Verified Security for Browser Extensions

Arjun Guha
Brown University

Matthew Fredrikson
University of Wisconsin, Madison

Benjamin Livshits, Nikhil Swamy
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 IBEX, a new framework for authoring, analyzing, verifying, and deploying secure browser extensions. Our approach is based on using type-safe, high-level languages to program extensions against an API providing access to a variety of browser features. We propose using Datalog to specify fine-grained access control and data flow policies to limit the ways in which an extension can use this API, thus restricting its privilege over security-sensitive web content and browser resources. We formalize the semantics of policies in terms of a safety property on the execution of extensions and develop a verification methodology that allows us to statically check extensions for policy compliance. Additionally, we provide visualization tools to assist with policy analysis, and compilers to translate extension source code to either .NET bytecode 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 sources estimate that a third of all users of Firefox (some 34 million) use extensions [27], while the 50 most popular Chrome extensions have each been downloaded several hundred thousand times [13].

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, and can make use of local storage. A malicious or buggy extension can easily void many security guarantees 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 extensions. 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 request privileges they need in an explicit manifest [3], and, when installing an extension, the user is prompted to grant it these privileges.

We view the Chrome model as a step in the right direction—privileges in the manifest can be inspected independently of extension code; and the browser assumes the responsibility of enforcing access controls. However, from an empirical study of over 1,000 Chrome extensions (Section II), we find that this model is often not very effective in limiting the privileges of extensions. For example, nearly a third of the extensions we surveyed request full privileges over data on arbitrarily many web sites; and as many as 60% have access to a user's entire browsing history. In many of these cases, the language of Chrome's security manifests makes it impossible to state finer-grained policies to more precisely capture extension behavior.

In an effort to alleviate some of these shortcomings, we propose IBEX, a new framework for authoring, analyzing, verifying, and deploying secure browser extensions. Our model speaks to three main groups of principals: extension developers, curators of extension hosting services, and end-users.

While this paper focuses primarily on the subject of browser extensions, our work is motivated by, and speaks to, several important trends in software distribution. As evidenced by *app stores* for iOS, Windows, and Android devices and web apps in Chrome OS [32], software distribution is increasingly mediated by a centralized, curated service. In this context, automated software checking for both security and reliability becomes a plausible alternative to manual vetting, since curators have the ability to reject distributing applications that risk compromising the integrity of the ecosystem. Our work also explores the space of policies that apply to a growing number of HTML5 applications, running on the web, on the desktop, a mobile device, or within a browser. (Trends in Chrome OS suggest a convergence between these forms of applications.) A key component of IBEX is a lightweight, logic-based approach to policies that aims to find a balance between resources and rights specified at a flexible level of granularity, while still allowing for efficient and reliable enforcement.

A. *Overview of* IBEX *and contributions*

We discuss the key elements of IBEX (illustrated in Figure 1) in conjunction with our technical contributions, below.

Browser-agnostic API for extensions. We provide developers with an API that exposes core browser functionality to extensions. We expect programmers to write extensions in high-level, type-safe languages that are amenable to formal analysis, including, for example, the .NET family of languages, or JavaScript subsets like those explored in Gatekeeper [17]. Our API is designed for the static verification of extension security and thus mediates access to features that can be abused by buggy or malicious extensions.

A policy language for stating extension privileges. To describe an extensions privilege over specific browser resources, we propose using a logic-based policy language. Our language, based on Datalog, allows the specification of fine-grained authorization and data flow policies on web content and browser state accessible by extensions. We expect policies to be developed in conjunction with the extension code, either authored manually by extension developers, or, in the future, extracted automatically via analysis of extension code.

Tools for curators of an extension hosting service. We envisage the distribution of extensions to end-users via a curated extension hosting service, as adopted by Chrome, or Firefox. Extension developers submit extension code and policy to the hosting service and curators can avail of policy analysis tools we provide to determine whether or not an extension is fit for public distribution. Specifically, we discuss a policy visualization tool that helps a curator to estimate an extensions access rights on specific web pages.

A formal semantics of policies and extension safety. We give a formal notion of extension safety to define precisely when an extension can be said to be in compliance with a policy. 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. Our safety property is designed to be robust with regard to the composition of safe extension code with untrusted scripts.

Static checking of extension safety. We develop a methodology based on refinement typing (proven sound) to verify that extensions written in Fine [30], a dependently typed ML dialect, satisfies our safety condition. Static verification eliminates the overhead of runtime security monitoring, and promotes robustness of the browser platform since extensions can never raise unexpected security exceptions. We expect our verification tools to be used both by extension developers and, importantly, by curators prior to accepting extensions for distribution.

