavaScript, facilitating cross-browser deployment of extensions.

We evaluate our work by implementing and verifying 17 extensions with a diverse set of features and security policies. We deploy our extensions in Internet Explorer, Chrome, Firefox, and a new experimental HTML5 platform called C3. In so doing, we demonstrate the versatility and effectiveness of our approach.

I. INTRODUCTION

Like operating systems, IDEs, and other complex software systems, web browsers may be extended by third-party code. Extensions provide unforeseen new functionality and are supported by all major browsers. Although a precise count for each browser is hard to obtain, various sources¹ estimate that a third of all users of Firefox (some 34 million) use extensions, while the 50 most popular Chrome extensions have each been downloaded several hundred thousand times.

Notwithstanding their popularity, extensions can pose a significant risk to the security and reliability of the browser platform. Unlike JavaScript served on web pages, extensions can access cross-domain content, make arbitrary network requests, access security-sensitive HTML form fields such as passwords, and can make use of local storage. A malicious or buggy extension can easily void any guarantee of security that a browser tries to provide, e.g., with extensions installed, the same-origin restriction enforced by browser to prevent cross-domain flows is easily circumvented. Additionally, extensions affect page load times and browser responsiveness.

In light of these concerns, browser vendors have put in place various processes to control how extensions are distributed, installed, and executed. Mozilla, for example, manages a hosting service for Firefox *add-ons*. Newly submitted extensions are subject to an *ad hoc* community review process to identify extensions that violate best practices, e.g., polluting the global JavaScript namespace. In contrast, Google Chrome extensions explicitly specify the capabilities they need [3], requiring Chrome extensions to be written against a specific API and for their security privileges to be stated in an explicit manifest. When installing an extension, the user is prompted to grant it privileges that appear in the manifest.

We view the Chrome model as a step in the right direction, because it requires extensions to explicitly request privileges; these privileges can be reviewed independently of the extension code, and access controls are enforced by the browser. However, we find this model inadequate, for several reasons.

First, Chrome security manifests cannot be used to express many common policies. Access control specifications in manifests are extremely coarse grained, e.g., extensions may request access to a user’s entire browsing history, to all data on a set of web sites, etc. We also observe, empirically (Section II), that nearly a third of over 1,000 Chrome extensions we surveyed request full privileges over data on arbitrarily many web sites; and as many as 60% have access, often quite needlessly, to a user’s entire browsing history. From this we conclude that Chrome’s extension model offers inadequate protection for a user’s security- and privacy-related data.

This paper sets out to develop a new, comprehensive security model for browser extensions. Our primary aim is to provide a means to formally specify fine-grained data confidentiality and integrity policies for extensions and to have such policies enforced efficiently and with high reliability. Our

model, as illustrated in Figure 1, addresses three main principals that exist in this space: extension developers, curators of extension hosting services, and end-users.

Briefly, we develop a policy language that allows extension developers to distribute their extensions with manifests that describe fine-grained access control and information flow rights. We provide visualization tools to help extension curators and end-users understand policies. We apply static verification tools to automatically check extension code for policy compliance, eliminating runtime overhead and security exceptions. Finally, our approach allows extensions to be authored in a platform-independent way. We provide compiler support to deploy extensions in multiple browsers.

A summary of our technical contributions follows:

- We conduct an empirical study of a large corpus of Chrome extensions and their security privileges, and conclude that a significant fraction of Chrome extensions are over-privileged.
- We propose a policy language based on Datalog for specifying fine-grained authorization and data flow policies on web content and browser state accessible by extensions. We provide visualization tools to help understand the impact of authorization policies on specific web pages. Our choice of Datalog is motivated by a number of factors, including its expressiveness; its amenability to formal analysis; its widespread use in the prior art of authorization logics [4]; and, most importantly, because Datalog, as we will see, allows us to provide a sound basis for reasoning about the security behavior of extensions in the presence of untrusted third-party code and other extensions.
- We formalize the semantics of security policies and extensions. A distinctive feature of our semantics is that it accounts for an execution model that involves arbitrary interleavings of extension code with other untrusted scripts on a web page. These semantics enable the definition of a security property, $(\mathcal{L}; \mathcal{P})$ -safety, suitable for use with extensions that interact with other, untrusted code.
- We develop a methodology based on refinement typing (proved sound) that can be used to verify that extension code written in Fine [19], a dependently typed ML dialect, satisfies $(\mathcal{L}; \mathcal{P})$ -safety. Static verification eliminates the overhead of runtime security monitoring, and promotes robustness of the browser platform since extensions can never raise unexpected security exceptions.
- Our evaluation includes programming 17 extensions in Fine, specifying a range of fine-grained authorization and information flow properties for each, and automatically verifying them for policy compliance. Among these 17 extensions are several ported versions of widely-used Chrome extensions, which shows that our model also brings benefits to existing legacy extension architectures.
- Finally, we extend the publicly available Fine compiler with a code generator that emits JavaScript (in addition to .NET bytecode). This enables the development of extensions in a platform-independent manner, while allowing deployment in multiple browsers, including Internet Explorer 8, Chrome,

¹<http://blog.mozilla.com/addons/2009/08/11/how-many-firefox-users-use-add-ons/>, <http://chrome.google.com/extensions/list/popular>

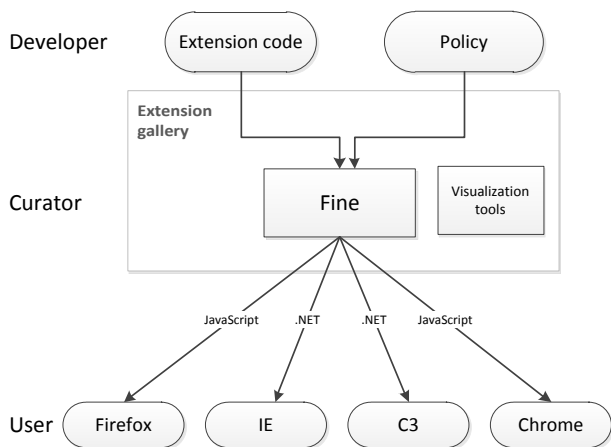


Fig. 1: System architecture.

and Firefox. Additionally, we show how to deploy extensions in C3 [15], a new platform for HTML5 experimentation developed entirely in a type-safe, managed language.

Outline of the paper: We begin in Section II by discussing existing extension security models, motivating the need for our work. We sketch of our entire solution in Section III, before addressing each element in detail. Section IV presents our policy language its visualization tool. Section V formalizes the semantics of policies and defines $(\mathcal{L}; \mathcal{P})$ -safety. Section VI shows how to statically verify extension code for safety. Section VII presents our experimental evaluation and discusses the code of two extensions in detail. Section VIII discusses our support for cross-browser deployment of extensions. Section IX discusses related work, and Section X concludes.

II. A SURVEY OF EXISTING EXTENSION MODELS

Extensions have access to resources and capabilities that are far more powerful than those available to scripts running on web pages. Therefore, malicious extensions are a security concern. Moreover, since extensions interact with web pages, a malicious page could exploit a vulnerable extension to access capabilities that web pages do not ordinarily possess. Extensions can customize browsers' interfaces, either extensively (e.g., Firefox) or in limited ways (e.g., Chrome). Unlike scripts on web pages, which can only affect the page on which they are hosted, extensions can read and modify arbitrary web pages. Furthermore, extensions are generally not subject to the same-origin policy that applies to scripts on web pages — this allows them to communicate with arbitrary web hosts.

These and others are powerful capabilities that should not be granted wholesale to all extensions. Below, we discuss security mechanisms employed by Internet Explorer, Firefox, and Chrome to motivate the design of our extension system. Of all three browsers above, we believe that Chrome has the most security-aware extension system to date. We perform a detailed study of over 1,000 Chrome extensions to study the effectiveness of its security model and conclude that many, if not most, extensions are unnecessarily over-privileged.

A. IE Extension Model: BHOs

Internet Explorer supports several extension mechanisms of which browser helper objects or BHOs are probably the most commonly used. BHOs (usually native binaries) have virtually unrestricted access to IE's event model and, as such, have been used by malware writers in the past to create password capturing programs and key loggers. This is especially true because some BHOs run without changes to the user interface. For instance, the CISpring Trojan² uses BHOs to install scripts to provide a number of instructions to be performed such as adding and deleting registry values and downloading additional executable files, all completely transparent to the user. Even if the BHO is completely benign, but buggy, its presence might be enough to open up exploits in an otherwise fully patched browser.

B. Firefox Extension Model: Overlays

Firefox extensions are typically written in JavaScript and can modify Firefox in fairly unrestricted ways. This enormous flexibility goes with few if any security guarantees. Extensions run with the same privilege as the browser process, so a malicious extension can cause arbitrary damage. Firefox extensions often employ highly dynamic programming techniques that make it difficult to reason about their behavior [14].

To protect end-users, Firefox relies on a community review process to determine which extensions are safe. Only extensions deemed safe are added to Mozilla's *curated extension gallery*. Firefox ordinarily refuses to install extensions that do not originate from this gallery. Users are thus protected from unreviewed extensions, but reviews themselves are error-prone and malicious extensions are sometimes accidentally added to the gallery. An example of this is an extension called Mozilla Sniffer which was uploaded to the extension gallery on June 6th, 2010 to be removed on July 12th, 2010, having been downloaded close to 2,000 times [18].

C. Chrome and Coarse-grained Access Control: Manifests

Google Chrome extensions are written in JavaScript and hosted on extension pages, but they have access to extension APIs that are not available to web pages. Extension pages run in the context of the extension process, different from the browser processes and has the ability to both access and augment the browser UI. Extension pages can register to listen to special browser events such as tab switching, window closing, etc.

