Verifying Web Browser Extensions' Compliance with Private-Browsing Mode

Benjamin S. Lerner, Liam Elberty, Neal Poole and Shriram Krishnamurthi

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-13-02
March 2013

Verifying Web Browser Extensions' Compliance with Private-Browsing Mode

Benjamin S. Lerner, Liam Elberty, Neal Poole and Shriram Krishnamurthi
Brown Computer Science
Brown University
Providence, United States
{blerner, lelberty, nbpoole, sk}@cs.brown.edu

***Abstract*—Modern web browsers implement a *private browsing mode* that is intended to leave behind no traces of a user's browsing activity on their computer. Currently, browser vendors ensure they have implemented this intent through subtle, whole-program reasoning about the browser. As a result, this feature is in direct tension with support for *extensions*, which let users add third-party functionality into their browser. To date, Mozilla and Google have taken different approaches in their support for extensions in private browsing mode, but both approaches are suboptimal and neither can reliably verify the behavior of extensions.**

We tackle this problem by creating a type system to analyze JavaScript extensions for observation of private browsing mode. Using this type system, extension authors and app stores can convince themselves of an extension's safety for private browsing mode. In addition, some extensions intentionally violate the private browsing guarantee; our type system accommodates this with a small annotation overhead, proportional to the degree of violation. These annotations let code auditors narrow their focus to a small fraction of the extension's codebase.

This paper explains the design of the type system, highlighting the new features needed to handle potentially-but-not-always unsafe APIs. We evaluate our system by retrofitting type annotations to Firefox's APIs and to a sample of actively used Firefox extensions, and discuss some of the challenges involved. Finally, we used the type system to verify several extensions as safe, find actual bugs in several others (which have been confirmed by their authors), and find dubious behavior in the rest.

I. Introduction

Modern web browsers are many-featured systems, providing a highly user-customizable environment for browsing, running web apps, and downloading content. People use browsers for a wide variety of reasons, and now routinely conduct sensitive transactions with them. Accordingly, recent browsers have added support for so-called *private browsing mode*, in which the browser effectively keeps no record of the user's activities: no history or cache is preserved, no cookies are retained, etc. The precise guarantee provided by private browsing mode is, however, rather more subtle, since a strict policy of retaining absolutely no record would preclude standard browsing activities such as downloading any files.

Ensuring the correctness of private browsing mode is therefore challenging on its own, but the situation is trickier still. Most browsers support *extensions*[1], written in JavaScript (JS), that allow users to customize the browser with third-party code—which run with the browser's full privileges and can hence also save files or other traces of the user's activity. In short, the presence of extensions can completely undermine the browser's privacy efforts.

The extension community is vibrant, with over 60 million daily extension users, and billions of total installations [18]. The potential privacy harm from faulty extensions is correspondingly huge.[2] Unfortunately, all three involved parties—browser authors, extension authors, and end-users—have difficulty properly protecting end-users from these problems. For browser authors, there is no universally safe default: running extensions automatically is clearly unsafe, but disabling them by default would also disable extensions such as AdBlock, which serve to enhance the overall privacy of the browser! Moreover, users currently have no way to make an informed decision about which extensions to re-enable. And worse still, even extension authors do not fully understand what private browsing mode entails: in the course of this work, for instance, one extension author we contacted replied, "when I wrote [the extension], the private browsing stuff didn't exist (to be honest, I'm only peripherally aware of it now)."

To date, browser vendors—primarily Mozilla and Google, whose browsers feature the most sophisticated extension support—provide extension authors with only rather vague guidelines on proper behavior when in private browsing mode. These guidelines are fail-open: extensions must actively check the browser's current mode; forgetting to do so breaks the privacy guarantee. Searching for such omissions is hard as, by definition, there is nothing to find! Mozilla has also instituted a manual code review of all extensions hosted on their site, but such manual processes scale poorly and are error-prone, and must be wholly repeated if policies change.

[1] These are distinct from binary *plugins* such as Flash or Java; we exclude plugins in this work.

[2] Even Dilbert is aware of the issue: see http://www.dilbert.com/fast/2012-05-03/

Contributions

We propose a new mechanism for verifying that extensions behave properly in private browsing mode. Our approach uses a lightweight type system for JS that exposes all potentially privacy-violating actions as type errors: the lack of type errors proves the extension is privacy-preserving. Authors can tell the typechecker precisely which errors to ignore, and only these annotations must then be audited in a security review. This paper makes the following contributions:

- We design a type system that segregates "potentially unsafe" code from "provably safe" code. Our system is *lightweight*—we typecheck *only* the code that may run in private browsing mode, and extension authors must only annotate code that is not provably safe. Most utility code is easily safe and requires no annotation. (Section IV)
- We implement our approach for the extension APIs found in Mozilla Firefox. Ascribing types to Firefox's APIs is non-trivial, and must match their quirky idioms with high fidelity in order to be useful. (Section V)
- We evaluate our system by retrofitting type annotations onto 12 real-world extensions (6.8K non-comment lines of code). Relatively few annotations are needed. We verify six extensions as safe, finding private-browsing (but not necessarily privacy) violations in the rest; three were confirmed by their authors as bugs. (Section VI)
- We generalize our experience with Firefox to lessons on how to design a privacy-conscious API. (Section VII)

II. Background: Extensions and Privacy

Browser extensions define both UI and behavior, the former written in markup that varies between browsers and the latter written in JavaScript. Though interesting in its own right [20], the UI code is essentially inert markup and therefore largely irrelevant for our security concerns here. We present a brief overview of how extensions are written in the two main extensible browsers: Firefox and Chrome.

A. Classic Firefox Extension Model

Firefox extensions define their functionality in JS, and trigger it via event handlers in the markup of their UI. These event handlers can use the same DOM methods as typical web pages, but in addition, they are given a suite of APIs providing low-level platform features such as file-system access and process management, as well as access to history and preferences, and many other functions. These APIs are obtained via a factory; for example, the following constructs a file object:

```
var file = Components
   .classes["@mozilla.org/file/local;1"]
   .createInstance(Components.interfaces
                     .nsILocalFile);
```

The `Components.classes` array contains "contract IDs" naming the various object types available via the API. Its elements are factories for constructing the desired instances; `Components.interfaces` names all the interfaces implemented by those instances. As of version 13, Firefox defines 847 contract IDs and 1465 interfaces: a huge API surface.

One of these APIs, namely `Components.interfaces.nsIPrivateBrowsingService`, allows code to check if it is running in private-browsing mode, and to cause the browser to enter or exit private-browsing mode. The former check is essential for writing privacy-aware code; the latter methods are particularly troublesome (see Section V).

B. Chrome and Jetpack Extension Model

The traditional security response to such a situation is to lock down and narrow the API surface area. Chrome's architecture has done so. The back-end of a Chrome extension is written against a much smaller API: a mere 26 objects to access bookmarks, cookies, tabs, etc [13]. Though there are no APIs to access the filesystem directly, there are experimental APIs for local storage, and extensions have unrestricted access to cross-origin XHR: extensions can still persist state. This relatively spartan API means Chrome extensions are inherently less capable than Firefox ones, and despite that they can still violate incognito-mode guarantees. In particular, an extension can create an implicit channel that leaks sensitive data from an incognito tab to a concurrent public one; from there the data can easily be saved to disk. See Section III-B for further discussion.

In a similar vein, Firefox has been working on a new extension model, known as "jetpacks" or "SDK addons". These extensions are a hybrid: they have access to a small API similar to Chrome's, but if necessary can get access to the `Components` object and access the rest of the platform. Such access is discouraged, in favor of enhancing the SDK APIs to obviate the need.

III. Subtleties of Private Browsing Mode

The intuitive goals of the various private browsing mode implementations are easy to state, but their precise guarantees are subtly different. In particular, the security versus usability trade-offs of private browsing mode are particularly important, and lead browsers to compromise a cleanly-engineered but unusable solution for a messier pragmatic one. We explore this trade-off, and then see how it leads to complications for supporting extensions.