Cross-browser deployment. We utilize multiple code generators implemented by the Fine compiler (including a new JavaScript backend) to allow the same extension source to be deployable in multiple browsers. A key enabler of this feature is the use of a browser-agnostic core extension API, combined with the use of a standard ML-like source language. To date, we have deployed extensions in Internet Explorer 8, Chrome, and Firefox. Additionally, we show how to deploy extensions in C3 [23], a new platform for HTML5 experimentation developed entirely in a type-safe, managed language.

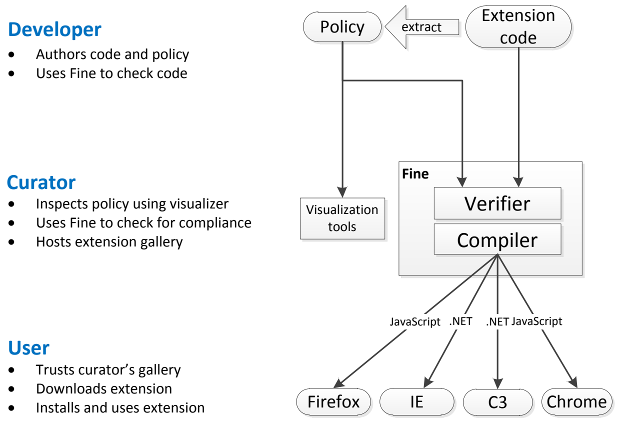


Fig. 1: Users, developers, and curators: an overview of IBEX.

Empirical evaluation. Our evaluation includes programming 17 extensions in Fine, specifying a range of fine-grained authorization and data 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.

B. *Outline of the paper*

We begin in Section II by discussing existing extension security models. Section III presents an overview of the design of IBEX. Section IV discusses our policy language and its visualization tool. Section V formalizes the semantics of policies and our safety property. Section VI shows how to statically verify extensions using refinement type checking. 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 limitations and future work. Section X discusses related work, and Section XI concludes.

II. A Survey of Existing Extension Models

Extensions have access to browser resources not usually available to scripts running on web pages. Unlike scripts on web pages, which can can only affect the page on which they are hosted, extensions can read and modify arbitrary web pages, and can even customize browsers' interfaces. Extensions are also not subject to the same-origin policy that applies to scripts on web pages—this allows them to communicate with arbitrary web hosts. With access to these and other capabilities, extensions, if malicious, pose a security risk. 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.

Below, we discuss the security mechanisms employed by Internet Explorer, Firefox, and Chrome to motivate the design of IBEX. Of these browsers, 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. *Internet Explorer's extension model*

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 ClSpring Trojan [4] 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's extension model*

Firefox extensions are typically written in JavaScript and can modify Firefox in fairly unrestricted ways. This flexibility comes with few 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 [22].

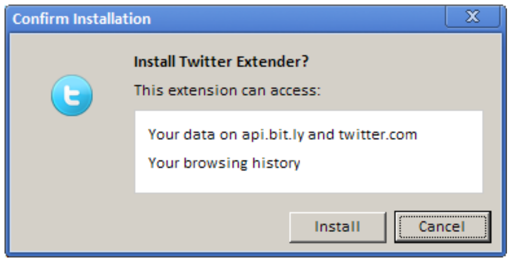
To protect end-users, Firefox relies on a community review process to determine which extensions are safe. Only extensions deemed safe are added to Mozilla's *curated extension gallery*. Firefox ordinarily refuses to install extensions that do not originate from this gallery. Users are thus protected from unreviewed extensions, but reviews themselves are error-prone and malicious extensions are sometimes accidentally added to the gallery. An example of this is Mozilla Sniffer [28], an extension which was downloaded close to 2,000 times, before being removed from the gallery after it was deemed malicious.

C. *Chrome's extension model*

Google Chrome extensions are written in JavaScript and hosted on extension pages, but they have access to 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 access to events related to browser tab manipulations. In

```
"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/" ]
```


Confirm Installation

Install Twitter Extender?

This extension can access:

Your data on api.bit.ly and twitter.com

Your browsing history

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"

Chrome's model, access to tabs implies that the extension has access to the user's browsing history. This is unfortunate—this extension 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 over-privileged 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 uninformative and the extensions require 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 sub-domains 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%
history (`tabs`)	694	60%
`bookmarks`	66	5%
`notifications`	15	1%

Fig. 4: Chrome extensions permissions statistics.