Extension manifests: Extensions specify their resources and the capabilities they require in an extension manifest file. When a user tries to install an extension, Chrome reads the extension manifest and displays a warning. Figure 2 shows the manifest of an extension called *Twitter Extender* and the warning raised by Chrome before the extension is installed. In this example, the manifest requests (roughly) read and write privileges over all content on <http://api.bit.ly> and <http://twitter.com>. Additionally, this extension requires

²<http://www.ca.com/us/securityadvisor/virusinfo/virus.aspx?ID=42280>

```

"update_url": "http://clients2.google.com/service/...",
"name": "Twitter Extender", "version": "2.0.3",
"description": "Adds new Features on Twitter.com ",
"page_action": { ... }, "icons": { ... }, \
"content_scripts": [ {
  "matches": [
    "http://twitter.com/*", "https://twitter.com/*"],
  "js": ["jquery-1.4.2.min.js", "code.js"]
} ],
"background_page": "background.html",
"permissions": [ "tabs", "http://api.bit.ly/" ]

```

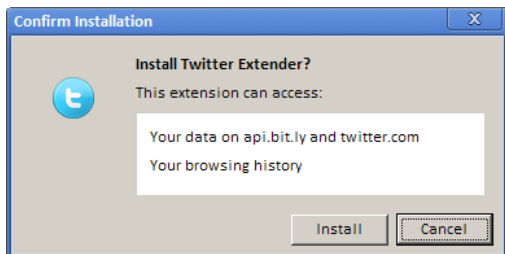


Fig. 2: A fragment of Twitter Extender’s manifest and the dialog that prompts a user for access privileges when the extension is installed

Name	Behavior
Google Reader client	Sends RSS feed links to Google Reader
Gmail Checker Plus	Rewrites mailto: links
Bookmarking	Sends selected text to delicious.com
Dictionary lookup	sends selected text to online dictionary
JavaScript toolbox	edits selected text
Password manager	stores and retrieves passwords per page
Short URL expander	sends URLs to longurlplease.com
Typography	modifies values on <input> elements

Fig. 3: Some over-privileged Chrome extensions that require access to “your data on all websites”

access to events related to browser tab manipulations. In Chrome’s model, access to tabs implies that the extension has access to the user’s browsing history. This is rather unfortunate — this extension certainly does not need access to all of a user’s browsing history to function properly, but Chrome’s model makes it impossible to restrict its privilege any further.

Over-privileged extensions: Twitter Extender’s access to browsing history is not an isolated example of an overprivileged extension. Chrome’s model also allows extensions to request rights over other resources, including, the privilege to access “your data on all websites”. Unfortunately, many simple, seemingly benign operations require extensions to request access to this very coarse privilege — Figure 3 lists several of these. In all these cases, manifests are essentially uninformative, and reviewing the security of these extensions requires manual code review.

Extension study: We conducted a simple analysis of the manifests for over 1,139 popular Chrome extensions, to determine how many require the capability to read and write to all websites. Our results are shown in Figure 4. Over 10% of all extensions require access to all `https://` sites, and event more need access to `http://` sites. About half of all extensions use wildcards, such as `http://*.facebook.com`

to specify the sites they want to access. Since new subdomains can and do appear under a domain such as `facebook.com`, policies that use wildcards can be overly permissive. Only a small percentage of extensions restrict their access to only several URLs (about 17%).

What is perhaps most troubling about the Chrome access control model is that about 60% of all extensions have access to a combination of browser tabs and local storage. Using these two facilities, an extension can monitor which sites the user goes to, collecting browser history.

Resource	Count	
all https	143	12%
all http	199	17%
wildcard	536	47%
1 URL	149	13%
2 URLs	30	2%
3 URLs	15	1%
4 URLs	6	<1%
5 URLs	1	<1%
86 URLs	1	<1%
<hr/>		
history (tabs)	694	60%
<hr/>		
bookmarks	66	5%
notifications	15	1%

Fig. 4: Chrome extensions permissions statistics.

III. AN OVERVIEW OF OUR SOLUTION

Internet Explorer’s BHOs and Firefox’s JavaScript extensions are very hard to secure reliably. Chrome’s extension system, while being the most advanced browser extension model in everyday use, still admits a large number of over-privileged extensions. Our work aims to redress these difficulties using a number of mutually complementary measures. This section briefly sketches our solution, starting from our policy language, which enables the specification of fine-grained policies for extensions. Policies can be audited by curators independently of the code of extensions, and we provide visualization tools to assist with the task of understanding policies. We advocate a model in which extensions are authored in high-level, type-safe languages in a style that allows them to be automatically verified for compliance with policies. Additionally, through the use of compiler tools, extensions can be deployed in multiple browsers.

A. FacePalm: A Running Example

FacePalm, an extension we wrote, will be the running example of the next several sections. This is an extension that allows a user to manage an address book built from contact information that their friends make accessible on Facebook, a social networking site. When a user visits a friend’s Facebook page in a browser extended with FacePalm, the extension crawls the page to identify any updated contact information and, if it finds anything, automatically sends the information to an online address book for the user maintained on a third-party bookmarking service, say, `delicious.com`.

While useful, FacePalm raises several potential security concerns. For one, it violates the browser’s same-origin restrictions by sending data from the `facebook.com` domain to `delicious.com` — however, this behavior is entirely by design. More significantly, a user may be concerned that FacePalm manipulates her Facebook data in other, less desirable ways. For example, FacePalm may automatically send, accept, or reject friend requests on the user’s behalf, it might

send more than just contact information to Delicious (e.g., a user’s photographs), update status messages etc. We would like to be able to specify a security policy for FacePalm that limits its behavior to its advertised functionality, thus increasing a user’s confidence in the extension. Existing approaches are inadequate for this purpose. For example, in the language of Chrome’s security manifests, all that can be said about FacePalm is that it may access all data on both `facebook.com` and `delicious.com`.

B. A Policy Language for Fine-grained Specifications

We base the design of our policy language on the insight that the structure of web content can be exploited to specify precise security policies. For example, the tree structure of the DOM can be used to grant extensions access to certain fragments of a page; the structure in various URL schemes can be used to control cross-domain information flows, etc.

Our language takes the form of an ontology for Datalog, where the predicates in our ontology are chosen with the structure of web content in mind. As an informal first example, we show below a simplified version of the policy for FacePalm:

```
(* Extension policy *)
∀e, p. (EltParent e p && EltTagName p "div" && EltAttr p "class" "website")
  ⇒ CanRead e
```

Informally, this policy allows an extension to read text contained within `<div class="website">` elements. Literally, it says that for all DOM elements `e` and `p` on a web page, if `p` is the parent of `e`, and if `p` is a `<div>` node, and if `p` has the `class="website"` attribute; then, the extension holds the `CanRead` privilege on the element `e`. Of course, the structure of real Facebook web pages are considerably more complicated than this first example suggests. However, our experience shows that other features of real pages (e.g., style attributes), serve as effective markers for the specification of fine-grained data-driven security policies on web content.

C. Semantics and tool support to understand policies

Rather than requiring end-users and curators to examine and understand Datalog, our visualization tool interprets policies on specific web pages, highlighting the content on a page to which an extension has been granted access. While our visualization tool helps provide an informal understanding of policies, it can also be imprecise. We provide a formal semantics of policies and define a property on program executions that policies are intended to induce.

D. Static Verification of Extensions

The richness of our policy language makes it easy to specify fine-grained, extension-specific policies. However, ensuring that such policies are properly enforced poses several challenges. We show how to surmount these challenges by programming extensions in Fine, an ML dialect, and statically verifying them against fine-grained security specifications using dependent and refinement types provided by Fine.

Our verification methodology involves developing an API for extensions annotated with refinement types to capture

security-related pre- and post-conditions. We program extensions against this API and verify them for compliance with a security policy \mathcal{P} . We present a detailed review of Fine in Section VI, but give a taste of our approach below.

```
(* Refined DOM API *)
val tagName: e:elt → t:string{EltTagName e t}
val firstChild: p:elt → e:elt{EltParent e p}
val getAttr: e:elt → a:string → v:string{EltAttr e a v}
val textContent: e:elt{CanRead e} → string
```

The code above declares types for four common functions in our API that allows extensions to manipulate the DOM. The type of `tagName` says that it is a function that takes a DOM element `e` (given the abstract type `elt`) as an argument, and returns a string `t` as a result. Additionally, the type of `tagName` is annotated with a post-condition asserting that the returned string `t` is related to the argument `e` according to `EltTagName e t`, a proposition used in our authorization policies. The types of `firstChild` and `getAttr` are similar. In contrast, the type of `textContent` shows it to be a function from DOM elements `e` to strings, where the returned string could be security-sensitive data on a page, e.g., it could represent the contents of a password field. To ensure that extensions cannot access such sensitive content without appropriate privileges, the type of `textContent` is annotated with a pre-condition that requires the caller to have the `CanRead e` privilege on the argument `e`.

```
(* Extension code *)
let extensionCode (e:elt) =
  if tagName e = "div" && getAttr e "class" = "website"
  then textContent (firstChild e) else ""
```

Extension code (above) is written against this API is verified statically using refinement type checking for compliance with its policy. Failed policy compliance checks result in compile-time errors. However, this particular extension is well-behaved — it inspects the structure of a web page, and manipulates data only from the fragments of the page to which it (via the policy rule shown previously) has been granted access. Importantly, extension code is untrusted and never has to be manually inspected for potential vulnerabilities or malice. Curators and end-users need only look at their policies.

E. Authoring and Deploying Extensions

In addition to verifying extensions, our approach allows extensions to be developed in a platform-independent manner. Our extensions may be implemented, depending on the browser, either using a JavaScript bridge to the underlying browser implementation (Chrome, Firefox), directly in .NET (C3), or via bindings from .NET to native code (Internet Explorer). Our tools include a new code generator that allows us to compile extension code either to JavaScript or to .NET, allowing extensions to be authored once in Fine, and deployed on multiple browsers.

In addition to cross-browser deployment, JavaScript code generation allows our approach to be used in combination with existing extension security models. In particular, we show how to verify authorization properties for Chrome extensions by partially porting their *content scripts* (the interface of a

Chrome extension to the DOM) from JavaScript to Fine — the much larger *extension core* can remain in JavaScript and inter-operates with code generated from Fine. While such hybrid approaches are attractive for the ease of use and migration, the security guarantee in such a configuration is, of course, weaker; for instance, unverified extension cores are free to violate information flow properties.

IV. A LANGUAGE FOR FINE-GRAINED POLICIES

This section introduces our policy language, a Datalog-based framework for specifying fine-grained data confidentiality and integrity policies for browser extensions. We present our policy visualization tool, and discuss how policies may be analyzed for robustness.