A. Usability Trade-Offs

Private browsing mode is often described as the browser "not remembering anything" about the current session. This is a statement about the state of users' computers *after* their sessions have terminated; it says nothing about what happens *during* their sessions. In particular, a user might reasonably expect that the browser "work like normal" while browsing,

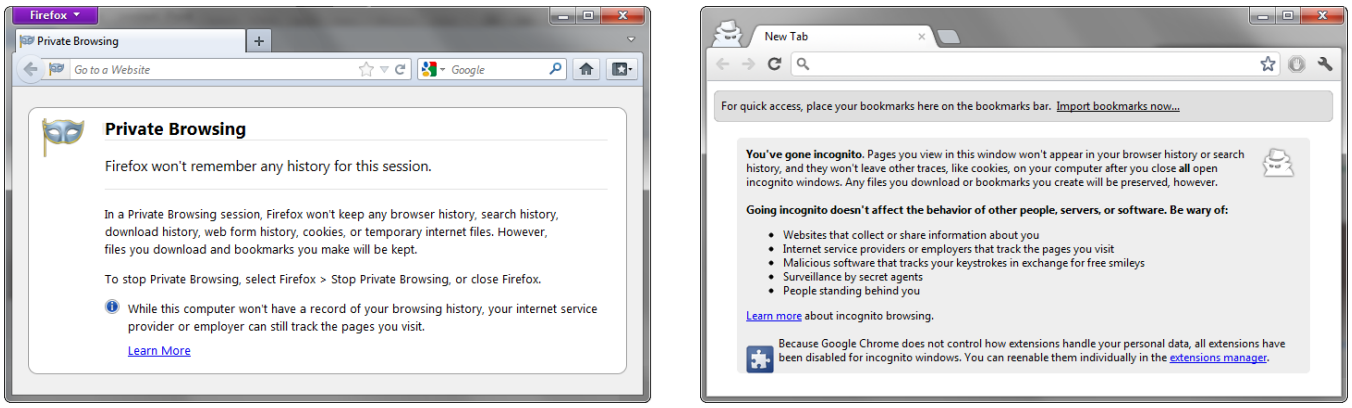

Figure 1. Firefox's and Chrome's initial incognito mode screens: note the warning Chrome provides regarding extensions

and then forget everything about the session afterward. But such an expectation is hard to engineer. The browser routinely handles many forms of persistent state on behalf of the user, such as history, cookies, or cache. (Even if this were wholly confined to memory, it would still pose a problem: as Aggrawal et al. [3] point out, virtual memory might swap private browsing mode information to persistent storage.) Additionally, a user can ask the browser explicitly to take certain stateful actions, such as downloading a file or updating bookmarks. *All* of these actions involve writing information to disk, and therein lies the problem. How can these actions be permitted and yet still meet the expectations of "not remembering anything" and "working like normal"?

One strict approach might be to prohibit all disk writes altogether. Indeed, the initial planning for Firefox's private browsing mode [24] states, "The bullet-proof solution is to not write anything to disk. This will give users maximum confidence and will remove any possible criticism of the feature from security experts." This is straightforward to implement, but it clearly breaks normal operations, which can substantially hamper the feature's utility to users.

A more pragmatic approach—and the one actually taken by both Firefox and Chrome—might permit the user's explicit actions, but disable the browser's internal implicit ones. This comes closer to preserving normal operations, but it muddies the security model of what the browser is permitted to do. This in turn creates usability and engineering challenges. The former is at best partially solved as both browsers present an introductory warning upon entering private browsing mode (Fig. 1) to clarify this to the user.

The engineering challenge, however, is harder: browsers must now distinguish data permitted to be written from prohibited data, an information-flow determination that is practically impossible to make soundly and automatically [4]. Instead, browser vendors currently revise their browsers' internals to avoid writing to disk in private mode, except on code paths triggered by user actions. This solution is tenable, but fail-open: it requires expert knowledge of the entire browser, specifically all possible code paths that access disk—but browser extensions can add new such code paths!

B. Extension Abilities and Private-Mode Transitions

While it is convenient to refer to "private browsing" as a *mode*, such that the browser is either in private mode or not, the reality is substantially murkier. Firefox allows the user to switch in and out of normal and private modes without restarting the browser. Chrome even allows the user to have both "normal windows" and "incognito windows" open simultaneously. Since our concern in this work is to prevent sensitive information from leaking out of private browsing sessions, it might seem that existing browsers make this impossible. An extension could conceivably hoard sensitive data in memory while in private mode, wait for a user-initiated transition to public mode, and then freely leak the data without consequence.

The mitigation for this attack differs in its details between the two browsers, but amounts to isolating extensions' state to within a single window, which is then the unit of normal or private modes. In particular, all the scripts that implement the behavior of windows in Firefox run in the context of that window.[3] When Firefox transitions between private and public modes, it closes all currently-open private windows and re-opens the public ones from the prior session. Crucially, this means extensions' state is effectively destroyed and re-initialized as these windows are opened, so extensions cannot passively hoard data and have it automatically leak into public mode. Instead, they must actively use APIs to persist their data, and we detect and flag such uses.

Firefox is considering adopting a model closer to Chrome's, where normal and incognito windows are open simultaneously. In Chrome, however, extensions are partitioned into two pieces: a background task running in a standalone context, and content scripts running in the context

[3] The expert Firefox programmer might know about "backstage pass" contexts, which might persist across such changes. Such contexts are internal to Gecko and to our knowledge cannot be intentionally accessed by script.

of individual pages. The background task can execute higher-privileged operations than content scripts can, but cannot obtain data about the user's open pages directly. Instead, the content script must actively send such sensitive data to the background task over a well-specified API, and again we can detect and flag such uses.

In short, browsers are currently engineered such that there is no implicit communications channel between private-mode windows and public ones. Persisting any data from one to the other requires explicitly using an API to do so, and our system (described in Section IV) is specifically designed to raise warnings about just those APIs. Accordingly, for the remainder of this paper, we can safely assume that the presence or absence of private mode is effectively *constant* while analyzing an extension, because it *is* constant for the duration of any given JS context.

C. Extension Challenges and Current Guidance

In this work, we assume that the browser vendor has correctly implemented private-browsing mode. (Verifying this assumption is an interesting but orthogonal problem. Recent efforts [17] have begun making progress in formally verifying properties of browser implementations; we anticipate work of this nature to apply here.) What happens to the privacy guarantee when a user has installed extensions? After all, extensions might also write to disk either automatically or in response to explicit user actions. The same engineering problem then arises for extensions too, but browser vendors cannot feasibly rewrite extensions' code (as they rewrote their own) to ensure compliance. The only mechanism available to browsers must be some form of interposition baked into the browser—but any sufficiently-flexible such mechanism would surely have worked for the browser itself! This quandary is vexing enough that it delayed Chrome's inclusion of incognito mode altogether [29]. Chrome's current pragmatic approach dodges the problem by disabling all extensions by default in incognito mode, and allowing users to re-enable them manually (see Fig. 1). This is reasonable but suboptimal: by default it excludes privacy-enhancing extensions such as AdBlock, and users are certainly less well-equipped to decide whether a particular extension is respecting of incognito mode than Chrome is.

As technical enforcements so far are inadequate, both Google and Mozilla have issued guidelines for well-intentioned extension authors to follow when writing extensions. These guides are cheerfully informal, and not prescriptive rules. For instance, Google provides only a "rule of thumb" [14] saying, "If a piece of data might show where a user has been on the web or what the user has done, don't store it if it's from an incognito window." Mozilla's guidelines are more specific, but still somewhat vague [25]: add-ons must not record "sensitive data", which they define as the URLs, domains, or content of pages the user has visited; all data "related to those pages" (including cookies and form data); and "any data used to customize the Firefox user interface based on what happens in private browsing mode".

Mozilla enforces its guidelines via manual code audits on all the extensions uploaded to their site. Unfortunately, these audits are a bottleneck in the otherwise-automated process of publishing an extension [40, 41]. It is also possible for violations—sometimes flagrant ones [27]—to slip through. Moreover, if policies ever changed, Mozilla would face the daunting task of re-auditing thousands of extensions.

IV. Approach

Mozilla's manual review process is the current most-stringent test for extension compliance with private-browsing mode. But with thousands of extensions of unknown quality and clarity, reviewers are challenged even to detect the privacy-relevant parts of the extensions' code. We instead want to automate much of that review process, and help focus the reviewers' attention on the relevant code directly. We design this as an *affirmative privacy burden*: rather than require all developers to annotate all code, we require annotations only where code might violate private browsing expectations. Extensions that adopt our system can be reviewed expeditiously by Mozilla, or enabled by default by Google, making it advantageous for all parties.