III. An Overview of Ibex

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 describes our solution using FacePalm, an extension we wrote, as a running example.

A. A running example: FacePalm

FacePalm 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 is part of the intended behavior of the extension. 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 manipulate all data on both `facebook.com` and `delicious.com`.

B. Programming type-safe extensions against a browser API

In contrast to Internet Explorer's native binaries, or Firefox extensions that make heavy use of dynamic programming techniques (e.g., "monkey-patching"), in Ibex, we advocate extensions to be programmed in high-level languages that are amenable to formal analysis. In this paper, we focus on extensions authored in an ML dialect for .NET called Fine. Our approach also applies naturally to other statically typed languages such as those provided by the .NET platform. In the future, we anticipate extending our work to handle extensions authored in statically analyzable subsets of dynamic languages like JavaScript.

As in Chrome, we provide APIs that allow extensions to access to browser resources like the DOM, as well as features like browsing history and the local file system not usually available to scripts on web pages. We show a fragment of this API below as a typed ML interface (we refine this API shortly).

```
(* Simple DOM API *)
val tagName: elt → string
val firstChild: elt → elt
val getAttr: elt → string → string
val textContent: elt→ string
(* Extension specific functionality *)
val readFile: filename → string
val sendRequest: url → string → string
val historyOnSite: string → list url
```

Two points about the design of this API are worth noting. First, we aim to provide extensions with functionality that is a strict super-set of the functionality available to web pages. However, we also aim for our interface to be browser-agnostic (to the extent that it is possible) to enable cross-browser deployment. Second, we provide access to features like browser history; however, our API is designed to allow restricting access to these resources at a fine granularity. For example, rather than providing an extension with access to all or none of a user's browsing history, functions like historyOnSite provide access to browsing history on a per-site basis. Further refinements of this interface to, say, browsing history restricted to a particular time interval are also possible.

We show a fragment of the code of FacePalm below. The getWebsite function inspects the tag and attributes of an element e, and returns the contents of e if it is a <div> node tagged with a website CSS class attribute. The rest of FacePalm traverses the DOM of a Facebook page, calls this function at various points, and, if appropriate, sends its result to `delicious.com`,

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

C. Policies and tool-support fine-grained specifications

Rather than provide all extensions with unfettered access to the entire extension API, we provide a policy language to provide specific privileges to extensions. 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 data flows, etc.

Our policy 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 a 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
```

Our aim is for the policies to capture the security-relevant behavior of extensions, allowing reviewers to audit extensions for security without necessarily having to conduct detailed code reviews. The policy statement above summarizes the behavior of getWebsite, the part of FacePalm that reads sensitive data out of a Facebook page, while hiding other details of FacePalm's implementation. Informally, this policy allows an extension to read text contained within `<div class="website">` elements. (The complete policy for FacePalm also describes the cross-domain flow from Facebook to `delicious.com`.)

Of course, the structure of real Facebook web pages are considerably more complicated than this first example suggests, leading to policies that are also more complicated. Rather than requiring reviewers to examine and understand Datalog, we provide a visualization tool that interprets policies on specific web pages, highlighting the content on a page to which an extension has been granted access.

D. Static verification of policy compliance

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, $(\mathcal{L};\mathcal{P})$-safety, on program executions that policies are intended to induce. The main technical development of this paper shows how, despite the richness of our policy language, we can statically verify extensions for compliance with a policy.

Our verification methodology involves annotating the API exposed to extensions with refinement types that capture security-related pre- and post-conditions. For example, the fragment of the DOM API shown earlier is annotated as shown below. This API makes use of dependent refinement types as provided by the Fine programming language—Section VI includes a detailed review of Fine, but we give a taste of our approach here.

```
(∗ 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 (like getWebsite) can be statically verified against this API for policy compliance using refinement type checking. Extensions that pass the type checker are guaranteed to be $(\mathcal{L};\mathcal{P})$-safe.

Static verification has a number of benefits. (1) Extension code is untrusted and never has to be manually inspected for potential vulnerabilities or malice. Curators (and interested end-users) need only look at their policies. (2) Verification also rules out potential runtime failures that can compromise the reliability of the browser platform. (3) By requiring access privileges to be determined statically, we avoid the pitfalls of dynamic discovery of access privileges identified by Koved *et al.* [21] in the context of Java access rights, namely that it is either error-prone or leads to over privilege. (4) We also observe that certain policies are not easily or efficiently enforced dynamically, including those based on $(\mathcal{L};\mathcal{P})$-safety, since this requires maintaining additional state at runtime, and also requires adding taint tags to arbitrary data values. Despite recent advances, dynamic taint tracking can be prohibitively expensive [8]. (5) Finally, we note that IBEX's deployment model makes the centralized extension hosting service a natural place for enforcement based on static analysis; such a facility is absent in decentralized software distribution.