A. Language Design

Distinguishing data from metadata: We take the view that the structure of web content can be interpreted as security metadata, and can be used to restrict the privilege of extensions at a fine granularity. As such, we think of page structure as inducing a kind of dynamic, data-driven, security labeling [23] on web content. From this perspective, since the extension’s behavior depends on the metadata of a page, it is most convenient if the metadata itself can be considered to be not security sensitive.

Determining which elements of semi-structured web content constitute metadata is a design decision that involves weighing several factors. In this work, we choose to the view the tag and some of the attribute structure of a web page as being security metadata that an extension can freely inspect but not modify. In contrast, the text, links, images, and all other content on a web page is considered, by default, to be high confidentiality (secret) and immutable. Extension-specific policies must explicitly grant an extension privileges to access or modify non-metadata content. Our experience indicates that this choice represents a good balance of concerns — it leads to a familiar programming model for extensions, while still providing good protection for a user’s sensitive web content.

Stability of a security policy and the choice of Datalog: Another constraint in the design of our policy language is driven by the execution model for extensions. Specifically, JavaScript that appears on the web page can interact with extensions via shared state in the DOM. Furthermore, while JavaScript and extension code share a single thread of control, their execution can be interleaved arbitrarily. A key property that we wish for our policy language is that the security policies should be *stable*. This notion is spelled out in the next section; intuitively, stability ensures that a well-behaved extension that is deemed to comply with a policy will never become insecure because of the actions of unanticipated JavaScript on the web page.

Accounting for these considerations, we choose to base our policy language on Datalog. We define a set of predicates to use with policies, where these predicates reflect the structure of web content. Importantly, Datalog’s restricted use of negation ensures that policies are always stable.

Metadata predicate	Description
DocDomain doc string	the document, <code>doc</code> has domain <code>string</code>
EltDoc elt doc	the element <code>elt</code> is in the <code>doc</code>
EltParent elt p	<code>p</code> is the parent-element of <code>elt</code>
EltTagName elt tagName	<code>elt</code> ’s tag-name is <code>tagName</code>
EltTextValue elt v	<code>elt</code> ’s text-value is <code>v</code>
EltAttr elt k v	<code>elt</code> has an attribute <code>k</code> , with value <code>v</code>
EltStyle elt sty	<code>elt</code> ’s style is <code>sty</code>
UrlScheme url s	<code>url</code> ’s scheme is <code>s</code> (e.g., “http:”, “ftp:”, etc.)
UrlHost url h	<code>url</code> ’s host is <code>h</code>
UrlQuery url p	<code>url</code> ’s query parameters are <code>p</code>
FlowsFrom a b	<code>a</code> was derived from <code>b</code>
Permission predicate	
CanReadSelection doc	the extension can determine user’s selection on <code>doc</code>
CanAppend elt	the extension can append elements to <code>elt</code>
CanEdit elt	the extension can modify <code>elt</code>
CanReadValue elt	the extension can read the text value of <code>elt</code>
CanWriteValue elt	the extension can write text to <code>elt</code>
CanWriteAttr elt k v	the extension can write <code>v</code> to the <code>k</code> -attribute of <code>elt</code>
CanReadAttr elt k	the extension can read the attribute named <code>k</code> on <code>elt</code>
CanStyle sty	the extension can modify the style <code>sty</code>
CanRequest str	the extension can send HTTP requests to url <code>str</code>
CanFlowTo a b	the extension is allowed to write <code>a</code> to <code>b</code>
CanReadHistory site	the extension is allowed to read history on <code>site</code>
CanReadFile file	the extension is allowed to read the local <code>file</code>

Fig. 5: A selection of the predicates in our policies.

Figure 5 shows a selection of the predicates we provide. The figure is split into two parts, the top showing the predicates we use to speak about security metadata; the bottom showing predicates that grant privileges to extensions. Most of the predicates listed in the figure are self-explanatory. However, a few are worth further discussion. The predicates `EltTextValue` and `EltAttr` appear in the metadata section of the figure. However, both the text and attribute content of a web page are, by default, considered sensitive information. In order to be able to access the text values and attributes of an element `e`, an extension must be granted explicit `CanReadValue` and `CanReadAttr` privileges on `e`. We show an example of this shortly. Note also that we provide predicates `FlowsFrom` and `CanFlowTo`, which allow a policy to impose data flow constraints on extensions — this is particularly important for controlling access to resources such as browsing history (Section VII-B).

An example policy: The top of Figure 6 shows part of the policy we use with FacePalm. The first rule grants the extension the ability to read `class` attributes on all elements in the page, i.e., `class` attributes are considered metadata in this policy. The second rule states that for all elements `e` that have their `class` attribute set to the value “label”, the extension has read access to the text content of their immediate children. The third rule is the most complicated: it states, roughly, that for a specific sub-element `website` of a node tagged with the “label” attribute and “Website:” text value, the extension has the right to read a link stored in the `website` node.

- (* Required to select elements by class (i.e., the "label" class) *)
1. $\forall e. \text{CanReadAttr } e \text{ "class"}$
 (* Requires to read the label text *)
 2. $\forall e. \text{child. EltParent child } e$
 $\&\& \text{EltAttr } e \text{ "class" "label"} \Rightarrow \text{CanReadValue child}$
 (* Permission to read website links *)
 3. $\forall \text{data, label, labelText, website, parent.}$
 $\text{EltParent data parent } \&\& \text{EltParent label parent}$
 $\&\& \text{EltParent website data } \&\& \text{EltParent labelText label}$
 $\&\& \text{EltAttr label "class" "label"} \&\& \text{EltTextValue labelText "Website:"}$
 $\Rightarrow \text{CanReadAttr website "href"}$

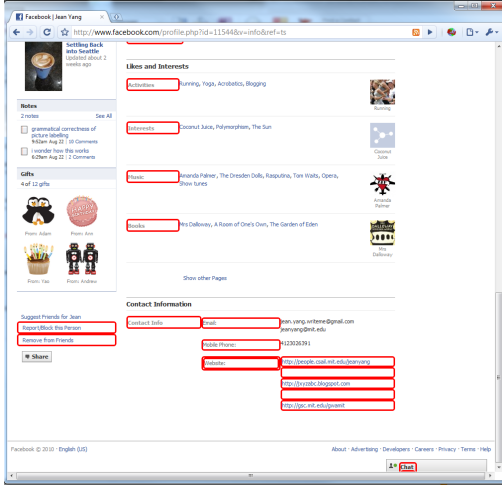


Fig. 6: FacePalm’s policy and its visualization on a Facebook page.

B. Understanding Policies

Extensions and their policies are often designed with specific websites in mind, e.g., FacePalm’s code and policy is closely tied to the structure of a Facebook web page. Such precise policies are meaningless, unless the reader understands the structure of the HTML used on the relevant websites. For example, the author of FacePalm must understand the structure of HTML on Facebook to write both its security policy and its code. However, it is unreasonable to expect reviewers and end-users to study Facebook’s HTML source.

We can visualize security policies by interpreting them as selectors, and highlighting readable/writable elements. We built a Chrome extension that reads a policy and then highlights the accessible elements on any web page. For example, the bottom of Figure 6 shows the elements accessible to FacePalm on a particular Facebook profile. Various labels such as “Interests”, “Chat”, “Music”, and “Website:” are highlighted, since the extension needs to search through the labels until it finds “Website:”. The websites on the profile are highlighted, since they are the data that FacePalm reads and sends to delicious.com. Most important, consider the data that is not highlighted — email addresses, phone numbers, likes and dislikes, etc. — this data is inaccessible to FacePalm, as advertised. Therefore, we can be confident that FacePalm is secure when it runs on this particular web page. To gain further confidence, we should visualize the policy on other kinds of Facebook page: our own profile, profiles that are unreadable, the settings page, etc.

There are three issues with visualization. First, it does not provide complete coverage, since visiting all Facebook pages is impractical. Second, there are elements of policies that our visualizer cannot handle, such as information-flow policies. Finally, it is possible to write a policy that only triggers on very specific Facebook pages, such as the one below.

$\forall e1, e2, x. \text{EltAttr } e1 \text{ "rare" } x \Rightarrow \text{CanReadValue } e2$

Unless the reviewer happens to sample a page containing an element with the “rare” attribute, the visualizer will not highlight anything. As such, visualization is not a substitute for reading the policy, but it does help understand them.

Robustness of a policy: Visualization is one tool to assist with understanding and vetting policies. We envisage building several other useful tools for policy analysis. An advantage of using Datalog as the basis of our language is the availability of tools on which to base such analyses. One obvious analysis is to check for policies that use specific undesirable patterns. For example, a policy should not grant an extension the privilege to modify a page in a way that allows the extension to grant itself access to protected resources. The following policy exemplifies this undesirable pattern: the attribute (class) that protects access to an element is mutable by the extension.

$\forall e. \text{CanWriteAttr } e \text{ "class"}$
 $\forall e, k. \text{EltAttr } e \text{ "class" "readable"} \Rightarrow \text{CanReadValue } e$

Detecting such situations is relatively straightforward since Datalog policies can be automatically analyzed to enumerate the set of attributes over which an extension has write privilege. A simple syntactic check to ensure that none of these attributes ever appear within a metadata predicate ensures the integrity of security-sensitive metadata. We leave the implementation of such an analysis to future work.

V. THE SEMANTICS OF SECURITY POLICIES

This section formalizes a core language and execution model for browser extensions. The distinctive feature of this model is that the execution of extension code is interleaved arbitrarily with JavaScript on the web page. We use this model to provide a semantics for security policies and define a safety property for extensions — safe extensions never cause runtime security failures. In the following section, we show how refinement type checking can be used to soundly decide extension safety.

A. λ^{BX} : A Core Calculus for Browser Extensions

The listing below shows the syntax of λ^{BX} , a (tiny) lambda calculus that we use to model extensions and their interactions with the DOM. We also show a syntax (\mathcal{P}) for a model of the policy language of the previous section. Both λ^{BX} and \mathcal{P} are to be understood as minimal core models — we leave out many elements of our practical implementation, including network access, event handling, local storage, and browsing history.