To do so, we define a *type system* for JS that can capture most of the idioms used in practice, but that will allow us to prevent access to potentially-harmful APIs. In particular, any such accesses must be syntactically marked in the code, making the reviewers' job a simple search, rather than a reading of the entire extension. Our type system is based on Typed JS [16], with several variations. For non-experts in type systems, we begin by informally describing the errors our type system raises to alert the programmer of violations, and the design considerations for our type system. For readers more familiar with type systems, we then present the technical features of our type system.

A. Informal Description

Type systems are program analyses that determine whether a semantic property holds of a given program, based on that program's syntactic structure. A type system is comprised of three parts: a type *language* for describing the types of expressions in the program, a type *environment* assigning types to the predefined APIs, and a type *checker* that takes as input a program to be checked and the type environment, and then attempts to validate the program against a set of rules; programs that pass the typechecker possess the desired semantic property.

Our particular type language defines a type, `@Unsafe`, which our environment assigns to the potentially-unsafe APIs (e.g., `file.create`) to prevent them from being called, and to the potentially-unsafe objects (e.g., `localStorage`) to prevent their properties from being accessed. This can be quite refined: objects may contain a mix of safe and unsafe

methods. For example, in our system, it is perfectly fine to open and read from existing files, but it is unsafe to create them; therefore, only the first line below causes an error:

```
file.create(NORMAL_FILE_TYPE, 0x644);
var w = file.isWritable();
```

The type checker will complain:

```
Cannot dereference an @Unsafe value at 1:0-11
  (i.e., file.create).
```

The programmer has three options to resolve this error: 1) rewrite the code to eliminate the call to `file.create`, or 2) prove to the type checker that the code is never called in private browsing mode, or 3) "cheat" and insert a typecast to hide the error without actually fixing it. Often, very simple changes will suffice as proof:

```
if (WeAreNotInPrivateBrowsingMode()) {
  file.create(NORMAL_FILE_TYPE, 0x644);
}
var w = file.isWritable();
```

(We show in Section IV-D4 how to implement the `WeAreNotInPrivateBrowsingMode` function.)

B. The Type System Guarantee

Extensions that have been annotated with types and pass the typechecker enjoy a crucial safety guarantee. This guarantee is a direct consequence of Typed JS's own type-safety theorem [16], the soundness of Progressive Types [33], and the correctness of our type environment:

If an extension typechecks successfully, using arbitrary type annotations (including `@Unsafe`), and if an auditor confirms that any "cheating" is in fact safe, then it does not violate the private-browsing mode invariants. Moreover, the auditor must check *only* the "cheating" code; all other code is statically safe.

As with any similar system, the soundness of our analysis assumes a model of the underlying system: here, that JS contexts in private mode are not reachable from any JS contexts in public mode or vice versa. (In Section III-B we explained why this model is appropriate.) We reiterate that verifying that the browser obeys this model is a fascinating challenge in its own right, but orthogonal to—and thus beyond the scope of— this paper.

In the following subsections, we explain how the type-checker recognizes the example above as safe, and make precise what "cheating" is and why it is sometimes necessary.

C. Type System Ergonomics

A well-engineered type system should be capable of proving the desired properties about source code and be flexible enough to prove others, with a minimum of invasive changes to the code. Typically, these properties are phrased as *preservation* and *progress* guarantees: respectively, well-typed programs preserve their types at each step of execution, and can make progress without runtime error. Our goal here is a relatively weak progress guarantee: we only prevent extensions from calling `@Unsafe` APIs; other runtime errors may still occur, but such errors cannot cause private-browsing violations.

As a strawman, one inadequate approach to proving privacy-safety might be to maintain a list of "banned words"—the names of the unsafe APIs—and ensure that the program does not mention them or include any expressions that might evaluate to them. Such an approach is inflexible: developers cannot name their functions with these banned words. It also prohibits much of JS's expressiveness, such as using objects as dictionaries (lest some subexpression evaluate to a banned word which is then used as a field name), which is very limiting for developers.

Another approach might graft checks for `@Unsafe` calls onto a more traditional, stronger progress guarantee. This is costly: for example, consider the information needed to ensure the (safe) expression `1+o.m("x")` makes progress. The type system must know that `o` is an object with a field `m` that is a safe function that accepts string arguments and returns something that can be added to `1`. Conveying such detailed information to the type system often requires substantial annotation[4]. But this is overkill in our setting: if any of the facts about the expression above were false, it would likely cause a runtime error, but *still would not call anything* `@Unsafe`.

Instead, we design a type system that *can* support such precise types, but that does not *force* them upon the developer. We provide a default type in our system that can type everything except the unsafe APIs: code that never calls anything unsafe does not need any annotation. Using such a type relaxes the progress guarantee to the weaker one above: nonsensical expressions may now typecheck, but still will never call `@Unsafe` APIs. Developers that *want* the stronger progress guarantee can add precise type annotations *gradually* to their code. The next subsection explains how our type system achieves this flexibility.

D. The Private-Browsing Type System

1) Preliminaries: Our type system contains primitive types for numeric, `null` and `undefined` values, variadic function types with distinguished receiver parameters (the type of `this` within the function), regular expressions, and immutable records with presence annotations on fields. It also contains type-level functions, equi-recursive types, and reference cells. For example, all of these features are necessary to define a (simplified) type for homogeneous arrays:

[4] Type *inference* for objects is of little help, as it is often undecidable [30].

```
type Array =
  typrec array :: * => * .
    typlambda α :: * . Ref {
      /[0-9]+|\+Infinity|-Infinity|NaN/ :? α,
      __proto__ :! Ref {
        concat :! forall β .
          [array<β>] array<β> -> array<β>,
      },
      * : _
    }
```

In words, `Array` is a recursive type-constructor (of kind `* => *`) whose numeric-like fields (i.e., field names matched by the regular expression)—if present—are bound to values of type α, whose prototype definitely is present and contains a method `concat` that operates on two arrays of the same type to produce a new one, and for which all other fields are missing (the underscore).

On top of this, we add support for (unordered) union and (ordered) intersection types. As a simple example, `True` and `False` are primitive types in our system, and `Bool` is the union type `(True + False)`. As a more complicated example, the addition operator has type

```
(Num * Num -> Num) & (Str * Any -> Str)
```

which will yield a number when given two numeric arguments, a string when given a string as its first argument, and otherwise will not apply [37]. This type is a *conservative approximation* and does not capture all of JS's behavior, but suffices for our needs (and will remain sufficient with the additional types below).

2) Safe Types: We define a (slightly-simplified) *extension type* that includes all possible JS values [36]:

```
type Ext = rec e .
  Num + Bool + Undef + Str + Null + Ref {
    __proto__ :! Src { },
    __code__ :? [e] e ... => e,
    * :? e
  }
```

In words, values of type `Ext` may be null, boolean, undefined, strings, numbers, or mutable objects whose fields, if present, are of the same type. The `__proto__` field is a read-only object (about whose fields we know nothing), and the `__code__` field models JS functions, which are objects with an internal code pointer. (In fact, we define the "thin arrow" syntax `[a] b -> c` to be syntactic sugar for the type `Ref { __code__ :! [a] b => c }`, where the "fat arrow" is the traditional arrow type of actual lambda expressions.)

`Ext` is the default type for all expressions, and any `Ext`-typed code need not be annotated, justifying our "lightweight" claims. As Section IV-C mentioned, developers are free to add more precise types to their code gradually. Any such types will be subtypes of `Ext`, meaning that richly-typed code will successfully interoperate with `Ext`-typed code without having to modify the `Ext`-typed code further.

3) Marking APIs with the `@Unsafe` Type: We add to our type language a new primitive type `@Unsafe`, which we ascribe in our initial type environment to all potentially-harmful APIs. This type is unrelated by subtyping to any other types besides `Top`, `Bot` or intersections or unions that already contain it. Accordingly, any attempts to use values of this type will cause the type checker to get stuck: because it is distinct from function types it cannot be applied; because it is distinct from object types it cannot be dereferenced, etc. `@Unsafe` values can be assigned to variables or fields, provided they have also been annotated as `@Unsafe`.

4) Checking for Private-Browsing Mode: Our efforts to segregate `@Unsafe` values from safe `Ext` code are overzealous: we do not need to prevent *all* usages of `@Unsafe` functions. Recall the revised example from Section IV-A: it specifically checks for private browsing mode, and now avoids performing `@Unsafe` actions as appropriate.