E. Cross-browser deployment of extensions

In addition to verifying extensions, our approach allows extensions to be developed in a platform-independent manner. Our tools include a new code generator that allows us to compile extension code either to JavaScript or to .NET. This allows extensions to be authored once in Fine, and deployed on multiple browsers, including, via JavaScript, in Chrome and Firefox; via bindings from .NET to native code for Internet Explorer; and directly in .NET for C3.

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 interoperates 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 [35] 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 view elements' tag-names and certain attributes (e.g., styles and identifiers) as 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.

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).

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

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

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.

B. *Understanding policies*

Extensions are often designed with specific websites in mind, e.g., FacePalm's code closely tied to the structure of a Facebook web page. Policies, being an abstraction of the code, can also be closely tied to the page structure. Such policies can be hard to understand, unless the reader also understands the structure of the HTML used on the relevant websites.

We provide a visualization tool to assist users with the task of understanding security policies. Our idea is to interpret

```
(* Required to select elements by class (i.e., the "label" class) *)
1. ∀e. CanReadAttr e "class"
(* Requires to read the label text *)
2. ∀e, child. EltParent child e
     && EltAttr e "class" "label" ⇒ CanReadValue child
(* Permission to read website links *)
3. ∀data, label, labelText, website, parent.
       EltParent data parent && EltParent label parent
    && EltParent website data && EltParent labelText label
    && EltAttr label "class" "label" && EltTextValue labelText "Website:"
    ⇒ CanReadAttr website "href"
```

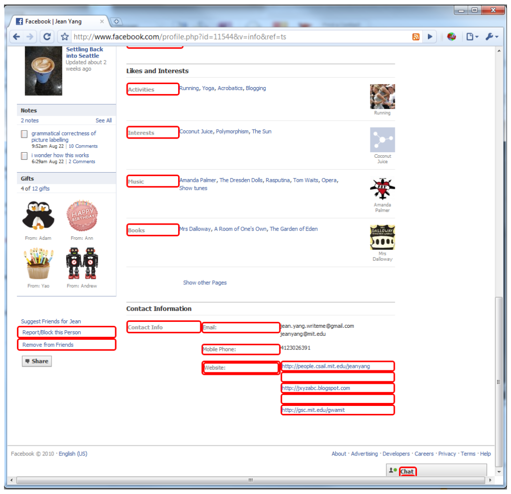

Facebook | Jean Yang
http://www.facebook.com/profile.php?id=11544&v=info&ref=ts
Settling Back into Seattle
Updated about 2 weeks ago
Notes
2 notes
See All
grammatical correctness of picture labelling
9:52am Aug 22 | 10 Comments
i wonder how this works
6:29am Aug 22 | 2 Comments
Gifts
4 of 12 gifts
From: Adam
From: Ann
From: Yao
From: Andrew
Suggest Friends for Jean
Report/Block this Person
Remove from Friends
Share
Likes and Interests
Activities
Running, Yoga, Acrobatics, Blogging
Running
Interests
Coconut Juice, Polymorphism, The Sun
Coconut Juice
Music
Amanda Palmer, The Dresden Dolls, Rasputina, Tom Waits, Opera, Show tunes
Amanda Palmer
Books
Mrs Dalloway, A Room of One's Own, The Garden of Eden
Mrs Dalloway
Show other Pages
Contact Information
Contact Info
Email:
jean.yang.writeme@gmail.com
jeanyang@mit.edu
Mobile Phone:
4123026391
Website:
http://people.csail.mit.edu/jeanyang
http://jxyzabc.blogspot.com
http://gsc.mit.edu/gwamit
Facebook © 2010 · English (US)
About · Advertising · Developers · Careers · Privacy · Terms · Help
Chat

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

predicates in a policy as XML selectors, and to highlight elements in a web page for which an extension has read or write access. Our tool takes the form of an extensions for Chrome and the bottom of Figure 6 shows a screen-shot of this extension when applied to FacePalm's policy. Specifically, it highlights 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.