Syntax of λ^{BX} and policies \mathcal{P}

values v	$::=$	$x \mid c \mid (v_1, v_2) \mid op \mid \lambda x.e$
expr. e	$::=$	$v \mid e_1 e_2 \mid (e_1, e_2) \mid \pi_1 e \mid \pi_2 e$ \mid $\text{if } e \text{ then } e_1 \text{ else } e_2$
const. c	$::=$	$() \mid \text{true} \mid \text{false} \mid s \text{ (string)} \mid \eta \text{ (nodes)}$
opers. op	$::=$	$\text{getAttr} \mid \text{setAttr} \mid \text{getChildren} \mid \text{strEq}$
policy \mathcal{P}	$::=$	$\cdot \mid \forall \bar{x}. \phi_1, \dots, \phi_n \Rightarrow \phi \mid \mathcal{P}, \mathcal{P}'$
preds. ϕ	$::=$	$\text{Parent } v_1 v_2 \mid \text{EltAttr } v_1 v_2 v_3$ \mid $\text{CanReadAttr } v_1 v_2 \mid \text{CanWriteAttr } v_1 v_2 v_3$ \mid $\text{FlowsFrom } v_1 v_2 \mid \text{CanFlowTo } v_1 v_2$

Values in λ^{BX} include variables x , constants c , pairs, and lambda abstractions. Expressions additionally include application, projection, and conditional forms. Constants include the unit value, booleans, identifiers η (which we use as abstract handles to DOM nodes), and string literals (for attributes of DOM nodes). The primitive operators of λ^{BX} are the most interesting parts. These include `getAttr` and `setAttr` to access and mutate the attributes of a node; `getChildren` to traverse a DOM tree; and `strEq` for primitive equality on strings.

A policy \mathcal{P} is a finite list of horn clauses. The base predicates ϕ are drawn from the ontology of Figure 5. Importantly, in order to establish a connection between λ^{BX} programs and their policies, the base predicates of \mathcal{P} are defined over the (first-order) values of λ^{BX} .

To relate the syntax of our core language to our other examples, we reproduce the extension code from Section III-D below and show its λ^{BX} version.

```
(* In Fine *)
let extensionCode (e:elt) =
  if tagName e = "div" && getAttr e "class" = "website"
  then textContent (firstChild e) else ""
(* In  $\lambda^{\text{BX}}$  *)
```

```
 $\lambda e.$ if (strEq(getAttr(e, "tagName"), "div"))
  then if (strEq(getAttr(e, "class"), "website"))
    then getAttr( $\pi_1$ (getChildren e), "textContent")
    else ""
  else ""
```

B. Dynamic Semantics of λ^{BX} .

This section presents a dynamic semantics for λ^{BX} programs governed by \mathcal{P} policies. Our semantics is carefully designed to account for the possibility of interleavings between untrusted, page-resident JavaScript and extension code. This design of our semantics and its corresponding safety property results in a fine-grained security model for extensions that is robust with respect to the effects of JavaScript on the web page.

To appreciate the design of our semantics, we first discuss (a strawman) security property that depends on the *instantaneous* dynamic state of a web page. In this model, consider a well-behaved implementation of an extension like FacePalm. Such an extension could query a metadata attribute on a DOM node (e.g., check that the `class` attribute of a node is `label`); decide according to the policy that it has read privilege over the node; and, could then proceed to read the contents of the node. If the node’s metadata changes just prior to the read (due the effect of page-resident JavaScript), under an instantaneous view of the policy, the read must be rejected as insecure. Effectively, due

to the behavior of unforeseen JavaScript, unpredictable time-of-check to time-of-use (TOCTOU) discrepancies can arise. Worse, under this model, an adversarial web page can cause extensions to throw runtime security exceptions, making the browser platform unstable.

To counter such difficulties, the key insight behind our semantics is to make the security behavior of extensions dependent only on a *dynamic log*, a monotonically increasing set \mathcal{L} of ground facts about page metadata. While page-resident JavaScript can cause additional facts to be introduced into the log, it can never remove facts from the log. In conjunction with our use of (strictly positive) Datalog as a policy language, this design ensures that page-resident JavaScript, and any TOCTOU discrepancies that it may introduce, can never cause security failures in extension code.

Figure 7 defines a reduction relation $\mathcal{P} \vdash (\mathcal{L}, e) \rightsquigarrow (\mathcal{L}', e')$, according to which a runtime configuration (\mathcal{L}, e) , consisting of a dynamic log \mathcal{L} of ground facts and a λ^{BX} term e , reduces to (\mathcal{L}', e') , while under the purview of an unchanging policy \mathcal{P} . This is a small-step reduction relation for a call-by-value language, with a left-to-right evaluation order, extended with reductions for the primitive operators of λ^{BX} . The definition of the relation makes use of an auxiliary judgment $\mathcal{L}; \mathcal{P} \models \phi$, a standard entailment relation for Datalog, stating that the fact ϕ is derivable from the database of ground facts \mathcal{L} and intensional rules \mathcal{P} . We omit the definition of the standard entailment relation for Datalog.

The rules (E-Ctx), (E- β), (E-If) and (E- π) are standard. The rule (E-Eq) is unsurprising — it represents an equality test on string values. (E-SetAttr) is more interesting. It represents an attempt by the extension program to alter the DOM by altering the attribute s_{key} on the node η to the value s_{val} . Our model views attribute mutation as a security-sensitive event, so the premise of (E-SetAttr) contains a security check. Specifically, we require the `CanWriteAttr` $\eta s_{key} s_{val}$ privilege to be derivable from the facts in the log \mathcal{L} and the policy \mathcal{P} .

As discussed in Section IV-A we view the tree structure of a page as security metadata not subject to access restrictions itself. This design is reflected in the rule (E-GetCh), which contains no security check in the premise — an extension is always free to traverse the structure of the page. However, in the conclusion of the rule, we record facts in the log \mathcal{L}' to indicate that the parent/child relationships between η , η_1 and η_2 . These facts can be used in subsequent security checks to grant privileges to extensions. Note that for the purposes of this model, we consider DOM trees as having infinite depth, i.e., it is always possible to access the children of a node. In practice (cf. Section VI-B), `getChildren` returns an option.

Finally, we have (E-GetAttr), which combines elements from (E-SetAttr) and (E-GetCh). Depending on the policy, some attributes of a node (say, its `innerText` field) are considered security sensitive and are subject to access controls; other attributes (say, a CSS class) can be treated as security metadata. For this reason, the premise of (E-GetAttr) contains a check to ensure that an extension has read privilege on the requested attribute. Additionally, we record facts in the log \mathcal{L}' .

$$\begin{array}{c}
\text{log } \mathcal{L} ::= \cdot \mid \phi \mid \mathcal{L}_1, \mathcal{L}_2 \qquad \text{eval. contexts } E[\bullet] ::= \bullet \mid E e \mid v E \mid (E, e) \mid (v, E) \mid \pi_i E \mid \text{if } E \text{ then } e_1 \text{ else } e_2 \\
\frac{\mathcal{P} \vdash (\mathcal{L}, e) \rightsquigarrow (\mathcal{L}', e')}{\mathcal{P} \vdash (\mathcal{L}, E[e]) \rightsquigarrow (\mathcal{L}', E[e'])} \text{E-Ctx} \quad \frac{}{\mathcal{P} \vdash (\mathcal{L}, \lambda x. e v) \rightsquigarrow (\mathcal{L}, e[v/x])} \text{E-}\beta \quad \frac{e' = e_1 \text{ when } v = \text{true} \quad e' = e_2 \text{ when } v = \text{false}}{\mathcal{P} \vdash (\mathcal{L}, \text{if } v \text{ then } e_1 \text{ else } e_2) \rightsquigarrow (\mathcal{L}, e')} \text{E-If} \\
\frac{}{\mathcal{P} \vdash (\mathcal{L}, \pi_i(v_1, v_2)) \rightsquigarrow (\mathcal{L}, v_i)} \text{E-}\pi \quad \frac{v = \text{true when } s_1 = s_2 \quad v = \text{false otherwise}}{\mathcal{P} \vdash (\mathcal{L}, \text{strEq}(s_1, s_2)) \rightsquigarrow (\mathcal{L}, v)} \text{E-Eq} \quad \frac{\mathcal{L}; \mathcal{P} \models \text{CanWriteAttr } \eta \ s_{key} \ s_{val}}{\mathcal{P} \vdash (\mathcal{L}, \text{setAttr}(\eta, (s_{key}, s_{val}))) \rightsquigarrow (\mathcal{L}', ()))} \text{E-SetAttr} \\
\frac{\mathcal{L}' = \mathcal{L}, \text{Parent } \eta \ \eta_1, \text{Parent } \eta \ \eta_2}{\mathcal{P} \vdash (\mathcal{L}, \text{getChildren } \eta) \rightsquigarrow (\mathcal{L}', (\eta_1, \eta_2))} \text{E-GetCh} \quad \frac{\mathcal{L}; \mathcal{P} \models \text{CanReadAttr } \eta \ s_{key} \quad \mathcal{L}' = \mathcal{L}, \text{EltAttr } \eta \ s_{key} \ s_{val}, \text{FlowsFrom } \eta \ s_{val}}{\mathcal{P} \vdash (\mathcal{L}, \text{getAttr}(\eta, s_{key})) \rightsquigarrow (\mathcal{L}', s_{val})} \text{E-GetAttr}
\end{array}$$

Fig. 7: Dynamic semantics of λ^{BX} : $\mathcal{P} \vdash (\mathcal{L}, e) \rightsquigarrow (\mathcal{L}', e')$.

The first fact indicates that the node η indeed has the attribute (s_{key}, s_{val}) ; the second records the fact that the value s_{val} was derived from η . The latter fact is useful for enforcing data flow properties — we discuss this in Section VII-B.

Modeling the effects of JavaScript via non-determinism:

Extensions and page-resident JavaScript interact via shared DOM state. In most browsers, extensions and JavaScript share a single thread of control. An event handler, whether JavaScript or extension, runs to completion on receiving an event, and then yields control back to the browser, which can then schedule another event handler. In general, when extension code regains control, the page may have evolved arbitrarily since the last time the extension had control.