To capture this intuition, we define the typechecking rules for `if` statements as follows:

$$\frac{\Gamma \vdash \texttt{c} : \texttt{True} \qquad \Gamma \vdash \texttt{t} : \tau}{\Gamma \vdash \texttt{if c t f} : \tau}\ \textsc{If-True}$$

$$\frac{\Gamma \vdash \texttt{c} : \texttt{False} \qquad \Gamma \vdash \texttt{f} : \tau}{\Gamma \vdash \texttt{if c t f} : \tau}\ \textsc{If-False}$$

$$\frac{\Gamma \vdash \texttt{c} : \texttt{Bool} \qquad \Gamma \vdash \texttt{t} : \tau \qquad \Gamma \vdash \texttt{f} : \tau}{\Gamma \vdash \texttt{if c t f} : \tau}\ \textsc{If-Other}$$

For conditionals where we statically know whether the condition is `True` or `False`, we only typecheck the relevant branch: the other branch is statically known to be dead code. Otherwise, we must typecheck both branches. Under these rules the dead code could be arbitrarily broken; nevertheless it will never run. Note that here, "dead code" really means "not live in private-browsing mode".

This leads to our key encoding of the `nsIPrivateBrowsingService` API's `privateBrowsingEnabled` flag. Normally, this flag would have type `Bool`. But we only care when it is *true*; when it is false, it is fine to use `@Unsafe` values. We therefore give it the type `True`. If-True then permits the example in Section IV-A to typecheck without error.

5) "Cheating": As pointed out in Section III-A, we may want to allow extensions to use `@Unsafe` APIs even in private-browsing mode, to preserve "normal operations". This may be because they do not store "sensitive" information, or because they are run only in response to explicit user action. Statically determining whether information is sensitive is an information-flow problem, a thorny one we deliberately avoid. Moreover, the control flow of browsers is quite complex [21], making it challenging to link code to the event that triggered it. We are not aware of any successful efforts to apply static information flow to JS, and certainly none that also apply to the browser's control flow. Instead,

we require extension authors to annotate all those uses of `@Unsafe` values that cannot statically be shown to be dead code. To do this, we introduce one last type annotation: `cheat` τ. When applied to an expression, it asserts the expression has type τ and does not actually check its type.

Obviously, gratuitous cheating will let even unsafe extensions typecheck. Therefore all `cheat` typecasts must either be manually audited by a human to confirm their safety, or verified by more sophisticated (perhaps expensive) runtime systems. For now we assume a human auditor; by having these annotations we let future researchers focus on the remaining problems of fully-automated audit.

V. From Theory to Practice: Typing Firefox Extensions and APIs

We have laid out our technique that developers would ideally use from the outset as they develop new extensions, and the theorem in Section IV-A ensures that their efforts would be worthwhile. We now examine whether it is applicable to *existing* code. In this section, we present a worked example, where we annotate an extension to verify its safety. We then explain the details of instantiating our system for Firefox's APIs to support that example. In Section VI, we use this system to verify (or find bugs in) twelve extensions. In Section VII, we distill from this case study lessons in designing privacy-conscious and analysis-friendly APIs.

A. Worked Example: Extension Annotations

The "Open As Webfolder" extension [34] defines a popup menu item that allows users to open a given URL in Windows's WebDAV client. It it recently updated and is compatible with the latest Firefox version; it ought to be compliant with Mozilla's private browsing policies. Intuitively, it must be `@Unsafe`: it spawns a process (`wscript.exe`) and sends it instructions to open the current URL. However, in this extension these activities are only triggered by explicit user actions; perhaps this does not fall afoul of Mozilla's policy after all. (Recall from Section III-A that unsafe calls might be permitted to maintain normal operations.) Our goal in typechecking this extension is to push the `@Unsafe`ty of its actions up the call stack, until we reach a point where an auditor could decide whether an individual call is permissible.

An excerpt (edited for brevity) of the extension appears in Fig. 2. This code, with no additional annotations, will typecheck under the type environment we define below. In particular, the environment allows our type system to detect that `process` is of type `nsIProcess` on line 9, and therefore determine that the call to `process.run` is `@Unsafe` on line 23; similar reasoning holds for `tmpfile` and `foStream`. The boxed expressions are calls that our

[5] Type annotations are written in comments beginning with a colon; typed JS programs can therefore be run unmodified by existing JS engines.

```
1  var Cc = Components.classes;
2  /*: cheat @Unsafe */[5]
3  function openWebfolderViewForUrl(url) {
4    ...
5    var wscript = Cc["..."].getService(Ci.nsIProperties).
6      get("WinD", Ci.nsIFile);
7    wscript.append("system32"); wscript.append("wscript.exe");
8    ...
9    var process = Cc['...'].getService(Ci.nsIProcess);
10   process.init(wscript);
11   ...
12   var tmpfile = Cc["..."].getService(Ci.nsIProperties)
13     .get("TmpD", Ci.nsIFile);
14   tmpfile.append("open-webfolder.js");
15   tmpfile.createUnique(Ci.nsIFile.NORMAL_FILE_TYPE, 0x664);
16   var foStream = Cc["..."]
17     .createInstance(Ci.nsIFileOutputStream);
18   foStream.init(tmpfile, 0x02 | 0x08 | 0x20, 0x664, 0);
19   foStream.write(content, content.length);
20   foStream.close();
21   ...
22   var args = [..., tmpfile.path, url];
23   process.run(true, args, args.length, result);
24   tmpfile.remove(false);
25 }
26 /*: cheat @Unsafe*/
27 function openWebfolderView() {
28   openWebfolderViewForUrl(gBrowser.currentURI.spec);
29   return true;
30 }
31 function(event){/*:cheat Ext*/openWebfolderView();};
```

Figure 2. Excerpt from the "Open as Webfolder" extension

type system detects as `@Unsafe`—with no annotation needed. Because these calls are essential to the function's behavior, we annotate the entire function as `@Unsafe` (in the comment on line 2). The type system now accepts the body of the function without further complaint, but will instead complain about calls to this function: we have pushed the problem up one stack frame. Similar reasoning now shows that `openWebfolderView` must also be marked as `@Unsafe`, and finally we arrive at the last line, which is the event handler for the extension's popup menu item. Here we "cheat" the call to `Ext`: a suspicious typecast! Note that if the extension author inserted checks for private browsing (as in the example in Section IV-A) on lines 31 or 28, this last `cheat` would be unnecessary, and the code would typecheck cleanly.

With merely these three annotations, the code now passes our type checker. An auditor can look at the first two and dismiss them—it is always acceptable to state something is `@Unsafe`—and might decide that the last annotation is permissible after all, because it was triggered by explicit user action. No other part of the extension need be read; the reviewer has full confidence that the remainder of the code is privacy-preserving.

Encoding challenges: With this example in mind, our goal becomes clearer: we need an initial type environment in which the `process`, `tmpfile` and `foStream` variables (on lines 9, 12 and 16 respectively) are inferred to have types

nsIProcess, nsIFile and nsIFileStream, so that the remainder of the program is typed accordingly.

Retrofitting a type system onto an API that was not necessarily designed with types in mind is challenging. But recall that we merely want to ensure that while extension code runs, it does not call @Unsafe functions. We therefore can be fairly relaxed about permitting expressions that cause runtime errors—after all, an extension that aborts with an error will not subsequently write to disk. Instead we only require *type preservation*: we must ensure that no @Unsafe types are inadvertently made safe as the program runs. We face three key challenges in building the type environment: translating the available type information as well as possible, encoding @Unsafe values and the flag for private browsing mode, and encoding the Components object appropriately.

Analogously, retrofitting types onto extension code that was not designed with types in mind is likewise challenging. Our goal is a flexible type system that can analyze real, third-party extensions, but it is unreasonable to contort our analysis to handle every idiosyncratic program in existence; we expect some small level of cooperation from the developer. Accordingly, we also document the minor stylistic changes and local refactorings we applied to our sample extensions such that they are better analyzable by our system.

B. Translating Typed Interfaces

Most of Mozilla's APIs are defined in typed interface files (written in a variant of WebIDL [42]), which we parse into our type language. The translation begins smoothly: each interface is translated as a read-only reference to an object type; this ensures that extensions cannot attempt to delete or redefine built-in methods. Functions and attributes on interfaces are then translated as fields of the appropriate type on the translated object types.