While helpful, visualization is necessarily imprecise and is not intended to be a substitute for either manual inspection or formal analysis of the policy. Visualization only renders the impact of a policy on a particular web page and, as such, cannot be used to provide complete coverage since visiting all Facebook pages is impractical. Second, there are elements of policies which cannot easily be depicted in visual manner, e.g., information-flow policies.

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 illustrates this undesirable pattern: the attribute (class) that protects access to an element is mutable by the extension.

```
∀e. CanWriteAttr e "class"
∀e,k. EltAttr e "class" "readable" ⇒ 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 $\mathcal{P}$

const.	c	::=	$()$ | *true* | *false* | *op* | s (string) | η (nodes)
values	v	::=	x | c | (v_1, v_2) | $\lambda x.e$
expr.	e	::=	v | $e_1\ e_2$ | (e_1, e_2) | $\pi_1 e$ | $\pi_2 e$
		|	if e then e_1 else e_2
opers.	op	::=	getAttr | setAttr | getChildren | strEq
policy	$\mathcal{P}$	::=	$\cdot$ | $\forall \vec{x}.\phi_1, \ldots, \phi_n \Rightarrow \phi$ | $\mathcal{P}, \mathcal{P}'$
preds.	ϕ	::=	Parent $v_1\ v_2$ | EltAttr $v_1\ v_2\ v_3$
		|	CanReadAttr $v_1\ v_2$ | CanWriteAttr $v_1\ v_2\ v_3$
		|	FlowsFrom $v_1\ v_2$ | 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 the DOM (modeled as a binary tree of nodes); 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 λBX*)
```

$$
\begin{array}{l}
\lambda e.\text{if } (\mathsf{strEq}(\mathsf{getAttr}(e, \text{"tagName"}), \text{"div"})) \\
\quad \text{then if } (\mathsf{strEq}(\mathsf{getAttr}(e, \text{"class"}), \text{"website"})) \\
\qquad \text{then } \mathsf{getAttr}(\pi_1(\mathsf{getChildren}\ e), \text{"textContent"}) \\
\qquad \text{else ""} \\
\quad \text{else ""}
\end{array}
$$

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 straw-man) 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 $\mathsf{CanWriteAttr}\ \eta\ s_{key}\ s_{val}$ 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), $\mathsf{getChildren}$ 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 $\mathcal{L}'$. 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 log $\mathcal{L}$ ::= $\cdot \mid \phi \mid \mathcal{L}_1, \mathcal{L}_2$ eval. contexts $E[\bullet]$ ::= $\bullet \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid \pi_i E \mid$ if E then e_1 else e_2

$$\frac{\mathcal{P} \vdash (\mathcal{L}, e) \rightsquigarrow (\mathcal{L}', e')}{\mathcal{P} \vdash (\mathcal{L}, E[e]) \rightsquigarrow (\mathcal{L}', E[e'])}\ \text{E-Ctx} \qquad \frac{}{\mathcal{P} \vdash (\mathcal{L}, \lambda x.e\ v) \rightsquigarrow (\mathcal{L}, e[v/x])}\ \text{E-}\beta \qquad \frac{e' = e_1 \text{ when } v = \mathit{true} \quad e' = e_2 \text{ when } v = \mathit{false}}{\mathcal{P} \vdash (\mathcal{L}, \text{if } v \text{ then } e_1 \text{ else } e_2) \rightsquigarrow (\mathcal{L}, e')}\ \text{E-If}$$

$$\frac{}{\mathcal{P} \vdash (\mathcal{L}, \pi_i(v_1, v_2)) \rightsquigarrow (\mathcal{L}, v_i)}\ \text{E-}\pi \qquad \frac{v = \mathit{true} \text{ when } s_1 = s_2 \quad v = \mathit{false} \text{ otherwise}}{\mathcal{P} \vdash (\mathcal{L}, \mathsf{strEq}\ (s_1, s_2)) \rightsquigarrow (\mathcal{L}, v)}\ \text{E-Eq} \qquad \frac{\mathcal{L}; \mathcal{P} \models \mathsf{CanWriteAttr}\ \eta\ s_{key}\ s_{val}}{\mathcal{P} \vdash (\mathcal{L}, \mathsf{setAttr}\ (\eta, (s_{key}, s_{val}))) \rightsquigarrow (\mathcal{L}', ())}\ \text{E-SetAttr}$$