We model this characteristic feature of the extension execution model by making the rules (E-GetCh) and (E-GetAttr) non-deterministic. The non-determinism in our formal model is at an arbitrarily fine level of granularity, e.g., successive calls to (E-GetAttr) with the same arguments are allowed to return different results, modeling the fact that JavaScript code can be interleaved between the two calls. In practice, interleavings are not arbitrarily fine — extension code in a single event handler runs to completion without preemption. However, closures and shared state across event handler invocations allow extensions to observe the effects of JavaScript, essentially, between any pair of syntactically adjacent instructions.

C. $(\mathcal{L}; \mathcal{P})$ -safety: A Security Property for λ^{BX}

The main security definition of this paper is a notion of safety of λ^{BX} programs, defined above as a traditional progress property on the reduction relation.

Definition 1 (Safety): An extension e is $(\mathcal{L}; \mathcal{P})$ -safe if either e is a value, or there exists an expression e' and a log \mathcal{L}' such that $\mathcal{P} \vdash (\mathcal{L}, e) \rightsquigarrow (\mathcal{L}', e')$ and e' is $(\mathcal{L}'; \mathcal{P})$ -safe.

$(\mathcal{L}; \mathcal{P})$ -safety has the pleasing property that the security of an extension does not depend on the actions of page-resident JavaScript. However, it also limits the kinds of security policies that can be defined. In particular, policies that involve dynamic revocations cannot be modeled using $(\mathcal{L}; \mathcal{P})$ -safety. While revocation is an important feature in many security settings, and may, in some cases, be relevant for browser extensions too, our view is that supporting revocations for extensions is undesirable, for a number of reasons.

First, $(\mathcal{L}; \mathcal{P})$ -safety applies to a single run of an extension on a web page. If a fragment of a page needs to be withheld

from an extension, the page author needs only to update the page and the next time the page is loaded, the extension will not have access to the withheld content. Second, revoking the privilege of an extension while it is executing is problematic is at odds with our goal of isolating the notion of extension security from the actions of JavaScript—if dynamic revocation is possible, we need some way to prevent arbitrary untrusted scripts from revoking extension privileges; this is not possible without modifications either to the said scripts or to the browser platform itself. Additionally, dynamic revocation leads to a more complex programming model.

A final point about $(\mathcal{L}; \mathcal{P})$ -safety: although cast essentially as an authorization property, $(\mathcal{L}; \mathcal{P})$ -safety can also be used to place restrictions on the kinds of information flows permitted in a program. We discuss this in detail, in the context of an example, in Section VII-B.

VI. STATIC ENFORCEMENT OF EXTENSION SAFETY

This section describes a methodology based on refinement type checking that we use to statically verify that extension programs are $(\mathcal{L}; \mathcal{P})$ -safe. In comparison to runtime enforcement, our approach has a number of benefits.

Improved performance: Static verification of extension safety removes the performance cost of runtime monitoring. In the context of $(\mathcal{L}; \mathcal{P})$ -safety, runtime monitoring is particularly expensive, since it requires a dynamic log to be maintained at runtime as well as a Datalog interpreter to be invoked (potentially) on each access to the DOM. Static enforcement allows the dynamic log to be virtualized, so no log need be maintained at runtime, and, of course, no runtime Datalog interpretation is necessary either.

Robustness: Static verification guarantees that extensions can never raise runtime exceptions, thus improving the robustness of the browser platform.

Expressiveness: In addition to the enforcement of $(\mathcal{L}; \mathcal{P})$ -safety, we show how we can enforce data-flow like taint-based properties with no runtime overhead. Dynamic enforcement of such policies is a subject of active research and adds further runtime overhead.

Our approach involves programming extensions in Fine, linking them against libraries that implement extension APIs, where the APIs are decorated with refinement types describing security-relevant pre- and post-conditions. We verify extensions using the Fine refinement type checker. We begin this

section with a brief review of refinement types and Fine. We then discuss the high-level architecture of our verification methodology and present fragments of the refined APIs that we expose to extensions. We then present several small examples of extension code and show how these are verified against the APIs. The section concludes with a discussion of the main theorem of the paper, namely that well-typed Fine programs are $(\mathcal{L}; \mathcal{P})$ -safe.

A. A Review of Refinement Types in Fine

Fine is a verification system for a core, functional subset of F#. The principal novelty of Fine is in its type system, which is designed to support static verification of safety properties via a mixture of refinement and substructural types — for the purposes of this paper, substructural typing is unimportant. This section describes the syntax and intuitions behind refinement types in Fine. For details, we refer the reader to a recent comprehensive presentation of Fine and other related languages [20].

Value-indexed types: Types in Fine can be indexed both by types (e.g., `list int`) as well as by values. For example, `array int 17` could represent the type of an array of 17 integers, where the index `17:nat` is a natural number value. Value indexes on types can be used to specify a variety of security constraints, e.g., example, `labeled int x` could represent the type of an integer whose security label is described by the program variable `x`.

Dependent function types: Functions in Fine are, in general, given dependent function types, i.e., their range type depends on their argument. These are written $x:t \rightarrow t'$, where the formal name `x` of the parameter of type `t` is in scope in `t'`. For example, the type of a function that allocates an array of `n` integers can be given the type `n:nat → array int n`. When a function is non-dependent, we drop the formal name.

Refinement types: A refinement type in Fine (technically, a *ghost* refinement) is written $x:t\{\phi\}$, where ϕ is a formula in which the variable `x` is bound. Fine is parametric in the logic used for formulas, ϕ , however, in practice, the logic is often a first-order logic with equality. In this paper, rather than use the full power of first-order logic, we limit the formula language to strictly positive Datalog, which, as explained earlier, is suitable for $(\mathcal{L}; \mathcal{P})$ -safety. Formulas are drawn from the same syntactic category as types, although, for readability, we use italicized fonts for formulas.

Refinements as pre- and post-conditions: We can use refinement types to place pre- and post-conditions on functions. For example, we may give the following (partial) specification to a list permutation, where the refinement formula on the return value `m` corresponds to a post-condition of the function, relating the return value to the argument. $\forall \alpha. !:\text{list } \alpha \rightarrow m:\text{list } \alpha \{ \forall x. \text{In } x \mid \Leftrightarrow \text{In } x \ m \}$. Refinement types can also be used to state pre-conditions of functions. For example, to rule out divide-by-zero errors, we could give the following type to integer division: `x:int → y:int{y != 0} → int`.

Kind language: Types in Fine are categorized according to a language of kinds. Types are divided into four basic kinds,

although we only consider two of these kinds in this paper. The kind \star is the kind of normal types; and, `prop`, the kind of propositions. Type constructors are given arrow kinds, which come in two flavors. The first, $\alpha::k \Rightarrow k'$ is the kind of type functions that construct a k' -kinded type from a k -kinded type α . Just as at the term level, type-level arrows are dependent—the type variable α can appear free in k' . Type functions that construct value-indexed types are given a kind $x:t \Rightarrow k$, where x names the formal of type t and x can appear free in k . In both cases, when the kind is non-dependent, we simply drop the formal name. For example, the kind of `list` is $\star \Rightarrow \star$; the kind of the value-indexed `array` constructor is $\star \Rightarrow \text{nat} \Rightarrow \star$; the kind of the propositional connective `And` is `prop ⇒ prop ⇒ prop`; the kind of the user-defined predicate `In` is $\alpha::\star \Rightarrow \alpha \Rightarrow \alpha \text{ list} \Rightarrow \text{prop}$.

Top-level assumptions: The predicates that appear in a refinement formula can be axiomatized using a collection of user-provided assumptions. For example, in order to axiomatize the list membership predicate `In`, the standard library of Fine contains assumptions of the form `assume ∀hd, tl. In hd (Cons hd tl)`. In the context of this paper, in addition to axiomatizing standard predicates, top-level assumptions are used to specify the security policy that applies to an extension.

Refinement type checking: A refinement type $x:t\{\phi\}$ is inhabited by values $v:t$, for which $\phi[v/x]$ is derivable. Formally, derivability is defined with respect to assumptions induced by the program context (e.g., equalities due to pattern matching), the top-level assumptions, and any formulas in a purely virtual dynamic log \mathcal{L} , where the contents of the log is itself soundly approximated using refinement types. The derivability of refinement formulas is decided by Fine’s type checker by relying on Z3 [6], an SMT solver. We show an example program and its typing derivation in Section VI-C.

B. Refined APIs for Extensions

Our verification methodology involves giving refinement-typed interfaces to browser functionality that is exposed to extensions. This section presents a fragment of this interface in detail and discusses how the types of these interfaces map to the semantics of Section V. We focus here on the API for the DOM; our implementation uses a similar approach to provide refined APIs for local storage, network, and browsing history.

The listing below shows a fragment of the refined DOM API we expose to extensions. It begins by defining two abstract types, `doc` and `elt`, the types of web documents and document nodes, respectively. Since these types are abstract, well-typed extensions can only manipulate values of these types using our exposed APIs.

Next, we define a number of type constructors corresponding to the predicates of our policy language (Figure 5)—Fine’s type and kind language makes it straightforward to define these predicates. We start at lines 4–8 by showing the definitions of several metadata predicates that can be used to speak about the structure of a web page. Lines 10–14 show predicates corresponding to authorization privileges. For example, at line 4, `DocDomain` is defined to construct a proposition (a `prop`-kinded

type) from a `doc` and a `string` value. Fine’s kind language also makes it possible to define polymorphic propositions. For example, the `FlowsFrom` proposition at line 8 relates a value v_1 of any type α to another value v_2 of some other type β , to indicate that v_1 was derived from v_2 ; `CanFlowTo` is similar.