However, these IDL files have three problems: they can be overspecific, underspecific, or incomplete. For example, a function declared to expect a string argument can in fact be given any JS value, as the glue code that marshals JS values into C++ will implicitly call toString on the value. By contrast, functions such as getElementsByClassName return an nsIDOMNodeList, whereas the semantics of the method ensure that all the nodes in that list are in fact of the more specific type nsIDOMElement. Finally, the contents of the Components object are not specified in any interface, but rather dynamically constructed in C++ code; similarly, some XUL elements are defined entirely dynamically by XBL code. We need a mechanism to address each of these difficulties.

Rather than hard-code a list of corrections, we can exploit two existing IDL features for a flexible resolution. First, IDL permits "partial" interfaces, which are simply inlined into the primary definition at compilation time. Second, IDL syntax includes "extended attributes" on interface members, functions and their parameters, which may affect the translation to types: for instance, [noscript] members are not included in the JS environment, and [array] parameters are of type Array<τ> rather than τ. For missing types, we create a "type-overrides" file and add new type definitions there. For over- and under-specific types, we define a new extended attribute [UseType(τ)] to replace the type specified by the IDL, and in the type-overrides file define partial interfaces whose sole purpose is to revise the shortcomings of the input IDL. For example, we define a "DOMElementList" type, and override getElementsByClassName to return it instead.

C. Encoding @Unsafe Values and the Flag for Private Browsing Mode

We define two additional IDL extended attributes: [Unsafe] and [PrivateBrowsingCheck], and annotate the relevant methods or properties in the Mozilla environment with them. As described in Section IV, these are translated to the types @Unsafe and True, respectively.

As mentioned in Section II-A, the nsIPrivateBrowsingService object also allows extensions to switch Firefox into and out of private browsing mode. Even though Firefox does not wholly restart, it does effectively recreate all JS contexts. Nevertheless, we consider exiting private-browsing mode to be poor behavior for extensions, so we mark these APIs as @Unsafe as well. Any justifiable uses of this API must now be cheated, and their use indeed justified in a security review.

D. Encoding the Components Object

All of Mozilla's APIs are accessed via the idiom

```
Components.classes[cID].createInstance(
    Components.interfaces.fooInterfaceName)
```

or a similar variant (such as getService on line 12 above). An accurate but imprecise type for this function would simply be nsIJSIID -> nsISupports: the argument type is an "interface ID", and the result is the root type of all the Mozilla interfaces. But this function can return over 1400 different types of objects, some (but not all) of which are relevant to private-browsing mode. We therefore need a more precise return type, and since this function is used ubiquitously by extension code, we must avoid requiring developer annotations. The key observation is that the set of possible return types is known *a priori*, and the specific return type is selected by the provided interface ID argument. This is known as "finitary overloading" [31], and can be encoded in our system with intersection types.

Consider a specific instance of the idiom above, to obtain an nsIFile:

```
((((Components[8].classes)[3][cID[1]])[2].createInstance)[6]
    ((Components.interfaces)[7].nsIFile)[4])[5]
```

We explain the types in stages, for each numbered subexpression above:

1) This string is a "contract ID"; we simply give it the general type String. (The precise string doesn't matter for our purposes; we use a placeholder variable instead.)

2) The object returned here is of type `nsIJSCID`, notable only for the presence of the `createInstance` and `getService` methods.
3) This array therefore can take the (overapproximate) type `Cc = Src { * : nsIJSCID }`.
4) We need an encoding that assigns to each known interface ID a distinct type, all of which are subtypes of their common base type `nsIJSIID`. One such encoding is

```
nsIFile_IID = Src {
  "##nsIFile##" : Bool,
  "IID" : Bool,
  * : Undef
}
nsIJSIID = Src {
  "IID" : Bool,
  * : (Bool + Undef)
}
```

   Each interface (e.g. `nsIFile`) has a uniquely named field that is definitely a `Bool`, and a field marking it an `nsIJSIID`; everything else is undefined. No two interfaces X and Y are subtypes of each other (since X has a boolean field "##X##" that Y requires to be undefined, and vice versa), but both are subtypes of `nsIJSIID` (objects whose fields could be boolean *or* undefined).
5) We want the return type in this case to be an `nsIFile`—because the argument (4) says so.
6) The `createInstance` method therefore must have type `nsIFile_IID -> nsIFile`. But this is obviously only one of 1400 types it can have: in general, it must have type X`_IID -> `X for every known interface X. Accordingly, we give it the type

```
(nsIFile_IID -> nsIFile) &
(nsIProcess_IID -> nsIProcess) &
(nsIPrefBranch_IID -> nsIPrefBranch) &
... &
(nsIJSIID -> nsISupports)
```

   (The final fallback case is needed for extensions that define custom components with their own instance types.)
7) This object is a dictionary of the known types of interfaces, along with a fallback for unknown ones:

```
type Ci = Src {
  "nsIFile" : nsIFile_IID,
  "nsIProcess" : nsIProcess_IID,
  ...
  * : nsIJSIID
}
```

8) The `Components` object is simply

```
type C = Src {
  "components" : Cc,
  "interface" : Ci
}
```

While we need to distinguish the various interface IDs from each other (step 4), we do not need to distinguish contract IDs (step 1). Buggy code that mixes the contract ID for one object with the interface ID for another will result in a runtime error, which cannot lead to a privacy violation.

The reader may well be concerned by the length of the type for `createInstance`. However, it is never manually written down or even ever displayed; it is computed from the input IDL files. We again use the type-overrides file (Section V-B) to indicate which methods ought to have this specialized type, rather than the ones declared in the IDL.

E. Un-Representable API*s*

Firefox 3 introduced a new API for code reuse, `Component.utils.import("script.js", [obj])`, that has three completely different effects: it evaluates the script in the current global scope (if no `obj` parameter was passed), or in the scope of the provided object (if it exists), or returns it as a new object (if it is null). This API is "used for sharing code between different scopes easily" [26], but its effects are quite challenging to typecheck. In particular, its return value depends on what script is passed in, but scripts are not directly associated with types, so we cannot use the same representation that we used with `createInstance` above.

We do not attempt to typecheck this API; any uses of it must be cheated and manually audited.

F. Accommodating Real-World Extension Code

Our type system is designed to be used during the development process rather than after, but existing extensions were not written with our type system in mind. We therefore permitted ourselves several local, minimal code refactorings to make the code more amenable to the type checker. These changes let us avoid many typecasts, and (arguably) make the code clearer as well; we recommend them as best practices for writing new code with our type system.

First, we ensured that all variables were declared and that functions were defined before subsequent uses. Additionally, developers frequently used placeholder values of the wrong type—`null` instead of `-1`, or `undefined` instead of `null`—that we corrected where obvious.

Second, our type system naïvely infers the type of variables from the type of their initializer, for which it infers the strictest type it can. For instance, initializers of `false`, `"foo"` and `null` yield types `False`, `/foo/` (the regular expression matching the literal string) and `Null` respectively—rather than the likely-intended types `Bool`, `String`, and whichever particular object type is eventually used. This is deliberate: distinguishing `True` from `False` values lets us elide dead branches of code and thereby check for private browsing mode. Similarly, distinguishing string literals from each other enables support for `JS` objects' first-class field names. For now, we manually annotate the initializers with

Extension	Size	Typedefs	Casts	Cheats	Violates PBM?	Confirmed by author?	Violates Privacy?
The Middle Mouse Button v1.0	51	0	3	0	No		
Print v0.3.4	59	0	4	0	No		
Rapidfire v0.5	119	0	12	2	No		
Commandrun v0.10.0	147	2	7	6	Yes	Yes	Yes
Open As Webfolder v0.28	153	1	7	6	Yes	No	No
CheckFox v0.9.2	188	1	15	0	No		
The Tracktor Amazon Price Tracker v1.0.7	232	3	8	2	No		
Fireclam v0.6.7	437	4	17	14	Yes	No	No
Cert Viewer Plus v1.7	974	1	27	35	No	No	Yes
Textarea Cache v0.8.5	1103	8	41	85	No		
ProCon Latte Content Filter v3.3	2015	10	107	97	Yes	Yes	Yes
It's All Text v1.6.3	2623	3	165	125	Yes	Yes	Yes
Total	8101	33	413	372	3	4	

Figure 3. Sizes (LOC, including comments and annotations) and counts of type definitions, checked type casts and unchecked cheats, for the extensions analyzed: note that not all private-browsing violations are actual privacy violations.

their more general types; a production-ready system might try automatically generalizing types.