$$\frac{\mathcal{L}' = \mathcal{L}, \mathsf{Parent}\ \eta\ \eta_1, \mathsf{Parent}\ \eta\ \eta_2}{\mathcal{P} \vdash (\mathcal{L}, \mathsf{getChildren}\ \eta) \rightsquigarrow (\mathcal{L}', (\eta_1, \eta_2))}\ \text{E-GetCh} \qquad \frac{\mathcal{L}; \mathcal{P} \models \mathsf{CanReadAttr}\ \eta\ s_{key} \quad \mathcal{L}' = \mathcal{L}, \mathsf{EltAttr}\ \eta\ s_{key}\ s_{val}, \mathsf{FlowsFrom}\ \eta\ s_{val}}{\mathcal{P} \vdash (\mathcal{L}, \mathsf{getAttr}\ (\eta, s_{key})) \rightsquigarrow (\mathcal{L}', s_{val})}\ \text{E-GetAttr}$$

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

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 type soundness 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. We leave to future work the investigation of a security property for extensions that is suitable for use with revocation, while still being robust to the effects of untrusted JavaScript on the page.

VI. Static Enforcement of Extension Safety

This section describes a methodology based on refinement type checking that we use to statically verify that extensions comply with their policies. Section VI-A briefly reviews 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.

Our approach has a number of benefits, some of which were discussed in Section III-D. In light of the presentation of our safety property, we begin this section by highlighting two further benefits of our approach.

Robustness and modular verification: While $(\mathcal{L};\mathcal{P})$-safety is weak in the sense that it cannot model revocation, we find it particularly useful since it lends itself to a modular verification strategy. We can verify extensions for compliance with this property independently of page-resident JavaScript, and reason that this property is still preserved under composition with JavaScript. As such, this notion of safety is similar to the notion of *robust safety*, as formulated for use with model checking concurrent programs [16], or for verifying authenticity properties of cryptographic protocols [14].

Efficient policy enforcement: 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. Additionally, $(\mathcal{L};\mathcal{P})$-safety also allows us to enforce data-flow like taint-based properties with no runtime overhead.

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 [31].

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. Note that for uniformity, unlike ML, type applications are written in prefix notation (e.g., list int instead of int list).

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 → 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{ϕ}, 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. $\forall\alpha$ .l:list $\alpha \rightarrow$ m:list α {$\forall$x. In x l $\Leftrightarrow$ 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 $\rightarrow$ y:int{y != 0} $\rightarrow$ 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, P, 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 type level, kind-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 $\star \Rightarrow$ nat $\Rightarrow \star$; the kind of the propositional connective And is $P \Rightarrow P \Rightarrow P$; the kind of the user-defined predicate In is $\alpha{::}\star \Rightarrow \alpha \Rightarrow$ list $\alpha \Rightarrow P$.

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** $\forall$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{ϕ} is inhabited by values v:t, for which ϕ[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 [7], 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. 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 P-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 ⇒ P
 6 type EltDoc :: elt ⇒ doc ⇒ P
 7 type EltTagName :: elt ⇒ string ⇒ P
 8 type EltAttr :: elt ⇒ string ⇒ string ⇒ P
 9 type FlowsFrom :: α::⋆ ⇒ β::⋆ ⇒ α ⇒ β ⇒ P
10 (* DOM permission predicates *)
11 type CanAppend :: elt ⇒ elt ⇒ P
12 type CanEdit :: elt ⇒ P
13 type CanReadAttr :: elt ⇒ string ⇒ P
14 type CanWriteAttr :: elt ⇒ string ⇒ string ⇒ P
15 type CanFlowTo :: α ::⋆ ⇒ β ::⋆ ⇒ α ⇒ β ⇒ P
16 (* Metadata queries *)
17 val getChild : p:elt→ int→
18       r:option elt{∀ ch. r=Some ch⇒ EltParent p ch && 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 s_{key} was guarded by a premise that required the proposition CanReadAttr η s_{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 s_{key} reduces to an attribute s_{val}, and. importantly, records the facts EltAttr η s_{key} s_{val} and FlowsFrom η s_{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, for the example below, 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 reveals two subtleties, which we discuss next.

First, this model of side-effects rules out the possibility of strong updates, or, equivalently, dynamic revocation. Despite this weakness, as discussed earlier, the monotonic nature of our model lends itself to 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 is a characteristic feature of $(\mathcal{L};\mathcal{P})$-safety, and our model of side effects is set up to precisely model this property. Additionally, the robustness of $(\mathcal{L};\mathcal{P})$-safety with regard to the effects of JavaScript 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 of the programming model.

Second, 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 recording the history of parent/child relationships between nodes.