The DOM API (partial)

```

1 module DOM
2 type doc (* abstract type of documents *)
3 type elt (* abstract type of DOM element nodes *)
4 (* DOM metadata predicates *)
5 type DocDomain :: doc => string => prop
6 type EltDoc :: elt => doc => prop
7 type EltTagName :: elt => string => prop
8 type EltAttr :: elt => string => string => prop
9 type FlowsFrom ::  $\alpha :: * \Rightarrow \beta :: * \Rightarrow \alpha \Rightarrow \beta \Rightarrow \text{prop}$ 
10 (* DOM permission predicates *)
11 type CanAppend :: elt => elt => prop
12 type CanEdit :: elt => prop
13 type CanReadAttr :: elt => string => prop
14 type CanWriteAttr :: elt => string => string => prop
15 type CanFlowTo ::  $\alpha :: * \Rightarrow \beta :: * \Rightarrow \alpha \Rightarrow \beta \Rightarrow \text{prop}$ 
16 (* Metadata queries *)
17 val getChild : p:elt -> int ->
18   r:elt option{ $\forall \text{ch. r}=\text{Some ch} \Rightarrow \text{EltParent p ch} \ \&\& \ \text{FlowsFrom r p}$ }
19 val parentNode : ch:elt -> p:elt{EltParent p ch}
20 val getEltById : d:doc -> x:string -> c:elt{EltDoc c d & EltAttr c "id" x}
21 val tagName : ce:elt -> r:string{EltTagName ce r}
22 (* Protected access to data *)
23 val getAttr : e:elt -> k:string{CanReadAttr e k} ->
24   r:string{EltAttr e k r & FlowsFrom r e}
25 val setAttr : e:elt -> k:string -> v:string{CanWriteAttr e k v} ->
26   _:unit{EltAttr e k v}
27 val getValue : e:elt{CanReadValue e} -> s:string{EltTextValue ce s}
28 val createElt : d:doc -> t:string ->
29   e:elt{EltDoc e d & EltTagName e t & CanEdit e}
30 val appendChild : p:elt -> c:elt{CanAppend c p} -> _:unit{EltParent p c}

```

Lines 16-21 show a sampling of functions that extensions can use to inspect the structure of a page. Each of these functions is given a refined type, where the refinement on the return value corresponds to a post-condition established by the function. At lines 24-31 we show functions that provide extensions with access to security sensitive data, e.g., the attributes of an element. The types of these functions are refined with both pre- and post-conditions, where the pre-conditions correspond to authorization privileges that the caller must possess in order to access, say, an attribute; while the post-conditions, as with the metadata queries, record properties of the page structure.

At one level, one can understand pre- and post-conditions as predicates that relate the arguments and return value of each function. However, a more precise reading is in terms of the dynamic semantics of λ^{BX} . To illustrate, consider the primitive operator `getAttr` of Figure 7. In our formal model, the reduction rule for `getAttr` η_{key} was guarded by a premise that required the proposition `CanReadAttr` η_{key} to be derivable from the policy and the facts in the log. We capture this requirement by giving `getAttr` a type that records the corresponding `CanReadAttr e k` predicate as a pre-condition. Going back to the formal model, if the policy check succeeds `CanReadAttr` η_{key} reduces to an attribute s_{val} , and, importantly, records the facts `EltAttr` $\eta_{\text{key}} s_{\text{val}}$ and `FlowsFrom` $\eta_{s_{\text{val}}}$ in the log. We capture this effect on the log

by giving `getAttr` a type that includes the corresponding version of these predicates in its post-condition.

With the understanding that log effects correspond to post-conditions, and that policy checks in the premises of our reduction rules correspond to pre-conditions, we discuss the remaining functions in our DOM API. The function `getChild` is the analog of the operator `getChildren` of our formal semantics, adapted for use with a more realistic DOM. At the moment, our logical model of the DOM ignores the relative ordering among the children of a node—we simply record the fact that a pair of nodes are in a parent/child relationship. Enhancing this model to include ordering constraints is certainly possible, however, our examples have so far not required this degree of precision on the structure of a page to state useful security policies. Extensions can traverse the DOM in both directions, using `getChild` and `parentNode`. The DOM also includes a function, `getEltById`, which provides random access to nodes using node identifiers—notice that the post-condition of this function is relatively weak, since the exact placement of the returned nodes in the DOM is undetermined.

Our API also provides functions that allow extensions to mutate the DOM. For example, using `createElt` and `appendChild`, a suitably privileged extension can alter the structure of a web page. The observant reader may wonder how such side-effecting operations can be soundly modeled using refinement types in a functional language. The key point here is that we model such mutation effects purely in terms of their effects on the dynamic log. Since the log grows monotonically, a property that was once true of an `elt` remains valid in the logic even after the element is mutated.

Concretely, suppose we have a pair of `elt` values `e1` and `e2`. Then, in a context where `CanAppend e2` is derivable, the predicates derivable at each line are shown in comments.

```

let p1 = getParent e1 in (* EltParent p1 e1 *)
    appendChild e2 e1 (* EltParent p1 e1 && EltParent e2 e1 *)

```

Importantly, even after `e1` has been added as a child of `e2` on the second line, the predicate `EltParent p1 e1` continues to be derivable, since it remains as a ground fact in the dynamic log. This behavior has a number of implications, which we discuss next.

First, this model of side-effects rules out the possibility of strong updates, or, equivalently, dynamic revocation. However, this model of side effects is *just right* for verifying properties of extensions that are interleaved with arbitrary JavaScript code. By ensuring that all log effects are strictly positive formulas, we ensure that the effects of unverified JavaScript cannot undo properties established by extensions. This strict positivity condition and its corresponding monotonic behavior of side effects is set up to precisely model this property.

Should dynamic revocations be required eventually for certain extensions, our model is likely up to the task. Prior work on Fine [19] specifically shows how to verify security properties of programs that must make strong updates. However, further research is required to truly provide a convenient programming model for working with revocations and

strong updates. For example, the DOM supports operations like `getEltById` which provides random access to nodes. Most programming disciplines that allow strong updates require programs to obey certain aliasing disciplines. Random access into pointer structures via functions like `getEltById` make it particularly hard to reason about aliasing.

As a final remark, when programming against this model, intuitions about the meaning of certain predicates, like `EltParent`, have to be adjusted slightly. Specifically, we must view `EltParent` as a many-to-many relation, since, as the example above illustrates, the element `e1` can have more than one parent. As such, our logical model of the DOM is a graph, rather than a tree. However, our experience indicates that this imposes little, if any, burden on the programmer. On the contrary, programming extensions with $(\mathcal{L}; \mathcal{P})$ -safety as the goal allows extension authors (at least from a security standpoint) to be largely unconcerned with the interleavings of extension code and JavaScript, which is a significant simplification.

C. Safety by Typing

The listing below shows a highly simplified fragment from FacePalm, code that was presented informally in Section III. We discuss how this code is verified against the DOM API.

A simplified fragment of FacePalm

```

1 prop EltAncestor :: elt ⇒ elt ⇒ prop
2 assume ∀e1, e2. EltParent e1 e2 ⇒ EltAncestor e1 e2
3 assume ∀e1, e2, e3. EltParent e1 e2 && EltParent e2 e3 ⇒ EltAncestor e1 e3
4 assume ∀(e:elt). CanReadAttr e "class"
5 assume ∀(e:elt), (p:elt). (EltAncestor e p && EltTagName p "div" &&
6   EltAttr p "class" "website") ⇒ CanReadValue e
7 let extensionCode (e:elt) =
8   let t = tagName e "div" in
9   let a = getAttr e "class" in
10  if t = "div" && a = "website"
11  then match getChild e 0 with
12    | Some c → Some (getValue c)
13    | None → None
14  else None

```

Lines 1–6 above show the policy used with the extension written in Fine using a collection of assumptions. The policy defines a relation `EltAncestor`, the transitive closure of `EltParent`, and at lines 4 and 5, grants the extension the privilege to 1) read the “class” attribute of every element on the page; and 2) to read the contents of any sub-tree in the page rooted at a div node which is whose class attribute is “website”.

Lines 7–14 show the code of the extension. At line 8, we extract the tag `t` of the element `e`; the post-condition of this function allows the Fine type checker to conclude, after line 8, that the proposition `EltTagName e p` is in the dynamic log. In order to check the call at line 9, we have to prove that the pre-condition `CanReadAttr e "class"` is derivable—this follows from the top-level assumptions. After line 9, we can conclude that the fact `EltAttr e "class" a` is in the dynamic log. At line 11, in the **then**-branch of the conditional, the type checker uses the types of the equality operation $(=):x:\alpha \rightarrow y:\alpha \rightarrow b:\text{bool}\{b=\text{true} \Leftrightarrow x=y\}$ and of the boolean operator $(\&\&):x:\text{bool} \rightarrow y:\text{bool} \rightarrow z:\text{bool}\{z=\text{true} \Leftrightarrow x=\text{true} \ \&\& \ y=\text{true}\}$ to refine its

information about the contents of the dynamic log. In particular, the type checker concludes that if control passes to line 11, then both `EltTagName e "div"` and `EltAttr e "class" "website"` are in the dynamic log, and, using similar reasoning, it concludes that if control passes to line 12, `EltParent e c` is in the dynamic log. Finally, at the call to `getValue c` at line 12, we need to show that the pre-condition `CanReadValue c` is derivable. Given the top-level assumptions, and all the accumulated information about the contents of the dynamic log, the theorem prover Fine uses can establish this fact.

The main formal result of this section is the theorem below. It states that a program e that is well-typed against an interface Γ_{DOM} (representing the type and value signatures in the **module** `DOM` listing), a set of assumptions representing a Datalog policy \mathcal{P} , and a set of ground facts in an abstract dynamic log \mathcal{L} , is guaranteed to be $(\mathcal{L}; \mathcal{P})$ -safe.

Theorem 1 (*Type-correct programs are $(\mathcal{L}; \mathcal{P})$ -safe*): Given a policy \mathcal{P} and its translation to a signature $S = \llbracket \mathcal{P} \rrbracket$; a dynamic log \mathcal{L} and its translation to an environment $\Gamma_L = \llbracket \mathcal{L} \rrbracket$; such that $S; \Gamma_{DOM}, \Gamma_L$ is well-formed (i.e., $\vdash S; \Gamma_{DOM}, \Gamma_L$). Then, for any assumption-free program e and type t , if $S; \Gamma_{DOM}, \Gamma_L \vdash e : t$, then e is $(\mathcal{L}; \mathcal{P})$ -safe.