Third, we replaced the idiomatic field-existence check `if (!foo.bar)` with `if (!("bar" in foo))`, as the type-checker will complain when the field does not exist in the former, whereas the latter has the same dynamic effect but does not impose any type constraints. (When the field name is known to exist, this idiom also checks whether the field's value is not null, zero or false; we did not rewrite such usages.)

Additionally, we permitted ourselves two other refactorings to accommodate weaknesses in our current prototype system. First, our system does not model the marshaling layer of Mozilla's API bindings; for instance, passing a non-string value where a string parameter is expected will yield a type error. We therefore added `(""+)` to expressions to ensure that they had type `Str`.

Second, Mozilla APIs include `QueryInterface` methods that convert the provided value from one interface to another. Code using these functions effectively changes the type of a variable during the course of execution. Our type system cannot support that; we refactored such code to use auxiliary variables that each are of a single type.

VI. Case Study: Verifying Firefox Extensions

To evaluate the utility and flexibility of our type system, two of the authors (both undergraduates with no experience with engineering type systems) retrofitted type annotations onto 12 existing Firefox extensions, chosen from a snapshot of Firefox extensions as of November 2011. Some were selected because we expected them to have non-trivial correctness guarantees; the rest based on their brevity and whether they mentioned unsafe APIs. All extensions had passed Mozilla's security review: ostensibly they should all comply with private-browsing mode. We had no prior knowledge of the behavior or complexity of the extensions chosen beyond their description.

The results are summarized in Fig. 3. In short, we analyzed 6.8K non-comment lines of code (8.1KLOC total, including comments and our type definitions), and found private-browsing violations in six of the twelve extensions—of which only four truly violate privacy. Below, we highlight interesting excerpts from some of these extensions, and discuss the detailed use of annotations in them.

A. Example Extensions

In the process of making these extensions pass the type checker, we were forced to cheat 38 call-sites to `@Unsafe` functions as innocuous. Those 38 call sites are potential privacy violations appearing in five extensions, of which we think four truly violate private browsing mode (the other uses `@Unsafe` functions to setup initial preferences to constant—and therefore not privacy-sensitive—values). We have contacted the authors of these extensions, and two have responded, both confirming our assessment. A sixth extension uses cheats slightly differently, and the process of typechecking it revealed a large security hole that we reported: it was confirmed by its author and by Mozilla, and was promptly fixed. We highlight three of these extensions—one (almost) safe and two not—to highlight how subtle privacy violations can be.

1) (Almost) Safe: Textarea Cache [38]*:* This extension maintains backups of the text entered by users into textareas on web pages, to prevent inadvertently losing the data. Such behavior falls squarely afoul of Mozilla's prohibition against recording data "relating to" web pages in private browsing mode. The developer was aware of this, and included a check for private-browsing mode in one of the extension's core functions:

```
1  textareaCache.inPrivateBrowsing = function () {
2    if (this._PBS)
3      return this._PBS.privateBrowsingEnabled;
4    else return true; // N.B.: used to be false
5  };
6  textareaCache.beforeWrite = function(node) {
7    if (this.inPrivateBrowsing()) return;
8    ...
9    this.writeToPref(node);
10 };
```

Our system recognizes that `inPrivateBrowsing` has type `() -> True`, and therefore determines the `@Unsafe` call on line 9 is dead code. (Note that this is strictly more expressive than checking for the literal presence of the private-browsing flag at the call-site of the `@Unsafe` function: the flag has been wrapped in a helper function that is defined arbitrarily far from the `beforeWrite` function, yet `beforeWrite` is itself correctly typechecked as safe.) Several other unguarded code paths, however, result in writing data to disk, and these are all flagged as type errors by our system. Tracing carefully through these calls reveals that they are all writing only pre-existing data, and not recording anything from the private session.

An interesting subtlety arises in this code due to backward-compatibility: This extension is intended to be compatible with old versions of Firefox that predate private browsing mode, and in such versions, clearly `inPrivateBrowsing` is false. Accordingly, the default on line 4 used to be `false`, which prevents the function from having the desired return type. Annotating and typechecking this code directly revealed this mismatch; once changed to `true`, the modified code validates as safe.

A cleaner, alternate solution exists if we allow ourselves to refactor the extension slightly. As written, the `_PBS` field is initialized lazily, and so must have type `nsIPrivateBrowsingService + Undef`. That `Undef` prevents the type system from realizing the `return false` is dead code. If we rewrite the initializer to be eager, then `_PBS` has type `nsIPrivateBrowsingService`, which is never undefined, and again the function typechecks with the desired return type.

2) Unsafe: ProCon Latte Content Filter [28]*:* This "featured" (i.e., nominated as top-quality[6]) extension maintains keyword-based white- and black-lists of sites. A user can add URLs to these lists that persist into subsequent browsing sessions; this persistence is achieved by APIs that store preferences in the user's profile. These APIs all are flagged by the typechecker as `@Unsafe`—and as we annotated this extension, we determined that these APIs were reachable from within private browsing mode. In other words, the type checker helped determine that URLs could be added to these lists even while in private browsing mode, which seems like a clear violation of Mozilla's policy (see Section III-C).

One could try to make the case that this particular scenario is slightly more nuanced, though: as with Open as Webfolder, pushing the `@Unsafe` annotations up the call tree reveals that these functions are *only* called in response to explicit user actions, and as we saw earlier sometimes such `@Unsafe` actions should be permitted. Nevertheless, we contacted the extension author, who confirmed that this behavior is a bug: URLs were not intended to persist past private browsing mode.

3) Unsafe: Commandrun [1]*:* It is obvious that the Commandrun extension must be unsafe for private browsing. In fact, it is egregiously unsafe, as it allows an arbitrary website to spawn a process (from a whitelist configured by the user) and pass it arbitrary data. (Even worse, the version of this extension we analyzed had a further flaw that would allow websites to bypass the whitelist checking.) Yet counterintuitively, this extension produces no errors about *calling* `@Unsafe` functions: no such calls are present in the source of the extension! Instead, the extension creates an object that will launch the process, and then injects that object into untrusted web content (edited for brevity):

```
1  CommandRunHandler = function() {
2    this.run = /*:cheat @Unsafe*/function(command, args){ ... };
3    this.isCommandAllowed = function(command, args){ ... };
4  };
5  CommandRun = {
6    onPageLoad: function(event) {
7      var win = event.originalTarget.defaultView.wrappedJSObject;
8      win.CommandRun = new CommandRunHandler();
9    }
10 };
```

The `CommandRunHandler.run` function (line 2) is annotated as `@Unsafe`, but it is never directly called from within this extension, so it does not directly cause any further type errors.

The true flaw in this extension occurs where the object is leaked to web content on lines 7 and 8, and our type system *does* raise an error here. Gecko, by default, surrounds all web-content objects in a security wrapper to prevent inadvertent tampering with them, but exposes the actual object via a `wrappedJSObject` field on the wrapper. Our type environment asserts that such wrapped objects must only contain fields of type `Ext`, but the `CommandRunHandler` object has an `@Unsafe` field, and therefore the assignment on line 8 causes a type error. The only way to make this code type-check is to cheat either the reference to `wrappedJSObject` or to the `CommandRunHandler`, thereby exposing this flaw to any auditor. We contacted the author of this extension, who promptly confirmed and fixed the bugs.

B. Use of Annotations

Besides the immediate impact of finding real bugs in existing, audited extensions, we also use our case study to examine what type facts were needed to analyze these extensions, and what sorts of annotations were used to provide that information. We describe both the effort of

[6] https://addons.mozilla.org/en-us/developers/docs/policies/recommended

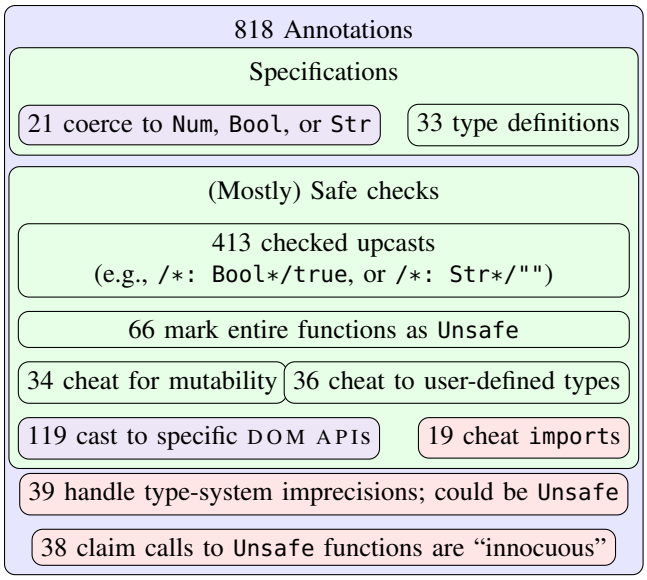

Figure 4. Detailed breakdown of all annotations used

setting up the type environment (and keeping it up to date), and the effort of annotating extensions.