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 ⇒ P
 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 =
 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 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:bool{b=true $\Leftrightarrow$ x=y} and of the boolean operator (&&):x:bool $\rightarrow$ y:bool $\rightarrow$ z:bool{z=true $\Leftrightarrow$ x=true && y=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 $S = [\![\mathcal{P}]\!]$; a dynamic log $\mathcal{L}$ and its translation to an environment $\Gamma_L = [\![\mathcal{L}]\!]$; 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. [31], 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 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. Experimental Evaluation

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. We plan future work to infer policies via program analysis, and expect this to reduce programmer burden.

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.

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 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 [11]. 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: Controlling data flows and browsing history

NewsPers is an extension that personalizes `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

```
 1 (∗ Partial API to local file system, URLs, network, and history ∗)
 2 type url
 3 type CanReadFile :: string ⇒ P
 4 type UrlHost :: url ⇒ string ⇒ P
 5 type CanRequest :: url ⇒ string ⇒ P
 6 type CanReadHistory :: string ⇒ P
 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} → list url
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 → list url
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:list url → history:list url → list url
28 val nodesInOrder: o:list url → r:elt → list (e:elt{FlowsFrom e r})
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.

write arbitrary elements to an `nytimes.com` page, it could, for example, insert image tags to send requests to a third 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, once and for all, 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 discuss this approach here. Other extensions, including several of the miners, adopt this approach (with additional programmer effort) to protect against leaks via implicit flows.

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 over-privileged 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 IBEX 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. Additionally, because we can compile from .NET to JavaScript, we have also retargeted some of our extensions to run on Firefox. Each browser employs distinct back-ends and TCBs. In this section, we discuss system-level security guarantees that these browsers provide.

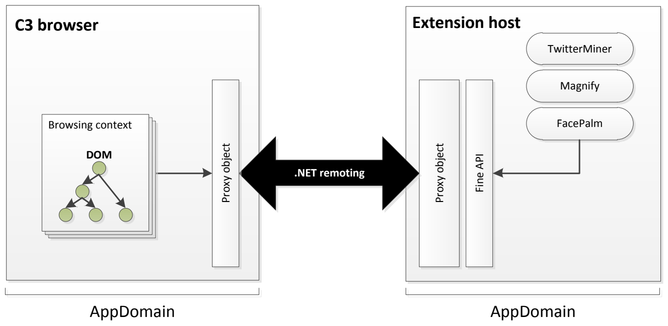


Fig. 11: C3 hosting architecture.

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 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 [19], we compile our Fine extensions to ordinary Chrome extensions by translating them to JavaScript. 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. C3: 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. Limitations and Future Work

This section discusses several limitations of approach and considers directions for future work.

Extension evolution and policy inference: Extension code is closely tied to the structure of the page. A web-site update can cause the extension to stop functioning properly. To help with this situation, we plan to investigate tool support to help extension authors update their code to account for page structure changes. In addition to assisting with code changes, we anticipate making use of weakest pre-condition inference for refinement types to automatically extract policies from code, reducing the programmer effort required to produce verified IBEX extensions.

Verified translation to JavaScript: We can deploy our extensions on various browsers because our compiler has two backends. To build extensions for Internet Explorer and C3, we use Fine's DCIL backend, which was previously proven type preserving [5]. To build extensions for Chrome and Firefox, we use Fine's new JavaScript backend. This paper does not establish the soundness of compilation to JavaScript; we leave this for future work.

Information flow: As presented, our extension APIs do not support non-interference based information flow control. Prior work shows that non-interference based information flow control can be enforced in Fine using monadic libraries equipped with value-indexed types. However, for simplicity, we restrict ourselves to policies based on taint-tracking, which yields a weaker security guarantee. In the future, we aim to make use of type coercions [29] to transform programs to automatically use monadic information flow controls.

Revocation: Our log-based model of DOM side effects rules out the possibility of specifying dynamic revocation policies. Devising a security property and a verification methodology that provides a higher fidelity model of effects, while still being robust to the effects of untrusted JavaScript is an open problem which we aim to address in the future.

X. Related Work

Browser extension security: Ter Louw *et al.* [25] 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. We survey the policies in use with Chrome extensions, and find many extensions to be needlessly over-privileged. Our survey results are complemented by recent unpublished work by Felt *et al.* [10], who also study the permissions used by Chrome extensions.