Proof: A straightforward extension of the main soundness result of Fine, as described by Swamy et al. [20], wherein a reduction relation for Fine is given while accounting for a dynamic log of assumptions. We extend the core reduction rules with four additional cases corresponding to (E-StrEq), (E-GetAttr), (E-SetAttr), and (E-GetCh). In each case, we show that reduction preserves typing, according to the types given to the primitive operations in Γ_{DOM} . Finally, we appeal to a relation between first-order and Datalog derivability, showing that the former subsumes the latter. ■

VII. WRITING REAL EXTENSIONS

We have, to date, written 17 extensions to evaluate our framework. Some of these extensions are prototypes written from scratch; others are third-party extensions that we partially ported and verified. This section summarizes these extensions, their security policies, and discusses our experience programming and verifying them in Fine. Our experience suggests that while authoring extension code is relatively easy and verification times reasonably fast, stating precise security policies for extensions still demands a non-trivial amount of work from the programmer. However, we note that policies are often dictated by the structure of the code. We plan future work to infer policies via program analysis, and expect this to reduce programmer burden.

A. Summary of results

Figure 8 summarizes our experimental results. It lists the 17 extensions we wrote, the number of lines of code, the number of policy rules (assumptions), and the time taken to verify and compile each extension, and the

Name	LOC	# Assumes	Compile (s)	#Z3 q's
Verified for access control properties				
Magnifier	23	1	6.0	11
PrintNewYorker	45	2	6.2	15
Dictionary lookup	70	3	6.6	24
FacePalm	142	5	10.7	26
Bib Parser	262	2	5.9	15
Verified for access control and data flow properties				
Password Manager	52	2	5.7	14
Twitter Miner	36	2	5.6	18
Bing Miner	35	4	5.7	37
Netflix Miner	110	17	6.2	57
Glue Miner	101	11	8.9	77
News Personalizer	124	7	13.1	125
Search Personalizer	382	12	83.6	339
Partially ported Chrome extensions				
Bookmarking	(6K) 19	1	5.8	9
Gmail Checker Plus	(7K) 43	3	6.5	19
JavaScript Toolbox	(2K) 19	1	6.3	9
Short URL Expander	(494) 22	1	5.2	9
Typography	(20K) 44	2	6.2	15
TOTAL	1,529	78	194.2	819

Fig. 8: Summary of experimental evaluation.

number of theorem prover queries that were issued during verification. Each of these extensions was programmed against some subset of our refined APIs. Figure 9 alongside shows the various components in our APIs and the lines of code in each. It is worth pointing out that although most of our extensions use only a few policy assumptions, as illustrated in Section IV-B, logic-based policies are not always easier to read than code—our visualization tools go some way towards assisting with policy understanding.

API	LOC
Events + network	31
Local storage	37
JSON + Utilities	58
Behavior mining	260
DOM, URLs, Styles	267
TOTAL	653

Fig. 9: Extensions APIs

Our extensions fall into three categories. This first group includes five extensions that we wrote from scratch and verified for access control properties. Magnifier is an accessibility extension: it enlarges text under the mouse on any web page—its policy ensures that only the styling of a page is changed. PrintNewYorker rewrites links on `newyorker.com` to go directly to print-view, removing ads and the multi-page layout of the site—its policy ensures that the host of a link is never changed and that only known constants are appended to the query string of a url. Dictionary queries an online dictionary for the selected word—only the selected word is allowed to be sent on the network. Bib Parser uses its own language of XML patterns to parse the contents of one of the authors’ bibliography from a web format to bibtex—its policy guarantees that it only reads data from a specific URL.

The second group of extensions are all verified for a combination of authorization and information flow properties. The miners and personalizers in this group were developed in conjunction with a project that was specifically investigating the use of browser extensions for personalizing web content by mining user behaviors [9]. Our work here is complementary in that it places behavior mining extensions on formal grounds.

This is particularly important, since behavior mining, while extremely useful, can also be worrisome from a security and privacy perspective. The next section discusses a variation of one of these extensions in detail—the others have a similar flavor. The last group of extensions includes 5 Chrome extensions that we partially ported to Fine. We discuss these in detail in Section VII-C.

B. NewsPers: Flow Controls and Browsing History

NewsPers is an extension that provides a news personalization service on top of `nytimes.com`. It re-arranges the news stories presented on the front page to link to stories more likely to be interesting to the user. It does this in four steps, outlined below.

- 1) When the user browses to `nytimes.com`, NewsPers reads a configuration file on the local file system, that specifies a user’s news preferences.
- 2) It sends data from this preferences file to `digg.com`, a social news website, and obtains a response that lists currently popular stories.
- 3) It consults the user’s browsing history to determine which of these popular stories on `nytimes.com` have not been read before by the user.
- 4) Finally, it re-arranges the `nytimes.com` page, placing unread popular stories towards the top.

For this extension, we aim to enforce a policy that ensures 1) that `digg.com` only obtains data from the configuration file, and 2) that no information about browsing history is leaked to `nytimes.com` (in addition to what it may already know). Figure 10 shows a fragment of NewsPers.

We begin by showing a fragment of our API that provides extensions with access to features beyond the DOM. We start with an API to access the local filesystem, using the `readFile` function, which is guarded by the `CanReadFile` privilege. Next, we show the API for working with URLs and making network requests. And, finally, we show the API to the local browsing history. Rather than providing extensions with access to the entire browsing history, our API provides finer controls by which an extension can request to view the history of URLs that a user may have visited at a particular site.

Using this API, our policy grants NewsPers the privilege to read the configuration file it needs and to read a user’s browsing history only for `nytimes.com`. The assumption at line 15 illustrates how $(\mathcal{L}; \mathcal{P})$ -safety policies can be used to enforce flow controls. Here, we state that only information derived from the `prefs` file can be sent to `digg.com`.

Lines 16–17 specify that the NewsPers has the privilege to append an element `e2` as the child of another element `e1`, but only if `e1` is a `nytimes.com` node, and if `e2` was derived from a node on the same domain. In other words, this assumption gives NewsPers to reorder the structure of nodes on an `nytimes.com` page, but not to add any new content. This specification is particularly important since NewsPers has access to a user’s browsing history. If it is able to write arbitrary elements to an `nytimes.com` page, it could, for example, insert image tags to send requests to a third


```

1 (* Partial API to local file system, URLs, network, and history *)
2 type url
3 type CanReadFile :: string ⇒ prop
4 type UrlHost :: url ⇒ string ⇒ prop
5 type CanRequest :: url ⇒ string ⇒ prop
6 type CanReadHistory :: string ⇒ prop
7 val readFile: f:filename{CanReadFile f} → s:string{FlowsFrom s f}
8 val mkUrl: s:string → h:string → ... → u:url{UrlHost u h && ...}
9 val sendRequest: u:url → s:string{CanRequest u s} → resp:string
10 val historyOnSite: host:string{CanReadHistory h} → url list
11 (* Policy *)
12 let prefs = ‘AppData\NewsPers\prefs.txt’
13 assume CanReadFile prefs
14 assume CanReadHistory ‘nytimes.com’
15 assume ∀s, u. FlowsFrom s prefs && UrlHost u ‘digg.com’ ⇒
  CanRequest s u
16 assume ∀e1 e2 e3. FlowsFrom e2 e3 && EltDomain e3 ‘nytimes.com’
17   EltDomain e1 ‘nytimes.com’ ⇒ CanAppend e1 e2
18 assume ∀e e2 e3. EltAncestor e2 e3 && FlowsFrom e e2 ⇒ FlowsFrom e e3
19 (* Sending request to digg.com *)
20 val parseResponse: string → url list
21 let getPopularStories () =
22   let p = readFile prefs in
23   let url = mkUrl ‘http’ ‘digg.com’ ... in
24   let resp = sendRequest url p in
25   parseResponse resp
26 (* Rearranging nytimes.com *)
27 val munge: digg:url list → history:url list → url list
28 val nodesInOrder: o:url list → r:elt → (e:elt{FlowsFrom e r}) list
29 let start root =
30   if (domain root) = ‘nytimes.com’ then
31     let popular = getPopularStories () in
32     let h = getHistoryOnSite ‘nytimes.com’ in
33     let ordering = munge popular h in
34     let nodes = nodesInOrder ordering root in
35     iter (fun e → appendChild root e) nodes
36   else ()

```

Fig. 10: A fragment of NewsPers.

party, leaking information about the browsing history. Of course, by rearranging the structure of the `nytimes.com` page, NewsPers reveals the user’s browsing history on `nytimes.com` to `nytimes.com` itself — but this is not a serious concern.

At lines 20–26, we show an implementation of a function that reads data from the local preferences file and sends it to `digg.com`. Lines 27–36 show the high-level structure of the code that rearranges `nytimes.com`. We elide the implementations of several helper functions, but show their signatures — these are largely free of security-sensitive operations. Notice that the implementation itself is pleasingly free of type annotations. While decorating APIs with precise types requires some effort, this burden is assumed by us, the API developers.

Finally, the model of flow controls we adopt here fits naturally into the $(\mathcal{L}; \mathcal{P})$ -safety framework. However, in comparison to noninterference-based approaches to information flow controls, the security property we obtain is relatively weak. In particular, what we obtain is a form of syntactic secrecy, rather than an observational equivalence property. Practically, what this means is that an extension can leak information about the browsing history to `digg.com` by choosing to send various fragments of the user preference information to `digg.com` depending on what URLs appear in the browsing history, i.e., via a form of implicit flow. While prior work on Fine shows how to eliminate this form of leak using value-indexed types,

for simplicity, we choose not to employ this approach here.

C. Retrofitted Security for Chrome Extensions

In section II-C, we argued that many Chrome extensions are over-privileged because Chrome’s access-control system is too coarse-grained. We also described the innocuous behavior of eight overprivileged extensions (figure 3). Now that we have a fine-grained security system, we can consider securing them.

The last section of Figure 8 lists five full-featured extensions. Chrome extensions are split into two components — the *content script* and the *extension core* — that communicate by message-passing. The size of the extension core ranges from 500–20,000 lines of JavaScript (shown in parentheses). The extension core can perform various privileged operations (e.g., local storage, cross-domain requests, etc.), but it cannot directly read or write to web pages. Content scripts, on the other hand, can modify web pages, but they cannot access the resources that the extension core can. Of course, the two components can cooperate to provide extension core with access to the web page, and vice versa, or content script with access to storage. Nevertheless, the separation does provide a reasonable degree of isolation.