1) The Type Environment: Our type-overrides file is currently 376 lines long, and revises the types for 123 APIs. For scale, the original 1465 interfaces are defined in 130KLOC of commented interface files. *No* changes were made to Firefox source code to construct this file, and it can be maintained independently of the original sources: little investment would be needed for Mozilla to adopt this system.

2) Annotation Counts: Figure 3 shows the counts of extension size and numbers of annotations in our sample extensions. All told we needed 818 annotations: 33 that define types, 413 that are checked type assertions, and 372 that are unchecked `cheat` typecasts. Anecdotally, nearly all type annotations are tiny, inline comments, whereas the type definitions can be lengthy, proportional to the complexity of the extension's objects. However, the raw numbers obscure how the annotations are used.

Figure 4 presents a clearer accounting of the various forms of annotation, grouped by three "severities" of annotation, rather than by "checkedness" of annotation. Green regions indicate reasonably-safe casts, either because they are checked or because they are "obvious", such as casting to user-defined types. Purple regions indicate slightly-complicated casts that nevertheless cannot cause safety violations, such as coercing from a string to a number, or specializing a generic `DOMElement` to a specific element type. Only the red regions are worrisome, casting values to `Ext` or claiming `@Unsafe` functions are innocuous.

Sixty-six member functions are marked as `@Unsafe` because they (transitively) call `@Unsafe` functions. As we have noted, such annotations are always admissible, and should not be construed as dangerous. Conversely, thirty-eight more cheats mark the calls to these functions as `Ext` — and these are precisely the calls that we wanted to find!

Many of the cheats are due to gaps in our environment: for example, 42 calls to `document.getElementById` need to be cheated down to the specific element type being returned, so that element-specific properties can be accessed. However, our type system is unaware of the document structure, so it cannot infer any type more specific than `nsIDOMElement`, which often does not suffice.

Thirty-five cheats cast `document` to `nsIDOMDocument` or its relatives, and 13 more cast `window` to `nsIDOMWindow`. These typecasts are usually trivially correct; but our environment ascribes the `Ext` type to both variables by default.

An additional 34 `cheat`s were needed to permit the mutation of fields on DOM nodes. Our current encoding of Mozilla's types represents each object as a read-only reference to the immutable object. This is necessary for subtyping to proceed soundly: if an event listener only requires the generic fields on an `nsIDOMEvent`, it should be callable with an `nsIDOMMouseEvent` too, but (mutable) references do not subtype covariantly. (Another possible encoding of objects might make each mutable field be itself a reference cell; preliminary attempts at this encoding failed and we have not pursued it further.) In general, it appears challenging to fit both dictionary-like objects and class-like object hierarchies into the same type system, and more work is needed to find an appealing solution.

Of the remaining 136 `cheat`s, roughly half are safe casts to `Ext`, to avoid unnecessarily-precise type checking in other parts of code, and the rest handle (safe) functions we have not yet modeled in our environment. These are slightly suspicious (because they assert without proof that the expression is safe), but are readily distinguished by a human as valid.

3) Eliminating Unnecessary Annotations: Because our prototype system was engineered for parsimony of features, we expressed "upcasts" to `@Unsafe` as cheats; similarly, a quirk of our encoding of Mozilla API objects required cheats to make their fields mutable. A more robust system would handle these more easily, lowering the annotation burden. Similarly, the 413 checked upcasts could be elided in favor of using a straightforward intraprocedural analysis [16] to infer them. A variation of that analysis could be used to infer most of the 119 unchecked casts to specific DOM APIs.

VII. Discussion

Anecdotally, most of the extensions were relatively easy to annotate—most of the annotation was done by two undergraduates with no prior experience with extensions or typed JS—though it required many iterations with the typechecker to annotate an extension fully. If the original author added annotations incrementally during development, the interactions would be greatly streamlined.

Broader issues of API design surface when trying to typecheck real-world extensions. We focus on two here: ambient reachability of `@Unsafe` values, and interoperating between "safe" and "richly-typed" code.

A. Ambient `@Unsafe` Values

The `window` object holds a reference to the `localStorage` object, which is `@Unsafe` in private-browsing mode. Therefore, `window` cannot have type `Ext`. The `document` object's `defaultView` field points to `window`, so it too cannot be of type `Ext`. All DOM nodes point to their `ownerDocument`, so they cannot have type `Ext`. Does this mean that any code dealing with DOM manipulation must always be annotated? More generally, are there paths through fields or methods on global objects that lead to `@Unsafe` values? If so, then no `Ext` value is truly safe.

One alternative is to assert that the `defaultView` field is `@Unsafe`. If this is truly the only way to obtain an `@Unsafe` value from `window`, then we have successfully contained the problem. However, such a solution is brittle and not future-proof to new web APIs. It also makes writing "generic" code harder: when accessing an arbitrary field of an object, the code must prove that the field name is not `"defaultView"`. (Arguably, such checks should be mandatory; real-world bugs can occur when they are omitted.[7])

Instead, a more principled approach is to *restrict the ambient environment*, and provide access to these features through tighter, curated APIs, along the lines of `Components` in Firefox, or the analogous `chrome` object in Chrome. We advocate for this approach: this work has shown that it is feasible to retrofit a security-oriented type system onto systems with more ad-hoc designs, but a cleaner starting point would yield a simpler system, and enable proofs of security in more cases and with less burden.

B. Interoperation Between "Safe" and "Richly-Typed" Code

We set out to build a type system where most of an extension's code did not need any annotations in order to typecheck. And if the extension is simple enough, or if it only operates on HTML web content, then this is indeed readily possible. The `Ext` type encompasses all values that such code produces, and no further effort is needed.

However, most interesting extensions must talk to the XUL chrome of the browser, and these APIs are "richly typed": they expect objects with very specific sets of fields, their member functions are truly methods that must not be called as arbitrary functions, and their fields may or may not be mutable. But objects are passed by reference in JS, and references subtype in the "obvious" direction only when they are read-only; this makes it unwieldy to handle mutable fields on these objects. Since mutability is a core part of the current DOM APIs, a cleaner solution is needed here.

One approach attempts to redesign the APIs [22] to curtail mutability. We offer another alternative: recall from Section IV-C that we are not primarily concerned with traditional progress guarantees, and so do not need to ensure that APIs are called *correctly*, rather just that no `@Unsafe` ones are called. As a result, a second advantage of restricting the ambient environment would be to *separate most of the* DOM APIs *from any possible* `@Unsafe` *values.* Then even XUL-interacting extension code could remain `Ext`-typed.

VIII. Related Work

Our work clearly builds upon a rich area of security research and a growing body of work analyzing JS. We consider each in turn.

A. Security-Related Efforts

Many recent projects relate to extension security, authoring, or analysis. Several entail pervasive changes within the browser [5, 9–11]; we focus on techniques that do not need such support, and briefly describe the most relevant such projects. None of them handle our present use cases.

1) Mozilla's Add-on Validator: Mozilla has had an automated add-on validator tool for several years. It looks for usage of deprecated APIs, unsanitary coding practices such as setting event handlers via a string, various blacklisted components, and other lint-like checks. To our knowledge, it does not at all aid in detecting private-browsing violations in extensions.

2) ADsafety: The closest relative of our work is ADsafety [32], which uses Typed JS [16] to verify the correctness of ADsafe [2]. That work focused primarily on verifying the ADsafe sandbox itself, and then used a type similar to our `Ext` to typecheck "widgets" running within that sandbox. Unlike extensions here, the environment available to widgets is entirely `Ext`-typed; indeed, the whole purpose of a sandbox is to eliminate *all* references to unsafe values!