A number of researchers have explored the use of information flow for browser extension verification. Dhawan *et al.* present Sabre [8], a tool that instruments Firefox's JavaScript interpreter to track security labels at runtime. 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, other projects such as Chugh *et al.* [6] and Guarnieri *et al.* [17, 18] present information flow analyses for JavaScript that look for specific patterns of suspicious flows. However, because of the inherently dynamic nature of JavaScript, fully static approaches are difficult to apply to large segments of existing JavaScript code, generating interest in runtime enforcement [26]. Our Fine-based approach allows us to statically and soundly verify authorization and data flow properties of extensions; 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 [9, 20, 24]; 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 [12] and Google's Native Client [34] considers system-level sandboxing techniques for browser extensions. The OP [15] and Gazelle [33] 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 is effective at verifying device drivers. More recently, Zhou *et al.* explore the use of type safety to provide fine-grained isolation for drivers [36], and show how to apply their findings in a nearly backwards-compatible manner. Our work is in this tradition of static extension verification, but rather than focusing on system-level properties, we target those relevant to browser extension functionality.

XI. 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. It is our hope that IBEX will pave the way for a static verification mechanism of HTML5-based centrally-distributed browser extensions and applications on top of the HTML5 platform.

Acknowledgements: We thank Juan Chen, Cédric Fournet, Ben Lerner, Jean Yang, and the anonymous reviewers for both useful feedback and assistance with this work.

References

[1] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *ACM POPL*, 2002.

[2] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting browser extensions for security vulnerabilities. In *USENIX Security*, 2010.

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

[4] CA Technologies. Virus details: Win32/clspring family. `http://www.ca.com/us/securityadvisor/virusinfo/virus.aspx?ID=42280`, 2006.

[5] J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *ACM PLDI*, 2010.

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

[7] L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[8] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *ACSAC*, 2009.

[9] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *USENIX Security*, 2007.

[10] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of install-time permission systems for third-party applications. Technical report, University of California at Berkeley, 2010. UCB/EECS-2010-143.

[11] M. Fredrikson and B. Livshits. RePriv: Re-envisioning in-browser privacy. In *IEEE S&P*, 2011.

[12] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *USENIX Security*, 1996.

[13] Google Chrome. Popular Google Chrome extensions. `http://chrome.google.com/extensions/list/popular`, 2010.

[14] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11, July 2003.

[15] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE S&P*, 2008.

[16] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM TOPLAS*, 16, 1994.

[17] S. Guarnieri and B. Livshits. GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security*, 2009.

[18] S. Guarnieri and B. Livshits. Gulfstream: Incremental static analysis for streaming JavaScript applications. In *USENIX Conference on Web Application Development*, 2010.

[19] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *ECOOP*, 2010.

[20] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *USENIX Security*, 2006.

[21] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *ACM OOPSLA*, 2002.

[22] B. Lerner and D. Grossman. Language support for extensible web browsers. In *APLWACA*, 2010.

[23] B. Lerner, H. Venter, B. Burg, and W. Schulte. C3: An experimental extensible, reconfigurable platform for HTML-based applications, 2010. In submission.

[24] L. Li, X. Wang, and J. Y. Choi. SpyShield: Preserving privacy from spy add-ons. In *Conference on Recent Advances in Intrusion Detection*, 2007.

[25] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal of Computer Virology*, 4(3), 2008.

[26] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *oakland*, 2010.

[27] Mozilla Foundation. How many Firefox users use add-ons? `http://blog.mozilla.com/addons/2009/08/11/how-many-firefox-users-use-add-ons/`, 2009.

[28] Mozilla Foundation. Add-on security vulnerability announcement. `http://blog.mozilla.com/addons/2010/07/13/add-on-security-announcement/`, 2010.

[29] N. Swamy, M. Hicks, and G. Bierman. A theory of typed coercions and its applications. In *ACM ICFP*, 2009.

[30] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *ESOP*, 2010.

[31] N. Swamy, J. Chen, C. Fournet, K. Bhargavan, and J. Yang. Security programming with refinement types and mobile proofs. Technical report, MSR, 2010. MSR-TR-2010-149.

[32] K. Thomas. Chrome browser acts more like an OS, but security is unclear. `http://www.pcworld.com/businesscenter/article/220756/chrome_browser_acts_more_like_an_os_but_security_is_unclear.html`, 2011.

[33] H. Wang, C. Grier, E. Moshcuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *USENIX Security*, 2009.

[34] 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 S&P*, 2009.

[35] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *Workshop on Formal Aspects in Security and Trust*, 2004.

[36] 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 *USENIX OSDI*, 2006.