In principle, we could port the entire Chrome extensions to Fine and verify them for end-to-end properties. However, we chose to rewrite only the content scripts in Fine, leaving extension cores in JavaScript. This approach, while involving much less effort, provides Chrome extensions with a measure of the benefits of our fine-grained DOM authorization policies. As Figure 3 shows, these extensions interact with web pages in limited ways. However, their limited behavior cannot be precisely expressed in Chrome manifests, hence they require access to “your data on all websites”. We can precisely state the limited privileges that these extensions actually need, and to verify them automatically for compliance.

Our policy language and API remains the same, with the exception of trivial, Chrome-specific message-passing functions that allow our Fine-based content-scripts to communicate with extension cores. Deploying these extensions in Chrome involves compiling content-scripts written in Fine to JavaScript—we discuss this next.

VIII. CROSS-BROWSER EXTENSIONS

A significant benefit of the approach this paper advocates comes from the fact that once an extension is verified, it can be *re-targeted* to run in a variety of modern browsers. To date, we have run our extensions on four distinct web browsers: Internet Explorer, Google Chrome, and C3, a research Web browser under development at Microsoft Research. This section discusses our deployment configuration in each of these browsers. Additionally, because we can compile from .NET to JavaScript, we have also retargeted some of our extensions to run on Firefox.

Internet Explorer: BHOs: To target Internet Explorer, we compile our Fine extensions to .NET libraries. These libraries are then loaded by a single native IE extension, or a BHO, in IE parlance. The BHO is implemented in F# and hosts our

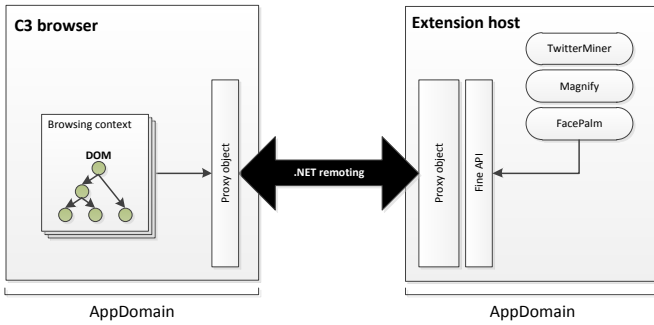


Fig. 11: C3 hosting architecture.

Fine extensions in an unprivileged *AppDomain*, a software-isolated process for .NET. The *AppDomain* allows us to easily load and unload extensions while the browser is running, but is not necessary for security guarantees, which are provided by Fine’s type system. Although, of course, both the .NET runtime and the browser itself are part of the TCB.

Google Chrome: porting the content scripts: As discussed in Section II, Google Chrome’s extension model cannot adequately express least privilege for a large class of extensions. Using a new JavaScript back-end for Fine, based on the λ_{JS} software [12], we compile our Fine extensions to ordinary Chrome extensions by translating .NET bytecode into JavaScript source code. In addition, we provide a trivial JavaScript runtime system that exposes JavaScript’s object-oriented DOM API as functions. Note that we can afford to only translate the content script of an extension, leaving the *extension core* of the extension running separately, in a different Chrome process. However, by rewriting extension content scripts in Fine, we gain the ability to reason and restrict how the extension interacts with HTML pages in a manner that is more restrictive and fine-grained than the default extension manifest.

A. A Fully-Managed Hosting Platform

C3 is an HTML5 experimentation platform written from the ground up exclusively in C#. Because C# code ultimately runs in a memory-managed environment, it is not susceptible to the memory corruption vulnerabilities that are responsible for many existing browser attacks. Our extension hosting architecture leverages this characteristic and benefits from the added safety.

Hosting architecture: Figure 11 illustrates the architecture we use to host Fine extensions inside of C3. When C3 initializes, it creates a new *AppDomain*, used to host all Fine extensions. C3 then loads a hosting module into the new application domain, which serves a dual purpose. First, the module searches a pre-defined directory for .NET assemblies that implement the interface supported by our Fine extensions. On finding such an assembly, the module loads it into the new application domain, and invokes its `main` function. This process is performed only once, on browser start-up.

Second, the hosting module acts as a “shim” layer between the Fine extension API and the internals of C3. This functionality is implemented using a .NET proxy object, which is a

type-safe cross-*AppDomain* communication mechanism. The proxy object contains one method for each internal C3 method needed by the API’s, which are then implemented in terms of methods on the proxy object. When an API function is invoked by an extension, each subsequent call to a proxy method causes the CLR to create a remote request to code in C3. Finally, C3 objects referenced by the proxy object are associated with integer GUIDs, communicated across *AppDomain* boundaries instead of serialized versions of the original objects.

We implemented extension APIs for C3 in about 270 lines of F#, and the proxy object implementation is 918 lines of F#. We find these requirements to be modest, and the gains due to the added type safety to be well worth the effort.

IX. RELATED WORK

Browser Extension Security: Ter Louw *et al.* [17] monitor calls by extensions to a subset of Firefox’s privileged APIs, in order to secure the extension installation process. While this establishes a form of access control for extension installation, the primary extension APIs remain unprotected, so extensions are still over-privileged. Barth *et al.* [3] develop the security model used for Google Chrome extensions. While this is the first extension model with native support for policy enforcement, the policies it supports are significantly more coarse-grained than the examples we presented in this paper, making it difficult to privilege extensions appropriately. Our work directly addresses this problem, by providing a fine-grained policy language and a static verification approach.

A number of researchers have explored the use of information flow for browser extension verification. Dhawan *et al.* present Sabre [7], a tool that instruments Firefox’s JavaScript interpreter to track security labels at runtime. The labels originate at sensitive data sources that they classify manually, and are checked at untrusted sinks that can result in leaks. However, because the overhead introduced by Sabre is as high as 5x, it is not suitable for policy enforcement, but rather for use as a forensic tool. In contrast, Bandhakavi *et al.* [2] presented Vex, a tool that statically analyzes Firefox extensions for a set pre-determined patterns of suspicious information flows. While not specifically tied to extensions, Chugh *et al.* also present an information flow analysis for JavaScript [5] that also looks for specific patterns of suspicious flows. However, because of the inherently dynamic nature of JavaScript, both approaches are not proven sound. In contrast, our approach allows us to soundly verify authorization and data flow properties of extensions; the properties are defined by a fine-grained logic-based extension specific policies; and our formal model characterizes safety even in the presence of unverified third-party code.

Many have addressed the problems that arise due to browser *plugins*, which consist of native code that executes in the context of the browser. Internet Explorer’s entire extension model fits into this description, and much recent research has addressed the problems that arise. In particular, spyware extensions have received attention [8, 13, 16]; these systems use binary taint-tracking to ensure that sensitive personal

information does not flow to untrusted parties. Addressing a more general set of concerns, Janus [10] and Google’s Native Client [22] considers system-level sandboxing techniques for browser extensions. The OP [11] and Gazelle [21] web browsers are constructed to address this issue, but do so by applying general principles of secure system design to the architecture of new browsers. In general, all these works target the enforcement of isolation and memory safety properties, not the more fine-grained authorization properties we address.

Verified Extensibility: Outside the specific setting of browser extensions, the question of providing verified extension mechanisms for system-level code has received much attention. With the SLAM project [1], Ball *et al.* show that software model checking proves effective at verifying the behavior of drivers. More recently, Zhou *et al.* explore the use of type safety to provide fine-grained isolation for drivers [24], and show how to apply their findings in a nearly backwards-compatible manner. Our work fits squarely in this tradition of static extension verification, but rather than focusing on system-level properties, we target those relevant to browser extension functionality.

X. CONCLUSIONS

This paper proposes a new model for authoring, verifying, distributing, and deploying safe browser extensions that can run on all the most popular browser platforms. Our motivation stems from the fact that even in the case of Chrome, which is, arguably, the most secure of the browser extension models in common use, extensions tend to be over-privileged, rendering many protection mechanisms useless. We propose a finer-grained access control model for browser extensions, formally characterize a security property for extensions, and develop a methodology to enforce safety statically. We evaluate our approach by developing 17 non-trivial browser extensions, demonstrating that our approach is viable in practice.

REFERENCES

- [1] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.
- [2] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*, 2010.
- [3] A. Barth, A. P. Felt, and P. Saxena. Protecting browsers from extension vulnerabilities. In *Network and Distributed System Security Symposium*, 2010.
- [4] P. C. Chapin, C. Skalka, and X. S. Wang. Authorization in trust management: Features and foundations. *ACM Comput. Surv.*, 40, 2008.
- [5] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.
- [6] L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *TACAS*, 2008.
- [7] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Annual Computer Security Applications Conference*, 2009.
- [8] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *USENIX Security Symposium*, 2007.
- [9] M. Fredrikson and B. Livshits. RePriv: Re-envisioning in-browser privacy. Technical report, Microsoft Research, 2010. MSR-TR-2010-116.
- [10] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *USENIX Security Symposium*, 1996.
- [11] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, 2008.
- [12] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, 2010.
- [13] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *USENIX Security Symposium*, 2006.
- [14] B. Lerner and D. Grossman. Language support for extensible Web browsers. In *APLWACA*, June 2010.
- [15] B. Lerner, H. Venter, B. Burg, and W. Schulte. C3: An experimental extensible, reconfigurable platform for HTML-based applications, 2010. In submission.
- [16] L. Li, X. Wang, and J. Y. Choi. SpyShield: Preserving privacy from spy add-ons. In *Conference on Recent Advances in Intrusion Detection*, 2007.
- [17] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3), 2008.
- [18] Mozilla Foundation. Add-on security vulnerability announcement. <http://blog.mozilla.com/addons/2010/07/13/add-on-security-announcement/>, July 2010.
- [19] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *ESOP*. Springer, 2010.
- [20] N. Swamy, J. Chen, C. Fournet, K. Bhargavan, and J. Yang. Security programming with refinement types and mobile proofs. Technical report, Microsoft Research, 2010. MSR-TR-2010-149.
- [21] H. Wang, C. Grier, E. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the gazelle web browser. In *USENIX Security Symposium*, 2009.
- [22] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.
- [23] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *FAST '04*. Springer, 2004.
- [24] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.