3) IBEX: Guha et al. [15] develop Fine, a secure-by-construction language for writing extensions. Fine is pure, dependently-typed, and bears no resemblance to idiomatic JS. Extensions must be (re-)written entirely in it in order to be verified. Accordingly, the barrier to entry in their system is quite high, and they explicitly do not attempt to model the browser APIs available besides the DOM.

4) VEX: Bandhakavi et al. [4] design a system that statically attempts to discover unsafe information flows in extension code, for instance from unsanitized strings to calls to `eval`. By their own admission, their system is neither sound nor complete: they explicitly check only for five flow patterns in extensions and so miss any other potential errors, and any errors they raise may still be false positives. This provides no reliable guarantee for browser vendors.

[7] www.google.com/support/forum/p/Google+Docs/thread?tid=0cd4a00bd4aef9e4

Additionally, they do not address the conditional safety of API usage which is the hallmark of the private-browsing mode problem.

5) Beacon: Karim et al. [19] design an analysis for Mozilla Jetpack extensions (see Section II-B) to detect capability leaks, where privileged objects (such as unmediated filesystem objects) are exposed to arbitrary extension code. While laudable, this approach does not work for detecting private-browsing violations: filesystem capabilities are entirely permitted in public mode. Additionally, their tool is unsound, as it does not model reflective property accesses.

B. Language-Level Analyses

1) Progressive Types: As mentioned, our type system is based on that of Guha et al. [16], with enhancements that simplify reasoning about our relaxed progress guarantees. These enhancements are a form of *progressive typing* [33], in which the programmers using a type system can choose whether to defer some static type checks until runtime, in exchange for a easier-to-satisfy type checker.

2) Type Systems for JS: Typed JS is one of a handful of type disciplines for JS. The two most fully-featured are the Closure compiler [12] and Dependent JS [8]. The former is informally defined, and makes no claims that its type system entails a soundness guarantee. Further, the type language it uses is too coarse to help with the problem examined here. Dependent JS, by contrast, uses dependent types to capture challenging idioms in JS, such as the punning between arrays and dictionaries, and strong updates that change variables' types. However, the largest example the authors checked using Dependent JS is barely larger than the third-smallest extension we examine. Moreover, their language imposes huge annotation overheads: the type annotations are comparable in length to the original program! In short, while powerful, such a system is impractical and overkill for our purposes, and we can achieve our desired guarantee without the proof obligations entailed by dependent type systems.

3) Language-Based Security: Schneider et al. [35] survey the broad area of language-based security mechanisms. Cappos et al. [7] build a language-based sandbox for Python, such that even privileged scripts cannot access resources they should not. And other sandboxes exist for JS along the lines of ADsafe [2, 23, 39] to try to corral web programs. But none of these approaches explicitly address the modal nature of enforcement that we need for private-browsing guarantees.

4) Certified Browsers: Jang et al. [17] present an implementation of a browser kernel implemented in Coq [6], which allows them to formalize desirable security properties of the browser kernel such as non-interference between separate tabs, and the absence of cookie leakages between sites. Their current development is for a fixed-function browser; enhancing it to support extensions and private-browsing mode are intriguing avenues of future work.

IX. Conclusion

We have presented the challenges faced by browser vendors in supporting both private-browsing mode guarantees and third-party extensions. We have designed and implemented a lightweight type system that precisely signals where potentially-unsafe APIs are used, and demonstrated the feasibility of retrofitting these types onto previously written extensions. Finally, we have suggested some lessons learned on designing security-conscious extensibility APIs.

References

[1] A. Abeling. commandrun :: Add-ons for Firefox. Retrieved Nov. 2011. https://addons.mozilla.org/en-us/firefox/addon/commandrun/.

[2] ADsafe. Retrieved Nov. 2009. http://www.adsafe.org/.

[3] G. Aggrawal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *USENIX Security Symposium*, Aug. 2010.

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

[5] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, Mar. 2010.

[6] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[7] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2010.

[8] R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2012.

[9] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Annual Computer Security Applications Conference (ACSAC)*, Dec. 2009.

[10] V. Djeric and A. Goel. Securing script-based extensibility in web browsers. In *USENIX Security Symposium*, Aug. 2010.

[11] M. Fredrikson and B. Livshits. RePriv: Re-envisioning in-browser privacy. Technical report, Microsoft Research, Aug. 2010. http://research.microsoft.com/apps/pubs/default.aspx?id=137038.

[12] Google. Closure tools — Google developers. Retrieved Nov. 2012. https://developers.google.com/closure/compiler/.

[13] Google. chrome.* APIs – Google Chrome extensions. Retrieved Apr. 2012. http://code.google.com/chrome/extensions/api_index.html.

[14] Google. Overview – Google Chrome extensions. Retrieved Apr. 2012. http://code.google.com/chrome/extensions/trunk/overview.html.

[15] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy (Oakland)*, May 2011.

[16] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming Languages and Systems (ESOP)*, 2011.

[17] D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security Symposium*, Aug. 2012.

[18] E. Jostedt. Firefox add-ons cross more than 3 billion downloads! Written July 2012. https://blog.mozilla.org/blog/2012/07/26/firefox-add-ons-cross-more-than-3-billion-downloads/.

[19] R. Karim, M. Dhawan, V. Ganapathy, and C.-c. Shan. An analysis of the mozilla jetpack extension framework. In *26th European conference on Object-Oriented Programming*, 2012.

[20] B. S. Lerner. *Designing for Extensibility and Planning for Conflict: Experiments in Web-Browser Design*. PhD thesis, University of Washington Computer Science & Engineering, Aug. 2011.

[21] B. S. Lerner, M. J. Carroll, D. P. Kimmel, H. Q. de la Vallee, and S. Krishnamurthi. Modeling and reasoning about DOM events. In *USENIX Conference on Web Application Development (WebApps)*, June 2012.

[22] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2009.

[23] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. Technical report, Google Inc., 2008. http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.

[24] Mozilla. PrivateBrowsing – MozillaWiki. Retrieved Apr. 2012. https://wiki.mozilla.org/PrivateBrowsing.

[25] Mozilla. Supporting private browsing mode – MDN. Retrieved Apr. 2012. https://developer.mozilla.org/En/Supporting_private_browsing_mode.

[26] Mozilla. Components.utils.import – MDN. Retrieved Apr. 2012. https://developer.mozilla.org/en/Components.utils.import.

[27] S. Newton. ant video downloader firefox addon tracking my browsing. Written May 2011. http://iwtf.net/2011/05/10/ant-video-downloader-firefox-addon-tracking-my-browsing/.

[28] H. Paolini. ProCon latte content filter :: Add-ons for Firefox. Retrieved Nov. 2011. https://addons.mozilla.org/en-us/firefox/addon/procon-latte/.

[29] M. Perry. Extensions in incognito. Written June 2010. http://blog.chromium.org/2010/06/extensions-in-incognito.html.

[30] B. C. Pierce. Bounded quantification is undecidable. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Jan. 1992.

[31] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[32] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: type-based verification of JavaScript sandboxing. In *USENIX Security Symposium*, Aug. 2011.

[33] J. G. Politz, H. Q. de la Vallee, and S. Krishnamurthi. Progressive types. In *ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, Onward! '12, 2012.

[34] J. Reschke. Open as webfolder :: Add-ons for Firefox. Retrieved Nov. 2011. https://addons.mozilla.org/en-us/firefox/addon/open-as-webfolder/.

[35] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. *Lecture Notes in Computer Science*, 2000:86–101, Nov. 2001.

[36] D. Scott. Lambda calculus: Some models, some philosophy. In *The Kleene Symposium*, 1980.

[37] V. St-Amour, S. Tobin-Hochstadt, M. Flatt, and M. Felleisen. Typing the numeric tower. In *14th Practical Aspects of Declarative Languages (PADL)*, 2012.

[38] H. SUN. Textarea cache :: Add-ons for Firefox. Retrieved Nov. 2011. https://addons.mozilla.org/en-us/firefox/addon/textarea-cache/.

[39] The Caja Team. Caja. Written Nov. 2009. http://code.google.com/p/google-caja/.

[40] J. Villalobos. The add-on review process. Written Feb. 2010. http://blog.mozilla.org/addons/2010/02/15/the-add-on-review-process-and-you.

[41] J. Villalobos. Queue weekly status 2012-04-20. Written Apr. 2012. https://forums.mozilla.org/addons/viewtopic.php?f=21&t=8719.

[42] W3C. Web IDL. Written Apr. 2012. http://www.w3.org/TR/WebIDL/.